A Systems Perspective of Software Runtime Bloat - Origin, Mitigation and Power-Performance Implications

A THESIS
SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy
IN THE FACULTY OF ENGINEERING

by

Suparna Bhattacharya

under the guidance of
Prof K. Gopinath and Dr. Manish Gupta

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012
MAY 2013
FOR

*My parents,*

to whom I owe my love for academics

AND FOR

*My husband,*

whose love kept me going
The dreams are so beautiful. Of course they are further than they seem ...

but that only means that the bridges I build must be longer.

- found in the notes of Satyajit Bhattacharya (August 1983- April 2006)

A Personal Note

For my 23rd birthday, an unusual surprise gift awaited me. It was a book of twenty-five poems written by my ten year old little brother. Putting it together in the form of a book took longer than anticipated, so the book ended up reaching my hands only on my next birthday with a “better late than never” note scribbled on the first page. The pages had been carefully typeset by my father and bound into a volume somewhat audaciously titled the “Golden Best of Satyajit Bhattacharya”. The poems were brimming with the joyful imagination of a child. It included topics ranging from how he loved the world of books to his dream of having an entire pail of ice-cream, from empathy for neglected animals to the fun of walking barefoot and the excitement of summer holidays. They still make me smile when I read them today.

Satyajit never turned 23. Just a few months short of his 23rd birthday, we lost him forever. He was the heart and soul of our family. One of his friends said he was a genius with a heart of gold, for another, he was the most loving person she knew, many others would miss his ready wit. For me, he is still the little brother I could never imagine growing older without.

A year after we lost him, I started my PhD. Perhaps it was inspired by a tiny hope of finding a way to honor how precious life is and what he meant to us, simply by following what I felt I could do best. I am sure it would have amused him no end to have a thesis in his memory. But then, life takes us all by surprise. Today, I prepare this manuscript in remembrance of the joy he brought us in his short life. This thesis is far from the masterpiece I had once hoped it would be – instead it is more like an imperfect snapshot of ideas struggling to come out into the light of learning, a light that only dawns slowly as we grow a little wiser each day. But in the words of my brother, when he was ten, “I still hope you like it, or think its fair ...”.
Acknowledgements

The best part of writing this thesis is the opportunity it gives me to thank those who made it possible. First of all, I would like to thank my advisors Professor K. Gopinath and Dr. Manish Gupta for the honor and privilege of working with them. I am grateful to Professor Gopinath for his wisdom in recognizing that I had picked a difficult and complex problem where success would not come easily and yet encouraging me to pursue it with whole hearted enthusiasm. With his rare breadth of knowledge in the diverse technical areas that this problem spanned, he opened my mind to exploring tools and techniques beyond my limited repertoire. This research would never have taken shape it did without his guidance and the belief he maintained in my abilities even when I was close to doubting them. I thank him for making my experience of research an immensely enjoyable and enriching one.

To Manish, I owe the inspiration to do a PhD, the choice of my research problem and insight into the compiler optimization perspective of the problem. It was during a discussion at ISLPED 2008 that he started it all by pointing me to the work on runtime bloat by Mitchell, Sevitsky and Schonberg. I have come away enriched in many ways from our discussions over the course of my research.

This dissertation traverses multiple technical areas and diverse research techniques. Such intersectional research was feasible only because I was fortunate to get a head start in each of these areas through the opportunity of working with some excellent collaborators. In particular, I would like to thank Dr. Karthick Rajamani, Dr. Mangala Gowri Nanda, Mrinal Kanti Das and Professor Chiranjib Bhattacharyya, who co-authored some of the publications related to this thesis and even joined the midnight oil camp with me near submission deadlines. I have gained a deep respect for their dedication, talent, depth of knowledge and skill in the art of research.
and look forward to the joy and privilege of collaborating with them again in the future.

As a person coming from an OS background, who had never touched Java before, let alone framework based applications, I owe a lot to Gary Sevitsky, Dr. Nick Mitchell, Dr. Edith Schonberg and Dr. Matthew Arnold for sharing their rich insights on the nature of Java runtime bloat and its root systemic causes, through regular conference calls and e-mail discussions. I also thank Bob Blainey for giving me the opportunity to lead an IBM Academy of technology study on “Software Bloat Causes and Consequences” and all the members of the study team for their insights and contributions. A special thanks to Kiran Subbaraman, Jojo Joseph and Ciju Rajan for a collaboration on lean middleware experiments and to Vijay Mann, Venkateswara Madduri and Dr. Manish Gupta – these case studies gave me a practical feel for the challenges in reducing bloat in real applications and shaped my explorations in the thesis. In addition, I thank Dr. Paul McKenney, Dr. C. Mohan and Kalpana Margabandhu for regular advice and my colleagues in the IBM Linux Technology Center energy management team, the Java Technology Center and IBM research for their help and support. A few of these contributions are acknowledged individually in the relevant chapters of this dissertation.

As an external registration student, my PhD experience involved bridging one more intersection – that between the worlds of academia and industry. Ultimately, I think I was fortunate to experience the best of both worlds. I thank Professor Y. Narahari and Professor M. Narasimha Murthy, the current and previous chairman of the CSA department at IISc for their support and encouragement all through. It was the support extended by Professor Narahari at a crucial stage that enabled me stay on course with my PhD. I also thank senior management at IBM India Software Lab, Harish Grama, Dr. Ponani Gopalakrishnan, Reena Malangone, Mickey Purohit and my manager Premalatha M. Nair for their support.

I would like to thank the faculty and students of the CSA department for creating a wonderful academic environment where “research is in the air”. In particular, I must thank Professor Y. N. Srikanth, Professor Mathew Jacob, Professor Govindrajan and Professor Varsha Apte for their feedback and advice at different stages of my research. I thank my comprehensive committee members Professor Y. N. Srikanth, Professor S. K. Nandy and Professor Chiranjib Bhattacharyya. I also thank Professor Deepak D’Souza, Dr. Sriram Rajamani, Dr. Aditya
Nori, Dr. Madhu Gopinathan, Professor Aditya Kanade and Professor K. V. Raghavan for introducing me to the fascinating area of program analysis and verification through a range of courses and engaging discussions.

I thank Ms Lalitha, Ms Suguna and Ms Meenakshi from the CSA office for taking special care to ensure I had a smooth experience all through these years. I also thank Kushal from the CASL lab for cheerfully handling any administrative help that I required.

Of course, the most enjoyable part of IISc campus life is the time spent with fellow research students, the only people who really understand what it is like. I thank Aparna, Archana, Arun, Arnab, Vasanta, Raghavendra, Meghana, Rupesh, Sripathi, Geetha, Sujit, Saheli, Mrinal and Abhijit for those times. I also thank my labmates in CASL, Kamala, Prachee, Shiva, Gupta, Ganesh, Pankaj, Srimugunthan, Prasenjit, Amar, Neeraj, Priyanka, Aravinda, Piyush and Hasmukh for the vibrant environment in the lab, my hostmates Piyali, Madhuja, Karthika, Bandita and Moitreyee and my first year classmates Neeraja, Priyank, Arun, Vivek, Bhupi and Vipul for all the good times we had.

I thank my father and my mother for being a constant inspiration to me through the difficult times in our lives and reminding me to find joy in little things. I am glad I finally listened to their advice to do a PhD even if only about 14 years late, because now I know what an amazing experience I would have missed if I did not. I thank my sister, Barnali and my nieces, Mitali and Mouli for always welcoming me with affection even though I could never quite come up with a satisfactory answer to their question about what it is that I really have been doing. A late-in-life PhD like mine inevitably becomes a family PhD. I thank my mother-in-law for being so understanding that despite all the extra load that this meant for her, she would still stand up for me whenever anyone, even my own relatives, showed the slightest sign of misreading my unavailability as a lack of caring.

It must require a strong sense of humour and loads of patience to deal with a spouse who is virtually absent most the time and drones on about “software bloat” during the rest. And so, finally, it is time to thank the person whose contribution to this thesis runs deeper than I could possibly express, my husband, Sourav, who shared it all.
Vita

Current (from 2007)  PhD Scholar (External Registration Program)
                   Indian Institute of Science
Current (from 1993)  Senior Technical Staff Member
                   IBM Systems and Technology Group, India
1989–1993           B.Tech (Hons), Electronics and Electrical Communication Engineering
                   Indian Institute of Technology, Kharagpur
Publications based on this Thesis


**Technical Reports:**

Abstract

Large flexible software systems tend to incur “bloat”, here defined as the runtime overhead induced by the accumulation of excess functionality and objects. Removing bloat is hard as these overheads are a side-effect of the same trends that have fuelled software growth. Even defining and measuring bloat is non-trivial, as software doesn’t come with built-in labels that indicate which portions of computation are necessary for a given application and which lead to bloat. Much progress has been made in novel analysis tools that aid (human experts in) the process of finding bloat by highlighting signs of excessive activity and data flow. However, there has been very little research focus on understanding the connection between sources of bloat and its system level implications.

In particular, excess resource usage due to bloat could be a significant source of power-performance inefficiencies, but the relation between bloat and energy efficient design remains unexplored. In order to systematically devise effective mechanisms for reaping power-performance benefits through bloat mitigation, we require a deeper insight into exactly when excess features can originate bloat, when the resource overheads of bloat are most pronounced and when bloat matters for power performance. This dissertation explores the problem of software bloat and its energy efficiency implications from multiple perspectives to develop a better understanding of these connections.

First, we establish the need for a whole systems perspective in assessing potential energy efficiency benefits of bloat reduction, based on a systematic empirical and analytical study that highlights a curious interplay between bloat, energy proportionality and system bottlenecks. Second, we present a novel static analysis algorithm to perform an automated code transformation for object reuse that mitigates bloat involving the generation of excess temporary objects.
within loops. Third, we introduce the idea of concern augmented program analysis (CAPA), to identify sources of bloat due to excess features; the technique uses externally supplied information about program concerns and their properties as an abstraction of underlying intent. Fourth, as an early diagnostic aid, we use a statistical topic model to automatically discover latent concerns from source code statements and then correlate these latent concerns with resource usage properties that vary at statement granularity. The statistical model has a built-in sensitivity to the context of individual statements so that it can discover even diffused concerns without any apriori concern information.

Together, our findings show that presence of excess features, in itself, may not lead to (runtime) bloat, unless these features have some structural interaction with essential features. Further, the overheads due to such structural interactions, in turn, may not cause substantial bloat in the resource consumption of a long running (server) application unless incurred repeatedly during program execution. Finally, even such bloated resource usage has a pronounced impact on power-performance only if it affects a system bottleneck or a hardware resource that has a high degree of energy proportionality and consumes a high fraction of power compared to the other system resources.

We conclude that energy wastage due to bloat need not be an inevitable consequence of over-provisioning flexibility. Instead, the extent to which excess features result in runtime bloat and poor power-performance is determined by certain characteristics of the program structure and of the underlying hardware system – these represent potential control points that could be exercised to develop principled design approaches for mitigating bloat without sacrificing flexibility or productivity.
# Contents

Acknowledgements iv  
Vita vii  
Publications based on this Thesis viii  
Abstract x  
Common Terms xxii  

1 Introduction 1  
1.1 Current State-of-the-Art and Open Challenges 3  
1.2 Our Focus and Approach 5  
1.3 Findings and Contributions 6  
1.3.1 Connection between resource bloat and power-performance 7  
1.3.2 Connection between recurring bloat and its automatic mitigation 8  
1.3.3 Connection between excess software concerns and execution bloat 9  
1.3.4 Systematically linking connections from origin to implications 10  
1.3.5 New research direction: Software models and hardware support to enable explicit management of propensity for bloat 11  
1.4 Organization of this dissertation 12  

2 The Problem of Software Bloat 14  
2.1 Different Notions of Software Bloat 15  
2.1.1 Principal aspects of bloat 17  
2.1.2 Subjective vs objective dimensions 19  
2.1.3 Definitions relevant for this thesis 19  
2.2 Causes and Consequences of Bloat 20  
2.3 Different Forms of Software Runtime Bloat 23  
2.4 Systemic Origin of Software Runtime Bloat 26  
2.5 Modeling and Measuring Bloat 28  
2.6 Mitigating and Avoiding Bloat 32  
2.6.1 Semi-automated approaches 32  
2.6.2 Automated code optimization 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.3</td>
<td>Can runtime bloat be avoided by construction?</td>
</tr>
<tr>
<td>2.7</td>
<td>Conclusions</td>
</tr>
<tr>
<td>3</td>
<td>Power-Performance Implications of Java Runtime Bloat</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Temporary (Java) object churn bloat and its effects</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Allocation wall effect</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Heap pressure effect</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Object construction computation overhead</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Influence of system configuration</td>
</tr>
<tr>
<td>3.3</td>
<td>Experimental study</td>
</tr>
<tr>
<td>3.3.1</td>
<td>SPECpower_ssj2008 and Metrics</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Varying extent of bloat</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Experimental setup</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Multi-platform experiments and results</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Experiments with other workloads</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Key observations</td>
</tr>
<tr>
<td>3.4</td>
<td>Quantifying power-efficiency impact of bloat: a</td>
</tr>
<tr>
<td>3.4.1</td>
<td>simple abstract model</td>
</tr>
<tr>
<td>3.4.2</td>
<td>System Bottlenecks and Bloat: A Curious Interaction</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Summary</td>
</tr>
<tr>
<td>3.5</td>
<td>Corroborating the model with experimental</td>
</tr>
<tr>
<td>3.5.1</td>
<td>observations</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Specific model predictions and corroborating</td>
</tr>
<tr>
<td>3.6</td>
<td>Related Work</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusions</td>
</tr>
<tr>
<td>4</td>
<td>Automated Object Reuse Transformation for</td>
</tr>
<tr>
<td>4.1</td>
<td>Mitigating Bloat</td>
</tr>
<tr>
<td>4.2</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.3</td>
<td>Preliminaries</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Finding potential sources of bloat</td>
</tr>
<tr>
<td>4.3.2</td>
<td>The Problem with Loop Carried Data Dependence</td>
</tr>
<tr>
<td>4.3.3</td>
<td>The Basic Algorithm</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Multiple Control Dependence Paths</td>
</tr>
<tr>
<td>4.4</td>
<td>Object reuse, recycle transformations</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Basic reuse-recycle algorithm</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Reusing collections</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Reusing strings</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Implementation Details</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Dynamic analysis guided filtering of candidate</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Discussion</td>
</tr>
<tr>
<td>4.5</td>
<td>Empirical evaluation</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Reuse site detection statistics (static analysis)</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Reuse site object churn statistics (dynamic</td>
</tr>
</tbody>
</table>

**CONTENTS**

- 2.6.3 Can runtime bloat be avoided by construction? .......................... 34
- 2.7 Conclusions .................................................................................. 35
- 3 Power-Performance Implications of Java Runtime Bloat ...................... 37
- 3.1 Introduction .................................................................................. 37
- 3.2 Temporary (Java) object churn bloat and its effects ...................... 40
  - 3.2.1 Allocation wall effect .......................................................... 41
  - 3.2.2 Heap pressure effect ............................................................. 41
  - 3.2.3 Object construction computation overhead .................................. 42
  - 3.2.4 Influence of system configuration .......................................... 42
- 3.3 Experimental study ........................................................................ 43
  - 3.3.1 SPECpower_ssj2008 and Metrics .............................................. 43
  - 3.3.2 Varying extent of bloat ........................................................... 44
  - 3.3.3 Experimental setup .................................................................. 46
  - 3.3.4 Multi-platform experiments and results ..................................... 47
  - 3.3.5 Experiments with other workloads ............................................ 53
  - 3.3.6 Key observations .................................................................... 54
- 3.4 Quantifying power-efficiency impact of bloat: a simple abstract model | 55
  - 3.4.1 The Model ................................................................................ 56
  - 3.4.2 System Bottlenecks and Bloat: A Curious Interaction ................. 59
  - 3.4.3 Summary .................................................................................. 61
- 3.5 Corroborating the model with experimental observations .................. 62
  - 3.5.1 Experimental study in perspective ........................................... 62
  - 3.5.2 Specific model predictions and corroborating experimental results | 63
- 3.6 Related Work ................................................................................. 65
- 3.7 Conclusions .................................................................................... 66
- 4 Automated Object Reuse Transformation for Mitigating Bloat ............. 68
  - 4.1 Introduction ................................................................................ 68
  - 4.2 Preliminaries .............................................................................. 72
  - 4.3 Finding potential sources of bloat .............................................. 75
    - 4.3.1 The Problem with Loop Carried Data Dependence .................... 75
    - 4.3.2 The Basic Algorithm ............................................................. 77
    - 4.3.3 Multiple Control Dependence Paths ..................................... 80
  - 4.4 Object reuse, recycle transformations ........................................ 84
    - 4.4.1 Basic reuse-recycle algorithm ............................................. 84
    - 4.4.2 Reusing collections ............................................................. 86
    - 4.4.3 Reusing strings .................................................................... 87
    - 4.4.4 Implementation Details ......................................................... 87
    - 4.4.5 Dynamic analysis guided filtering of candidate reuse sites ....... 89
    - 4.4.6 Discussion ............................................................................ 90
  - 4.5 Empirical evaluation .................................................................... 90
    - 4.5.1 Reuse site detection statistics (static analysis) ..................... 91
    - 4.5.2 Reuse site object churn statistics (dynamic analysis) ............. 93
4.5.3 Performance impact statistics ........................................ 94
4.6 Related work .................................................................. 96
4.7 Conclusion and future work .............................................. 98

5 CA PA: Concern Augmented Program Analysis for Bloat Detection 100
5.1 Introduction ................................................................... 100
  5.1.1 CA PA: concern augmented program analysis ..................... 103
  5.1.2 CA PA for Bloat Analysis .................................................. 104
  5.1.3 Contributions: ................................................................. 106
5.2 Execution bloat in the context of software concerns .................. 107
  5.2.1 When do excess features cause execution bloat? .................... 108
5.3 Preliminaries ................................................................... 111
  5.3.1 Working definitions ........................................................ 111
  5.3.2 Preliminary static analysis ................................................. 112
5.4 Microslicer .................................................................... 113
  5.4.1 Stepwise refinement microslicing ...................................... 113
  5.4.2 Micro-Slice Interaction Graph (MSIG) ................................. 116
5.5 Concern Augmentation ........................................................ 119
  5.5.1 CA PA overview ................................................................. 119
  5.5.2 Concern augmented static analysis to detect execution bloat..... 121
  5.5.3 Implementation ............................................................... 123
  5.5.4 Evaluating Concern Augmented Static Analysis for Bloat Detection 124
5.6 Probabilistic CA PA ............................................................ 128
  5.6.1 Experimental observations with probabilistic CA PA .............. 130
5.7 Related Work .................................................................. 133
  5.7.1 Java Bloat Analysis .......................................................... 133
  5.7.2 Concern Analysis ............................................................. 133
  5.7.3 Slicing ........................................................................ 134
  5.7.4 Topic Models ................................................................. 134
5.8 Conclusions ................................................................... 135

6 Modeling Statement Context to Surface Diffused Concerns Automatically for Probabilistic CA PA 136
6.1 Introduction ................................................................... 136
6.2 Problem Definition ............................................................ 140
  6.2.1 Concerns .................................................................... 140
  6.2.2 Problem statement ........................................................ 140
  6.2.3 Evaluation criteria ........................................................ 141
6.3 Background: Probabilistic Topic Modeling ............................... 142
  6.3.1 Latent Dirichlet Allocation (LDA) ...................................... 143
6.4 Assessing applicability of state-of-the-art topic models to the problem of statement level concern discovery .. 146
  6.4.1 FINDING 1: LDA cannot detect diffused concerns .............. 147
  6.4.2 FINDING 2: MG-LDA is ineffective for source code ............ 148
6.5 A statistical software concern model that is sensitive to the context of individual source code statements .............................................. 149
   6.5.1 Context Sensitive Concern Modeling (CSCM) ..................... 149
   6.5.2 Boosting capability to detect weak concerns ...................... 153
6.6 Attributing resource usage to latent concerns .......................... 154
6.7 Empirical Evaluation .............................................................. 157
   6.7.1 Key evaluation criteria ...................................................... 158
   6.7.2 Concern detection sensitivity ............................................. 158
   6.7.3 Concern coverage diversity .............................................. 160
   6.7.4 Programmer interpretability (human evaluation) study: ........ 162
   6.7.5 Summary ............................................................................ 170
6.8 Discussion ................................................................................. 171
6.9 Related Work ........................................................................... 172
6.10 Conclusions ............................................................................. 173

7 Putting It All Together - A Systems View 175
   7.1 Towards systematic strategies for bloat mitigation .................. 177
   7.2 Resource proportional features to reduce bloat propensity of software .................................................. 179
      7.2.1 Quantifying feature utilization and bloat propensity ........ 180
      7.2.2 Tradeoff between feature exploitation and provisioning overhead .................................................. 183
   7.3 Related Work .......................................................................... 185
      7.3.1 Lessons from architecture research ................................. 185
      7.3.2 Approximate computing .................................................. 187
   7.4 Conclusions ............................................................................. 188

8 New Research Direction: Exploring alternate software data models and hardware support to enable explicit management of propensity for bloat 189
   8.1 Radical data representation schemes for managing Java data structure bloat .................. 191
      8.1.1 Common patterns of Java memory data structure bloat ........ 192
      8.1.2 Alternate addressing schemes for reducing memory overheads .................................................. 193
      8.1.3 Content addressable memory data model .................................................. 196
      8.1.4 Applications of CAM based schemes for reducing run-time data bloat .................................. 198
      8.1.5 Analysis of space savings and overheads .................................................. 202
      8.1.6 Caveat: Power-performance tradeoffs with hardware TCAMs .............................................. 206
   8.2 Hardware support for resource proportional primitives by virtualizing bloat ? 206
      8.2.1 Established and emerging applications of TCAMs .............. 208
      8.2.2 Virtually cool TCAM - a possible way to minimize energy wastage due to unused associations ? .................................................. 209
      8.2.3 Content addressable VMM (CAVMM) ..................................... 210
      8.2.4 Content locality classification ............................................ 214
      8.2.5 Impact on resource proportionality ...................................... 215
      8.2.6 Implementation challenges ............................................... 216
      8.2.7 Conclusions ........................................................................ 217
9 Conclusions 219

Bibliography 224
List of Tables

2.1 State-of-the-art techniques for measuring different indicators of bloat categorized according to the underlying heuristics and extent of reliance on intrusive profiling .................................................. 31

3.1 Effects of temporary objects bloat observable by comparing specific code variants under specific configuration constraints .......................................................... 46

3.2 Systems Configurations: Processor Config is Sockets/ Cores per Socket/ Threads per Core. Cache size is for the last-level per socket. ................................. 46

3.3 Effect of bloat reduction in different scenarios when bloat affects a single resource $R_k$ .................................................. 59

3.4 Trends from cross-platform experiments .................................................. 63

4.1 Benchmarks analysed and freed memory for each benchmark .................. 91

4.2 Reuse site detection statistics .................................................. 92

4.3 Reuse site object churn statistics .................................................. 93

4.4 Performance impact statistics: The percentages are baselined against the corresponding results for the original benchmark without object reuse conversion; for example 100% reduction in temporary objects generated would mean that all object allocations were eliminated, 100% improvement in throughput would mean that throughput doubled. .................................................. 95

5.1 Bloat detection heuristics in decreasing likelihood of potential bloat contribution (computed only for MSIG nodes that are inside methods which can be executed by the client program even when no optional feature is used explicitly). .................................................. 123

5.2 Experimental Results: Concern Augmented Static Analysis for Bloat Detection 126

5.3 A few lines taken from the file TestMultiFieldQueryParser.java in Apache lucene, colored based on probabilistic concern partitioning .................. 131

5.4 Probabilistic CAPA for object churn .................................................. 132

6.1 Choice of $F, \zeta$ and its implications. If $F$ is 1, contextual concerns are selected, else modular concerns are selected. .................................................. 152
6.2 Byte-code execution summaries computed for sample Apache lucene concerns found by C SCM. Results are shown for two benchmarks DaCapo lusearch and DaCapo luindex. It reports cumulative bytecode execution cost attributed to a concern as a percentage of total bytecodes executed by the program. Shaded rows are examples of *diffused concerns* and they are undetected by LDA-CS.  
6.3 Four Java applications of various scales and domains used in our experiments.  
6.4 Example statements corresponding to a foreign concern “graphics” injected into BerkeleyDB (top). The concern is detected by C SCM, but LDA fails to detect it due to its weak presence (bottom).  
6.5 Context sensitivity experiment results: Pruned BerkeleyDB test concern topics found (5 most likely words of relevant topics are listed).  
6.6 Examples of BerkeleyDB concerns found by C SCM with 10 most likely words  
6.7 Interpretability of statement level concern assignments (most relevant topic).  
6.8 Concern topic interpretability results: ability to detect intrusion words or assign consistent labels to concern topic words or its relevant statements.  
6.9 Evaluation summary.  
6.10 Lines from SPECjbb2005/spec/jbb/Company.java, colored based on concerns  
7.1 Different levels of intervention to contain energy wastage due to bloat.  
8.1 Computing space savings vs overheads of CAM based representation schemes.  \( sz() \) is a function that computes the size of the portion of the CAM word that represents the fields specified as parameters.  \( entries \) is the number of data element entries, \( pieces \) is the number of pieces in a discontiguous object, \( ptrs \) refers to pointers, \( hdrs \) refers to object headers, \( typeptr \) refers to the type information part of an object header.  
8.2 Example: CAM based representation of hash set with 3 short strings.  
8.3 Example: Space overheads vs savings of using a CAM based representation of a TreeMap (from \textit{double} to \textit{double}) with 1 million entries.  
9.1 Research questions addressed by different chapters and techniques contributed.
List of Figures

1.1 Tradeoff between software engineering considerations of solution flexibility, runtime efficiency and development productivity .................................. 3
1.2 A Systems Perspective of Software Runtime Bloat - The Price of Flexibility ....... 11

2.1 Different aspects of bloat ................................................................. 18
2.2 Relationship between different types of bloat and their runtime manifestations ... 25
2.3 Accumulation of excess functionality across layers of redeployable components at many levels of granularity. The circles represent features implemented by a component; intersecting circles signify feature interactions. The arrows connect features in a component with features it uses from other components. Blue circles represent features that are needed by a particular application MyApp. Yellow circles represent features that are currently in excess for MyApp but exist to support other components or future needs. Not all excess features induce runtime bloat, however. ............ 27
2.4 Excessive transformations in framework based software. A sequence of expensive transformations with nested transformations [89] ........................................ 29
2.5 Data structure bloat: High percentage of representation overhead .................. 30

3.1 Effects of Java temporary object creation bloat ...................................... 41
3.2 Cross-platform comparison: All results are normalized with respect to the corresponding AllocOrig measurement ................................................. 47
3.3 Memory Statistics for SPECpower_ssj2008 on Power 750 in SMT4 mode (normalized wrt AllocOrig) .......................................................... 49
3.4 Heap size reduction impact with decrease/increase in bloat: All results are normalized with respect to the corresponding AllocOrig measurement at full heap size .... 51
3.5 Cache capacity effect on improvement from bloat reduction (on Power750): SMT2 has most cache resource/thread, SMT4 has half SMT2’s capacity/thread, SMT4-halfcache has half SMT4’s capacity/thread. For energy-efficiency and peak performance the numbers show the improvement (increase) for AllocLess relative to AllocOrig. For equi-performance power and peak power the quantities shown represent the saving (decrease) for AllocLess relative to AllocOrig. For all cases, a positive number signifies an improvement for that quantity with bloat reduction. .......................... 52
3.6 **Hardware energy proportionality differences:** (a) The Power 750 system employing new DVFS algorithms shows an *approximately cubic* relationship for CPU power; *memory power increases linearly but as a much smaller fraction of system power* on the test configuration (which uses adequate memory for SPECpower_ssj2008, but far below the system’s memory capacity of 512GB). (b) Without DVFS the relationship for the Power 750 system is *almost linear*, similar for HS21. (c) The x3650 M2 system has a *near quadratic* characteristic for the full system power (we do not have CPU and memory power measurements in this case).

4.1 Sample code. ................................................................. 70
4.2 Escape Analysis. ............................................................. 74
4.3 Loop header sharing for multiple control dependence paths .............. 83
4.4 Code transformation example ............................................. 88

5.1 Introducing concern-awareness in program analysis with CAPA ............ 104
5.2 Big endian to little endian transformation .................................. 107
5.3 Example: Logged Restorable Buffer from [72, 64] ................................ 109
5.4 Microslice Interaction Graph for Fig 5.3 ...................................... 117
5.5 Concern Augmented Program Analysis ....................................... 120
5.6 Concern Augmented Microslice Interaction Graph corresponding to Fig 5.4. The shaded nodes represent microslices in methods belonging to optional features (LOG: blue and RESTORE: gray). The arrows represent potential USE interactions (where the destination node USEs the source node’s feature). .......................... 121

6.1 Graphical model representation of LDA. Each node is a random variable. Edges represent conditional dependence, where incoming arrows at a node depict the variables it depends on. \( w \) is the only observed variable here (shaded node) representing words in a document. The hidden nodes (latent variables) are unshaded. This includes the document wise topic proportions \( \theta \) (and its Dirichlet prior parameter \( \alpha \)), the topic assignments \( z \) and the topics \( \beta \), where topics are distributions over words in the vocabulary. The rectangles are “plates”, a concise notation for replication – the outer and inner plates marked \( M \) and \( N \) imply iteration over \( M \) documents and then over \( N \) words in each document. ................................................................. 145
6.2 Example of 3 contexts, each as a set of 3 consecutive statements in a sliding window mechanism. Statement 8 or statement 9 alone may not be clearly identified as a *copy* or *buffering* concern, but along with statement 7 it becomes more apparent. .......... 149
6.3 Graphical model representation of CSCM. The shaded circles represent inputs and outputs of the model. The rectangles marked with M, C, S, N imply iteration over M modules, C contexts, S statements and N words. A directed arrow means that the “from” node influences the “to” node. \( w \) is the only observed variable here representing words in software code, while \( z \) represents the concerns. The suffixes “m” and “c” in the parameters distinguish modules and contexts respectively. ................................. 151
6.4 Generative process of CSCM model for each module \( f \) in a software application. .... 152
6.5 Jenson Shannon Divergence among concerns detected by CSCM and LDA-CS. .... 160
6.6 Out of 23 known concerns in BerkeleyDB, CSCM detects 19, and LDA-CS detects 17. All concerns detected by LDA-CS are detected by CSCM.  

6.7 Notched boxplots of the interpretability of statement level concern (topic) assignments (left) and concern (topic) interpretability (right) for CSCM and LDA-CS. These plots illustrate the variation in interpretability across topics, complementing the results in Tables 6.7 and 6.8 which showed the averages aggregated over all responses. The interpretability value (Y-axis) computed for each topic represents the fraction of subjects whose responses indicate that they perceive that topic to be interpretable.  

7.1 High level cause effect flow for guiding optimization  

7.2 Different types of resource proportionality characteristics. The curves plot the normalized actual resource usage of a fully generalized implementation as the feature utilization $\mu$ increases. The resource usage (Y-axis) is normalized wrt to the resource usage when all the features are utilized.  

8.1 Content addressable VMM example  

8.2 Sample content addressable space
Common Terms

( Runtime) Bloat  Runtime overhead induced by the presence of excess functionality and objects, resulting from optional concerns and poor implementation choices in framework based software, typically to support features that are in excess for a given deployment. Bloat can manifest as runtime resource consumption disproportionate to actual function delivered. Note that except when qualified otherwise by the context, we use the term bloat to specifically mean software runtime bloat.

Concern  Any consideration that can impact the implementation of a software program. According to [106] a concern could represent “anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, design idioms, and implementation mechanisms. We will be using the terms concern and feature interchangeably in this dissertation.
**Concern intent and extent**  A concern can be characterized in terms of its intent and extent [75]. A concern’s *intent* is defined as the conceptual objective of the concern and properties associated with that objective. A concern’s *extent* is the concrete representation of concern in source code, i.e. the source code modules and statements where the concern is implemented.

**Energy proportionality**  A measure of the extent to which the power consumed by a hardware resource changes in proportion to its actual utilization.
Chapter 1

Introduction

“Software girth has surpassed its functionality largely because hardware advances make this possible.

About 25 years ago, an interactive text editor could be designed with as little as 8000 bytes of storage (Modern editors request 100 times that much!). An operating system had to manage with 8000 bytes and a compiler had to fit into 32 Kbytes, whereas their modern descendants require megabytes. Were it not for a thousand times faster hardware, modern software would be utterly unusable.”

– Niklaus Wirth, A Plea for Lean Software, 1995 [133]

“From 1963 to 1966, I worked for the Pennsylvania State Drivers License Division, programming an IBM 1401 computer. The first widely used transistor-based computer, our 1401 handled all six million drivers in the state.

The machine had 12K of memory. There was no graphical interface, because there was no screen. We debugged our programs with memory dump printouts, and we didn’t even have an operating system. What for? We just wrote our own input/output routines.

But, we processed an entire state!

Know anyone these days processing a state on his or her desktop computer with 300,000 times as much memory as we had in 1963? Software bloat. Better believe it. Happy computing!”

– Alan Freedman

The emergence of powerful frameworks for redeployable software has transformed the pace of large scale software development. Framework based software systems are provisioned with a high degree of built-in excess flexibility to support standardization and cope with evolving requirements. These are desirable trends, considering the widespread adoption of such software and the impact it increasingly has on our lives. However, an unintended side-effect of
Chapter 1. Introduction

These trends is that software systems tend to accumulate “runtime bloat” - the resource overhead of excess function and objects. According to Mitchell, Schonberg and Sevitsky [86], bloat can manifest as a pervasive pattern of excessive memory, processing and IO overheads in framework based applications – these overheads can cause applications to “miss their scalability goals by an order of magnitude” – resulting in the deployment of additional hardware and complex system infrastructure to meet service level objectives.

Meanwhile, energy consumption of computing servers is a serious concern in sustaining the kind of technological advancements required to meet exploding information processing demands. As a result there is a renewed focus on ensuring more efficient utilization of hardware resources by software on these systems. Till date, advances in power-efficient software for server systems have included approaches such as (i) the incorporation of power-awareness in workload scheduling and resource management [111], (ii) restructuring distributed computation to avoid energy wastage in under-utilized components [69], (iii) the timely release of resources to avoid interference with hardware and system power management [111], (iv) the exploitation of power-efficient hardware\(^1\) by software applications [52, 68], (v) techniques for energy-efficient query-optimization [143] and (vi) approximate computing based power optimization [115, 10].

The presence of software runtime bloat indicates that there may be opportunities for significant improvement in energy efficiency along another dimension – if we could address inefficiencies caused by bloat, it would reduce the base resource consumption imposed by the software stack. *Does this represent an opportunity to develop fresh approaches for obtaining further energy savings in server systems?* Is it possible to achieve these improvements without sacrificing the very capabilities that have fueled the advancement of software and its deployment at scales which were unanticipated just two decades ago?
1.1 Current State-of-the-Art and Open Challenges

Concern about software code and data bloat is not a new issue - yet for over a decade and a half since the publication of Niklaus Wirth’s article “A Plea for Lean Software” [133], the incidence of bloat has mostly continued unabated. It has even accelerated as a consequence of prevalent software and hardware trends. The deep systemic cause is not one of “bad programmers”, but practical cost-benefit tradeoffs that have emerged in reaction to the inherent difficulty in meeting the pressures of simultaneously addressing the need for flexibility, productivity and efficiency\(^2\) of software solutions. Any two of these considerations can typically be achieved at the price of the compromising on the third consideration (Figure 1.1). In the case of framework based software the tradeoffs have leaned in favour of flexibility and productivity at the cost of

---

\(^1\)e.g. DVFS, solid state storage, heterogeneous cores

\(^2\)although there are other important non-functional considerations such as security, standardization, RAS (reliability, availability and serviceability), for simplifying the discussion we include those considerations either under the category of flexibility or productivity, as appropriate
efficiency [86].

For example, one explanation for why the problem of software bloat has received very little sustained attention until now is that there have been dramatic improvements in hardware performance over successive CMOS technology generations. Such improvements would tend to obviate the significance of inefficiencies caused by bloat in comparison with the huge gains in software flexibility and development productivity. Efficiency in software is now regaining its importance with shifting technology trends such as the gradual reduction in energy efficiency improvements over hardware generations and the rising operational costs of maintaining overly complex and bulky software stacks. At the same time, both flexibility and productivity continue to be important software engineering considerations. Hence, traditional solutions to bloat that rely on detailed attention to programming, finely crafted abstractions, tightly coupled hardware software development or carefully customized feature configuration are unsuitable for the scale and pace of software development required in large flexible IT solutions.

There are significant challenges in developing automatic approaches to measure and mitigate software bloat. Distinguishing excessive resource utilization due to bloat from resource utilization due to essential function is non-trivial, as it typically requires a deep knowledge of intended semantics of complex software stacks. As a result it is difficult to even arrive at a precise definition of bloat, let alone perform a systematic optimization of software to address the problem or conduct early cost-benefit assessments to determine where opportunities exist.

Good progress has been made in developing analysis tools that aid (human experts in) the process of finding bloat by highlighting signs of excess activity and data flow accordingly to different notions of bloat [87, 41, 139, 138, 140, 141]. Further, several anecdotes of bloat have been documented by previous studies which provide insight into the nature of certain types of bloat [89, 86, 36]. Yet, till date, we only have a limited understanding of the circumstances under which bloat arises and the extent to which it affects runtime resource usage and system power-performance. In the absence of such understanding, it is not clear how to systematically devise effective mechanisms for reaping power-performance benefits through bloat mitigation.

For example, in order to establish the relationship between bloat and energy efficient design, there is a need to quantitatively relate the overall power-performance impact of bloat in a
system to specific sources of bloat and to reason about energy savings expected from alternative strategies for bloat reduction. However, this is a very broad and challenging problem that raises several interesting questions:

• When do excess features lead to runtime bloat?
• When is the resource overhead due to bloat most pronounced?
• How much does bloat matter for power-performance?
• How can the extent of bloat attributed to a given source be determined?
• What information is necessary to automatically estimate the extent of bloat?
• What can be done to mitigate bloat once we have identified it?
• What information is necessary to automatically de-bloat software?
• How do we assess the propensity for bloat of a given software implementation?
• Can systems be redesigned to enable software to avoid propensity for bloat without losing flexibility or productivity?

1.2 Our Focus and Approach

Our research aims to develop a better understanding of these questions by investigating the problem of bloat and its power-performance implications at multiple levels. We explore novel approaches for reasoning about the systems level impact of software run-time bloat, its origin and mitigation. In particular, we adopt a variety of research techniques to investigate:

1. The system level impact of bloat in terms of the connection between runtime resource bloat and power-performance. We use:

• Controlled experiments involving real system power measurements on multiple hardware platforms (detailed in Chapter 3).
• Analytical modeling based on fundamental operational laws for abstract “what-if” reasoning (developed in Chapter 3).

2. **The automatic mitigation of bloat** in terms of the connection between repeatedly incurred overheads and resource savings achievable by amortizing these overheads. We apply:

   • Static program analysis and automated (source) code transformation (developed and evaluated in Chapter 4).
   
   • New OS and architectural mechanisms that exploit an alternate memory addressing scheme (early ideas discussed in Chapter 8)

3. **The origin of bloat** in terms of the connection between excess software concerns (or features) and sources of runtime bloat. We employ:

   • New techniques for enriching program analysis with externally supplied concern information (developed and verified in Chapter 5)
   
   • Statistical topic modeling of source code for automated summarization of bytecode and object churn profiles in terms of latent program concerns (developed and evaluated in Chapter 6)

### 1.3 Findings and Contributions

Our main insight is that the extent to which excess flexibility and features result in runtime bloat and poor power-performance is determined by certain characteristics of the program structure and of the underlying hardware system – these characteristics represent potential control points that could be exercised to develop systematic approaches for mitigating energy wastage due to bloat without sacrificing flexibility.
1.3.1 Connection between resource bloat and power-performance

The first key finding of this thesis is that a software-only view is inadequate when assessing the energy efficiency effects of bloat. The degree to which bloat impacts power-performance is not always obvious and can vary widely with hardware and software configuration. It could even depend on the hardware characteristics of non-bloated resources. This is because the presence of bloat shifts the relative utilization of different hardware resources in a system, some of which may have a non-linear power-vs-load variation.

Interplay of software bloat, hardware energy proportionality and system bottlenecks

We support this finding through two contributions, one empirical and the other analytical (both described in Chapter 3 [20, 22]).

1. We conduct the first systematic empirical study of the joint power-performance implications of software bloat, using SPECpower ssj2008, across a range of hardware and software configurations on modern server systems covering different processors/system designs, levels of multithreading, cache sizes, power management and heap sizes. The study employs controlled experiments that vary the degree of bloat and system characteristics to investigate different manifestations of the effects of one prevalent type of Java runtime bloat, excess temporary objects. Using multiple metrics for evaluating the impact of bloat – peak power, equi-performance power, peak performance, and the resulting energy efficiency measures – we systematically analyze and expose the effect of energy proportionality and bottleneck resources on the impact of bloat reduction.

2. We then develop an abstract analytical model that establishes the relation between resource pressure caused by bloat in general and its effect on energy efficiency. Our model incorporates the relative energy proportionality of all hardware resources as well as how close these resources are to a system bottleneck, as two important determinants of the extent to which bloat impacts power-performance. Application of the model to different scenarios allows a generalized “what-if” exploration of the power-performance implications of bloat. It also helps place our experimental study in perspective, with the results
confirming the presence of the effects predicted by the model in real systems.

We conclude that a whole systems perspective is necessary to properly evaluate the power-performance benefits of bloat reduction solutions.

1.3.2 Connection between recurring bloat and its automatic mitigation

The second key finding of this thesis is that an effective way to mitigate the runtime resource pressure due to bloat is to amortize overheads that are incurred repeatedly during program execution, such as the generation of excess temporary objects within loops. This method can be particularly useful given the intrinsic difficulty of first distinguishing excess flexibility from essential function in existing software and then eliminating it without losing the benefits afforded by such flexibility.

Automated object reuse transformation for de-bloating software In Chapter 4 [19], we make the following contributions in this direction:

1. We present a novel algorithm that can detect objects created within a loop and determine whether an object created within a loop can be reused at the end of each iteration. In the case of nested loops, the algorithm reports the innermost enclosing loop in which the object can be reused.

2. Next we develop a solution that can automatically transform the source code to reuse temporary container and string objects such as to mitigate the effects of bloat. Empirical evaluation indicates that our solution can reduce upto 40% of temporary object allocations in large programs, resulting in a performance improvement that can be as high as a 20% reduction in the run time, specifically when a program has a high churn rate or when the program is memory intensive and needs to run the garbage collector (GC) often.
1.3.3 Connection between excess software concerns and execution bloat

The third key finding of this thesis is that software bloat in terms of overprovisioned or excess features directly incurs a runtime execution overhead, i.e. “execution bloat”, only when statements that implement optional software concerns are structurally intertwined with code that implements essential concerns. These statements represent unnecessary instructions that can get executed or extra fields of data that may be maintained even when the optional concern is not in use. However, distinguishing different software concerns or features and assessing which ones are essential requires a higher level insight into programmer intent than that available solely using static and dynamic program analysis information. We observe that advances in concern analysis could provide a way to address this challenge: concern or feature assignments represent functional intent at a level of abstraction required for automating execution bloat analysis. We make the following contributions (in Chapters 5 and 6) to explore this possibility:

Making program analysis concern-aware to enable execution bloat estimation  We introduce CAPA (Concern Augmented Program Analysis), a novel approach (described in Chapter 5) that complements traditional static or dynamic program analysis with an abstraction of underlying intent in terms of software concerns, using the notion of “concern partitioning”. We show how CAPA can be applied for reasoning about software properties such as execution bloat which are hard to determine automatically without higher level insight about functional intent.

We develop a concern augmented static analysis for bloat detection that is effective in locating potential bloat contributing statements when information about methods corresponding to optional concerns are available apriori. The approach relies on automatic step-wise refinement microslicing, a novel underlying static analysis technique. The results of microslicing are combined with the supplied concern information to pinpoint statements involved in structural interactions between essential and optional features. The technique is able to detect execution bloat in six programs with well-known optional concerns.
Probabilistic concern augmented dynamic analysis which leverages statement context to surface latent concerns automatically For early diagnostic purposes in situations where concern information is not available apriori, we use a novel approach based on statistical topic modeling of source code (described in Chapter 6 [17]) to automatically discover latent concerns with a disproportionately high contribution to resource usage (or bloat).

We employ a context sensitive concern model (called CSCM) – the model assigns a mixture of concern topics to each statement via a probabilistic inference procedure that is sensitive to the surrounding statements (local context) in which the statement occurs – this enables it to detect even diffused concerns which are not prominent in any module. The statement-wise probability distribution of concern topics inferred by the model is leveraged by a probabilistic form of CAPA to correlate these latent concerns with runtime resource usage properties. As examples, we illustrate a probabilistic concern augmented dynamic object churn analysis (in Chapter 5) and an automated summarization of bytecode profiles in terms of latent concerns (in Chapter 6). We perform a detailed evaluation of the underlying model and highlight how the demanding requirements of statement level concern discovery presents practical challenges in the application of such statistical models for concern augmented analysis.

1.3.4 Systematically linking connections from origin to implications

Putting our three key findings together (Fig 1.2) shows that presence of excess features, in itself, may not lead to runtime bloat, unless these features have some structural interaction with essential features. Further, the overheads due to such structural interactions, in turn, may not cause substantial runtime resource bloat in a long running (server) application unless they are incurred repeatedly during program execution. Finally, even such resource bloat has a pronounced impact on power-performance only if it affects a system bottleneck or a hardware resource that has a high relative energy proportionality and consumes a high fraction of system power compared to the other system resources. In Chapter 7, we integrate these insights into a high-level cause-effect flow diagram as a foundation to enable the development of systematic strategies for guiding power-performance optimization through bloat mitigation.
Figure 1.2: A Systems Perspective of Software Runtime Bloat - The Price of Flexibility

**Resource proportionality of software features** This sequence of findings also suggests a possible approach for engineering “resource proportional features”, a term we introduce in Chapter 7 to describe software features that incur minimum bloat by consuming computing resources (and energy) in proportion to functionality and flexibility that is actually exploited in a given deployment scenario (even when the entire range of possibilities supported is much larger). As a contribution of this thesis, we outline a simplified framework for quantifying this notion of resource proportionality, a measure of the extent to which the features of a component are resource proportional.

1.3.5 **New research direction: Software models and hardware support to enable explicit management of propensity for bloat**

Designing software that avoids bloat by construction is non-trivial and requires significant effort. An interesting alternative is to design software in a way that makes it easier to detect and
mitigate bloat. For example, if the overheads of excess flexibility could be made explicit, opportunities might exist for novel system level support mechanisms (such as runtime, operating system and architecture enhancements) to optimize resource proportionality. We outline a few radical ideas in this direction in Chapter 8 [18] but leave the investigation of this possibility as an open problem for future research.

1.4 Organization of this dissertation

The rest of this dissertation is organized as follows.

In Chapter 2, we present an overview of the problem of software bloat based on a review of relevant literature and expert inputs. The expert inputs were obtained during the course of an IBM Academy of Technology study on the causes and consequences of software bloat (a study led in 2009 jointly by the author with Bob Blainey).

Chapter 3 is a detailed study of the effects of software runtime bloat on power-performance under different conditions of hardware energy proportionality and resource bottlenecks. We establish the need for a whole system analysis perspective to consider the overall effects on all system resources that are relevant for power-performance and not just those which are directly impacted by bloat. We observe that of all the statements that implement an excess concern, only a few may actually be responsible for execution bloat – those which have structural

3 to consider the overall effects on all system resources that are relevant for power-performance and not just those which are directly impacted by bloat.

4 a program characteristic that can potentially be controlled to create an optimization opportunity.
interactions with exploited concerns. By detecting such statements, our analysis exposes a new control point for tackling bloat. We also propose a probabilistic version of CAPA that can exploit statistical concern distribution information (such as that obtained via unsupervised machine learning) – the analysis is useful for early diagnostic purposes, e.g. for estimating concerns that are a major source of temporary object churn.

In Chapter 6, we employ a statistical topic model called Context Sensitive Concern Model (CSCM) to automatically discover latent concerns and their distribution from source code statements in situations where concern information is not available apriori. This model is an example of an unsupervised machine learning technique that enables probabilistic CAPA to correlate latent concerns with program properties that vary at statement granularity. CSCM leverages the context of individual statements to overcome a limitation of statistical topic models that have previously been used for concern discovery, the inability to detect diffused concerns, statement level concerns that are not prominent in any source module.

Chapter 7 links the different connections from the origin of bloat to its implications explored separately in previous chapters. We introduce the notion of resource proportionality of software features to enable a principled design approach in minimizing propensity for bloat – such an approach could exercise multiple control points from a systems perspective for tackling bloat and its power-performance effects.

In Chapter 8, we initiate a new research direction: an exploration of alternative software models and hardware support that could enable software to be designed in a way that makes it easier to detect and mitigate bloat. We illustrate an example of a ternary content addressable memory based software data model where the energy consumption overheads of unused flexibility can be automatically characterized in terms of rarely used associations, enabling system level control points to be devised for mitigating the overheads.

Finally, Chapter 9 summarizes our conclusions.
Chapter 2

The Problem of Software Bloat

“Anything that is produced by evolution is bound to be a bit of a mess”

– Sydney Brenner, Nobel laureate

“Software is getting slower more rapidly than hardware becomes faster”

– Wirth’s Law

(Wirth attributes this quote to Reiser)

We review the problem of software bloat based on a study of relevant literature and expert inputs, with an emphasis on runtime bloat in framework based applications.

Over the past two decades, software design paradigms have evolved to prioritize programmer productivity over runtime efficiency. In contrast to programs tuned for a specific use, large software systems are standardized around deeply layered frameworks that facilitate rapid development. Each (component) layer is designed to ensure composability of its functions to support a high degree of flexibility for reuse and interoperability. In a typical execution scenario, the system uses only a small subset of the functionality but still pays the overhead for supporting the full functionality. With more layers, the number of potential function combinations grows exponentially, compounding the hidden burden of largely unused combinations.

1Some portions of this chapter have appeared in the following article:
Chapter 2. The Problem of Software Bloat

As a result, although functionally richer and more flexible, newer software packages often incur a larger resource overhead in typical execution scenarios than their older editions [133]. Researchers have noted many forms of such runtime bloat—runtime resource consumption disproportionate to the actual function being delivered—including the execution of excess function calls, the generation of excess objects, and the creation of excessively large data structures. For example, Mitchell, Schonberg, and Sevitsky [86] mention cases involving (1) the creation of hundreds of thousands of method calls and objects to service a single web request that retrieves and formats a few database records and (2) the consumption of a gigabyte of memory per hundred users in an application that needs to scale to millions of users. They document 15 anecdotes drawn from their experience in analyzing real world applications to illustrate how current software engineering trends can lead to bloat. Our own experience and case studies support similar findings: such as a document-exchange gateway that creates six copies or transformations for each input document processed, a telecom application intended to support high transaction rates that generates over a megabyte of temporary objects per transaction and an object cache with 90% data structure overhead when used for storing small objects.

In the past, the benefits of flexibility have outweighed the perceptible cost of overheads incurred, given the pace of improvements in hardware performance enabled through CMOS technology. However, shifting hardware technology trends and operational challenges motivate a need for paying greater attention to these inefficiencies, as the overheads tend to negate gains from increases in computing hardware capacity and can result in higher power consumption and wasted energy.

2.1 Different Notions of Software Bloat

The concept of software bloat, although obvious in some respects, is elusive to formal definition. Part of the reason is that both in the popular press and in published research, the term “software bloat” is commonly used in a very broad sense to refer to a number of related issues.
For example, Samaai and Barnes have described software bloat as “a vague term encapsulating interface bloat, code bloat, feature bloat and size bloat”[107], while Xu, in his doctoral dissertation [137] uses the term bloat “to refer to the general phenomenon of using excessive work and memory to achieve simple tasks”. Furthermore, even for a specific issue such as runtime resource bloat (relevant for this thesis), it is difficult to arrive at a precise definition, because “bloat is relative” and identified “in comparison with a more optimal software implementation” for the same purpose, if one exists [139]. Below we list some of the definitions of software bloat that we have come across:

**Definitions in terms of trends in software growth**

1. “a process whereby successive versions of a computer program include an increasing proportion of unnecessary features that are not used by end users, or generally use more system resources than necessary, while offering little or no benefit to its users” - Wikipedia [131]

2. “the increased complexity of software. Modern operating systems and applications are huge in size and complexity compared to software in personal computers in the late 1970s and early 1980s. They are absolutely gigantic next to software of the 1950s and 1960s” - Alan Freedman, PCMAG encyclopedia

3. “the unnecessary growth of code. Most software based products contain an order of magnitude more software than is required.” - Gerrit Muller, 2003 [90].

4. “consuming ever greater amounts of memory and numbers of processor cycles with successive software version releases” - Nichols and Twidale, 2003 [94].

**Definitions in terms of out-of-proportion costs vs benefits**

1. “arises when the user expects the task to be simple and the device for doing the task is unnecessarily complicated or the amount of effort needed to learn a feature isn’t commensurate with its utility” - Kaufman and Weed, 1998 [61].
Chapter 2. The Problem of Software Bloat

2. “the result of adding new features to a program or system to the point where the benefit of the new features is outweighed by the impact on the technical resources (e.g., RAM, disk space or performance) and complexity of use” - Online Computing Dictionary.

3. “when execution time and memory consumption is high compared to what the program actually accomplishes” - Science of Runtime Bloat project (Mitchell et al.).

4. “a large and pervasive infrastructure tax” that results in “expending prodigious effort to accomplish relatively simple bits of functionality” - Mitchell, Sevitsky and Schonberg, IEEE Software 2010 [86].

Definitions in terms of excess functionality or overheads

1. “any unnecessary complexity or wasteful consumption of resources by software” - IBM Academy study, 2009.

2. “massive amounts of software distributed over thousands of nodes where much of the functionality is never used” - IBM Academy study external respondent.

3. “operations that while not strictly necessary for forward progress are executed nevertheless” - Xu et al., PLDI 2009 [138].

4. “a general situation where redundancy exists towards finishing a task which could have been achieved more efficiently” - Xu et al., FoSER 2010 [139].

2.1.1 Principal aspects of bloat

At a high level, we can categorize the principal aspects of bloat identified across these definitions in terms of the nature of perceived effects as well as the nature of perceived sources (Fig 2.1).

- (a) Perceived Effects:

1. Excessive complexity (and its impact on human performance)
2. Resource inefficiency (and its impact on system performance)

- (b) **Perceived Sources:**

1. Overprovisioning (excess or largely unnecessary) features
2. Poor implementation choices (sub-optimal coding)

![Diagram showing different aspects of bloat](image)

Figure 2.1: Different aspects of bloat

These aspects are not entirely independent, however. It is this characteristic of bloat which makes it difficult to arrive at self-contained definitions that clearly delineate a particular source-effect combination from the others. For example, resource inefficiency leads to higher demand for hardware capacity scaling which can translate into more complex configurations, installation and administration overheads. Likewise the more complex and obscure the software becomes, the harder it is to write efficient code and optimize resource usage. Poor implementation choices made when designing without context is a source of over-general framework components with excess features. On the other hand, the presence of unnecessary features
and abstractions can obscure the overheads of implementation decisions and thus cause poor choices to be made.

Note that in our use of the term “poor implementation choices”, we distinguish implementation choices that lead to runtime bloat\(^2\) from choices related to inefficient algorithms or traditional performance issues such as contention, scheduling and concurrency, which also impact the effective utilization of resources by software.

### 2.1.2 Subjective vs objective dimensions

A few studies [107, 80] have explored how end users perceive complexity due to feature bloat in software, particularly in the context of desktop applications and user interface design. McGrenere and Moore distinguish objective and subjective dimensions of bloat [79, 80, 81]. They observe that for an individual user, some features are desired by the user whether or not they are actually used; hence even if these features are not used they are not perceived as bloat. Objective bloat refers to the overhead of features that are not used or desired by majority of users. Subjective bloat refers to the overhead due to the particular subset of features that are not used and not desired by an individual user. This set varies from user to user, which explains why it is hard to avoid feature bloat.

In the case of server applications, operational complexity and resource inefficiency are more relevant issues than user interface complexity. Further, the subjective dimension is specific to a deployment context or client instead of an individual end-user.

### 2.1.3 Definitions relevant for this thesis

Our research is focused on the perceived effects on runtime resource inefficiency rather than the perceived complexity effects of software bloat. We use the following different variations of the definition of runtime bloat to qualify the nature of perceived sources (from 2.1.1(b)) that are relevant for a given analysis:

---

\(^2\)such as a combination of disparate local decisions and component reuse choices that can result in global inefficiencies, e.g. by inducing excessive data copies
• **Definition I** *(emphasizes both perceived sources)*: Bloat is the runtime resource overhead induced by the presence of excess functionality and objects.

• **Definition II** *(emphasizes poor implementation choices as the perceived source)*: Bloat is runtime resource consumption disproportionate to actual function delivered.

• **Definition III** *(emphasizes overprovisioning features as the perceived source)*: Bloat is the runtime execution overhead induced by the presence of excess features or program concerns.

### 2.2 Causes and Consequences of Bloat

In order to gain an overall perspective about the problem of bloat, its implications for large IT solutions and high level root causes, we co-led an IBM Academy of Technology study with IBM Fellow, Bob Blainey. This internal study, titled “Software Bloat, Causes and Consequences” consolidated observations drawn from discussions, debates, interviews and surveys capturing technical expertise and field experience from experts across product, solution and services engineering from both software and hardware systems disciplines. We highlight a few below.

**OBSERVATION 1: The perception of bloat depends on user expectations and the extent of scarcity of resources.** Observations from the study highlighted the subjective element of software bloat, where the problem of bloat can encompass different interpretations based on criteria such as:

- Which resource is affected by bloat?
  - It could be memory, CPU, disk, network capacity (physical computing resources)
  - It could be execution time, power, bandwidth
  - It could be installation and operational effort (human skill and attention)

- Where is the perceived impact of bloat?
Chapter 2. The Problem of Software Bloat

- On total cost of ownership (TCO), such as infrastructure, operational and maintenance costs
- On system performance, including start up, response time, throughput and scalability
- On complexity and consumability, including usability, dependability, productivity

- What is considered essential and what is perceived as wastage?
  - This is subjective based on deployment environment, usage scenarios and target segment expectations

- When is bloat perceived to matter?
  - When software becomes too complex to use or economically unviable to deploy
  - When there are hard limits in terms of hardware capacity, operating environment restrictions or real-time demands
  - When comparable light-weight alternatives exist

**OBSERVATION 2:** Attributing a cost to bloat is an open challenge  
Although most participants reported that bloat caused them problems and also considered it important to understand the cost of bloat, very few could attribute such a cost. Inability to attribute a cost to bloat makes it difficult to identify optimization opportunities and assess where cost-benefit tradeoffs are justified. A related finding was that even during software development, the lack of tools that predict costs and provide early feedback about the implications of implementation choices can lead to poor decisions resulting in bloat.

**OBSERVATION 3:** Shifting technology trends are driving the urgency of addressing bloat

- **Growing concern about software complexity and consumability** as solution stacks get unwieldy with mounting framework layers, to the point of hindering productivity.
• *Hardware limitations in coping with spiralling software demand* as the free ride on continuously increasing single thread performance growth, fast memory and energy efficiency improvements can no longer be taken for granted, necessitating software optimizations for exploiting new hardware efficiently.

• *Economics of server consolidation and cloud computing* as growing data center operational costs (including power consumption and administration effort) favour lightening software resource consumption to enable more applications to be provisioned on existing hardware infrastructure.

**OBSERVATION 4: Root causes of runtime bloat in Java arise from engineering practices that prioritize development productivity over runtime efficiency.** In large applications, bloat results from a pileup of inefficiencies stemming from a combination of disparate decisions across different development groups. The following high level causes were highlighted:

1. *Frameworks without a budget*, or the creation and use of frameworks without considering their impact on resource usage. Frameworks can have high resource costs which are compounded when they are easy to misuse, when they induce frequent data transformations and when API boundaries introduced by additional layers limit effectiveness of JIT (just-in-time compiler) optimizations and cause global efficiencies.

2. *Efficiency being an afterthought:* Performance and efficiency considerations are typically postponed to the end, if at all, leaving developers unaware of potential bloat introduced. For example, Mitchell and Sevitsky describe a case where developers estimated that they were using only about 2KB of state per session while actual measurements showed 200KB of per-session state, which explained why the application was scaling poorly and running out of memory at much lower loads than expected [88]. In this case the developers were two orders of magnitude off in their estimates.

3. *Feature creep and backward compatibility:* New features keep getting added to meet evolving standards, formats and changing requirements, but once the code for a feature is introduced it is very hard to remove it even when it is no longer useful.
Chapter 2. The Problem of Software Bloat

4. *Systemic bloat of the Java platform* Java’s data modeling limitations\(^3\) combined with high data representation and communication costs can force applications into resource heavy designs that are hard to optimize [86]. Automated optimizations performed by runtime systems are mostly local in scope and ineffective in tackling problems of bloat which typically tend to span components.

5. *Bloated specifications and protocols* Standard specifications and protocols built for high flexibility, interoperability and extensibility tend to result in bloat for common use cases. Excessive use of XML\(^4\) and SOAP\(^5\) introduces processing and communication overheads that reduce achievable peak transaction rates. For example, a legacy financial transaction message (ISO8583) of 122 bytes turns into a 15.3KB message when represented in XML. Accessing an NOAA\(^6\) weather service using a WSDL\(^7\) API was observed to generate 30X temporary Java objects (churn) and invoke 20X more method calls compared to a REST\(^8\) based API. Hence the use of these protocols causes bloat in situations which do not require the power of interoperability and flexibility provided.

2.3 Different Forms of Software Runtime Bloat

This thesis is primarily concerned with software runtime bloat, or the kinds of bloat that directly result in inefficient usage of runtime resources and can thus impact system power-performance. In particular we focus on Java based software where certain types of runtime bloat have been documented or analyzed by previous researchers. The following are some examples:

- Framework bloat
- Protocol bloat

\(^3\)for example, composition is often preferred over inheritance but composition in Java must be implemented as delegation, resulting in extra indirections and the use of several objects to represent even basic data structures

\(^4\)Extensible Markup Language
\(^5\)Simple Object Access Protocol
\(^6\)National Oceanic and Atmospheric Administration
\(^7\)Web Service Definition Language
\(^8\)Representational State Transfer
Chapter 2. The Problem of Software Bloat

• Feature bloat

• Data structure bloat [87]

• Temporary object churn bloat [41]

• Transformation bloat [89]

• Java heap bloat

• Java native memory bloat [96] (or internal bloat of the runtime system)

• Code size bloat

• Execution pathlength bloat

The above types are interrelated and sometimes overlapping. For example, both data structure bloat and temporary object churn lead to bloat in the Java heap. Framework and protocol bloat can cause unnecessary (bloated) transformations. Transformation bloat generates excess objects and bloats execution pathlength. Feature and framework bloat lead to increased code size and pathlengths. Protocol bloat can lead to higher communication payloads. Code size and communication bloat can lead to internal bloat in Java native memory usage especially as a side-effect of JVM optimizations such as the use of direct byte buffers and bytecode generated at runtime to optimize reflective invocations.

Runtime overhead manifestations We characterize the different forms in which bloat is manifested at runtime according to the following major categories:

• Excess Work (Code execution)

• Excess State (Data generated)

• Excess Communication (I/O transfer)

Fig 2.2 illustrates the interrelated nature of the categories. Data structure bloat (which can affect the size of both long lived and short lived objects), temporary object churn bloat (excessive generation of short lived objects) and Java native memory bloat are examples of bloat.
Chapter 2. The Problem of Software Bloat

Figure 2.2: Relationship between different types of bloat and their runtime manifestations

involving excess state (data). The presence of excess data is typically associated with excess work to maintain the state (e.g. to create, lookup, update, copy, parse, format and transform data). Likewise, excess work typically results in the generation of excess objects as a vehicle for processing relevant state, dynamic dispatch of relevant methods and ferrying data across API boundaries. Bloat at a higher level of abstraction, such as framework, protocol and feature bloat may be viewed as the inclusion of excess or incidental software concerns and can thus result in excess work as well as excess state and excess communication.
2.4 Systemic Origin of Software Runtime Bloat

Applications tend to accumulate excess function and objects as a side-effect of the same software engineering paradigms that enable support for ever more complex business logic, integration requirements and operational considerations. Mitchell, Schonberg and Sevitsky discuss four software development trends that lead to Java runtime bloat [86] in large framework based applications:

1. A culture of using objects (both long lived and temporary) to represent even the simplest of concerns without consideration of costs
   - Example: Fine grained data modeling with high per-object overheads
   - Effect: Memory bloat, object delegation costs, excess temporaries

2. A sea of (costly) abstractions and excessive layering
   - Example: Designing without context and inappropriate reuse
   - Effect: Barrier to optimization (both for humans and JITs)

3. Computers as communicators integrating disparate services and data sources
   - Example: Impedance mismatch between constituent services and data sources
   - Effect: Excessive transformations and data duplication

4. “Just-in-case” programming and anticipatory flexibility
   - Example: Insulation abstractions and (self-describing) dynamic types
   - Effect: Memory bloat (due to embedded strings and collections used for associating structural information description such as field names with data values) and execution bloat due to the overheads of data-driven dynamic dispatch

These trends can be collectively viewed as emphasizing a sound general design principle: opting for a safe (but heavy) overapproximation (superset) of requirements when choosing or
Figure 2.3: Accumulation of excess functionality across layers of redeployable components at many levels of granularity. The circles represent features implemented by a component; intersecting circles signify feature interactions. The arrows connect features in a component with features it uses from other components. Blue circles represent features that are needed by a particular application MyApp. Yellow circles represent features that are currently in excess for MyApp but exist to support other components or future needs. Not all excess features induce runtime bloat, however.

implementing standardized reusable functionality and data structures at all granularities from low-level libraries to high level solution composition.

This view highlights two related systemic sources of excess functionality (Fig 2.3). First, it is safer to overprovision features supported by components built for widespread deployment, where it is difficult to anticipate all potential usage situations. This emphasis on flexibility encourages the creation of overgeneral component and data structure implementations. Second, when choosing components and data structures to construct solutions from, software developers tend to rely on simpler and less detailed mental models that overapproximate desired requirements in a sound manner (similar to the manner in which abstract interpretation
techniques for constructing computable yet sound static analysis tools rely on an overapproximation of possible behaviors of a program by a less detailed and hence simpler to verify model). This emphasis on productivity encourages the selection of concrete components that satisfy more features than necessary in a given situation. The combination of these two effects (productivity pressure and flexibility pressure) at every layer leads to the presence of excess features in software.

Although this simplified systemic view illustrates why excess features can accumulate in redeployable software components (Figure 2.3), as we show later in this thesis, the presence of excess features does not necessarily explain the presence or intensity of runtime bloat even though it increases code size. In other words, runtime bloat need not be an inevitable consequence of overprovisioning features, often an appropriate design strategy for such applications. Runtime overheads can occur due to excess state that must be traversed or processed at runtime even when the corresponding feature is not relevant. This includes excess intermediate objects and transformations that are generated in order to reuse APIs that support some excess functionality. The challenge is to figure out how such side-effects could be mitigated to enable flexibility without incurring bloat.

2.5 Modeling and Measuring Bloat

Software does not come with built-in labels that indicate which portions of computation are necessary for a given application and which lead to bloat. Estimating the amount of resources a non-bloated implementation would have consumed for a specific execution is also difficult. Understanding the nature and sources of different types of software bloat is the first step to addressing the issue. The second is to quantify the magnitude of excess resource consumption attributed to each type of bloat. While the latter enables an estimate of how much room for improvement exists, the former provides insights on how to fix the problem.

In their study of large framework-based applications, Mitchell et al. observed that information-flow patterns for a data record involved a sequence of expensive transformations with multiple levels of nesting [89]. For example, moving a single date field from a SOAP message to a Java
Figure 2.4: Excessive transformations in framework based software. A sequence of expensive transformations with nested transformations [89].

An object in a stock-brokerage benchmark involved 58 transformations and generated 70 objects (Figure 2.4). Many of these were facilitative transformations for reusing existing parsers, serializers, and formatters. The observations highlight the considerable overhead expended in supplying data to the application’s core business logic. The study also proposed metrics based on modeling runtime information flow to classify and characterize the nature and volume of data transformations executed. However, these measures have not been automated till date.

Automated measures of bloat rely on a variety of heuristics to distinguish incidental overhead due to a specific category of bloat from strictly necessary resource usage. When analyzing Java heap snapshots, for example, it is possible to automatically differentiate data structure representation overhead such as bytes expended by the JVM object headers, pointers (object references), collection glue, and book-keeping fields from the actual application data contained in the structures. Mitchell and Sevitsky introduced the notion of a data structure health...
Chapter 2. The Problem of Software Bloat

Java memory bloat (long lived data structure health)

Analysis of heap snapshots

- Primitive Bookkeeping, 15.50%
- Null pointers, 1.60%
- Collection glue, 6.60%
- Delegation costs, 8.90%
- JVM object headers, 29.00%
- Actual data, 38.40%

60% bytes in representation overhead

Uses 2.5X memory of actual data size for long lived structures

Mitchell and Sevitsky: The Causes of Bloat, The Limits of Health, OOPSLA'07

Figure 2.5: Data structure bloat: High percentage of representation overhead

signature, a relative measure of memory consumed by actual data versus associated representational memory overhead (Figure 2.5), which reveals that data-structure bloat can increase the memory footprint of long-lived heap data structures by anywhere between a factor of two and five [87]. Their work was the first to automate the diagnosis and quantitative measurement of a type of bloat and to show how to generate formulas to predict asymptotic behavior of the health of a given data structure design at high data scales.

From a power-consumption standpoint, execution bloat and associated excess temporary-object generation are even more interesting. However such overheads are more difficult to distinguish and characterize automatically. Xu et al. and other researchers have made good progress in diagnosing some signs of potential bloat by focusing on data flow patterns in the use of temporary objects—for example, excessive or expensive data copies [138] and the creation of expensive data structures with low utility [140].

\footnote{In contrast with compiler optimizations and performance analysis tools that focus on control flow, such as expensive or frequently executed methods}
Table 2.1 categorizes existing diagnostic approaches in terms of whether they focus on identifying specific overheads\textsuperscript{10} or on identifying cost-benefit imbalances\textsuperscript{11}.

<table>
<thead>
<tr>
<th>Bloat characterization approach</th>
<th>Diagnostic measure</th>
<th>Analysis type</th>
<th>Profiling expense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead indicators</td>
<td>Data structure health [87]</td>
<td>(Dynamic) heap snapshot</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Loop invariant fields (hoistability) [142]</td>
<td>Static</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Copy profile [138]</td>
<td>Dynamic slicing</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Temporary object usage [41]</td>
<td>Blended (dynamic+static)</td>
<td>High</td>
</tr>
<tr>
<td>Out-of-proportion costs</td>
<td>Inefficient container usage [141]</td>
<td>Static (or dynamic)</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Low utility data structures [140]</td>
<td>Dynamic slicing</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2.1: State-of-the-art techniques for measuring different indicators of bloat categorized according to the underlying heuristics and extent of reliance on intrusive profiling

**Characterizing overhead indicators:** Data structure representation overhead [87] and chains of data copies [138] are both indicative of overheads with respect to actual data. Detecting the former requires only a heap snapshot [85] to be captured at runtime. Detecting the latter requires JVM instrumentation that incurs a high profiling overhead as it traces detailed information flows. Another indication of bloat is the repeated construction of data that is actually invariant across loop iterations and can potentially be hoisted outside the loop [142] – these indicators can be detected using a static analysis.

**Characterizing out-of-proportion costs:** On the other hand, inefficiently used containers and low utility data structures which are expensive to create are indicative of cost vs benefit imbalances. Xu et al. show how the former can be detected using a static analysis [141] and the latter can be identified by using abstract thin dynamic slicing [140] to compute relative object construction costs. These costs are computed in terms of the number bytecode instructions required to produce a value from already available fields; the analysis relies on JVM

\textsuperscript{10} to characterize what is \textit{in excess}

\textsuperscript{11} to characterize what is \textit{excessive}
instrumentation with a high profiling overhead.

None of the current state-of-the-art techniques can measure how much overall excess resource consumption and energy waste are attributable to bloat.

2.6 Mitigating and Avoiding Bloat

2.6.1 Semi-automated approaches

Tackling the source of bloat usually involves manual source-code fixes and assumes some domain knowledge about the application. In a few cases—for example, when the bloat originates in inefficient data structures—advisory tools can minimize manual effort. For example, tools such as Chameleon [45] for Java and Perflint [73] for C++ have adopted rule based approaches for recommending improvements in the choice of collection data structures used by an application. Chameleon has an option to apply changes automatically by enabling an adaptive selection of appropriate efficient collections at runtime. Chis et al. [36] have developed an analysis framework for discovering high impact patterns of memory inefficiency from heap snapshots along with suggested remedies. Their tool covers eleven commonly occurring patterns of memory misuse observed in Java applications.

Detection of performance anti-patterns in component based applications for enabling manual or even automatic performance optimization is an active area of research. Parsons and Murphy have developed monitoring techniques for capturing runtime paths and inter-component communication patterns and mining these to automatically detect several J2EE performance design and deployment antipatterns [99]. A few of these anti-patterns are likely to be related to bloat (e.g. the unused data object pattern, bloated session anti-pattern, fine grained remote calls) while others reflect high level performance problems, e.g. inefficient configuration of container pools or inappropriate transaction sizes. Wang et al. explore a pattern based performance optimization approach using automatic refactoring to replace such anti-patterns with more optimal patterns at runtime [127] by exploiting a reflective middleware framework. Such approaches are suitable only for bloat patterns that can be detected and fixed at a coarse level of introspection.
2.6.2 Automated code optimization

Traditionally, software development paradigms have evolved with the expectation that it the responsibility of system tools, runtime optimizers and underlying infrastructure to ensure ef-
cient program execution. However, runtime bloat occurs despite the best efforts of modern compiler and runtime optimizers to automatically minimize inefficiencies in software code. There are two key limitations that such optimizers face when it comes to addressing bloat. First, being inherently agnostic to the higher level purpose of the code (e.g. software fea-
tures provided), they cannot distinguish function and objects that are truly essential from those that are unnecessary (in excess) but still happen to be invoked or referenced during a possible program execution. Second, although some redundancies induced by bloat could in principle be detected purely by observing low level code, the problems of bloat typically span several API boundaries and deep call chains - situations where existing analysis approaches are too conservative and/or too expensive in practice to exploit optimization opportunities effectively.

Even so, automatic code-optimization techniques can mitigate finer symptoms of bloat. In particular, storage representation optimizations such as object fusing [132] (and inlining) or space-optimized object headers can reduce some of the overheads of fine-grained and highly delegated\textsuperscript{12} data models. Automated JVM level techniques have been developed for reducing string memory inefficiencies [62], an important source of memory bloat. JVM level approaches have also been explored for tackling the incidence of excessive temporary object generation in component based applications. For example, Shanker, Arnold and Bodik applied escape analy-
sis after inlining method call chains in high object churn regions identified using a lightweight dynamic analysis [112]. Language extensions such as additional modeling options and specification of design intent (relationships, transformations in type systems) have also been proposed to enable more effective automated optimization and efficient storage representation.

A runtime optimizer could exploit speculative optimization by monitoring and intercepting dynamic data flow behavior to realize de-bloating opportunities spanning several layers of API boundaries without becoming overly conservative. However, many of the dynamic analysis

\textsuperscript{12}the pointer indirection costs of using multiple levels of object delegation for representing composition
techniques for diagnosing bloat that are described in Table 2.1 incur a heavy runtime overhead - they can cause a ten fold slowdown and are only intended to help developers perform offline optimization. To achieve continuous object access profiling [95] of sufficient accuracy while incurring a low enough runtime overhead to be used as a basis for new online object optimizations, Odaira and Nakatani propose a memory protection based technique called pointer barrierization. Their object access profiles capture dynamic behaviour properties such as write-only objects, immutable objects and non-accessed bytes. Write-only object access profiles are used as basis for speculative compression of character arrays as an online optimization which employs pointer barrierization as a recovery mechanism. The profile of non-accessed bytes is used for another optimization - the dynamic adjustment of initial sizes of containers to optimal sizes to avoid memory wastage. Profiling immutable objects can enable copy-on-write optimizations. Although such online optimizations result in performance gains that well exceed the profiling overheads in some situations according to the paper, there are other situations where the overheads are too high for online profiling (the technique is still useful for offline profiling in these situations). Further, the energy consumption overheads of using continuous object profiling have not been studied.

2.6.3 Can runtime bloat be avoided by construction?

Modularity is fundamental to the composability of software packages and to their rapid development and deployment. However, the prevalent approach to achieving it can lead to significant software bloat, which is detrimental to power, performance, and energy efficiency. The real issue isn’t modularity itself but that, because of the difficulty in modularizing functions exactly as needed, programmers inadvertently introduce superfluous processing and data overhead for reuse.

The traditional maxim for creating lean software, as advocated by David Parnas [98] and Niklaus Wirth [133], among others, is to engineer it right by adopting minimalist design principles that avoid bloat. Such software is built in a series of stepwise refinements carefully crafted to provision each potential use case without sacrificing extensibility or reuse. The Linux kernel illustrates the successful adoption of this principle to efficiently satisfy diverse
environments and requirements [16]. In framework-based environments, however, this approach is impractical. Many redeployable components must be dynamically programmable by business analysts and integrate with dozens of heterogeneous systems and information sources. Thus, it is not easy to anticipate usage of a component at design time, nor is it feasible to incrementally change intermediate interfaces later. Programmers are often unaware of the overhead that systems might incur during actual deployment, and a low-level runtime optimizer can’t “know” their intentions. Improving cross-layer line of sight into high-level functional intent and interoperability overhead—for example, data-supply inefficiencies such as transformations and copies to facilitate reuse—will help both programmers and runtime systems deliver better energy-optimization solutions by minimizing bloat automatically. However, this remains an open problem.

2.7 Conclusions

We find that the first challenge in addressing the problem of bloat is that the term “software bloat” lacks a consistent definition. It has several connotations depending on the perceived effects of bloat (complexity or resource inefficiency), its perceived sources (excess features or poor implementation) and how the presence of bloat is gauged (in terms of trends over successive software versions, out-of-proportion costs compared to useful work achieved or indicators of excess functionality and overheads). For the purpose of this dissertation, we are interested in understanding potential systemic characteristics that influence the origin of software runtime bloat and its direct impact on system power-performance. Thus, we will focus mainly on bloat’s perceived effect on resource inefficiency gauged in terms of indicators of overheads due to the combination of perceived sources of excess features and poor implementation choices.

Techniques for measuring, managing, and mitigating runtime bloat face significant challenges but there has also been progress in these areas during the past few years. Most of this work has been focused on inefficiencies observable from data structure and data flow patterns in the form of specific overhead indicators or out-of-proportion costs. Although this has been
a very fruitful line of research, mitigating these inefficiencies requires substantial manual effort due to the limited scope of automated de-bloating techniques available. Thus, such efforts need to be complemented by a higher level analysis perspective to characterize the implications of bloat for system power-performance and to relate the popular (user level) view of software bloat [131] in terms of optional features to the deeper forms of inefficiencies identified as runtime bloat by previous researchers. For instance, it would be useful to qualify program characteristics that determine when feature bloat (or more generally, program concerns that are in excess for a given deployment situation) can result in runtime execution bloat. There is also a need for more automated techniques for mitigating bloat in existing software. Finally, it remains an open question whether a principled approach could be applied to guide the construction of new systems that possess a lower propensity for runtime bloat without compromising on flexibility or development productivity.

**Acknowledgments for this Chapter**

We thank Gary Sevitsky, Nick Mitchell, Matt Arnold, Edith Schonberg for their work on Java runtime bloat and regular discussions that helped shape our understanding of the deep systemic issues that lead to bloat. We also thank all the members of the IBM Academy of Technology study on “Software Bloat - Causes and Consequences” (2009) – in particular, the study co-lead Bob Blainey and the workstream leaders Gary Sevitsky, John Easton and Patrick Mueller – for their contributions and rich insights on the problem of software bloat in context of hardware and software technological trends.
Chapter 3

Power-Performance Implications of Java Runtime Bloat

We conduct a systematic empirical and analytical study of the impact of runtime resource bloat on energy-efficiency. We find that the impact could vary widely with hardware and software characteristics due to a curious interplay between bloat, energy proportionality and system bottlenecks.

3.1 Introduction

As mentioned in earlier chapters, excess resource usage from bloat could lead to energy inefficiency and reduced performance [139]. Energy savings from reducing bloat are anticipated via both direct and indirect effects (e.g., server consolidation opportunity) of reducing excess resources. However, reducing bloat is non-trivial, particularly because its origin is linked to the same software development trends [86] that have been extremely successful in fueling the

---

1The work described in this chapter has appeared in the following publications:

2In this chapter we will use Definition I from Chapter 2, Section 2.1.3: “Bloat” is the resource overhead induced by the presence of excess functionality and objects, typically as a result of using highly general components standardized around deeply layered frameworks.
growth and widespread impact of redeployable software. Hence, there is a need to develop approaches to assess cost-benefit implications of reducing bloat and focus these efforts where they matter the most.

While reduction of bloat is expected to (obviously) have an impact on power-performance, we find that the inherent slack in large scale IT solution architectures and the presence of elements in the system that are not energy proportional\(^3\) may require this intuition to be qualified.

Is the effect of bloat more pronounced or less pronounced in the presence of energy proportional hardware? Should we expect an increase or a decrease in peak power consumption with lower bloat? In two different experiments on the same hardware platform, but involving different kinds of bloat, one with Apache DayTrader and another with a microbenchmark, we obtained opposite answers. Even with the same workload and same de-bloating optimization, we have seen wide variations in the effects with different hardware and software characteristics. There is, therefore, a need for empirical and analytical studies that validate intuitions and provide a deeper understanding of the relationships between measures of bloat and energy consumption.

Conducting a systematic study of the effects of bloat is challenging, in part due to the same reasons why addressing bloat is non-trivial. Not only does finding and fixing bloat involve manual effort and domain expertise, the nature of bloat varies from case to case and its impact can be very workload and system specific. This can make it difficult to arrive at generalizable conclusions from such a study.

In order to conduct a systematic investigation of the impact of bloat on power-performance, we focus on a detailed study of single type of bloat that is very common in Java applications, the presence of excess temporary objects, that is relatively easy to find/vary, but exhibits a diversity of effects impacting multiple system resources. We devise a methodology for systematically varying bloat and hardware and software characteristics to manifest these effects in the context of a standard server side Java power-performance benchmark, SPECpower_ssj2008.

Our experimental findings reveal a wide variation in the impact of bloat, indicating that a

\(^3\)energy proportional [14] hardware consumes power in proportion to actual resource utilization
software only view is inadequate for assessing the energy-efficiency benefits from bloat reduction. We develop an abstract model to enable a generalized exploration of the effect of bloat on power-performance, from a whole system perspective.

The complexity of modern software and system layers make it impractical to compute the exact power-performance impact of run-time bloat (reduction) analytically. However, our analytical model is still useful in reasoning about implications of reducing bloat. It surfaces certain aspects of the complex behavior that determine the impact of bloat, by abstracting operational relationships at bottleneck zones. This creates a foundation for reasoning quantitatively about the impact of bloat reduction on system power, performance and energy-efficiency. For example, on the one hand, energy proportionality of a bloat-impacted resource can amplify benefits of reducing bloat. On the other hand, energy proportionality of the remaining resources can shrink those benefits when bloat affects a bottleneck.

Contributions:

1. We conduct the first systematic empirical study of the joint power-performance implications of software bloat, using SPECpower_ssj2008, across a range of server hardware and software configurations covering different processors/system designs, levels of power management, multithreading, cache sizes, and heap sizes. We undertake controlled experiments varying the degree of bloat for one prevalent type of Java runtime bloat, excess temporary objects, and the system/software characteristics to investigate the power-performance effects. Using multiple metrics for evaluating the impact of bloat - peak power, equi-performance power, peak performance, and resulting energy efficiency measures - we systematically analyze and expose the effect of energy proportionality and bottleneck resources on the impact of bloat reduction.

2. We develop a simplified analytical model for studying the implications of runtime bloat on power and energy efficiency under different situations. The model takes into account bottlenecks in the system as well as the energy proportionality characteristics of hardware in the system. It provides a generalized abstraction to enable bloat impact
estimation (or prediction of trends) for various scenarios. Analyzing different scenarios with the model enables us to put our experimental results in perspective, with the experimental results re-affirming the effects predicted by our model in real systems.

3. We show that a whole system perspective is required for an informed assessment of the impact of bloat and its reduction. The nature and extent of impact is dependent on whether a bloat-impacted resource is a bottleneck resource as well as on the energy proportionality of not just the bloat-impacted resource but other key resources as well.

The chapter is organized as follows. Section 3.2 details different effects of Java temporary object bloat and resources they impact. Section 3.3 describes our empirical study and summarizes the findings. Section 3.4 and section 3.5 describe the model, its implications and their corroboration by our experimental observations. Section 3.6 covers related work and Section 3.7 concludes.

3.2 Temporary (Java) object churn bloat and its effects

In Java, the programming model encourages the creation of many short-lived objects - these are allocated on the heap\(^4\) and may be passed across framework layers, only to be used briefly and eventually garbage collected. For example, temporary objects are generated repeatedly during data transformations required for framework based reuse.

This can lead to the problem of object churn, i.e. a high volume of temporary objects [112] spreading out the memory access footprint of the application (because the memory is not reused until after the next garbage collection cycle). Excessive allocation of these temporary objects, which we refer to henceforth as temporary objects bloat, is a common type of Java runtime bloat; it causes excess memory usage and has deep and pervasive effects on the performance of a system. We find it a good representative of bloat for our empirical studies, as it exercises multiple hardware resources including the CPU and the memory system.

\(^4\)state of the art escape analysis techniques in modern production JVMs can only optimize a small fraction of these allocations [112]
Figure 3.1: Effects of Java temporary object creation bloat

Figure 3.1 depicts some of the potential effects of temporary objects bloat in Java. We discuss these briefly.

### 3.2.1 Allocation wall effect

Researchers have previously observed [147] that a large volume of temporary object allocations causes high memory write bandwidth consumption which limits scalability of certain classes of Java programs on multi-core platforms. The term “allocation wall” has been used to describe this phenomenon, as this slowdown is due to allocation overheads and not garbage collection cost.

### 3.2.2 Heap pressure effect

Creating many temporary objects can increase garbage collection overhead. Given a sufficiently sized JVM heap (typical in well-tuned JVM configurations) and a generational garbage collector, the percentage time spent in GC\(^5\) becomes small enough that it could be ignored in

\(^5\) garbage collection
comparison to other effects. Even so, de-bloating software to reduce temporary object allocation can reduce its footprint by enabling lower heap sizes to be used for the same application. Heap size reductions are beneficial from a power-performance perspective, when the memory footprint savings is significant enough to (i) enable inactive ranks of memory to be switched to a low power mode (ii) enable use of a smaller memory configuration of the system for cost and power savings, or (iii) enable allocation of spared memory to another virtual partition for improved workload consolidation with same system configuration/power envelope.

3.2.3 Object construction computation overhead

If the construction of a temporary object involves a costly initialization sequence for setting up its fields, object creation can also have a significant instruction pathlength impact. Object construction computation overhead manifests as an increased CPU utilization due to increased instructions per transaction. It may or may not have a significant effect on the CPI\(^6\) depending on the nature of instructions added.

3.2.4 Influence of system configuration

A single category of bloat may have multiple effects as described above, each of which, in turn, impacts physical resource utilization in different ways. A given effect may have a low impact in one situation but can become crucial under a different set of system/workload characteristics.

For example, the “allocation wall” effect [147] is observable in the presence of bottlenecks caused by constraints in memory system resources like cache, memory bandwidth and latency. In this situation, it leads to increased (effective) CPU utilization through its impact on the CPI. This increased utilization impacts power consumption. However, if the system is configured with a processor DVFS\(^7\) algorithm that is memory slack aware [135], it could exploit the available slack to run the processor at a lower speed; hence, the change in CPI may have a relatively lower impact on the effective (scaled) CPU utilization and power consumption.

\(^6\)cycles per instruction
\(^7\)dynamic voltage and frequency scaling
In other situations with relatively large on-chip caches and powerful memory subsystems, where memory performance is not a constraint, the allocation wall can effectively become irrelevant. Reducing bloat can still be beneficial in this case, due to another effect. Lower temporary object allocation rates enable a Java application to be configured to use a smaller heap size without an increase in GC time. If this heap size reduction is sufficient to enable the working set of the application to be entirely contained in a (huge) on-chip cache (a realistic possibility in the future, with 3D integration), off-chip memory accesses may be avoided altogether, potentially leading to significant power savings.

3.3 Experimental study

We conduct a systematic study on the quantitative impact of bloat reduction on power and performance in the context of varying system characteristics and bottleneck scenarios with the following methodology:

• Adopt SPECpower_ssj2008 as the evaluation workload as it has well defined performance measures, power measurement methodology, load variation and both detailed result reports and a composite energy-efficiency measure over a full range of system loads.

• Create lower bloat and higher bloat variants of the baseline code using different degrees of object reuse to vary the extent of bloat.

• Conduct experiments across multiple platforms with differing energy proportionality and memory system characteristics to understand their impact on bloat reduction’s benefits. Further vary software/system characteristics to examine the impact of the availability of key resource(s) on the benefit of bloat reduction.

3.3.1 SPECpower_ssj2008 and Metrics

SPECpower_ssj2008 is a server side Java (“ssj”) energy-efficiency benchmark. As the first commercial workload benchmark from SPEC [116] requiring power consumption and energy
efficiency reports across the full range of system loads, it is well-suited for controlled experiments to study the power-performance impact of software runtime bloat in the context of server-side Java. Additionally the benchmark offers relatively easy means to vary the amount of temporary object allocation bloat, giving us the means to study the impact of the extent of bloat (or its reduction).

There are 11 measurement intervals; starting from 100% load as measured during calibration, the intensity drops by 10% for each subsequent interval, finishing with an idle interval. During the entire run, the system power consumption is measured. The benchmark metric is an energy-efficiency score composed of the ratio of the sum of the transaction throughput of every load interval to the sum of the average power of every load interval (including the idle period). This is a fixed time rather than a fixed work benchmark, i.e. each load level is maintained for a fixed time. For our analysis, in addition to the energy-efficiency score across the load-levels we also use comparisons of peak performance, power at peak performance and power at equi-performance points.

An equi-performance power comparison between two alternatives compares power measurements taken at the same performance level (application throughput) for the two alternatives. It also serves as an energy-efficiency measure at constant performance, useful in assessing relative efficiency impact of bloat reduction when there is also a change in performance as result of bloat (reduction). For SPECpower_ssj2008, we utilize the availability of measurement results over the different load levels to derive equi-work or equi-performance comparisons (interpolating for power between the load levels, if necessary).

### 3.3.2 Varying extent of bloat

To understand the impact of the extent of bloat, we introduce variation in temporary object bloat in the SPECpower_ssj2008 application by increasing or decreasing the degree of object reuse. Object reuse is a well-known coding technique for reducing object creation in Java. As it is non-trivial to automate [147], we had to implement the variants by inspecting the
benchmark code and making manual code changes. We employ three code variants for a different extent of bloat (or of bloat reduction):

- AllocOrig: the baseline SPECpower_ssj2008 code.

- AllocLess: a lower bloat variant obtained by implementing object reuse to eliminate more than half of the temporary object memory allocations in the original code. This reduces temporary object allocations from 8KB/transaction to 3KB/transaction on the systems running a 64 bit JVM, and from 6.9KB/transaction to 1.8KB/transaction on the system running a 32-bit JVM. The form of object reuse here only reuses the memory allocation, and not object content, hence it does not save object construction computation costs for initializing fields.

- AllocMore: a higher bloat variant obtained by turning off object reuse in the original code, introducing excess temporary object allocations with object construction computation overheads. AllocOrig had an explicit optimization to maintain a pool of previously constructed objects for reuse, where the content of many fields of the objects in question could be reused without being overwritten, thus saving on object construction cost in addition to memory savings. Turning off this optimization raises the temporary object allocations from 8KB/transaction to 11KB/transaction on the systems running a 64 bit JVM and from 6.9KB/transaction to 9.8KB/transaction on the system running a 32-bit JVM. While a code change was necessary to introduce bloat in this workload, it often occurs in practice when optimization opportunities are not all exploited.

The volume of temporary objects given for each variant was computed by post processing garbage collection logs. Table 3.1 summarizes the specific effects of temporary objects bloat (described in the previous section) in terms of the variants and configuration constraints under which they could be manifested.

---

8the technique has recently been automated [19] for select object types
Chapter 3. Power-Performance Implications of Java Runtime Bloat

### 3.3.3 Experimental setup

Our experiments were conducted on three platforms to explore cross-platform differences in the impact of software bloat. Table 3.2 describes the three system configurations.

<table>
<thead>
<tr>
<th>System</th>
<th>Processor Config</th>
<th>Cache</th>
<th>Mem</th>
<th>JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power 750 AIX 6.1</td>
<td>IBM POWER7 4/8/4</td>
<td>32MB</td>
<td>64GB</td>
<td>J9</td>
</tr>
<tr>
<td>IBM HS21 Blade Linux 2.6.33</td>
<td>Intel Xeon E5450 2/4/1</td>
<td>12MB</td>
<td>8GB</td>
<td>J9</td>
</tr>
<tr>
<td>IBM x3650 M2 Linux 2.6.33</td>
<td>Intel Xeon X5570 2/4/2</td>
<td>8MB</td>
<td>64GB</td>
<td>J9</td>
</tr>
</tbody>
</table>

Table 3.2: Systems Configurations: Processor Config is Sockets/ Cores per Socket/ Threads per Core. Cache size is for the last-level per socket.

All systems are instrumented for obtaining system-level power measurements from the on-board service processors at fine-grained, periodic intervals for reliable power and energy-efficiency metric calculations. The Power 750 system also additionally provided processor and memory sub-system level measurements, processor throughput and memory bandwidth information at fine-grain, periodic intervals.

The Power 750 system uses the IBM POWER7 processor [128] with EnergyScale firmware [78]
employing new dynamic voltage and frequency saving (DVFS) algorithms for aggressive load-based energy savings [102]. The x3650 M2 system with the Intel Xeon X5570 processor uses the Linux `CPUfreq` ondemand governor also exploiting DVFS for load-based energy savings. The HS21 blade system with Intel Xeon E5450 processors was employed without any active power management.

Figure 3.2: **Cross-platform comparison:** All results are normalized with respect to the corresponding `AllocOrig` measurement.

### 3.3.4 Multi-platform experiments and results

Figure 3.2(a) presents the results of comparing the baseline variant `AllocOrig` with the lower and higher bloat variants, `AllocLess` and `AllocMore`, on the test platforms. The energy efficiency scores are based on full benchmark runs for the Power 750 and x3650-M2 system and the 100% load level measurement for the HS21 blade system. All results are normalized with

---

9When utilization rises above a threshold, the ondemand governor sets the frequency to the highest available. When utilization falls below the threshold, it gradually decreases the frequency as long as the system remains underutilized.
respect to the original (default) implementation, \textit{AllocOrig}.

We note a wide variation in power-performance impact of bloat. On the HS21 system \textit{AllocLess} shows 1.59X the energy-efficiency of \textit{AllocOrig} from reduction in bloat and \textit{AllocMore} shows 0.61X the energy-efficiency of \textit{AllocOrig} because of increase in bloat. The relatively poorer off-chip memory system of the HS21 blade system is responsible for a pronounced “allocation wall” effect which combined with the poor energy-proportionality characteristic (no active power management enabled) results in the biggest energy-efficiency impact for bloat (reduction) among the three configurations.

On the two larger server systems with more powerful memory subsystems and power management, i.e. the Power 750 and x3650-M2 systems, there is a lower but still significant impact from bloat. \textit{AllocMore} shows 0.76X and 0.85X the performance of \textit{AllocOrig} (from increased bloat) and \textit{AllocLess} shows 1.06X and 1.11X the performance of \textit{AllocOrig} (from reduced bloat) on the Power 750 and x3650-M2 platforms, respectively.

Figure 3.2(b) shows the impact on equi-performance power and power at the different peak performances (peak power) across the Power 750 and x3650 M2 systems. The equi-performance power graph shows the relative power of \textit{AllocMore} and \textit{AllocLess} both compared to \textit{AllocOrig} at a constant performance point - in the former case it is the lower peak throughput of \textit{AllocMore} (compared to \textit{AllocOrig}) and in the latter case it is at the lower peak throughput of \textit{AllocOrig} compared to \textit{AllocLess}.

\textit{The equi-performance power numbers indicate the synergistic impact of DVFS and bloat reduction.} As bloat is lowered from \textit{AllocMore} to \textit{AllocOrig} and from \textit{AllocOrig} to \textit{AllocLess}, for a given performance more slack is introduced. The DVFS algorithm then works on the slack to translate that to increased power reduction and energy-efficiency. Additionally, one can see how the effectiveness of the DVFS capability magnifies the impact of bloat reduction by comparing the larger equi-performance power reduction for Power 750 when reducing bloat compared to x3650 M2, the former platform having a bigger take-down in power with load reduction (cubic vs quadratic) as seen in Figure 3.6.

An interesting observation from the cross platform results is that \textit{even though the energy efficiency score improvement with AllocLess is smaller on Power 750 than the x3650 as shown
in Fig 3.2 (a), the corresponding equiperformance power results in Fig 3.2 (b) shows a higher (25%) savings on Power 750 than on x3650 (9%). The efficiency score improvements in Fig 3.2 (a) are at different performance levels. Equi-performance comparisons in Fig 3.2 (b), by normalizing to the same performance, show the true impact on energy-efficiency of bloat reduction. Bloat reduction decreases wasted resource usage and a more energy proportional system converts the saved resource usage into higher power reduction/energy savings.

Figure 3.3: Memory Statistics for SPECpower ssj2008 on Power 750 in SMT4 mode (normalized wrt AllocOrig)

The power numbers at peak are comparable across the bloat options. At peak performance, the numbers correspond to roughly the same processor activity across the bloat options and consequently roughly the same processor power. Figure 3.3 shows the impact of bloat on the memory controller and memory power statistics for SMT4 mode on the Power750 system. Reducing bloat reduces memory traffic to 67% resulting in a lowering of memory power consumption to 87%. The memory bandwidth per ssjop dropped to 63%. However, the power impact of this is marginalized by the significantly lower power consumption of the memory sub-system in this configuration.

3.3.4.1 Object construction computation overhead

The AllocMore variation showed significant performance degradation compared to AllocOrig (e.g. 0.59X for HS21 blade, 0.76X for Power 750). However when we measured the reduction
in IPC\textsuperscript{10} for AllocMore, it was very small (around 6\% on HS21 blade, even smaller on Power 750). The instructions per transaction, on the other hand had increased by over 25\%. This extra computation is a manifestation of the object construction costs that are involved when the object is not reused across iterations. We did not see any computation cost savings in the AllocLess case because it involved only a memory allocation reuse and not content reuse.

To probe further, we created a variant that combines the two changes AllocLess and AllocMore - this variant reduces the allocation wall effect (relative to AllocMore) but maintains the same object construction overhead. On the HS21 blade system this showed a high gain in IPC, resulting in a performance improvement of 8\% over AllocOrig, despite the extra instruction costs; on the Power 750 too, the hybrid variant had a significant improvement in performance over AllocMore but was still 15\% below AllocOrig (AllocMore being 25\% below). This indicates that the allocation wall effect is the dominant overhead for the HS21 blade (with a relatively poorer memory system) whereas the object construction computation overhead is a sizeable component of the impact of bloat with AllocMore on the Power 750.

\subsection*{3.3.4.2 Heap size and memory pressure effect}

The experiments discussed so far were executed with a sufficiently large JVM heap size (typical in well-tuned JVM configurations) in order to isolate the “allocation wall” and object construction computation effects of bloat from the heap pressure effects. We now turn our attention to heap pressure effects on impact of bloat.

Figure 3.4 shows the results on the HS21 blade and x3650-M2 system with significantly reduced heap sizes. Reducing the heap size lowers base performance on both platforms. The benefit from bloat reduction is more pronounced with reduced heap sizes (bars shown are normalized to the AllocOrig numbers for the full heap size). Further, even with a substantially smaller heap size\textsuperscript{11}, AllocLess low heap shows 27\% improvement in energy-efficiency over the original full heap configuration on the HS21 blade and 6\% improvement on the x3650 M2 server. This suggests that as an additional benefit of bloat reduction one can get good

\textsuperscript{10}Instructions per cycle
\textsuperscript{11}we could bring the heap sizes down to 25\% on the HS21 blade and 37\% on the x3650 server when running AllocLess, the reduced bloat alternative, and still exceed the performance of AllocOrig at full heap size.
Figure 3.4: Heap size reduction impact with decrease/increase in bloat: All results are normalized with respect to the corresponding AllocOrig measurement at full heap size for performance even with reduced heap sizes which would allow more instances of the workload (virtual machines) to be run in the same physical memory footprint.

3.3.4.3 Cache Pressure Effect

Cache resources of a microprocessor often have a key impact on a workload’s performance. Runtime bloat can reduce the effectiveness of the caches leading to lower performance. Alternatively, the impact of bloat reduction can vary with the shortage/plentifullness of cache capacity of the microprocessor. Here we examine the impact of cache pressure by varying it on a single platform (Power750) to better isolate its impact from other characteristics. We increase cache pressure by increasing the degree of hardware multi-threading from 2-way (SMT2) to 4-way (SMT4) and further by reducing the cache size to half (SMT4-halfcache).

In Figure 3.5 we see the impact of bloat reduction (for AllocLess over AllocOrig) as cache capacity per thread is progressively halved from SMT2 (2MB/thread) to SMT4 (1MB/thread) to SMT4-halfcache (512KB/thread). The SMT modes are changed through the OS facilities for setting the mode and SMT4-halfcache is realized by booting the system with reduced L3 cache sizes. Difference in energy proportionality characteristics is examined by doing the runs...
Figure 3.5: Cache capacity effect on improvement from bloat reduction (on Power750): SMT2 has most cache resource/thread, SMT4 has half SMT2’s capacity/thread, SMT4-halfcache has half SMT4’s capacity/thread. For energy-efficiency and peak performance the numbers show the improvement (increase) for AllocLess relative to AllocOrig. For equi-performance power and peak power the quantities shown represent the saving (decrease) for AllocLess relative to AllocOrig. For all cases, a positive number signifies an improvement for that quantity with bloat reduction.

at fixed frequency as well as with DVFS.

Bloat reduction shows the highest benefit when cache capacity is most constrained, for SMT4-halfcache and benefits decrease going to SMT4 and then SMT2, as cache capacity becomes more plentiful. There is actually an increase in peak power (negative power savings) with bloat reduction as the underutilized compute resources are utilized more improving throughput. This power increase is higher when using DVFS for adaptive power management than with fixed frequency operation because of the superlinear power characteristics with DVFS. Consequently, the energy-efficiency improvement mirrors the performance improvement for fixed frequency while energy-efficiency improvements are lower than the performance improvements for DVFS. As earlier, savings in equi-performance power (and equi-performance energy efficiency increase) is higher than the improvement in energy-efficiency at peak performance. The savings in equi-performance power is highest for SMT4-halfcache as bloat reduction is most effective when the cache pressure is maximum (more severe a performance bottleneck).
With the lowest cache-constraint, SMT2, the performance impact is the lowest and we actually observe a slight decrease in system power consumption (positive power savings), which is more pronounced at fixed frequency. Using the memory power statistics (not shown), we find an 18% reduction in memory power consumption, which translates to a lower overall system power savings because the memory component contributes to a small fraction of system power.

### 3.3.5 Experiments with other workloads

The degree of impact of temporary objects reduction on performance is workload dependent, as has been established from results on object churn reduction in prior work [147, 112]. In addition to our systematic studies with SPECpower_ssj2008 we conducted some smaller studies with other workloads.

**Microbenchmark Experiments:** We chose a simple microbenchmark, called "AllocMark", from [147] and extended this to create a microbenchmark called “AllocReuseMark” that uses different degrees of object reuse as a way of varying temporary object allocation bloat. Power and allocation rate measurements were collected for different levels of object reuse. We notice that reducing bloat by 75% improved the effective performance by 287%. This improvement was associated with a 19% increase in power consumption. As the increase in power consumption was relatively smaller than the performance gain there was a net improvement in energy efficiency.

**Apache DayTrader Experiments:** Temporary objects bloat is often an indication of other inefficiencies like excessive data transformations. To gain an assessment of potential power-performance implications of complex patterns of bloat in large server workloads, we ran comparisons of the DaCapo [27] tradesoap and tradebeans benchmarks. Both tradebeans and tradesoap execute the same underlying workload, except that tradesoap uses indirect calls through SOAP from the client (like many real workloads do) while tradebeans executes

---

12 Based on the Apache DayTrader J2EE workload, included in DaCapo v9.12; executed within the Geronimo application server utilizing the Derby in-memory database.

13 Simple Object Access Protocol
direct calls on the server. Comparing these two configurations is useful for studying the additional costs of fine grained messaging, layering and transformations that are needed for interoperability in the tradesoap implementation. Such excess costs are typical in patterns of software bloat observed in real applications. We observe a 15% increase in power consumption on the x3650 M2 system running tradesoap vs tradebeans, with a 2X slowdown in performance. tradesoap generated almost 10X higher footprint of temporary objects and caused a significant increase in CPU utilization. This shows that finer grain messaging and associated transformations (in tradesoap) can have a significant power efficiency cost, especially for small, high volume transactions.

Comparing the two sets of experiments above, we make an interesting observation. We see an increase in power consumption with lower bloat in AllocReuseMark and an increase in power consumption with higher bloat in DayTrader. In AllocReuseMark, bloat affects a bottleneck resource (stresses the memory hierarchy). In DayTrader, in addition to memory resources, bloat also affected a non-bottleneck (or underutilized) resource (the CPU), as was observed from the increased utilization for tradesoap over tradebeans. This could explain the increase in power consumption despite the slowdown in performance.

3.3.6 Key observations

Reducing runtime software bloat in the form of excess temporary objects has significant energy-efficiency impact, with upto 59% improvement at peak load seen on a memory performance constrained HS21 blade (1.59X energy-efficiency for AllocLess over AllocOrig) and 45% equi-performance power savings on a energy proportional Power 750 (1.83X equi-performance power for AllocMore compared to AllocOrig). The extent of impact was dependent on the bottleneck pressure with respect to the memory hierarchy resources and energy proportionality of the physical resources.

- Energy efficiency benefits at peak performance tend to be most pronounced when bloat affects a performance bottleneck and underutilized resources are less energy proportional (e.g. HS21 system), while equi-performance power savings are highest when impacted resources are super-linearly energy proportional (e.g. Power 750 DVFS).
• A higher degree of energy proportionality (e.g. power management in the form of processor DVFS) has a complementary impact with respect to addressing software bloat. On the one hand it can reduce the peak energy-efficiency degradation caused by bloat, if the energy proportionality is working on underutilized resources. On the other hand, it magnifies the equiperformance energy efficiency improvements brought about by reduction in bloat, if the energy proportionality is working on the resources that get freed up due to bloat reduction.

• Performance effects of bloat are most acute when memory hierarchy resources are already strained, e.g., bottlenecks due to low off-chip memory bandwidth as with the HS21, low allocated memory or cache pressure with smaller caches or higher degree of multi-threading.

• Power consumption at peak performance may increase or decrease with bloat reduction depending on the impact on resources underutilized in the presence of bloat.

Addressing runtime software bloat was also found to facilitate heap size reductions. The reduced bloat alternative performs better even with significantly reduced heap sizes.

3.4 Quantifying power-efficiency impact of bloat: a simple abstract model

Could our experimental observations be a manifestation of certain general implications that are valid for any category of bloat? In this section, we construct an abstract analytical model to better understand the power performance implications of reducing bloat and when its effects are more pronounced. This is a highly simplified model relating the hardware resource pressure caused by bloat and its impact on system energy efficiency using a few fundamental operational laws [32, 59] of queueing theory\textsuperscript{14}. Hence, it is generally applicable to any kind of resource

\textsuperscript{14}The laws used express basic operational relationships, such as the utilization law/equality, hence widely applicable and general enough to hold without any assumptions about inter-arrival or service times; we do not use any additional assumptions from operational analysis beyond these laws, e.g. we do not rely on product form queuing network conditions and do not assume closed loop networks.
Applications use a variety of hardware resources on any given system, e.g., processor cores, on-chip caches, off-chip memory and disk storage. An imbalanced use of these resources can cause a performance bottleneck at one resource and under utilization of others. For example, high cache miss rates caused by profligate use of objects due to bloat can cause underutilization of the processor cores. The power efficiency characteristics differ in nature and magnitude across resources types, e.g., CPU with a super-linear power versus load characteristic when using dynamic voltage and frequency scaling (DVFS) versus main memory with a linear power characteristic and high standby power. This difference leads to a very different impact on energy-efficiency for bloat reduction depending on which resource(s) is impacted by bloat and which are under utilized.

When a resource becomes the performance bottleneck because of bloat, reduction of bloat can increase power consumption because of increase in throughput. The varied and sometimes non-linear power characteristics of resources with load can make the impact of bloat reduction on power difficult to assess when compared across different throughput levels. To enable comparison on an equal footing before and after bloat reduction, we also analyze comparisons of power at equiperformance levels with and without bloat.

### 3.4.1 The Model

Let $R_i, i = 1..N$, be various types of resources with service demands (time) $D_i$ in the software without bloat. Let $b_i, i = 1..N$, be the overhead due to bloat introduced in each of these resources in the bloated software, changing the service demands to $D_i(1 + b_i)$.

Using asymptotic bounds based on bottleneck analysis [59] to approximate achievable performance, peak throughput is given by $X = \min_i(1/D_i)$ whereas peak throughput with bloat, $X_b = \min_i(1/((1 + b_i)D_i))$.

Let $P_i(U_i)$ be the power consumed by resource $R_i$ with utilization $U_i$ (the exact relationship between utilization and power can be different for different resources). Utilization $U_i$ of resource $R_i$ is $D_iX$ [59], when running the non-bloated software and $D_i(1 + b_i)X_b$ with the bloated software.
Chapter 3. Power-Performance Implications of Java Runtime Bloat

Power without bloat, \( P = \sum(P_i(U_i)) = \sum(P_i(D_iX)) \)

Power with bloat, \( P_b = \sum(P_i(D_i(1 + b_i)X_b)) \)

Power efficiency (perf/watt metric) without bloat,

\[
E = \frac{X}{P} = \frac{\min_i(1/D_i)}{\Sigma(P_i(D_iX))}
\]

Power efficiency with bloat,

\[
E_b = \frac{X_b}{P_b} = \frac{\min_i(1/((1 + b_i)D_i))}{\Sigma(P_i(D_i(1 + b_i)X_b))}
\]

We are primarily interested in quantifying potential improvements from bloat reduction. Hence, we define the metrics of interest for relative throughput, peak power, power-efficiency and equiperformance power of the non-bloated software normalized with respect to that of the bloated software.

Relative throughput with bloat reduction (\( > 1 \) is good),

\[
\phi_x = \frac{X}{X_b} = \frac{\min_i(1/D_i)}{\min_i(1/((1 + b_i)D_i))}
\]

Relative peak power with bloat reduction (\( < 1 \) is good),

\[
\phi_p = \frac{P}{P_b} = \frac{\Sigma(P_i(D_iX))}{\Sigma(P_i(D_i(1 + b_i)X_b))}
\]
Relative power-efficiency\textsuperscript{15} with bloat reduction ($> 1$ is good),
\[
\phi_e = \frac{X/P}{X_b/P_b} = \frac{(X/X_b)\sum(P_i(D_i(1 + b_i)X_b))}{\sum(P_i(D_iX))} = \frac{\phi_x}{\phi_p}
\]
(3.3)

Relative equiperformance power with bloat reduction (i.e. comparing power consumed by original and bloated software at the same throughput $X_b$) ($< 1$ is good)
\[
\phi_q = \frac{\sum(P_i(D_iX_b))}{\sum(P_i(D_i(1 + b_i)X_b))}
\]
(3.4)

### 3.4.1.1 Effect of degrees of energy proportionality

An energy proportional hardware resource consumes power in proportion to actual resource utilization (a desirable property of server systems). In the presence of certain power management schemes (e.g. DVFS), this relationship may be super-linear; we characterize this with an exponent which we call the degree of energy proportionality of the resource.

Let us model the power consumed by each resource as $P_i(U_i) = a_iU_i^{\alpha_i} + c_i$, where $\alpha_i =$ degree of energy proportionality of resource $R_i$.

Let $P_{\text{static}} = \Sigma(c_i)$, be the static power of the hardware system, $P_{\text{dyn}} = \Sigma(a_iU_i^{\alpha_i})$ be the total dynamic (load dependent) power consumption. ($\alpha_i$ would be zero if resource $R_i$ is non-energy proportional)

Let $L_i = a_iU_i^{\alpha_i}$ be the load-dependent power for resource $R_i$. Thus $P_{\text{dyn}} = \Sigma(L_i)$, $P = P_{\text{static}} + P_{\text{dyn}}$.

Relative peak power impact with bloat reduction:
\[
\phi_p = \frac{\sum(L_i(D_iX_b)\phi_x^{\alpha_i}) + P_{\text{static}}}{\sum(L_i(D_iX_b)(1 + b_i)^{\alpha_i}) + P_{\text{static}}}
\]

\textsuperscript{15}Notice that in the case of SPECpower_ssj2008, the analysis extends to the energy-efficiency score across load levels (and not just the peak load), as the load levels are scaled by a fixed percentage with respect to achievable peak performance.
Defining $f_i = \text{fraction of load dependent power consumed by resource } R_i \text{ (wrt total system power)}$ when running bloated software, $f_s = \text{fraction of static power consumed with the bloated software } (f_s + \sum f_i = 1)$, the above can be re-written as:

$$\phi_p = \sum (f_i (\frac{\phi_x}{1 + b_i})^{\alpha_i}) + f_s$$  (3.5)

As for the relative equiperformance power,

$$\phi_q = \sum (\frac{f_i}{(1 + b_i)^{\alpha_i}}) + f_s$$  (3.6)

### 3.4.2 System Bottlenecks and Bloat: A Curious Interaction

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Relative peak perf $\phi_x$</th>
<th>Relative power at peak perf $\phi_p$</th>
<th>Relative equi-perf power $\phi_q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloat at non-bottleneck resource</td>
<td>1 (same)</td>
<td>$\frac{f_k}{(1+b_k)^{\alpha_k}} + \sum_{i \neq k} f_i + f_s \leq 1$ (same if $\alpha_k = 0$, else decreases)</td>
<td>$\frac{f_k}{(1+b_k)^{\alpha_k}} + \sum_{i \neq k} f_i + f_s \leq 1$ (same if $\alpha_k = 0$, decreases otherwise)</td>
</tr>
<tr>
<td>Bloat at bottleneck resource</td>
<td>$1 + b_k$ (improves)</td>
<td>$f_k + \sum_{i \neq k} (f_i(1 + b_k)^{\alpha_i}) + f_s \geq 1$ (same if $\alpha_i = 0, \forall i \neq k$, else increases)</td>
<td>ditto as above</td>
</tr>
<tr>
<td>Bloat reduction shifts bottleneck</td>
<td>$1 + b_{eff}$ (improves, but less)</td>
<td>$f_k(\frac{1+b_{eff}}{1+b_k})^{\alpha_k} + \sum_{i \neq k} (f_i(1 + b_{eff})^{\alpha_i}) + f_s$ (can increase or decrease or stay same)</td>
<td>ditto as above</td>
</tr>
</tbody>
</table>

Table 3.3: Effect of bloat reduction in different scenarios when bloat affects a single resource $R_k$

Consider the situation where bloat primarily affects the demand for a single resource. Let $R_k$ be the bloated resource, then $b_k > 0$ and $b_i = 0, \forall i \neq k$. We show how the impact of
bloat reduction depends on where the primary bottleneck is relative to the bloated resource. Table 3.3 summarizes the impact on performance, peak power, and equiperformance power.

**Bloat at non-bottleneck resource** When bloat does not affect the bottleneck resource, \( \phi_x = 1 \), i.e. there is *no change in performance with bloat reduction*. Substituting in equation 3.5, we obtain:

\[
\phi_p = \frac{f_k}{(1 + b_k)^{\alpha_k}} + \sum_{i \neq k} f_i + f_s \leq 1
\]

Peak power *decreases* with bloat reduction, showing a higher improvement when the bloated resource has a steeper (larger \( \alpha_k \)) power-to-load characteristic and consumes a higher fraction of system power (larger \( f_k \)).

Since there is no change in performance \( \phi_e = 1/\phi_p \), i.e., the relative power-efficiency improves with bloat reduction. And \( \phi_q = \phi_p \), i.e., the relative equiperformance power is the same as the relative power at peak performance.

**Bloat at Bottleneck Resource** If bloat affects the bottleneck resource, i.e. \( k = \text{argmin}_i (1/D_i) \), then \( \phi_x = 1 + b_k > 1 \), i.e. *throughput improves with bloat reduction*, maximum improvement being \( 1 + b_k \) when bloat is eliminated. Substituting in equation 3.5

\[
\phi_p = f_k + \sum_{i \neq k} (f_i(1 + b_k)^{\alpha_i}) + f_s \geq 1
\]

Peak power *increases* or remains the same depending on the power characteristics of the *non-bloated resources*, in contrast with the previous case. Reducing bloat allows more productive use of the bottleneck resource, improving peak throughput. If this increase in throughput increases the usage of resources that were under-utilized earlier because of the bloat-affected bottleneck, and this increases their power consumption, then the power consumed by the application at peak throughput can increase.

\[
\phi_e = (1 + b_k)/\phi_p \leq 1 + b_k
\]
Relative power efficiency improvement is less than or equal to the throughput gain. A steeper energy proportionality characteristic of the other (non bloated) resources lowers the efficiency improvement from bloat reduction, especially if their power consumption is significant compared to the power consumption of the bloat-impacted bottleneck resource. The highest improvement from bloat reduction occurs in the case when the non-bottleneck resources are non-energy proportional (when $\alpha_i = 0, \forall i \neq k$).

**Bloat reduction shifts bottleneck**  
If reducing bloat causes the bottleneck to shift from $R_k$ to $R_l$, then, $1 < \phi_x < 1 + b_k$, i.e. throughput improves with bloat reduction (but to a lower extent than the previous case). Let us term $b_{eff} = \phi_x - 1$ as the effective bloat factor. Now the analysis for peak power and power efficiency are similar to the previous case, adjusting for $b_{eff}$.

Equiperformance power is impacted to the same extent in all the three cases above.

### 3.4.3 Summary

The extent to which bloat impacts power-performance may not always be obvious because bloat shifts the relative usage of different resources, some of which may have a non-linear power-vs-load variation. Performance and equi-performance power are generally improved with bloat reduction. However, power at peak performance can increase with bloat reduction as a result of increased throughput following reduced pressure at a bottleneck resource. The degree of increase depends on the energy proportionality characteristics of the resources which are not bloated. Reducing bloat can also cause a shift in bottleneck from the bloated resource to another resource. In general, bloat can involve multiple resources, so we need to use equations 3.5 and 3.6 to assess the combined impact of these conditions.
3.5 Corroborating the model with experimental observations

3.5.1 Experimental study in perspective

Our model provides a generalized analysis for relating the power-performance impact of bloat reduction to resource bottlenecks and system energy proportionality. It highlights two important factors that determine the extent to which bloat impacts power-performance: (a) the relative energy proportionality of the system resources and (b) the extent to which the resource’s usage is a bottleneck to performance.

Figure 3.6: Hardware energy proportionality differences: (a) The Power 750 system employing new DVFS algorithms shows an approximately cubic relationship for CPU power; memory power increases linearly but as a much smaller fraction of system power on the test configuration (which uses adequate memory for SPECpower_ssj2008, but far below the system’s memory capacity of 512GB). (b) Without DVFS the relationship for the Power 750 system is almost linear, similar for HS21. (c) The x3650 M2 system has a near quadratic characteristic for the full system power (we do not have CPU and memory power measurements in this case).

Figure 3.6 shows the power characteristics with load for the different systems used in our experimental study with SPECpower_ssj2008. The energy proportionality goes progressively higher from HS21 blade/Power750 system (3.6(b), near linear) with no adaptive power management to x3650 M2 with DVFS (3.6(c), quadratic) to the Power750 system with DVFS (3.6(a), cubic).

In our study, the memory hierarchy resources are the ones primarily affected by bloat and
serve as the bottleneck resource on different occasions, e.g., memory bandwidth and capacity being relatively limited for the HS21 (section 3.3.4), main memory being limited with reduced heap sizes (section 3.3.4.2), and on-chip cache being limited (section 3.3.4.3).

<table>
<thead>
<tr>
<th>System</th>
<th>Mem stress</th>
<th>Degree of energy proportionality</th>
<th>Peak Perf/ Eff. Score impact</th>
<th>Equi-perf power impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS21 (noDVFS)</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>x3650 M2 (DVFS)</td>
<td>Low</td>
<td>Med</td>
<td>Med</td>
<td>Med</td>
</tr>
<tr>
<td>Power750 (DVFS)</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 3.4: Trends from cross-platform experiments

Table 3.4 summarizes the trends seen in cross-platform experimental observations of the impact of bloat reduction (Section 3) through object memory reuse (i.e. from AllocOrig to AllocLess), in the context of memory constraint (bottleneck strain) and energy proportionality differences between platforms.

The higher the degree of energy proportionality (higher $\alpha_k$), the higher the equi-performance power savings from bloat reduction in Table 3.3. This is supported by the increasing impact on equi-performance power in Table 3.4 going from HS21 to Power750. On the other hand, the higher the extent of bottleneck, the higher the performance improvement with bloat reduction in Table 3.3, consequently the higher the increase in efficiency. This is supported by the higher impact on energy-efficiency for HS21 over x3650 M2 and Power 750.

### 3.5.2 Specific model predictions and corroborating experimental results

1. **Peak power** can increase or decrease with reduction in bloat depending on whether it affects the bottleneck resource or other resources. In the situation where bloat affects the
bottleneck, the degree of impact can depend on the steepness of energy proportionality characteristic of the resources which are not bloated and the fraction of system power consumed by them.

In our cache pressure experiments (Figure 3.5), bloat induced cache pressure causes under-utilization of the compute resources for SMT4 and HC, hence throughput increases with bloat reduction. Consequently peak power actually rises (negative peak power savings) with reduction in bloat. This effect is more pronounced for the steeper power characteristics with DVFS than for fixed frequency. On the other hand, reducing bloat also reduces memory references and consequently memory power (by 13% in Power750, SMT4 mode, as shown in Figure 3.3, and 18% in SMT2); the overall system power reduction due to this effect, however, is small as memory consumes a low fraction of system power on this system. In the SMT2 case, where the performance impact of bloat is lowest, this reduction translates into a slight reduction in overall system power.

2. The energy-efficiency improvement from reducing bloat is likely to be most pronounced when bloat affects a bottleneck resource and the non-bottleneck resources are not energy proportional (per Table 3.3). Energy proportional hardware mitigates the effect of bloat on energy efficiency at peak performance - Figure 3.5 confirms this, showing the gains in energy efficiency are greater when running the cores at fixed frequency than with DVFS. Similar behavior can be seen in Figure 3.2 with HS21 getting a higher improvement in energy-efficiency with bloat reduction (Efficiency of AllocLess/Efficiency of AllocOrig = 1.59) than x3650 M2 (1.1) or Power 750 (1.06).

3. While energy efficiency improvement at peak performance is higher with non-energy-proportional resources, the improvement at equal performance can be significantly higher for energy proportional hardware. This can be seen as significantly higher equi-performance power savings for DVFS in Figure 3.5 compared to the energy efficiency improvements at peak. The same figure also shows that having more energy proportional resources (DVFS) can yield significantly higher equi-performance energy savings (compared to fixed frequency).
3.6 Related Work

Analysis and measurement of software bloat Mitchell and Sevitsky initiated studies of runtime bloat with their analysis of data transformations [89] in framework based Java applications and data structure health signatures [87]. Subsequently, different measures such as the volume of temporary objects (which we investigate in our empirical study), data copies and heavy object creation costs have been used to recognize the presence of bloat [41, 138, 140]. A summary of the state of the art in research on software bloat analysis and solutions is covered in Chapter 2 and also in [139]. However, to the best our knowledge, we are the first to investigate the power-performance implications of bloat.

Java energy characterization Much prior work on energy (and power) characterization of the Java runtime and applications [123, 46, 67] have been simulator based studies or primarily conducted on embedded platforms. Contreras and Martonosi [38] used real system power measurements (like we do) on some mobile platforms to show that a significant proportion of energy is consumed by the JVM, particularly components like the classloader, JIT and garbage collector. Recently, Esmaeilzadeh et al. [42, 43] examined measured power and performance of Java benchmarks across hardware generations of Intel based processors and found Java multithreaded benchmarks to be more power-intensive than their single threaded counterparts: the differences may be explained by noting that complex Java workloads are likely to be memory bound and more sensitive to cache size than clock speed. Our work differs in its focus on the impact of a specific category of bloat on full system power-performance for a given workload on large server systems running a production JVM.

Object churn analysis, impact and solutions Several compiler and runtime optimizations have been developed to reduce the overheads of allocating and reclaiming temporary objects, including escape analysis [37, 28, 47, 130], escape detection and improvements in memory management and garbage collection [9]. In spite of this, it has been observed that even a sophisticated escape analysis implementation in a high performance production JVM typically eliminates less than 10% of allocations (Shankar et al. [112]) in component based applications
Chapter 3. Power-Performance Implications of Java Runtime Bloat

– this is consistent with our experience in this chapter. Techniques have also been proposed for ensuring faster reclamation of temporary objects [35, 57, 136] and for guiding programmers in eliminating excess temporaries [41, 31]. In our controlled experiments, we applied object reuse manually for de-bloating excess temporaries - the technique has subsequently been automated in Chapter 4 [19] for certain object types (container and string objects). Our work in this chapter complements such efforts by studying the actual systems level power-performance impact under different conditions, to help determine which optimization opportunities are most worthwhile. We believe that we are the first to perform such a study, particularly with real power measurements on large scale server systems with dynamic power management capabilities. As we find, in many situations, an improvement in peak performance need not translate to an equivalent improvement in equi-performance power savings and vice-versa.

Zhao et al. [147] analyzed the implications of object allocation on scalability and performance. We were able to reproduce similar results during our experimentation, particularly on the HS21 blade server which appears to be closest in configuration to their experimental setup. However, power-efficiency was not a consideration in their work.

3.7 Conclusions

In this chapter, we presented the first systematic empirical study analyzing the power-performance implications of software runtime bloat. With a methodology using the SPECpower_ssj2008 energy-efficiency benchmark for suitable injection/reduction of temporary object bloat and a detailed multi-platform analysis we establish the quantitative power-performance impact of bloat reduction in the context of system energy proportionality and resource bottlenecks. We then presented a simplified abstract model for relating the power-performance impact of bloat reduction to energy proportionality and resource bottlenecks for “what-if” analysis. We further corroborated our model with the findings from our in-depth experimental study. Our work provides the first detailed insight into the varying impact of software runtime bloat on the power-performance characteristics of systems. The findings show that lean does usually imply green, but to tell the shade, a whole system analysis like ours is necessary.
Acknowledgments for this Chapter

We thank Kazuaki Ishizaki, Gary Sevitsky, Vaidyanathan Srinivasan, Hong Hua, Dibyendu Das, Amar Devegowda, Ankita Goel, Derek Inglis for their help. We also thank Varsha Apte, Rupesh Nasre, Mathew Jacob and our anonymous reviewers for their valuable feedback.
Chapter 4

Automated Object Reuse Transformation for Mitigating Bloat

We describe a novel static analysis algorithm and an automated code transformation for object reuse to mitigate runtime bloat involving the repeated creation of temporary container and String objects in a loop.

4.1 Introduction

Our findings in the previous chapter (Chapter 3) show that reducing bloat can result in significant energy savings depending on the underlying system characteristics. In order to realize such savings without sacrificing development productivity, there is a need to explore automated techniques for de-bloating software. We now address this problem for a particular form of bloat: the creation (and deletion) of many temporary objects in Java programs (also known as temporary object churn, per Section 3.2), the energy efficiency effects of which we studied experimentally in Chapter 3.

1The work described in this chapter has appeared in the following publication:
Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath and Manish Gupta. Reuse Recycle to Debloat Software. ECOOP 2011
Chapter 4. Automated Object Reuse Transformation for Mitigating Bloat

As illustrated by Jack Shirazi in his book on Java Performance Tuning [114], and earlier discussed in Section 3.2, creating too many temporary objects results in higher garbage collection overhead, object construction costs and higher memory system stress resulting in an increase in processing time and memory consumption. At the end of the chapter on object creation in his book, Shirazi gives a long list of performance improvement strategies of which we reproduce a few here:

- Reduce the number of temporary objects being used, especially in loops.
- Avoid creating temporary objects within frequently called methods.
- Reuse objects where possible.
- Empty collection objects before reusing them. (Do not shrink them unless they are very large.)

However, this is easier said than done, especially for Java programmers who have grown up with the luxury of creating and discarding temporary objects, on the assumption that the discards would be efficiently garbage collected. Consider, for example, a typical piece of Java code as shown in Figure 4.1(a):

In this program, foo() calls doSomething() which loads several objects into a stack work. Then foo() picks up each element in the stack, checks for and discards any duplicates using seen and loads the unique objects into heap. At the end, based on some condition, heap is stored into either the field variable this.ftab or into the Hashtable tab\(^2\) passed in as a parameter\(^3\). The method bar() calls foo() iteratively and then dumps the contents of tab while the method driver() calls bar() iteratively.

Here we observe that foo() is called from inside a loop. Hence HashSet seen, Stack work and Vector heap will be created once for every iteration of the loop. Also, it is intuitively clear that seen and work can be reused, but heap may not be reusable.

---

\(^2\)For ease of exposition we model a hashtable as directly containing the key and value fields e.g., tab.value instead of containing the fields only indirectly e.g., tab.bucket[i].element[j].value.

\(^3\)This code was modified from Xylem code (refer Section 4.5), the only modification being the addition of ftab and the corresponding lines of code at lines 14 and 15 to highlight that an object may be reusable along one path but not another.
• Consider Stack work: it is created locally and is not accessible outside foo()—that is, it does not escape foo(). It may be reused as shown in Figure 4.1(b). Note, however, that the enclosing loop is in a different method than the objects being reused and thus requires interprocedural analysis. Nevertheless, work is reusable within the innermost enclosing loop and hence is termed a “Level 1” reusable object.

• Consider Vector heap: it is created locally but it is accessible outside foo()—that is, it escapes foo().

– Consider the case when it escapes via tab: heap does not escape the method

Figure 4.1: Sample code.
bar, but it does “escape” the loop inside bar. Going further back up the call flow graph, we find that bar is called from within a loop. Since heap does not escape from this loop, it is potentially reusable. In this case, heap is not reusable within the innermost enclosing loop, but it is reusable within the next enclosing loop and hence is termed a “Level 2” reusable object.

– When it escapes via ftab: ftab is accessible outside bar and driver and hence so is heap. Therefore, heap is not reusable along this path.

When we reused seen and work as shown in Figure 4.1(b), we observed a 9% reduction in execution time (on a dual core Intel(R) Core(TM)2 Duo system with 2GB RAM running Java Hotspot(TM) Server VM on Linux).

Thus we see that objects may be reused within the immediately enclosing loop or a higher level loop. The same object may be reusable along one path but not another. Similarly, the same object may be reusable at different levels along different paths. Besides these, there are many issues related to this kind of code transformation:

1. How do we determine automatically which variable can be reused and which one cannot be

2. Which data structures do we target and how do we know how to “clear” the structure before reuse

3. How do we determine when to perform the allocation and the “clear”. In the example, we have given a trivial solution which does not always work

4. Where do we insert the reuse code so that it does not become an overhead in itself

Although, in principle, it is possible to reuse any data structure, in our implementation, we address only certain Collection classes—specifically HashSet, Vector, Stack, TreeMap, LinkedList, ArrayList and PriorityQueue. This makes it easy to clear the objects using the clear() method from the Collection class. We also reuse memory in Strings (here we are referring to the reuse of the underlying arrays and not reuse of the string representation by string interning). This is far more complex and the details are given in Section 4.4.
Contributions In this chapter we give a novel algorithm for automatically finding sources of software bloat and then we give a solution to transform the code to reduce the bloat. The main contributions are:

- An algorithm that can detect objects created within a loop and determine whether an object created within a loop can be reused at the end of each iteration. In the case of nested loops, the algorithm will tell us the innermost enclosing loop in which the object can be reused.

- A solution that can automatically transform the source code to reuse the object such as to mitigate the effects of software bloat.

- An implementation that validates our claims and shows that we can get upto 40% reduction in bytes of temporary objects generated and 20% improvement in speed of execution.

Organization We start off with some definitions and a description of escape analysis used in this chapter in Section 4.2. In Section 4.3 we explain how to find safe reusable allocations and in Section 4.4 we give an algorithm that achieves the reuse through a source-to-source transformation. In Section 4.5 we report the empirical justification for using our analysis, Section 4.6 positions our work with respect to related work and we conclude in Section 4.7.

4.2 Preliminaries

The control-flow graph (CFG) for a method $M$ contains nodes that represent statements in $M$ and edges that represent potential flow of control among the statements.

We define here some terms used in the chapter.

Definition 4.2.1. **Dominator**: A node $S_i$ dominates a node $S_j$ iff $S_i \neq S_j$ and $S_i$ is on every path from $Entry$ to $S_j$.

Definition 4.2.2. **Postdominator**: A node $S_j$ postdominates a node $S_i$ iff $S_i \neq S_j$ and $S_j$ is on every path from $S_i$ to $Exit$. 
**Definition 4.2.3. Control dependence:** A node $S_j$ postdominates a branch of a predicate $S_i$ iff $S_j$ is the successor of $S_i$ in that branch or $S_j$ postdominates the successor of $S_i$ in that branch.

A node $S_j$ is control dependent on a predicate $S_i$ iff $S_j$ postdominates a branch of $S_i$ but $S_j$ does not postdominate $S_i$. (For example, in Fig 4.2, node 15 is control dependent on node 14.) A node can be directly control dependent on itself. Note that a node with only one successor can never be the source of a control dependence edge.

**Definition 4.2.4. Backedge:** A backedge in the CFG is an edge where the destination of the edge dominates the source of the edge. (For example, in Fig 4.2, the edge from node 19 to node 9 is a backedge as node 9 dominates node 19.)

**Definition 4.2.5. Data Dependence:** A node $S_j$ is data dependent on a node $S_i$, if $S_i$ defines some variable $x$, $S_j$ uses the variable $x$, and there exists a path from $S_i$ to $S_j$ without intervening definitions of $x$.

**Definition 4.2.6. Loop Carried Data Dependence:** A node $S_j$ has a loop carried data dependence on a node $S_i$, if $S_i$ defines some variable $x$, $S_j$ uses the variable $x$, and there exists a path from $S_i$ to $S_j$ without intervening definitions of $x$ and the path contains a backedge.

**Escape Analysis**  To locate reuse possibilities, we use escape analysis which is a method for determining the dynamic scope of pointers. After constructing the control-flow graph of each method, our solution uses flow- and context-sensitive pointer analysis and escape analysis. The escape analysis computes the escape-in and escape-out sets for each method.

- The *formal-in* set for a method $M$ contains the set of formal parameters. The implicit *this* parameter (in non-static methods) is also a formal-in.
- The *formal-out* set for a non-void method $M$ contains a single parameter $R$, the designated return value. The *formal-out* set is empty for a void method.

---

4A context-sensitive analysis propagates states along interprocedural paths that consist of valid call–return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it. A flow-sensitive analysis, on the other hand, takes into account the order of statements in a program.
Figure 4.2: Escape Analysis.

- The escape-in set for a method $M$ contains direct and indirect fields of the formal parameters of $M$ that are used, before possibly being defined, in $M$. These represent the upwards-exposed uses in $M$.

- The escape-out set for $M$ contains direct or indirect fields of the formal parameters of $M$ and the return value of $M$ that are defined in $M$.

- At each Call site $c$ that calls method $M$, the algorithm uses the escape-in and escape-out information, to compute the actual-in and actual-out sets, where

- we generate an actual-in for each formal-in and each escape-in and

- we generate an actual-out for each formal-out and escape-out in $M$. 

Chapter 4. Automated Object Reuse Transformation for Mitigating Bloat
The algorithm associates escape-in and formal-in sets with the *Entry* node of the CFG, and escape-out and formal-out sets with the *Exit* node of the CFG; likewise, the actual-in and actual-out sets are associated with call sites.

In the example shown in Figure 4.2, `num` (node 2) and `tab` (node 3) are formal-in parameters in the method `foo` as is the `this` parameter although it is not shown explicitly in the figure. `this.ftab` (node 4) is an escape-in parameter where `ftab` is a field of the formal-in parameter `this`. There is no formal-out parameter as both the functions are void functions, but `this.ftab.key` (node 20) and `this.ftab.value` (node 21) are escape-out parameters generated from the `put` method of the Hashtable. Similarly, `tab.key` (node 22) and `tab.value` (node 23) are escape-out parameters since they are fields of the formal-in `tab`.

In method `bar` at the call site for `foo` (node 37) we have generated actual-ins and actual-outs and mapped them appropriately to the formal-in and escape-out parameters in the called function.

### 4.3 Finding potential sources of bloat

In this section, we describe how to locate object allocation sites within loops that can be reused. Our analysis first identifies whether an allocation site is within a loop. This allocation site can be converted into a reuse site on the condition that

1. it does not have a loop carried dependence
2. it is not accessed outside the loop. To determine whether it is local to an enclosing loop, we need to check if it escapes the scope of the loop.

#### 4.3.1 The Problem with Loop Carried Data Dependence

Consider the following piece of Java code

```java
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
```
Here there is a loop carried dependence from vsucc to vprev. So if we reset and reuse vsucc inside the loop, then after the first iteration vsucc and vprev will always point to the same Vector, which is not correct.

Knowing that vsucc can be reused after $N$ cycles, it is possible to design reuse as follows:

```java
Vector tmp[] = new Vector()[N];
for (int j=0; j<N; j++) {
    tmp[j] = new Vector();
}
Vector vprev = new Vector();
int i=0;
while ( cond ) {
    tmp[i].clear();
    vsucc = tmp[i];
    i++;
    if ( i == N ) {
        i = 0;
    }
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
        vprev = vsucc;
    }
}
```

Finding loop carried dependence is relatively simple. However, it is not always possible to determine statically after how many cycles vsucc would be reusable, as in the example below.

```java
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() >= vprev.size() ) {
        vprev = vsucc;
        ...
    }
}
```

Hence, we conservatively ignore reuse when there is a loop carried data dependence.
4.3.2 The Basic Algorithm

Our basic analysis algorithm records the closest enclosing loop along an inter-procedural control dependence path where reuse may be implemented safely, if at all, for an allocation site. This involves a backward exploration along control dependence edges to find closest enclosing loops. A combination of forward slicing and escape analysis is used to determine whether accesses to the object remain within the scope of an enclosing loop. If not, the backward exploration continues looking for the next enclosing loop.

We start with preliminary analysis which consists of the following steps

- We compute flow and context sensitive data and control dependence, wherein the data dependence includes points-to and escape analysis described in the previous section. Each data dependence that is a loop carried dependence is flagged appropriately.

- We determine which conditionals in the Java bytecode are loop conditionals. We define a loop header as the destination of an edge in the CFG such that the node at the destination of the edge dominates the node at the source of the edge.

- We find all allocation sites in the code. For each allocation site \( S_{new} \) we compute the transitive closure of control dependencies. This process is performed interprocedurally. If \( S_{new} \) is not directly or transitively control dependent on a loop header, then we can discard it as being uninteresting from the point of reuse.

**Computing transitive closure of control dependencies** Intra-procedurally speaking, every node is eventually control dependent on the ENTRY node of the method. The ENTRY node is inter-procedurally control dependent on the CALL node from where the method is invoked. The transitive closure thus includes all nodes that the CALL node is control dependent upon.

In Figure 4.2, \( \text{HashSet } \text{seen} = \text{new HashSet()} \) is control dependent on the ENTRY node \( \text{void foo()} \). The ENTRY node is inter-procedurally control dependent on the CALL node \( \text{foo}(n, \text{tab}) \) in the method \( \text{bar()} \). The CALL node is control dependent on the for conditional \( n<\text{num} \). Hence, the allocation \( \text{seen} = \text{new HashSet()} \) is inter-procedurally and transitively control dependent on a loop header.
Removing Unnecessary Loop Header Dependencies  Note that a node that is control dependent on a loop header is not necessarily within the loop. However, we are only interested in finding allocations that are within a loop and hence need to perform additional computation.

All nodes within the loop are directly or transitively control dependent on the loop header. However there may be nodes outside the loop that are also control dependent on the loop header. This happens when there is a return from within the loop or when there is an exception flow edge from within the loop. Since we are interested only in nodes within a loop, we need to filter out these external-to-the-loop nodes. We do this by simply checking if there is a path from the node to the loop header that ends with a back edge.

Having found an allocation site that lies within a loop, we perform the algorithm given in Algorithm 1. The algorithm takes as input $S_{\text{new}}$, the allocation site $v = \text{new Collection}()$, where Collection is one of the classes mentioned in Section 4.1. It then computes the forward slice for $S_{\text{new}}$ as explained at lines 27–36. The forward slice consists of the transitive closure of all def-use sets starting with the definition at $S_{\text{new}}$. The nodes in the slice are separated into two bags, one called the “Escape” bag $\phi_{\text{esc}}$ that contains any formal-outs or escape-outs in the slice and the other bag $\phi_{\text{reg}}$ that contains all other nodes.

Next we track backward along the control dependence edges.

- If we come to a loop conditional we check if $\phi_{\text{esc}}$ is empty and every node in $\phi_{\text{reg}}$ lies within the loop and does not have a loop carried dependence. If yes, then we have found the closest enclosing loop inside which $S_{\text{new}}$ can be reused—along this particular path. We record this path and stop traversing the control flow graph any further for this path. For all other loop conditionals or branch conditionals, continue climbing up the control dependence graph.

- If we come to the Entry node of a method $S_M$, then for each invocation site, $S_{\text{call}}$, we map each node in the $\phi_{\text{esc}}$ set to the corresponding actual-out nodes. The old $\phi_{\text{esc}}$ and $\phi_{\text{reg}}$ sets are discarded and fresh sets are computed as the union of the forward slices of the actual-out nodes at the given invocation site. Then the analysis continues up the control dependence graph.
Algorithm 1 Locating reusable allocations within a loop

1: INPUT: $S_{new}$
2: $\phi_{esc} \leftarrow$ new Collection()
3: $\phi_{reg} \leftarrow$ new Collection()
4: computeForwardSlice($\{S_{new}\}$, $\phi_{esc}$, $\phi_{reg}$)
5: $N_{CD} \leftarrow$ controlDepPred($S_{new}$)
6: while $N_{CD} \neq$ null do
7:   if $N_{CD}$ is a loop header then
8:     level++
9:     if $\phi_{esc} = \{\}$ and contains($N_{CD}$, $\phi_{reg}$) and noLoopDD($\phi_{reg}$) then
10:        OUTPUT(level, $S_{new}$)
11:   end if
12:   else if $N_{CD}$ is an Entry node then
13:     for all $N_{invoke}$ a call site of Entry do
14:       newset $\leftarrow$ map($N_{invoke}$, $\phi_{esc}$)
15:       $\phi_{esc} \leftarrow$ new Collection()
16:       $\phi_{reg} \leftarrow$ new Collection()
17:       computeForwardSlice(newset, $\phi_{esc}$, $\phi_{reg}$)
18:     end for
19:   else
20:     reached the top of the call graph
21:     report and exit
22:   end if
23:   $N_{CD} \leftarrow$ controlDepPred($N_{CD}$)
24: end while
25:
26: computeForwardSlice(newset, $\phi_{esc}$, $\phi_{reg}$) {
27:   while !newset.empty() do
28:     $N \leftarrow$ newset.removeLast()
29:     for all $N_{dd}$ such that $N_{dd}$ is data dependent on $N$ do
30:       if $N_{dd}$ is a formal-out or an escape-out then
31:         $\phi_{esc} \leftarrow \phi_{esc} \cup N_{dd}$
32:       else
33:         $\phi_{reg} \leftarrow \phi_{reg} \cup N_{dd}$
34:       end if
35:     end for
36:   end while
37: }
• If we come to the top of the call flow graph, we conclude that $S_{new}$ may not be reusable along this path.

**An illustrative example** Consider the example in Figure 4.2.

1. We determine that the conditional nodes $n < \text{num}$ (node 34) in `bar` and `!work.isEmpty()` (node 9) in `foo` are loop headers.

2. Consider the statement `seen = new HashSet()` (node 5) in method `foo` in Figure 1. It is an allocation site for the Collection class `HashSet`. This node is control dependent only on the entry node `void foo` (node 1). This node is call dependent on the `invoke foo` node (node 37) which in turn is control dependent on the loop header $n < \text{num}$. Hence the allocation statement is interprocedurally called from inside a loop and has potential to be reused.

3. The forward slice is computed as $\phi_{reg} = \{ \text{seen.contains(w)}, \text{seen.add(w)} \}$ and $\phi_{esc}$ is empty as none of the nodes are formal-outs or escape-outs.

4. Now we traverse the control dependence path. At the entry node there is nothing to be mapped to the `invoke foo` site as $\phi_{esc}$ is empty. We discard the current $\phi$ sets and enter the method `bar` with empty sets. Next the `invoke foo` is control dependent on the loop header $n < \text{num}$. Here the requisite conditions are trivially true. Hence the allocation may be converted to reuse.

   If we consider the allocation site `heap = new Vector()`, it has four escape-outs in its $\phi_{esc}$; these map into actual-out nodes in the calling function `bar`. These actual-out statements are inside the loop but their forward slice contains nodes that are outside the scope of the loop. Two of these escape out of `bar` as well as its caller `Driver()`. Hence, this node is correctly not marked for reuse.

**4.3.3 Multiple Control Dependence Paths**

Since there may be multiple paths to an allocation site, several situations may arise:
1. The site is not reusable along any control dependence path.

2. The site is reusable along some control dependence paths, but not reusable along other control dependence paths.

3. The site is reusable along all its control dependence paths, but the closest enclosing loop where reuse can be implemented is not the same for all control dependence paths.

4. The site is reusable inside the same closest enclosing loop along all its control dependence paths.

One could take a conservative approach where only Case 4 is assumed to be safe for reuse conversion. However, this tends to miss several sites with potentially large churn (as we observe experimentally). A second approach is to introduce extra code to perform runtime tracking of the conditions for safe reuse in all situations. While this can enable more opportunities for reuse, it can become fairly complicated and invasive. For example, in the worst case, this might involve interprocedurally tracking path history along every branch leading to an allocation site from enclosing loops located several call levels away.

Instead, we use a simpler scheme that achieves greater precision than the conservative analysis but only exploits runtime state that needs to be introduced anyway for implementing object reuse.

Let us define the *height* $h$ of a loop $L$ along a control dependence path from an allocation site as the number of enclosing loop headers along that path upto and including $L$. Then, the reuse level $k$ for an allocation site along a particular control dependence path is defined as the height of the closest enclosing loop where the site is reusable for that path. This means that object reuse state for that allocation site must be maintained across iterations of all the inner loops upto height $k - 1$, and can only be reset across iterations of the loop at height $k$ or above. As long as this condition can be met across all control dependence paths for the site without conflict, the object can be safely converted for reuse along certain paths (where it is found to reusable) without affecting correctness along its other control dependence paths. This logic can be extended to address not just Case 3, but Case 2 as well, since a path that does not support reuse can be treated as a path with a very high reuse level. In other words, at
some outer loop level we can setup one control flow path to reuse and the another to not reuse, provided the two paths do not intersect within the same outer loop iteration.

Illustration: Consider the following variation of example in Figure 4.2, without lines 14-15, so that the allocation to heap no longer escapes directly via ftab. Now, suppose we add a couple of routines as follows:

```c
void barPersist(int num) {
    for (int n=0; n<num; n+=10) {
        foo(n, ftab);
    }
}

void driverPersist() {
    for (int num=0; num<100; num+=5) {
        barPersist(num);
    }
}

void mainDriver() {
    if (init()) {
        driverPersist()
    } else {
        driver();
    }
}
```

The allocation site heap = new Vector() in foo() is reusable at level $k = 2$, the loop in driver() that calls bar(), along one control dependence path, but is not reusable along another path that goes through driverPersist() and barPersist(). In this case, we notice that there is no conflict between these two cases as the corresponding loops do not intersect.

Now, let us say the routines driverPersist() and mainDriver() were removed and instead, the routine driver() modified as follows:

```c
void driver() {
    for (int num=0; num<100; num+=5) {
        if (init()) {
            barPersist(num);
        } else {
            bar(num);
        }
    }
```
This time, the outer loop is common to the two paths, which indicates a potential conflict. We perform an analysis of the loop sharing structure across control dependence paths to eliminate such potential conflicts.

Figure 4.3 illustrates some examples of loop header sharing across control dependence paths starting from two distinct nodes P and Q respectively. The loop header nodes are numbered according to the height of the loop with respect to the allocation site. In 4.3(a) the loops are embedded along both paths, hence there is a conflict if a reuse site in the inner loop is not reusable along either P or Q. In 4.3(b), the loops are disjoint and both inner loops invoke the method containing the reuse site. In this case, for reuse level $k = 2$ and above, the $k - 1$ loop of one path never falls inside the other. Hence if the site is reusable at level 2 (or higher) along paths from P, then, even if it is not reusable from Q, our transformation can be set up to safely exploit object reuse along the former. In 4.3(c), the innermost loop is shared, but the outer loops are disjoint. Thus a reuse transformation upto level 2 would be unsafe unless both
paths share the same reuse level. However, if the site is reusable at level 3 or higher along one path, then, even if it isn’t reusable along the other, our transformation can be set up safely to exploit object reuse at the appropriate level along that path.

4.4 Object reuse, recycle transformations

In the previous section we described an approach for finding allocation sites that are candidates for object reuse and the closest enclosing loops where they can be safely reused. Now we discuss our automated code transformations for implementing object reuse.

Object reuse optimizations may involve object memory reuse or object content reuse. The former recycles the memory and structural representation state of objects of the same type, instead of allocating fresh objects each time. The latter reuses at least some part of the actual object content (a form of memoization/caching) to save repeated content construction costs\(^5\). Our static analysis based detection technique mainly identifies the first kind of opportunities, hence this forms the focus of our implementation. Our code transformations can, however, be used to support the second category of reuse as well, with slight modifications\(^6\).

4.4.1 Basic reuse-recycle algorithm

The static analysis phase reports the safe reuse level and the corresponding loops for each allocation site identified for reuse (hereafter referred to briefly as a reuse site). We use this information to implement a basic reuse/recycle algorithm for these reuse sites.

The simple reuse transformation illustrated in the introduction is efficient but does not work in many situations. The allocation has been moved to a static initializer where constructor parameters that are specified at the allocation site may not be available. The conversion is only applicable for level 1 reuse, i.e. for objects allocated in an inner loop which can be recycled at the next iteration of this loop. In this case, it suffices to allocate a single reusable slot for object canonicalization is an extreme example of content reuse; object pooling involves memory and sometimes partial content reuse
\(^6\) e.g. steps such as clearing the object or simulating effects of a constructor may be skipped when reusing object content, thus simplifying the implementation
an allocation site, e.g. the variable REUSE.st_01 for Stack work. However, when allocated objects need to be preserved across iterations of an inner loop, and can only be recycled at a subsequent iteration of an outer loop, multiple reusable slots must be maintained for the same allocation site. This happens for a level $k$ reuse with $k > 1$, (where $k$ is the height of the closest enclosing loop at which the object can be reused), e.g. $k = 2$ for Vector heap when foo() is invoked via bar(). In this case, static initialization cannot be used as the number of distinct slots required (minimum outstanding allocations) may not be known until the inner loop completes its first iteration sequence. It could even change dynamically. The number of inner loop iterations and hence reuse slots maintained for Vector heap varies with the loop upper bound $num$, e.g. when $num = 50$, 10 reuse slots are used. We note that this also means that the number of reuse slots created must be bounded to avoid causing memory overhead due to a blowup in the number of inner loop iterations.

Therefore in our generalized implementation (Algorithm 2), the allocation statement is not moved, but instead, tracked during the first iteration of the level $k$ loop by creating reuse slots as needed and initializing them with the result of the allocations in inner loop iterations (lines 9-12). Note that Reusevar.numslots is statically initialized to zero. It is incremented (line 10) each time a reuse slot is created for this site, using Reusevar.addslot() (line 11). The objects from these slots are then reused sequentially (lines 16-17) during subsequent iterations of the level $k$ loop. If the inner loop iterations exceed the number of available reuse slots Reusevar.numslots (e.g. due to a varying loop bound), then additional slots are created as required (upto a maximum allowed capacity) (lines 9-12). If the maximum capacity of reuse slots is exceeded for a given allocation site, then allocations required beyond the capacity simply fall back to a non-reusable mode (lines 13-14).

This approach has the downside of an extra check in every inner loop iteration to distinguish the first iteration\footnote{and checks for dynamic expansion of slots} of the level $k$ loop from iterations which reuse previous allocations. The overhead may be optimized using loop peeling and specialization for common scenarios (such as level 1 reuse for collection objects which do not require a constructor parameter).

To enable an existing object to be recycled instead of issuing a fresh allocation, some type
specific steps need to be executed to re-initialize the object for reuse. In general, this may require simulating (a part of) its constructor functionality. We focus on reusing collection objects and strings.

4.4.2 Reusing collections

Preparing a collection object for reuse is particularly simple. Most collections provide a clear() method to reset a collection to zero entries while keeping the capacity of the collection intact. The larger the collection being reused, the greater the benefit as it saves a large portion of object construction costs.
4.4.3 Reusing strings

Recycling String objects requires simulating a part of its constructor functionality to re-populate the underlying character array with new content. Since a String object is an immutable data structure, this can be implemented efficiently only with special extension support from the class library or the JVM. For our experimental evaluation, we use reflection to access/clear/overwrite the array as required. This incurs a performance penalty, which is mitigated to some extent by caching the reflection results when the object is first allocated to avoid the overhead on every iteration. Therefore our results provide a conservative estimate of performance gain that can be attained through object reuse in this case.

4.4.4 Implementation Details

We used a source to source transformation approach to evaluate the feasibility of our automated object reuse/recycle conversion. Simplicity and clarity were our primary motivation for choosing this approach, e.g. ability to perform a visual inspection of the changes in source code after transformation. The conversion may also be implemented using byte-code manipulation and JVM level optimizations as discussed later.

The inputs required for the transformation are the output of the static analysis stage, and the source files of the application to convert.

Figure 4.4 illustrates our running example before and after automatic reuse conversion. The transformations are performed in a single-pass over the source. This is a slightly modified version of our running example from previous sections, without the ftab field. As heap no longer escapes via ftab, it is now reusable at level 2 (i.e. the loop starting at lineno 38 in driver()).

At each listed allocation site to convert for reuse (lines 14, 15 and 16), we replace the call to new with a call to an allocation site specific reuse method which performs allocation tracking and reuse. At the statement preceding a listed allocation site’s level \(k - 1\) loop header, we insert code to reset the reuse slot index for the allocation site and specify the maximum slots that may be created. Line 31 is the level 1 loop header corresponding to the level 2 reusable
class Klass {
    void foo(int num, Hashtable tab) {
        HashSet seen = new HashSet();
        Stack work = new Stack();
        Vector heap = new Vector();
        doSomething(work, num);
        while ( !work.isEmpty() ) {
            Object w = work.pop();
            if ( seen.contains(w) ) continue;
            seen.add(w);
            heap.add(w);
        }
        Integer inum = new Integer(num);
        tab.put(inum, heap);
    }
    void bar(int num) {
        Hashtable tab = new Hashtable();
        for ( int n=0; n<num; n+=10 ) {
            foo(n, tab);
        }
        dumpTabContent(tab);
    }
    void driver() {
        for ( int num=100; num > 0; num-=5 ) {
            bar(num);
        }
    }
}

(a) Before transformation

public class REUSE{
    static HashSet HashSet_14;
    public static HashSet ReuseHashSet_14() {
        if (HashSet_14 != null) {
            REUSEUtil.clearHashSet(HashSet_14);
        } else {
            HashSet_14 = new HashSet();
        }
        return HashSet_14;
    }
    static Stack Stack_15;
    public static Stack ReuseStack_15() {
        if (Stack_15 != null) {
            REUSEUtil.clearStack(Stack_15);
        } else {
            Stack_15 = new Stack();
        }
        return Stack_15;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }
    static int idxVector_16;
    static ArrayList<Vector> Slot_Vector_16 =
            new ArrayList<Vector>();
    static Vector Vector_16;
    public static Vector ReuseVector_16() {
        if (idxVector_16 < Slot_Vector_16.size()) {
            Vector_16 =
                    Slot_Vector_16.get(idxVector_16++);
            REUSEUtil.clearVector(Vector_16);
        } else {
            Vector_16 = new Vector();
            if (idxVector_16 < maxVector_16) {
                Slot_Vector_16.add(Vector_16);
                idxVector_16++;
            }
        }
        return Vector_16;
    }

(b) After transformation

Figure 4.4: Code transformation example
Chapter 4. Automated Object Reuse Transformation for Mitigating Bloat

A reuse context area and allocation site specific reuse methods are generated by the transformation. The reuse context fields maintain state corresponding to every allocation site that is converted for reuse. Stack maintains a reference to the level 1 reusable allocation for work. The reuse methods encapsulate allocation tracking and reuse logic specific to these allocation sites, e.g. ReuseVector uses SlotVector to keep track of the reuse slots for the allocation of heap at line 16. The size of the arraylist SlotVector thus corresponds to Reusevar.numslots in Algorithm 2. For level 1 reuse, as in the case of seen and work, there is a single reuse slot which is accessed directly from HashSet and Stack respectively. Before returning the reusable reference, these methods invoke a type-specific utility method to enable reuse for that object (clearHashSet() for seen, clearStack for work and clearVector for heap).

Reuse context entries are typically stored in a thread local reuse context area. For single-threaded programs like the above example, we maintain a global reuse context, to avoid the overhead of thread local context accesses in the interprocedural case.

4.4.5 Dynamic analysis guided filtering of candidate reuse sites

A purely static analysis based detection scheme has insufficient information to prioritize which allocation sites to convert based on an estimate of expected savings. We complement it with a dynamic analysis that profiles allocation sites with high object churn to guide the selection of statically identified candidate reuse sites that are worth converting. We then apply our object reuse transformation for those reuse sites.

The dynamic analysis phase takes as input the reuse sites reported by static analysis and the output of an allocation profiler that captures the volume of allocated and live vs freed bytes generated at each allocation site under a typical run of the program. It then generates statistics about the proportion of churn generated by reuse sites which use collections or

---

8i.e. bytes that are garbage collected
9%churn attributed to an allocation site = (Collected bytes which were allocated by the allocation site/Total collected bytes)*100
strings, and selects the top sites with significant contribution to overall volume of temporary objects generated.

4.4.6 Discussion

4.4.6.1 Alternatives to full source to source transformation

Instead of using a pure source to source transformation approach, object reuse transformations could also be implemented using byte code manipulation and JVM level optimizations. A JVM can avoid costs of reflection and thread local accesses that we incur and optimize the overhead of the check required in each iteration to distinguish first time allocation and reuse iterations. It can also enable profile guided object reuse conversion to be applied at runtime for the reusable allocation sites that exhibit the potential for highest savings.

4.4.6.2 Extending the technique to non-collection objects

The technique may be generalized further to any object type that is designed to support a special reuse interface with a type specific reuse method. This method provides an alternative to the constructor that is called to clear a previous instance of the object or re-populate it with new content. Such an approach can also be used to enable partial content reuse by implementing the reuse method to selectively preserve the content of some fields of the object.

4.5 Empirical evaluation

We apply our analysis to a few large applications, the SPECjbb2005 benchmark and the DaCapo benchmarks [27] lusearch, ps, pmd, antlr. We also apply it to Xylem [92], a proprietary tool that has been built to statically detect null dereferences in Java. In this paper we analyze only a subset of Xylem. Table 4.1 lists a brief description of the benchmarks used. The freed memory was measured from the garbage collection (GC) logs saved during execution of the applications and represents the total bytes freed over all GC cycles.

We apply the static analysis to all these applications, and use our dynamic analysis to
Table 4.1: Benchmarks analysed and freed memory for each benchmark

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Freed Memory (Object Churn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjbb2005</td>
<td>Server-side Java Benchmark</td>
<td>8 KB/txn</td>
</tr>
<tr>
<td>xylem</td>
<td>Proprietary tool to detect null references</td>
<td>1203 MB</td>
</tr>
<tr>
<td>DaCapo lusearch</td>
<td>An text search tool</td>
<td>4913 MB</td>
</tr>
<tr>
<td>DaCapo pmd</td>
<td>A source code analyzer for Java</td>
<td>1178 MB</td>
</tr>
<tr>
<td>DaCapo ps</td>
<td>A postscript interpreter</td>
<td>2366 MB</td>
</tr>
<tr>
<td>DaCapo antlr</td>
<td>A parser generator and translator generator</td>
<td>884 MB</td>
</tr>
</tbody>
</table>

select the applications and candidate reuse sites to convert from the safe reuse sites found. As shown in Table 4.3, SPECjbb2005, xylem and lusearch indicate the greatest potential for savings from object reuse for collections (including Strings and arrays). Hence we apply our automatic transformation to these applications and report the results in Table 4.4.

The following section presents our experimental results and analysis.

4.5.1 Reuse site detection statistics (static analysis)

Table 4.2 summarizes the results from the static analysis phase that finds safe reuse sites and the closest enclosing loops where objects may be reused. We notice that most opportunities exist at level 1 or level 2 reuse, and that a significant number of sites are only reusable along some paths and not others. Less than half of the safe reuse sites found are reusable at a single level along all paths. Except for ps, most benchmarks have a significant number of collection or string reuse sites.

4.5.1.1 Discussion: Analysis Time and Scalability

Table 4.2 also reports the times for analysis, and how much of that is spent on the preliminary analysis. We rely on an underlying context-sensitive flow analysis. This is, in general, slow, however, with suitable engineering, it can be reasonably scalable. Our analysis is built on top
Chapter 4. Automated Object Reuse Transformation for Mitigating Bloat

<table>
<thead>
<tr>
<th></th>
<th>SPECjbb</th>
<th>xylem</th>
<th>lusearch</th>
<th>pmd</th>
<th>ps</th>
<th>antlr</th>
</tr>
</thead>
<tbody>
<tr>
<td>functions</td>
<td>864</td>
<td>1679</td>
<td>2614</td>
<td>5587</td>
<td>1022</td>
<td>2486</td>
</tr>
<tr>
<td>statements</td>
<td>864</td>
<td>33924</td>
<td>69688</td>
<td>126312</td>
<td>19078</td>
<td>100359</td>
</tr>
<tr>
<td>total time</td>
<td>23s</td>
<td>33s</td>
<td>55s</td>
<td>8m 15s</td>
<td>20s</td>
<td>3m 16s</td>
</tr>
<tr>
<td>prelim time</td>
<td>18s</td>
<td>25s</td>
<td>41s</td>
<td>3m 41s</td>
<td>16s</td>
<td>2m 29s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Results</th>
<th>SPECjbb</th>
<th>xylem</th>
<th>lusearch</th>
<th>pmd</th>
<th>ps</th>
<th>antlr</th>
</tr>
</thead>
<tbody>
<tr>
<td>no. alloc sites</td>
<td>1014</td>
<td>1853</td>
<td>1549</td>
<td>2252</td>
<td>1013</td>
<td>2712</td>
</tr>
<tr>
<td>no. alloc sites in loops</td>
<td>784</td>
<td>1456</td>
<td>776</td>
<td>1076</td>
<td>146</td>
<td>1852</td>
</tr>
<tr>
<td>no. of (safe) reuse sites found</td>
<td>251</td>
<td>400</td>
<td>688</td>
<td>577</td>
<td>77</td>
<td>375</td>
</tr>
<tr>
<td>no. collection reuse sites</td>
<td>84</td>
<td>220</td>
<td>266</td>
<td>125</td>
<td>12</td>
<td>97</td>
</tr>
<tr>
<td>no. string reuse sites</td>
<td>27</td>
<td>0</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>51</td>
</tr>
<tr>
<td>no. of sites reusable only along some paths</td>
<td>273</td>
<td>148</td>
<td>657</td>
<td>507</td>
<td>67</td>
<td>401</td>
</tr>
<tr>
<td>pure level 1 reuse sites</td>
<td>90</td>
<td>274</td>
<td>197</td>
<td>166</td>
<td>28</td>
<td>79</td>
</tr>
<tr>
<td>pure level 2 reuse sites</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>min level 1 reuse</td>
<td>280</td>
<td>416</td>
<td>766</td>
<td>640</td>
<td>93</td>
<td>477</td>
</tr>
<tr>
<td>min level 2 reuse</td>
<td>15</td>
<td>6</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Reuse site detection statistics

of a basic slicer which has been previously run on programs that are larger than 450,000 lines of code and the preliminary analysis took less than 10 minutes as reported in [92].

The additional analysis that we apply does an all-path exploration to determine the reusability of an object. This is clearly an exponential algorithm. pmd, at a little over 126K bytecode instructions analyzed, took 8 minutes and 15 seconds to analyze, of which the preliminary analysis took 3 minutes and 41 seconds. Here again, we use standard engineering tactics to contain the exponential state space exploration. If, for a given object, the analysis takes too long (currently curtailed at 30 seconds), we abort analysis of the object and conservatively mark it as not reusable.
4.5.2 Reuse site object churn statistics (dynamic analysis)

<table>
<thead>
<tr>
<th></th>
<th>SPECjbb</th>
<th>xylem</th>
<th>lusearch</th>
<th>pmd</th>
<th>ps</th>
<th>antlr</th>
</tr>
</thead>
<tbody>
<tr>
<td>%churn at (safe) reuse sites</td>
<td>54%</td>
<td>18.4%</td>
<td>77.5%</td>
<td>16.4%</td>
<td>6%</td>
<td>14.6%</td>
</tr>
<tr>
<td>%churn at collection</td>
<td>46.6%</td>
<td>16.5%</td>
<td>63%</td>
<td>5%</td>
<td>3.7%</td>
<td>4.25%</td>
</tr>
<tr>
<td>(and string) reuse sites</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%churn at sites reusable only</td>
<td>6.8%</td>
<td>3.28%</td>
<td>77.2%</td>
<td>15.9%</td>
<td>6%</td>
<td>14%</td>
</tr>
<tr>
<td>along some paths</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no of reuse sites with</td>
<td>7</td>
<td>3</td>
<td>22</td>
<td>9</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>more than 1% churn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no of collection reuse sites</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with more than 1% churn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%churn at top 3 reuse sites</td>
<td>48%</td>
<td>16%</td>
<td>46%</td>
<td>10%</td>
<td>5.8%</td>
<td>5.8%</td>
</tr>
<tr>
<td>Distribution of reuse levels</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%churn at level 1 reuse sites</td>
<td>81%</td>
<td>99.98%</td>
<td>36%</td>
<td>87.2%</td>
<td>50%</td>
<td>84%</td>
</tr>
<tr>
<td>%churn at level 2 reuse sites</td>
<td>18%</td>
<td>0.02%</td>
<td>64%</td>
<td>12.4%</td>
<td>50%</td>
<td>16%</td>
</tr>
<tr>
<td>Distribution of reuse levels</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for collections</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%churn at level 1 reuse sites</td>
<td>89%</td>
<td>99.98%</td>
<td>25%</td>
<td>42.8%</td>
<td>100%</td>
<td>99.3%</td>
</tr>
<tr>
<td>%churn at level 2 reuse sites</td>
<td>11%</td>
<td>0.02%</td>
<td>75%</td>
<td>57.2%</td>
<td>0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

Table 4.3: Reuse site object churn statistics

Table 4.3 captures some of the statistics gathered during the dynamic analysis phase based on simple allocation profiling to identify reuse sites that generate more temporary objects.

We observe that in many cases, a few potentially reusable sites cause a perceptible amount of object churn, particularly in SPECjbb2005, lusearch and xylem. The results also reflect the importance of being able to handle reuse sites which are safely reusable along some paths but not others. In some benchmarks, e.g. lusearch, the sites that are the top contributors to
temporary objects bloat are of this nature.

### 4.5.3 Performance impact statistics

Object reuse conversion was applied only to the reuse sites that are indicated by dynamic analysis to have a major contribution to object churn. The performance comparisons between the original and converted application are presented in Table 4.4.

In general, the performance impact of reducing object allocations depends on the workload, choice of JVM used and both JVM and system parameters (as we have already seen in Chapter 3). For example, the JVM heap size, the garbage collection algorithm, system memory bandwidth characteristics (esp. on multi-core systems [147]) and workload specific tuning [23] can affect results of comparisons. However, in our evaluation for this chapter, we focus on the effectiveness of our technique rather than characterization of the degree of performance improvement expected from reducing object churn under different conditions. Hence we directly use default configurations instead of explicitly varying/tuning JVM and system parameters.

**System Configuration**  Our performance measurements were taken on a dual core Intel(R) Core(TM)2 Duo T7500, 2.2 GHz with 2GB RAM running Linux, Java HotSpot(TM) Server VM (build 14.3-b01, mixed mode). For the SPECjbb2005 measurements, we used an 8-core Intel server (Intel(R) Xeon(R) X5460, 3.16 GHz) with 16GB RAM, running Linux, Java HotSpot(TM) Server VM (build 1.6.0-b105, mixed mode).

**JVM settings**  For Xylem, we used a heap size of 1.6GB. We used out-of-the-box configuration parameters for the other benchmarks. In the case of SPECjbb2005, the heap size specified in the default benchmark properties file was 256MB. For the DaCapo benchmarks, the default heap size was as determined by the JVM. In all cases, the default garbage collection policy was determined by the specified JVM.

Since the execution time impact of reducing object creation can be highly dependent on the JVM and system parameters, we also measure other metrics such as the percentage reduction in bytes of temporary objects used (as estimated from garbage collection statistics) and relative
Chapter 4. Automated Object Reuse Transformation for Mitigating Bloat

<table>
<thead>
<tr>
<th></th>
<th>SMALL INPUT SIZE</th>
<th>LARGE INPUT SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPECjbb</td>
<td>xylem</td>
</tr>
<tr>
<td>No. of objects reused</td>
<td>24/txn</td>
<td>144851</td>
</tr>
<tr>
<td>No. of element allocations reused</td>
<td>1920/txn</td>
<td>-</td>
</tr>
<tr>
<td>% Reduction in temporary objects generated</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>% improvement in execution time or throughput</td>
<td>7.9</td>
<td>16.4</td>
</tr>
</tbody>
</table>

Table 4.4: Performance impact statistics: The percentages are baselined against the corresponding results for the original benchmark without object reuse conversion; for example 100% reduction in temporary objects generated would mean that all object allocations were eliminated, 100% improvement in throughput would mean that throughput doubled.

scaling with larger input sizes. This enables us to evaluate whether our transformation is efficient enough to exploit the potential for performance gains where opportunities exist.

We observe 20-40% reduction in object churn with our transformation. The execution time improvements range between 6-20%. Recall that our results in Chapter 3 showed how the power-performance implications of reducing temporary object bloat can vary widely with underlying hardware and software characteristics and that the energy savings can be significant under certain configurations. We also note that execution time improvements do not uniformly reflect the percentage reduction in objects. As observed by previous researchers [112, 147, 51] the relationship between percentage reduction in objects and performance is complex and depends on many factors ranging from workload and program characteristics, object construction costs, JVM tuning and hardware/system characteristics.

In SPECjbb2005, a single heavy allocation site dominates the reuse counts. Despite the fact that this is a string object and there are overheads due to reflection and accessing thread local context, we see significant benefits from object reuse automation. These improvements
appear to be consistent with those reported for a manual implementation of object reuse by researchers of [147]; their results were for a well-tuned setup (large heap, GC tuning).

4.6 Related work

Object churn analysis, impact and solutions Many compiler and runtime optimizations, such as escape analysis [37, 28, 47, 130], escape detection and improvements in memory management and garbage collection techniques [9] have been developed to reduce the overheads of allocating and reclaiming temporary objects. As part of their work on escape analysis, Blanchet [28] considered the problem of stack size limitations in using stack allocation for loop objects and implemented a simple liveness check to enable reuse of stack allocated space in loops. Their solution however does not consider higher levels of reuse in nested loops. They also rely on the use of inlining in case the loop header and allocation site are not within the same method, which is not practical in framework based applications where the allocation may lie several levels deep in the call chain from the closest enclosing loop.

Shankar et al. [112] found that even a sophisticated escape analysis implementation in a high performance production JVM typically eliminates less than 10% of allocations in component based applications. They experimented with the use of aggressive guided inlining of regions with high object churn to enable the JIT to detect more opportunities for stack allocation of objects. In contrast to their approach we use static analysis approach to perform source code transformations for object reuse, which enables us to detect additional opportunities without incurring a runtime overhead.

Performance understanding techniques have been proposed [41, 31] for guiding programmers in eliminating excess temporaries that cannot be automatically detected by runtime optimizers. For example Buytaert et al. [31] identify locations where code refactoring can be applied to reduce object creations. While their goal is similar to ours, they do not propose

---

10 [147] also reports results of experiments conducted across a whole range of JVM settings (heap size, GC policies) to show that performance degradation from excessive object allocation in this case is not a mere artifact of the GC algorithm or JVM parameters.
automated transformations. Their detection scheme uses dynamic traces unlike our static analysis approach where the dynamic analysis phase is only used to estimate potential benefits from the conversion.

Other approaches that help reduce the impact of excess temporary objects include advancements in memory management techniques for ensuring faster reclamation or reuse of temporary objects, e.g. taking better advantage of allocation phases in the application [136], or combining the benefits of explicit object release [35, 57, 51] with garbage collection or scoped batch reclamation. Our technique is complementary to these efforts as it avoids the creation of objects wherever possible.

Zhao et al. [147] analyzed the implications of object allocation on scalability and performance. They proposed the notion of an allocation wall that limits multi-core scalability programs that perform high volumes of temporary objects. For their experimentation they perform manual code modifications to implement a form of object pooling for objects that are allocated very frequently and showed significant benefits for SPECjbb and SPECjvm derby. In their paper they observe that the process of manually converting an application for object reuse is time consuming and hence impractical for application developers to use. Our work succeeds in efficiently automating such optimizations for collection objects and strings.

Analysis and measurement of software bloat Recall the available bloat analysis techniques that we reviewed earlier in Chapter 2. Mitchell, Sevitsky and Srinivasan [89] defined metrics based on modeling runtime information flow to classify and characterize the nature and volume of data transformations executed, though these measures have not been automated till date. The notion of data structure health signatures proposed by Mitchell and Sevitsky [87] has been used very effectively in characterization and automated measurement [85] of Java memory bloat in long lived heap objects. This is a relative measure of total memory bytes consumed by actual data vs associated representational memory overhead. For some categories of bloat, including the problem of temporary objects bloat which we address in this chapter, an explicit model may not always be available for distinguishing overhead from necessary data or activity. Researchers have therefore used different measures of excesses to recognize the presence of
bloat. For example, Xu et al. use an instrumented JVM to summarize chains of runtime data copies [138] and an abstract thin dynamic slicing technique to identify data structures with high cost-benefit ratios [140]. Most approaches for detecting bloat have employed dynamic analysis. [141] applies a static analysis scheme to detect inefficient uses of container objects, particularly for underpopulated and overpopulated containers. All of these techniques are focused on aiding the process of reducing bloat, however, unlike the solution we described in this chapter, they are intended for interpretation by experts, not as fully automated solutions to de-bloat software.

Dufour et al. [41] apply blended static and dynamic analysis techniques to runtime traces for characterizing the usage of temporaries. Their results show that a significant number of temporary objects may be used several call levels away from their allocation site, which makes them particularly difficult to optimize. This motivates the need for techniques like ours.

4.7 Conclusion and future work

We presented an analysis technique to automatically detect and convert opportunities for object reuse in Java programs where there is significant potential for benefit from reuse. This is a challenging problem because an object may be reusable in loops that may be several levels above it in the callgraph. Further, as our empirical results show, very often objects may be reusable only along certain paths and not others. In this situation a conservative analysis can miss most opportunities for reuse. We are able to improve precision in such situations by checking whether the conditions required for the correctness of our runtime transformation are met in the event these paths share the same loop header. Our results show that this solution can detect such opportunities in real large programs and reduce the generation of temporary objects significantly.

Further improvements in scalability and precision of our solution can be attained by incorporating feedback from our dynamic analysis to focus static analysis on the allocations sites that are likely to yield most benefits. Other future work includes extending the applicability of our automated transformation to other types of objects and using a combination of byte code
manipulation and JVM level optimizations to improve the performance of the transformed code.

**Acknowledgments for this Chapter**

We thank Gary Sevitsky, Matt Arnold, Kazuaki Ishigaki, Dibyendu Das, Vijay Mann and Prasanna Kalle for their help, particularly their contribution to discussions on the problem of Java temporary objects bloat. We also thank Rupesh Nasre, and our anonymous reviewers for their excellent feedback.
Chapter 5

CAPA: Concern Augmented Program Analysis for Bloat Detection

We show how excess concerns contribute to execution bloat (only) through statements involved in structural interactions with exploited concerns; such bloat can be detected using a novel approach to augment program analysis with external information about program concerns.

5.1 Introduction

In the previous chapter, we focused on temporary object generation in loops, a form of bloat which can be detected and even mitigated automatically without any knowledge of why it was generated\(^1\). We now turn our attention closer to the origin of bloat by exploring sources of bloat that are more difficult to detect.

Even though an abundance of redeployable components and frameworks has greatly eased software development, automatically reasoning about the properties of such software can be challenging. The difficulty arises due to a large amount of incidental context accrued across framework layers\(^2\), which can obscure underlying functional intent. As a result, it is non-trivial

---

\(^1\) for example, to implement transformations needed for framework based reuse

\(^2\) Such context resides not only inside the source code of all the components deployed by the application but
to efficiently surface key insights and details relevant for a given bloat analysis task.

For example, existing approaches for bloat analysis covered in Section 2.5 lack information about what was meant to be accomplished by the code statements responsible for the pattern of excess activity seen – insight that is required to determine why the excesses occurred and whether it can be de-bloated. Instead, the results for each site highlighted by these tools must be interpreted individually by manually looking at the source code in order to arrive at a conclusion about its purpose\(^3\). Consider also the example in Fig 4.1, Chapter 4. We observed that \texttt{Vector heap} is not reusable as it escapes via \texttt{ftab} along the path where \texttt{init()} returns \texttt{true}. If we knew that the only reason \texttt{init()} and \texttt{ftab} were introduced is to support auditing, which is not actually required in a given application context where this class is deployed, then \texttt{heap} could also have been optimized for reuse to achieve further savings in temporary object allocations.

Lack of higher level insight about underlying functional intent not only affects the quality or efficiency of existing static and dynamic analyses of bloat, but also limits the kinds of bloat analysis problems that can be addressed using these techniques alone. In this chapter, we tackle one such problem – \textit{the analysis of execution bloat induced by the presence of excess features (Definition III from Chapter 2, Section 2.1.3)} – this type of bloat is a typical consequence of the high level of generality supported by framework based components. We use the specific term “execution bloat” in this chapter to emphasize instruction execution overheads (i.e. excess work) in addition to (and including) runtime data overheads due to these excess features. For example, of the 58 transformations executed in converting a date field in a SOAP message to a Java business object as depicted earlier in Fig 2.4, 18 transformations result from an unnecessary feature of decimal number specific processing being applied to whole number sub-fields – overheads that are incurred repeatedly for every request processed.

\(^3\)For instance, several low utility data structures and data copies have been found in the DaCapo bloat benchmark using existing bloat analysis techniques [138, 140]. A manual inspection of the source code revealed that many of these were for the purpose of constructing strings to be printed by \texttt{Assert} statements and could thus be easily optimized [138, 140]. If higher level intent information were available, the results could have been automatically grouped into such categories instead of requiring a site by site manual interpretation.
Advancement in techniques for detecting symptoms of bloat have focused mostly on inefficiencies related to data structure construction and usage (both temporary and long lived) [139, 86, 41, 138, 140, 21] as covered in Section 2.5 and Chapter 4. Meanwhile, the problem of detecting statements that are sources of execution bloat due to excess features remains unsolved till date. This is because distinguishing different features and assessing which ones are essential and which might be in excess requires insight into program intent, particularly when features may be tangled in that source code and hence during execution. In fact, as we illustrate later, it is this tangling that gives rise to execution bloat, hence it cannot be ignored. Due to the variety of possibilities in how features might be defined or implemented, this problem does not easily lend itself to be modeled using simple rules based on language semantics and structural or dynamic program analysis.

Is it feasible to augment existing program analysis approaches with the kind of high level intent information suitable for such tasks? How do we obtain this information in practice?

We observe that the current state of the art in software engineering research already provides some answers to the latter question through a rich variety of (mostly semi-automated) techniques for identifying, locating, analyzing [106], annotating and separating software concerns and features. Different sources of information, techniques and their combinations have been used, such as source code mining [11, 145], formal concept analysis, program slicing [53, 25], execution traces [39], human input, natural language processing and programmer annotations for concern separation like feature oriented [4] or aspect oriented programming (AOP) [63]. Software concerns are features, design idioms or other conceptual considerations that can impact the implementation of a program (please see Common Terms).

Thus concern or feature assignments (which associate concerns and their properties, i.e. concern intent, with the source code statements where they are implemented, i.e. concern extent) can represent functional intent at a level of abstraction required for automating execution bloat analysis. The focus of our exploration is to determine how exactly such extracted concerns could be used to augment program analysis and whether this can be generalized in a way that smoothly integrates with current practices in analyzing dynamic and static program properties. The nature of concern information available may be incomplete or coarse grained or
only approximate in some cases. Hence its exploitation in program analysis requires careful adaptation depending on the target problem and concern input source.

This chapter introduces the notion of “concern partitioning” to develop abstractions that can be used to make traditional program analysis concern-aware. Concern partitioning partitions a set of known concerns based on properties of their intent, in turn inducing a partition on program statements based on the granularity at which the extent of the concerns is known (both deterministic or probabilistic assignments could be incorporated). We show how this can be used to develop concern augmented (static or dynamic) program analysis techniques to answer “what-if” questions such as whether a given optional concern could lead to execution bloat and which particular statements are the sources of bloat when that concern is optional.

5.1.1 CAPA: concern augmented program analysis

**Concern partitioning:** It is common practice to partition code in terms of syntactic modules (e.g. by method, component or package) for scalable analysis of large programs or for summarizing the sheer volume of analysis output. In well modularized programs, the method, class and component hierarchies are designed to reflect code elements that implement separate concerns. Thus, this structural hierarchy is a natural first level indicator for grouping code statements in terms of functional intent; yet it is too simplistic for our purpose since program concerns can be dispersed (scattered) across several code modules and interspersed (tangled) with other concerns.

Therefore, we propose an approach that augments static or dynamic program analysis with code partitions based on an abstraction that broadly reflects properties of the underlying concerns of the program. We call these partitions “concern partitions”. Concern partitioning groups together statements that implement similar or related concerns, where the relation may be defined as an abstraction over properties associated with a concern, providing a way to capture additional domain knowledge or human understanding as part of the reasoning. In bloat analysis, for example, the property of interest could be whether a concern is optional as determined based on human interpretation; here, code statements that are known to implement optional concerns would form an abstract concern partition. Once concern partitions
are created, all subsequent reasoning is independent of the mechanism by which information about the concerns and their location was obtained, whether provided explicitly by an expert or inferred automatically using concern analysis techniques.

Such an analysis that combines static or dynamic program analysis (or both) with concern based abstractions is called a concern augmented program analysis (CAPA) (Fig 5.1).

5.1.2 CAPA for Bloat Analysis

In the rest of this chapter, we develop and explore the use of concern augmented program analysis (CAPA) techniques in execution bloat detection and estimation. Our key observation is that not all statements that implement optional concerns are relevant for identifying bloat. Instead, bloat detection amounts to looking for only those statements that implement optional concerns that are also structurally intertwined with code that implements an essential concern (Section 5.2). These statements can get executed as a side-effect of using an essential concern even when the optional concern is not in use; thus they are a potential source of bloat.
The simplest case to handle is the situation when detailed concern/feature assignment information is available at statement level granularity. In this case, locating bloat due to a known optional concern is trivial. This is because we can directly identify which of the statements assigned to the optional concern occur inside methods introduced by essential concerns. (A method is said to be introduced by a particular concern if it is defined by statements that are assigned to the concern.)

**CAPA bloat microslicer:** However, more typically, only method level concern assignment information for certain optional (or essential) concerns may be available. A method level input concern partition is too coarse grained for detecting bloat – we need to refine it to the statement level. In Section 5.4 we introduce a novel static analysis technique, called step-wise refinement microslicing, that performs an automatic decomposition of each method into microslices, the smallest incremental units (statement groups) that could possibly have arisen due to distinct feature extensions. It then builds a graph that connects microslices that could potentially be inter-related but are present in different methods. When used in conjunction with the original concern information, this enables us to implement a concern augmented static analysis that refines the original method level concern partition and discovers fine-grained structural interactions between optional concerns and the rest of the code to estimate candidate bloat statements (Section 5.5).

**Evaluating CAPA:** Tools using bottom up techniques for bloat analysis [138, 140] are typically evaluated for usefulness in terms of the performance improvement that could be achieved by fixing program code to address the candidate bloat patterns highlighted by the tools. In contrast, our approach amounts to a directed “what-if bloat analysis” for a given concern (or set of concerns) and hence must be evaluated in terms of how well it answers the “what-if” question, rather than by the volume of bloat uncovered. However, no benchmarks exist for validating such results of execution bloat analysis. Besides, our approach requires externally supplied concern information. Hence, we have compiled a set of well-understood control examples and case studies (drawn mostly from existing literature on feature oriented programming and

---

4e.g. as generated by running `grep` on method names that match a known concern keyword pattern
5Notice that microslicing is based purely on a structural dependence analysis and entirely concern-unaware – it does not assume any information about which methods correspond to distinct features
library specialization) for our evaluation. This ensures that we have reliable concern information as input as well as the ability to ensure a line by line validation of the results. To confirm the scalability of our analysis we also run it on larger examples with some manually verifiable form of concern bloat.

**Probabilistic CAPA:** In the case of large and unknown programs no apriori information about concerns may be available. To aid the process of finding bloat, we draw on approximate concern input from a previously developed statistical topic model (described in Chapter 6) that performs automatic discovery and assignment of likely concerns at statement level granularity [17]. We combine the concern information with the output of standard execution profilers to obtain concern based summaries of resource usage, e.g. to surface concerns with disproportionately high object churn which may be candidates for de-bloating (Section 5.6). This is an example of a probabilistic concern augmented dynamic analysis.

### 5.1.3 Contributions:

- An approach to augment traditional program analysis with concern based abstractions using the notion of *concern partitions*.

- A novel concern augmented static analysis for bloat detection that pinpoints statements that are sources of execution bloat given information or rules to identify methods that correspond to optional concerns. The technique is validated using an experimental evaluation covering a carefully compiled set of six case studies of Java programs of different sizes.

- A novel probabilistic concern augmented dynamic object churn analysis to help estimate potential bloat for early diagnostic purposes when no apriori concern information is available.
5.2 Execution bloat in the context of software concerns

Runtime bloat can manifest in intricate ways in large framework based applications as discussed earlier using the illustration in Fig 2.2 (Chapter 2) depicting the relationship between different forms in which bloat is manifested at runtime as observed at different levels of abstraction. Availability of concern information in CAPA enables us to directly focus on unnecessary or incidental execution overhead due to software (feature/concern) bloat. This connects the user level view software bloat in terms of feature bloat to deeper forms of runtime inefficiencies highlighted by previous research on bloat.

For example, typical causes of bloat such as misuse of reuse, designing without context and just-in-case programming [86] result in excess code for supporting more capabilities and conditions (concerns) than strictly required [21]. This manifests at runtime as extra checks, bloated transformations and the creation of heavy-weight data structures for simple tasks. In Fig 5.2 the code to canonicalize data from big endian to little endian (line 11-15, including the cost of parsing and formatting from string to integer and back, intermediate objects generated and checking isBigE) implements a software concern that may be unnecessary in deployment situations when all systems or data sources involved are uniformly big endian. Note that if the method big2LittleEndian() is labelled as an optional concern, we need our analysis to deduce that not only line 12, but also lines 11, 13 and 14 are potential sources of bloat associated with this optional concern.
In this chapter we adopt a view similar to most existing techniques for bloat analysis which are intended as an aid for humans, in that we limit our focus to estimating candidate bloat statements and do not attempt to perform automated de-bloating. Dis-entangling execution bloat is usually more involved than merely moving around such bloat contributing statements or creating specialized versions of a function. Observe that in the above example, one possible approach to manually reduce bloat in the common case while preserving the ability to handle little endian data sources would be to switch to big endian as a canonical representation.

Although this is a highly simplified example, the resource overheads of bloat could be substantial when a sequence of such data conversions occur repeatedly in a loop (e.g. iteratively over a collection of data elements in each input request in a server side application that receives a high volume of request transactions). For instance, recall the example in Fig 2.4 where 58 transformations were invoked just to convert a single date field from a SOAP message to a Java business object for every input transaction request. By zooming into the nested transformations involved in parsing these fields, Mitchell et al found that the use of a general purpose DecimalFormat class results in 6 transformations for each sub-field of the date (month, day, year, hour, minute and second) [89] – this alone accounts for 36 of the 58 transformations. They also observed that three out of six transformations for each field (i.e. a total of 18 out of the 36 transformations) are only required for handling decimal numbers [89], an excess concern when the class is used to process whole numbers, such as the sub-fields of a date.

5.2.1 When do excess features cause execution bloat?

Our main insight is that the presence of excess features does not necessarily imply runtime bloat - instead execution bloat occurs when the excess features induce an extra execution cost (e.g. via extra statements or data objects) during the usage of essential features.

To make this notion more precise, consider a Java component (library or package) which provides several methods that may be used by client programs. A given client program may only use some features of the library; the remaining features of the component are in excess of what is needed for this particular client. If the features are perfectly separated into independent methods, then the excess code does not cause execution bloat as it never gets executed by
the client program. In practice, however, the code for excess features can get tangled with the code for necessary features or interact with necessary features via shared data structures. This causes some excess code statements to be executed when the client program is run. The execution cost of these statements is the execution bloat incurred by the client program when it uses the library.

```java
public class Buffer {
    int buf = 0;
    int back = 0;
    public Buffer(int x) {
        buf = x;
    }
    void logit() {
        System.out.println("LOG: buf = " + buf);
        System.out.println("LOG: back = " + back);
    }
    int get() {
        int tmp3 = buf;
        logit();   [<---- deriv(LOG, BUFFER)
        return tmp3;
    }
    void set(int x) {
        int tmp = buf; <---- deriv(BUFFER, RESTORE)
        back = tmp; <---- deriv(BUFFER, RESTORE)
        buf = x;
        logit();   [<---- deriv(LOG, BUFFER)
    }
    void restore() {
        int tmp2 = back;  [<---- deriv(BUFFER, RESTORE)
        logit();   |---- deriv(LOG, RESTORE)
        buf = tmp2; <---- deriv(BUFFER, RESTORE)
    }
}
```

Figure 5.3: Example: Logged Restorable Buffer from [72, 64]

We will use a variation of a simple working example from [64], for illustration (Fig 5.3). This code has three concerns named BUFFER, LOG and RESTORE. Let us assume that the concern partition for LOG includes statements in the `logit()` method, RESTORE includes statements in the `restore()` method and BUFFER includes statements in the remaining
Consider a client program that uses the Buffer class just for the BUFFER feature (init(), get() and set() methods); it does not require the functionality provided by the RESTORE or LOG features. In this case, BUFFER is a necessary feature, while RESTORE and LOG are excess features. The restore() method is in excess as it belongs to the RESTORE feature, but it is not invoked when the client program executes and hence has no execution cost.

The statements tmp = buf; back = tmp; (lines 17, 18) also correspond to the RESTORE feature, but are tangled with code for the BUFFER feature (due to shared fields back and buf) and hence get executed by client program. Thus they contribute to execution bloat.

The logit() method is also excess because it belongs to the LOG feature. It is invoked through statements in the get() and set() methods (lines 18 and 20). These statements are in excess as they correspond to the LOG feature, but they are tangled with the BUFFER feature and hence get executed when the client program runs. Thus, these statements also contribute to execution bloat.

The problem of finding statements that might contribute to execution bloat thus involves finding those statements belonging to excess features that are structurally intertwined with executable code statements for necessary features (such as lines 13, 17, 18, 20 in Fig 5.3).

To identify such statements automatically, we observe for instance, that statements in a method corresponding to a necessary feature may write data to some heap variables, which are later read only by methods corresponding to optional features. For example, line 18 in the set() method belonging to the BUFFER feature updates the variable back, which is read only by the RESTORE feature. Further, line 17 defines a value that is used solely by line 18 to update back. Therefore, the concern partition should be refined so that such statements are labelled as contributors to an optional feature.

One way to detect these statements is to compute the intra-procedural backward slices of such heap variables in the method corresponding to a necessary feature. Likewise, we require intra-procedural forward slices when statements in a method assigned to a necessary feature
read heap variables whose values are written only by methods corresponding to optional features. However, the slices computed can contain statements that also affect other heap variables, some of which might not be optional. Hence we need to further partition the statements in each slice in terms of the other heap variables they affect.

In the next few sections we generalize this procedure for fine-grained partitioning as a basis for developing bloat detection heuristics.

5.3 Preliminaries

5.3.1 Working definitions

Some working definitions of terms that we use:

We introduce the term concern partitioning to describe the task of partitioning the statements of a program into groups that implement the same concern or a set of concerns related by their properties (i.e. by concern intent). There may be many ways to partition the program depending on the concerns of interest. The implementation of a concern in source code, i.e. its extent, may be spread across multiple underlying program units (scattering) and the same program unit may contribute to multiple concerns (tangling). For example, different statements in a method may support different concerns.

**Concern Partition**: A concern partition is a subset of program statements which implement the same concern or a set of concerns that either share a common property or are related in some other manner (as specified by a partitioning rule).

**Essential Concern**: A concern that is essential for a given usage/deployment scenario

**Excess Concern**: A feature/concern that is not required for a given usage/deployment scenario

**Optional Concern**: A concern that is not required (i.e. is in excess) under some usage/deployment scenarios, but could be essential in other usage/deployment scenarios.

**Mandatory Concern**: A concern that is essential under all usage/deployment scenarios

**Incidental Concern**: A concern that is required to an extent as it is used by some essential concern but is a somewhat ancillary artifact of an implementation choice and hence not entirely
relevant to the usage/deployment scenario

5.3.2 Preliminary static analysis

Our implementation is built as an extension of the same underlying static analysis infrastructure [92, 19] that we used in Chapter 4.

Escape Analysis  To locate data structure fields that might be shared across methods (and could cause structural interactions between the concerns assigned to those methods), we rely on escape analysis. Recall that the control-flow graph (CFG) for a method $M$ contains nodes that represent statements in $M$ and edges that represent potential flow of control among the statements. After constructing the control-flow graph of each method, our underlying tool [92, 19] performs flow- and context-sensitive pointer analysis and escape analysis.

Also recall that for each method the escape analysis computes the following:

- The *formal-in* set for a method $M$ contains the set of formal parameters. The implicit *this* parameter (in non-static methods) is also a formal-in.
- The *formal-out* set for a non-void method $M$ contains a single parameter $R$, the designated return value. The *formal-out* set is empty for a void method.
- The *escape-in* set for a method $M$ contains direct and indirect fields of the formal parameters of $M$ that are used, before possibly being defined, in $M$. These represent the upwards-exposed uses in $M$.
- The *escape-out* set for $M$ contains direct or indirect fields of the formal parameters of $M$ and the return value of $M$ that are defined in $M$.
- At each *Call site* $c$ that calls method $M$, the algorithm uses the escape-in and escape-out information, to compute the actual-in and actual-out sets, where
  - we generate an *actual-in* for each formal-in and each escape-in and
  - we generate an *actual-out* for each formal-out and escape-out in $M$. 
The algorithm associates escape-in and formal-in sets with the *Entry* node of the CFG, and escape-out and formal-out sets with the *Exit* node of the CFG; likewise, the actual-in and actual-out sets are associated with call sites.

**Forward and Backward Slicing**  Our microslicer employs a combination of intra-procedural forward and backward slicing of the *escape-in* and *escape-out* parameters identified for each method of interest. Note that although the slices we compute are intra-procedural, the underlying pointer analysis is inter-procedural and aliasing is accounted for when computing control and data dependencies.

An intra-procedural forward slice $\text{ForwardSlice}(M, S_i)$ consists of all CFG nodes of $M$ in the transitive closure of def-use sets starting from the node $S_i$, i.e. all nodes that are eventually data dependent on $S_i$ (and hence affected by $S_i$).

An intra-procedural backward slice $\text{BackwardSlice}(M, S_i)$ consists of all CFG nodes of $M$ in the transitive closure of data and control dependencies of the node $S_i$, i.e. all nodes that affect $S_i$ through a direct or indirect data or control dependence.

### 5.4 Microslicer

We describe how to build a novel static analysis based representation of a program or component\(^6\) that is useful for bloat detection. We call this analysis *step-wise refinement microslicing* as it breaks up each method into many fine (micro) slices which represent the smallest incremental units within the method that could possibly be assigned to different features or unique combinations of features. Each *microslice* can contain a non-contiguous set of statements, and microslices within a method may be composed in an incremental step-wise refinement order.

#### 5.4.1 Stepwise refinement microslicing

Our analysis is based on the following simplified model of step-wise (feature) refinement:

\(^6\)e.g a Java library package
1. Each method in the component is introduced by a potential feature. (The information about which feature introduced the method is not used in this static analysis stage but is part of separately supplied concern information as described in Sec 5.5, where each feature is pre-categorized as mandatory or optional as a basis for concern partitioning)

2. The statements in the method are generated through a stepwise code refinement, where statements in each refinement step are introduced due to inter-dependencies (or structural interactions) with another potential feature. In the parlance of feature oriented programming (FOP), these statements constitute a derivative of the other feature and the original feature which introduced the method.

**Implications:** The advantage of this model is that the statements corresponding to each refinement step can represent structurally intertwined code between multiple features; thus they are candidates that contribute to potential execution bloat when the feature because of which the refinement occurred is optional. Secondly, the statements corresponding to the subsequent refinement steps (and dependent refinements in other methods) indicate where fixes may be required in order to enable bloat contributing statement to be removed or separated from the method.

Continuing with our working example from Fig 5.3, BUFFER is actually a group of three method level features, INIT (init()), SET (set()) and GET (get()). Observe that lines 17, 18 and 23, 25 represent structural interactions between BUFFER and RESTORE, while lines 13, 20 represent interactions between LOG and BUFFER and line 24 between LOG and RESTORE as shown in Fig 5.3 by the lines marked as deriv(). Here, deriv(X, Y) is a derivative of two features X and Y, where the term derivative represents statements that are required only if both features X and Y are required.

We use static analysis to perform an automatic decomposition of a program according to this model based purely on structural dependencies (e.g. data and control dependence). To achieve this, we further assume that each refinement step either uses an additional input field or method or affects an additional output field (where input fields correspond to escape-in and

---

7 there are subtle differences between the FOP notion of feature derivative and the pure program dependence based interpretation we use, but the principle is very similar
output fields to escape-out or formal-out nodes for a Java method). Potential dependencies between refinements in different methods can occur when a refining statement in one method uses a field and a refining statement in another method defines (updates) the field. Therefore an automatic stepwise refinement decomposition can be performed using a combination of intra-procedural forward and backward slicing to partition the statements of a method into candidate refinements (or microslices, as we call them).

A forward slicing criterion may be an input variable or field directly used within the body of the method, and a backward slicing criteria may be an output variable or field that is directly updated within the body of the method. In addition to input and output data, we also introduce a forward (input) criterion for each method call statement, and a backward (output) slice criterion for the method definition. The set() method in Fig 5.3 has two input (forward slicing) criteria, the field buf and the method call to logit(); here the field buf is also an output (backward slicing) criteria as it is updated by the method, as is the field back.

Consider a method with two input criteria and two output criteria. Let us label the intra-procedural forward slices of the input variables F1, F2 and the intra-procedural backward slices of the output variables B1, B2. Then, the following partitions may exist: F1, F1F2, F1B1, F1B2, F2, F2B1, F2B2, B1, B1B2, B2, F1F2B1, F1F2B2, F1B1B2, F2B1B2, F1F2B1B2. Here, F1B2 represents the set of statements that are only in the slices for both F1 and B2 (and not included in either F2 or B1).

The non-empty partitions among these are “microslices” and correspond to candidate feature refinements, labelled in a way that captures step-wise refinement ordering.

**Definition 5.4.1.** Let $\phi^M_{in}$ and $\phi^M_{out}$ represent the set of input and output slicing criteria of a method $M$ respectively. The set of microslices $\mu^M$ of $M$ consists of all tuples $\mu = \langle M, \phi_f, \phi_b, S \rangle$ where $\phi_f \subseteq \phi_{in}$, $\phi_b \subseteq \phi_{out}$ and $S$ is a non-empty set of statements contained in the microslice, where a statement $s \in S$ if and only if all the following conditions hold:

- $s$ belongs to method $M$
- $s \in \text{ForwardSlice}(M, \phi)$ $\forall \phi \in \phi_f$
- $s \in \text{BackwardSlice}(M, \phi)$ $\forall \phi \in \phi_b$
- $s \notin \text{ForwardSlice}(M, \phi)$ $\forall \phi \notin \phi_f$
s \notin \text{BackwardSlice}(M, \phi) \forall \phi \notin \phi_b

The above conditions ensure that the microslices do not overlap and that all the statements in a single microslice use and update the same set of heap variables and no others. This set of heap variables is a subset of the input and output criteria. If a method contains \(N_s\) statements with \(N_f\) input criteria and \(N_b\) output criteria, at most \(\min(N_s, 2^{N_f+N_b} - 1)\) microslices could exist. This is because there are \(2^{N_f+N_b} - 1\) such possible subsets, the microslices must be non-overlapping and each microslice should contain at least one statement.

The actual number of microslices is typically lower. For example, in Fig 5.3, the \texttt{set()} method contains microslices: \(<\text{set()},\{\text{buf}\},\{\text{back}\},\{17,18}\>, <\text{set()},\{\},\{\text{buf}\},\{19}\>\) and \(<\text{set()},\{\text{logit}\},\{\},\{20\}>\).

Suppose line 19 in the program is replaced by \(\text{buf} = \text{tmp} + x_i\). Now line 17 affects two heap variables, \text{buf} and \text{back}, unlike line 18 which only affects \text{back}. Thus lines 17 and 18 would no longer belong to the same microslice. Hence, the microslices in this case are: \(<\text{set()},\{\text{buf}\},\{\text{buf},\text{back}\},\{17\}>\), \(<\text{set()},\{\text{buf}\},\{\text{back}\},\{18\}>\), \(<\text{set()},\{\text{buf}\},\{\text{buf}\},\{19\}>\) and \(<\text{set()},\{\text{logit}\},\{\},\{20\}>\).

### 5.4.2 Micro-Slice Interaction Graph (MSIG)

Next we proceed to find associated (interacting) refinements in other methods for each of these candidate feature refinements by building a cross-method \textit{micro-slice interaction graph} (MSIG) as follows:

For each field or object that is shared between methods, we find the methods where the field is used as an input and identify the microslices that are in a \textit{forward slice} of that input criterion. Likewise, we find the methods where the same field or object is an output and identify the microslices that are in a \textit{backward slice} affecting that output criterion. Now we create a directed edge from each microslice node in the second set to each microslice node in the first set.

This creates a directed graph where a (target node) microslice is a descendant of the microslices it might require (e.g. parent nodes generate state that is stored in fields which might
be later be used by statements in the child nodes). A leaf node has no outgoing edges. Notice that this graph captures potential dependencies regardless of interprocedural calling context since we do not assume information during the static analysis phase about the client programs or contexts in which the component is deployed.

**Definition 5.4.2.** A *micro slice interaction graph (MSIG)* of a component is a directed graph where the *nodes* are microslices and there is a directed edge from node \( \mu_i = \langle M_i, \phi_{f_i}, \phi_{b_i}, S_i \rangle \) to node \( \mu_j = \langle M_j, \phi_{f_j}, \phi_{b_j}, S_j \rangle \) iff \( \exists \phi_i \in \phi_{b_i}, \phi_j \in \phi_{f_j} \) such that \( \phi_i \) and \( \phi_j \) refer to the same field, variable or method name and \( M_i \neq M_j \).

![Microslice Interaction Graph for Fig 5.3.](image)

**Figure 5.4:** Microslice Interaction Graph for Fig 5.3.

Fig 5.4 represents the MSIG for the example code in Fig 5.3. Each node in this graph represents a microslice and the line numbers that it contains (we have omitted the microslice partition labels in this diagram). Each edge represents a potential cross-method interaction between microslices and is labeled with the dependence criterion (e.g. shared fields like buf and back, or called methods e.g. logit()). The destination node of a directed edge represents
a node that may depend on the source node, i.e. it has a MAY USE structural interaction. In some cases we can detect a MUST USE interaction, e.g. if the destination node has a statement that invokes a method call with a single call target which is defined by the source node. Here the notion of use, whether MAY USE or MUST USE, refers to feature usage (not necessarily data flow). For example, note that in the case of a method call, we treat the caller as a user of the method definition (e.g. the microslice containing the call to \texttt{logit()} in line 20 uses the code feature defined by the microslices in line 8 and line 9).

The MSIG enables many interesting analyses when combined with concern partition information for each node:

- Edges in this graph represent candidate structural feature interactions if they connect nodes which correspond to different concern partitions, while the microslices at the corresponding vertices constitute potential feature derivatives. For example, the nodes containing line numbers 13 and 20 each represent the target of a structural interaction between the features BUFFER and LOG.

- Cycles in this graph are merged into a single node which constitutes a derivative if the constituent nodes belong to different concerns. For example, node (17,18) and node (23,25) are part of a cycle, which means that they must be present together and are needed only when both the BUFFER and RESTORE features are required.

- An inclusion relationship between the partition label strings for two nodes (e.g. F1F2B1 and F1B1) may represent a feature refinement if the corresponding microslices belong to different concerns.

- The leaves of this graph represent microslices that do not have any dependent nodes and hence likely to be easily separable if their constituent statements are part of an optional concern.

Although the MSIG is a powerful abstraction to surface statements involved in inter-method interactions which could give rise to bloat, the microslicer alone does not possess the “concern-awareness” needed to capture conceptually related feature refinements. It may
generate numerous overly narrow candidate features and refinements because it does not have any information about the true concern partitions of interest. For this reason, micro-slicing cannot be used in isolation to detect potential execution bloat statements. We need to perform a concern augmented analysis by combining our MSIG representation with externally supplied concern information to distinguish mandatory and optional features and the methods introduced by them.

5.5 Concern Augmentation

We first present an overview of the CAPA (Concern Augmented Program Analysis) framework and then describe how it is applied to detect statements that contribute to execution bloat.

5.5.1 CAPA overview

Fig 5.5 shows an example of a concern augmented program analysis that combines static and dynamic program analysis with concern partition analysis. Here, each different type of analysis (static, dynamic, concern), can implement multiple partitioning functions and analysis functions.

For example, the microslicing analysis can be viewed as a partitioning function that partitions the statements of a program into micro-slices and an analysis function that computes the structural interactions (i.e. the may use and may be used by relations) associated with each partition. The concern analysis separately implements partitioning in terms of concern assignment information (e.g. it groups methods introduced by the same concern). We can then define concern abstraction rules which group concerns such that each group has concerns that must be available together but different groups may be mutually optional. As this kind of information may not be available for all methods, a conservative approximation is applied to a default partition which has all the unassigned statements.

---

8 as an aside, we note that it could also miss detecting refinements introduced for intermediate transformations which do not involve additional input or output fields

9 this figure is a slightly detailed version of Fig 5.1
Chapter 5. CAPA: Concern Augmented Program Analysis for Bloat Detection

In the last block shown in the figure the results of the different types of analyses are combined in a task specific manner (expressible as a datalog formula over analysis functions and partitioning predicates, for example) to arrive an output partitioning, an output analysis function and its results (aggregated) for each partition. Later we will illustrate some specific rules for bloat detection.

For example, the statements in the microslice partitions associated with structural interactions across mutually optional groups of concerns (or across a mandatory and optional concern) are relevant for execution bloat. Thus, a combination of static analysis and concern analysis can be used to find a list of these statements. The output of this analysis information could further be combined with the results of measuring dynamic resource usage profiles (i.e. the output of the dynamic analysis phase in Fig 5.5) to compute an estimate of the execution overhead due to bloat and further narrow down the list to the top bloat contributing statements.


5.5.2 Concern augmented static analysis to detect execution bloat

Let us assume that we are given a concern-based abstraction that partitions the concerns of a component into two groups - mandatory (always required) and optional. The exact mechanism used to realize this grouping may range from a purely manual assignment by an expert to automated classification rules based on an analysis of representative client programs. We also have information about which methods are introduced by these concerns.

We perform a CAPA analysis using this information and the statically computed MSIG to identify candidate excess statements that cause execution bloat when none of the specified optional features are required. Estimating the runtime resource overhead induced by these potential bloat contributing statements (dynamically or statically) provides an approximate quantitative measure of execution bloat.

Figure 5.6: Concern Augmented Microslice Interaction Graph corresponding to Fig 5.4. The shaded nodes represent microslices in methods belonging to optional features (LOG: blue and RESTORE: gray). The arrows represent potential USE interactions (where the destination node USEs the source node’s feature).
While the microslicing analysis described earlier finds candidate feature refinements and possible structural interactions between them, it cannot conclusively determine which of these are valid feature derivatives or potential bloat contributors. One problem is that as described earlier, the dependence graph is built in a MAY USE fashion. As a result, not all edges represent real structural interactions. Also, as we note from the graph, a feature refinement derivative due to an optional feature can involve dependencies in either direction, i.e. both uses and used by relationships are possible. For example, in Fig 5.6, the statements at lines (17,18) constitute a refinement which is used by the optional RESTORE feature, while the call to \texttt{logit()} in line 13 is a refinement that uses the optional LOG feature. With multiple MAY USE edges in both directions all through the hierarchy (and the possibility of missing edges due to other kinds of dependencies), unresolved decisions accumulate rapidly. Hence, we need a few heuristics or additional information to constrain the space and obtain an approximate analysis of potential bloat.

Table 5.1 lists a few heuristics and the statements from our working example that are detected as potential bloat contributors using each heuristic. To keep the analysis simple, we only use the first two heuristics, expressed below as a datalog formula:

\[
\begin{align*}
\text{optional}(s) : & \quad -\text{inmethod}(s, m), \text{inconcern}(m, c), \text{isoptional}(c) \\
\text{mandatory}(s) : & \quad -\text{inmethod}(s, m), \text{inconcern}(m, c), \text{ismandatory}(c) \\
\text{potentialbloat}(s) : & \quad -\text{mandatory}(s), \text{usedby}(s, \text{null}), \text{mustuse}(s, s'), \quad \text{optional}(s') \\
\text{potentialbloat}(s) : & \quad -\text{mandatory}(s), \text{usedby}(s, s'), \text{optional}(s')
\end{align*}
\]

(5.1)

The first rule simply states that a statement is marked as optional if it inside a method that is introduced by a concern which is known to be optional. Concerns that are not optional are conservatively treated as mandatory. Thus, according to the second rule, the statements in methods introduced by those concerns are marked as mandatory. The next rule encodes the

\footnote{where a directed edge typically signifies that the destination node may have a structural interaction that uses or depends on the source node}
heuristic that a statement can be reported as a potential contributor to bloat if it is mandatory (contained in a method that is needed) but MUST USE an optional concern statement (e.g. calls an optional method) and no statements from other methods depend on it. The last rule marks a statement as potential bloat if it mandatory but is only required by optional concerns.

<table>
<thead>
<tr>
<th>Used By</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 None</td>
<td>Must use at least one optional node [e.g. (13), (20) in Fig 5.6]</td>
</tr>
<tr>
<td>2 Solely used by optional nodes</td>
<td>Any</td>
</tr>
<tr>
<td>[e.g. (3), (17, 18) in Fig 5.6]</td>
<td></td>
</tr>
<tr>
<td>3 Solely used by nodes along MSIG paths that eventually contain an</td>
<td>Any</td>
</tr>
<tr>
<td>optional node</td>
<td></td>
</tr>
<tr>
<td>4 Any</td>
<td>Must use at least one optional node</td>
</tr>
</tbody>
</table>

Table 5.1: Bloat detection heuristics in decreasing likelihood of potential bloat contribution (computed only for MSIG nodes that are inside methods which can be executed by the client program even when no optional feature is used explicitly).

5.5.3 Implementation

As stated in Sec 5.3.2, we exploit an underlying interprocedural analysis previously used in Chapter 4 and in [92, 19] that maintains flow and context sensitive data and control dependencies including escape analysis and pointer analysis (built over the WALA\textsuperscript{11} framework). The microslices, themselves, however, are intraprocedural and thus reasonably small. Computing the microslice interaction graph can be expensive when there are a large number of shared fields (or methods).

The escape analysis results are used to identify input and output fields potentially shared across methods (e.g. the \textit{escape-in} and \textit{escape-out} sets of the method). We contain the number

\textsuperscript{11}http://wala.sourceforge.net
of slicing criteria by filtering out some variables, e.g. pass-through escaping fields. The underlying analysis maintains information about the field names or access paths. We currently use this for determining which fields are shared between which methods; the approach introduces imprecision in some situations where the matching is not accurate.

5.5.3.1 Sources of Imprecision and Tradeoffs

We have chosen a conservative choice of heuristics to reduce the possibility of incorrectly marking a statement as potential bloat (mis-detected bloat) while tolerating some undetected bloat. However, we found that this is sometimes too conservative and required further engineering tradeoffs to improve effectiveness.

Remark: We avoid using the terms false positive and false negative as it does not have the same connotation here as in the context of program verification where a sound analysis is associated with program safety properties. Unlike a bug, the presence of bloat does not result in an incorrect program – undetected bloat is just a missed opportunity for optimization. As we do not use the results to apply automated debloating transformations, mis-detected bloat does not cause serious problems either, although it increases the manual validation effort.

In addition to imprecision in our bloat detection logic (where we rely on heuristics), inherent imprecision in the underlying static analysis can lead to both misdetected or undetected bloat. For example, when mapping virtual call targets, we might miss a potential callee or over-estimate callees - the first causes missing edges, the second causes extra edges in the MSIG, both of which affect accuracy. We did not attempt to study these in depth as there are alternative approaches to handle issues of precision in the underlying analysis (e.g. with supporting dynamic analysis) that are independent of the logic for bloat detection/concern augmentation.

5.5.4 Evaluating Concern Augmented Static Analysis for Bloat Detection

To evaluate the correctness of the static CAPA technique we have compiled a set of chosen case studies of six Java programs of different sizes.
Chapter 5. CAPA: Concern Augmented Program Analysis for Bloat Detection

1. Three small to medium sized programs with precisely-known fine-grained feature assignments and optional concerns that cause execution bloat, where we could ensure a line by line validation of results.

2. Three large programs with some optional concern assignment that is practical to manually validate line by line.

We do not rely on a specific concern location tool in our evaluation. Our analysis takes concern information as supplied, without regard to how the information was obtained. The input to our analysis is a jar file of the component or program and a list of methods corresponding to one or more optional concerns. The output is a list of statements that contribute to potential execution bloat due to the specified optional concerns. We take this approach because concern input can differ with the choice of the external concern analysis source. Choosing to focus on programs with clearly defined optional concern assignments provides us an oracle to test the correctness of our “what-if” analysis independently of the efficacy of the concern analysis tools. The smaller examples test ability to detect bloat due to various fine-grained structural interactions, while the larger examples with well-known concerns confirm scalability of the approach for large code bases.

Illustrative examples: Before presenting our results for these six case studies, we discuss the analysis results for the two working examples described in Fig 5.3 and Fig 5.2.

Buffer is the 3 feature BUFFER-LOG-RESTORE example from prior work on feature oriented programming [72] described earlier. We specify 2 optional concerns, LOG (the logit() method) and RESTORE (the restore() method). The analysis correctly located the statements that are the sources of execution bloat (lines 3, 13, 17, 18 and 20). It could recognize the assignment to back in the set() method that occurs through a temporary variable (via intra-procedural forward slicing) and also recognized the initialization of the field back.

Adaptor is an adaptor pattern that includes an optional big endian to little endian converter, illustrated in Fig 5.2, similar to a feature previously described by Prehoofer [101]. We created this example because many anecdotes of bloat highlight excess or unnecessary transformations and checks, and we wanted to determine if our technique could detect such patterns. The
method `big2LittleEndian` was specified as the optional concern. Our analysis was able to detect as bloat, the excess parsing (lines 12, 14) and condition check (line 11) that was needed only to enable this converter, besides the call to `big2LittleEndian` in line 13.

**Case study results**: Table 5.2 summarizes the results for the six case studies. We observe that our analysis is effective in correctly detecting bloat in all these examples. We elaborate on the case studies\[superscript 12\] in a little more detail below:

![Table 5.2: Experimental Results: Concern Augmented Static Analysis for Bloat Detection](image)

\[superscript 12\]The first two case studies below can be made available for other researchers on request. The rest of the case studies use externally available benchmarks

5.5.4.1 *Programs with precisely known fine-grained feature assignments*

**Stack** is an enhanced implementation of the classic FOP example proposed by Prehoofer [101], a 5 feature STACK with COUNTER, LOCK, UNDO and BOUND checking. We specify 4 optional features: COUNTER, LOCK, UNDO and BOUND checking. This example spans multiple classes and captures several possible nuances of the way features interact to cause bloat. Our technique is able to detect most of them. The statements that it missed were calls to a `save()` method which belongs to the UNDO concern but was not listed in the concern information supplied. Even so, our technique was able to detect the main state saving statement
savedState = new String(state) in the save() method as excess because it is in a microslice that is only used by the UNDO concern. This illustrates that our analysis may be effective even with slightly incomplete concern information.

**c5list** is a (simplified) Java implementation of a LinkedList with support for Views from the C5 collection library for C#. This example has been used in a study of library specialization techniques [2], so it was an interesting experiment to explore whether our technique is capable of automatically detecting bloat due to the optional feature “Views” in this example. We found that the analysis could indeed detect such statements, e.g. it could identify statements of the form if (underlying != null) underlying.counter++; without the information that underlying is only relevant for “Views”.

**Graph** is the well-known Graph Product Line implementation available at the FEATURE-HOUSE web-site [5]. We started with a variant that supports weighted directed graphs and used our analysis to find execution bloat due to the optional weighted feature. There were only a small number of such statements. Our method was able to detect these statements.

### 5.5.4.2 Large programs with a manually verifiable optional concern

In these cases we had to rely on prior experience / knowledge to specify an optional concern that contributes to bloat which we also know how to find manually so that it is practical to validate results line by line.

**xml benchmark from SPECjvm2008** is a Java benchmark from SPEC [116]. We specified validation as an optional concern (it is relevant from a benchmark perspective but could be optional otherwise). We picked this concern as it is likely to occur at fine grained levels in different places in the code and it is possible to manually confirm whether a statement is related to validation. Our analysis found 22 instances in the code of which 21 were correct. The incorrect instance occurred because of imprecision introduced by the underlying analysis in mapping virtual call targets. Since we did not have prior knowledge of all instances of this bloat we do not evaluate the extent of undetected bloat.
SPECjbb2005 is a Server-side Java benchmark from SPEC [116]. From an inspection of the code during prior research on object churn bloat, we have observed bloat due to transformations to XML format for temporary in-memory storage. As XML formatting is not strictly an essential feature here, we specified it as an optional concern. We were then able to use our analysis to find the statements that contribute bloat due to this concern. These included string copies that we know (from our prior results [19] in Chapter 4) to be responsible for 40% temporary object churn, by de-bloating which we had observed a 20% improvement in benchmark performance.

DaCapo bloat is a benchmark from the DaCapo 2006 benchmark suite [27]. Other researchers have noted that the bloat benchmark incurs considerable object churn bloat in building strings just for the purpose of assertions [138, 140] even when the assertion conditions do not occur. We specified assertions as an optional concern and were able to find these potential bloat statements. Even though assertions themselves are fairly straightforward to find using a code search, our approach has the advantage that it can also automatically catch statements that build up data structures just for the purpose of debugging.

5.6 Probabilistic CAPA

The concern augmented program analysis technique that we just described for detecting sources of execution bloat works well, but only given the availability of at least some reliable (even if not necessarily complete) information about optional concerns at a coarse (method level) granularity. However, when working with a large unfamiliar code base, one may not have any apriori knowledge of concerns to start with. As an early diagnostic or triaging aid in these situations, we now explore a different kind of CAPA technique that is probabilistic in nature, and relies on the use of a statistical topic model [29] to automatically surface potential concern partitions from source code.

The key idea behind the use of topic models for concern discovery [11] is that method names, class names, variable names, strings, annotations and comments contain natural language words that provide clues about functional intent. By modeling the generative process of
how concerns manifest in code, a reverse engineering procedure using a probabilistic modeling approach can discover the latent concerns from software code text.

In this section we develop a probabilistic CAPA technique that combines this kind of approximate concern information with dynamic program analysis to surface concerns with a probability of causing disproportionately high object churn, concerns which may therefore be candidates for de-bloating. Such a probabilistic concern augmented dynamic analysis of object churn involves the following steps:

1. **Probabilistic concern partitioning** We rely on a particular statistical topic model variation that we have previously developed in [17], wherein the connection between source code and the latent concerns they implement is established at statement level granularity. This model is described in Chapter 6. The concern analysis output provided by the model is a probability distribution of concerns assigned to each statement in the program. This information enables concern partitions to be generated using a concern abstraction step that implements a standard clustering algorithm based on a computation of concern distribution similarity between statements. We used a cosine similarity measure and chose KMEANS as the clustering algorithm. Other alternatives could also be explored in future, such as grouping statements in terms of their most probable concern topic.

2. **Dynamic analysis of object churn** We perform a dynamic analysis step where we compute a profile of temporary objects churn of an application by subtracting live bytes from allocated bytes for each allocation site as reported by the java hprof profiler (similar to that previously used in Chapter 4 [19]). We focus on object churn because the generation of high volumes of temporary object (churn) is a well known form of Java runtime bloat [112, 19] that we have already studied in Chapter 3 and Chapter 4. In Chapter 6 we illustrate an application of probabilistic CAPA for summarizing bytecode profiles, another category of dynamic analysis.

3. **Combined concern-aware analysis** Finally we combine the object churn profile output with the concern partition information (see Fig 5.5) to generate a concern augmented profile. This shows the cumulative total of the churn due to statements belonging to a
concern partition and the methods it uses reported as a percentage of the total object churn incurred by the application\textsuperscript{13}. As object allocations can often occur in library methods or helper routines, a flat concern-wise profile may not provide us a useful picture of which concerns might actually be driving the need for allocations. Computing a cumulative profile is more tricky with concern probability distributions because it may not be clear how to attribute costs when different statements in the call trace have different concern probability distributions. Concern partitioning simplifies this problem as each statement is assigned to only one partition. For each partition we aggregate the churn attributed to call traces in which a statement from the partition appears. In this manner we obtain the cumulative object churn caused by this partition and its callees.

As the concern partitions are generated via a probabilistic inference which depends on the statistical characteristics of meaningful words in source code text, this approach is suitable mainly as an aid for early diagnostics in the absence of any known concern information as the resulting partitions may not always have a meaningful interpretation.

### 5.6.1 Experimental observations with probabilistic CAPA

A detailed evaluation of the underlying statistical topic model is covered in Chapter 6; e.g. we found a statement level concern assignment accuracy that agrees 70\% of the time with typical programmer interpretation. In this section, we highlight some observations from qualitative case studies with probabilistic concern augmented object churn analysis using the model for two large applications, an open source application, Apache lucene v1.9.1 and the SPECjbb2005 benchmark.

We run lucene in the context of two DaCapo benchmarks [27] \textit{lusearch} and \textit{luindex} which exploit different high level concerns (SEARCH and INDEX). The concern partitions are computed only once for lucene by running a statistical topic model on the source code, while the dynamic analysis (concern augmented post processing of the object allocation profile) is performed separately for the two benchmarks. Table 5.3 illustrates a snippet of code from one

\textsuperscript{13}notice that typically there is an overlap between allocations attributed to different concern partitions as a concern uses other concerns
String[] fields = {b, t};
Query q = MultiFieldQueryParser.parse(query, fields,
new StandardAnalyzer());
assertEquals(expected, s);
String[] fields = {b, t};
String[] queries = {one, two};
Query q = MultiFieldQueryParser.parse(queries,
fields, new StandardAnalyzer());
String[] queries2 = {one, two};
q = MultiFieldQueryParser.parse(queries2,
fields, new StandardAnalyzer());
String[] queries3 = {one, two};
q = MultiFieldQueryParser.parse(queries3,
fields, new StandardAnalyzer());

Table 5.3: A few lines taken from the file TestMultiFieldQueryParser.java in Apache lucene, colored based on probabilistic concern partitioning

of the source files (picked at random) with statements colored based on the concern assigned by the model. Notice how the concern that represents test data (dark blue) is distinguished from the main query analysis concern even though they appear in consecutive statements in the code.

Topic models typically require a human evaluation of the results to assign meaningful labels based on the concern topic words (and in our case, a perusal of a few representative statements from the partition, as highlighted using a summarization tool). This is reasonable in situations intended for early diagnostics or assessment of potential opportunities for optimization. We present the output summaries to the user to decide whether any of the partitions appear to represent optional or incidental concerns that are unusually resource heavy.

Table 5.4 summarizes a few results of interest. The generation of partitions and their
cumulative object churn is fully automatic and unsupervised. Concern labelling and categorization as optional or incidental is performed by the user. For example, we label the SEARCH and QUERY partitions for lucene based on the main terms in the corresponding concern topics such as (searcher, similarity, term, doc, freq, hits) and (query, clauses, range, phrase, disjunction) respectively. We can also guess that SEARCH and QUERY are required when running lusearch but not when running luindex, so we can categorize it as an optional concern. For SPECjbb2005, looking at the representative summary statements we noticed a partition with a lot of data conversions between numerical and string values, hence we labeled it as FORMAT CONVERSION.

We find that the SEARCH/QUERY concern in lucene causes churn only when it is required (when running lusearch) and has a minimal object churn overhead when it is optional or unused (when running luindex). Thus, it is not a contributor to execution bloat in terms of object churn. The FORMAT CONVERSION concern partition in SPECjbb2005 is likely to be an incidental concern which appears to be causing excessive churn and hence a likely candidate for debloating. Analyzing the top object churn contributing statements of this partition highlights 38% churn due to string copies for XML logging which matches our expectations based on our previous findings [19] described in Chapter 4, where we have seen upto 40% object churn reduction through de-bloating these object creations. In the same partition we also find an additional 12% churn in BigDecimal to string conversion for logging Dollar values, which

<table>
<thead>
<tr>
<th>partition name</th>
<th>lusearch</th>
<th>luindex</th>
<th>SPECjbb</th>
</tr>
</thead>
<tbody>
<tr>
<td>source file documents</td>
<td>564</td>
<td>564</td>
<td>63</td>
</tr>
<tr>
<td>words</td>
<td>120979</td>
<td>120979</td>
<td>42594</td>
</tr>
<tr>
<td>concern partitions</td>
<td>30</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>partition name</td>
<td>SEARCH/QUERY</td>
<td>SEARCH/QUERY</td>
<td>FORMAT CONVERSION</td>
</tr>
<tr>
<td>%object churn of partition</td>
<td>12%</td>
<td>0.016%</td>
<td>62%</td>
</tr>
<tr>
<td>optional or incidental in this execution?</td>
<td>No</td>
<td>Yes</td>
<td>Maybe</td>
</tr>
</tbody>
</table>

Table 5.4: Probabilistic CAPA for object churn
has not been reported by previous techniques. These results highlight one of the benefits of a concern augmented analysis approach that we emphasized earlier: The approach can point us to what was meant to be accomplished by the code statements responsible for the pattern of excess activity seen, in this case “data format conversion”. Such insight can be useful both in understanding the reason for bloat and in de-bloating it most effectively.

5.7 Related Work

5.7.1 Java Bloat Analysis

Several approaches have been explored for detecting and eliminating runtime bloat in Java applications [139] (see Chapter 2). Data structure health signatures [87] use the structural semantics of Java’s data model to distinguish data structure representation overhead to measure memory bloat. Different notions of bloat have been considered in the absence of an explicit model for distinguishing overhead from necessary data or activity. A variety of symptoms of excesses have been used to recognize the presence of bloat, such as high volumes of temporary objects, unbalanced costs vs benefit of object creation and consumption [41, 138, 140, 141, 19] and object reuse opportunities in loops (Chapter 4) [19]. However, bottom up techniques are limited by lack of higher level insight about the purpose of code statements responsible for the pattern of excess activity suspected. Augmenting the analysis with information about program concerns and their properties (such as when a concern is necessary and when it isn’t) enables us to tackle the hitherto unsolved problem of detecting execution bloat due to excess features. Existing runtime bloat detectors can also benefit from such higher level insight to aid de-bloating.

5.7.2 Concern Analysis

There is a large body of existing literature on concern location and discovery, most of which is oriented towards program comprehension, maintenance and re-engineering tasks. A variety of techniques ranging from formal concept analysis, exploiting program topology [106].
information retrieval, graph mining, program slicing [53, 25] and dynamic analysis [39] have been employed in this context. Solutions that combine multiple approaches [104, 146, 109] and exploit multiple sources of information including test cases, program documentation and evolution history have also been used to improve the quality of results. It is the existence of these techniques that makes the concern augmentation step in our analysis feasible or usable in practice. Traditionally program analysis has been used to aid concern location. In contrast we propose the use of concern information to enrich program analysis for solving new problems.

5.7.3 Slicing

The use of slicing in refining concern location, key statement analysis [53] and building concern dependence graphs for program understanding is fairly well-established. Several heuristics such as the use of barriers during dependence graph traversal have been proposed for containing the size or (interprocedural) span of slices. Our microslicing analysis for automatic feature refinement decomposition addresses a novel objective - to unearth candidate fine grained structural feature interactions that could result in execution bloat and to isolate the corresponding feature refinements that potentially contribute to bloat. We incorporate concepts from feature oriented programming in (micro)slicing - we model the decomposition (slices) intra-procedurally as candidate feature derivatives [72, 64] (which could be be peeled out in a reverse step-wise refinement fashion) and build relations between these derivatives to enable bloat inference when augmented with requisite concern information.

5.7.4 Topic Models

Statistical topic models have been applied in several program comprehension and software maintenance tasks such as mining source code repositories for software concerns [71, 11, 76, 110, 118], estimating semantic coupling metrics [48], bug localization [74], statistical debugging [3] and software traceability [7]. We believe we are the first to explore the use of such models as an input for dynamic analysis of resource usage or bloat summarization. In the work [17] described later in Chapter 6, we explore a statistical model variation that infers
concerns at statement granularity, while in the current chapter, we have illustrated the use of such a model in concern augmented dynamic analysis as an early diagnostic aid for bloat detection.

5.8 Conclusions

Advances in concern analysis techniques provide an opportunity to address difficult problems such as reasoning about execution bloat which require higher level insight into underlying functional intent and therefore cannot be tackled using static and dynamic program analysis techniques alone. However, making program analysis concern-aware is non-trivial in practice because concern information can be incomplete, overly coarse or even uncertain. We introduced CAPA, concern augmented program analysis, and demonstrated its application in two different approaches to detect and estimate execution bloat in Java programs. We use conservative abstractions when concern properties are not available for all statements, microslicing when concern assignments are too coarse and probabilistic approximation when concerns are uncertain. Our results show that despite the caveat that the effectiveness of the CAPA methodology is highly dependent on the quality of concern information available, it can provide a fresh approach to problems that require ability to jointly reason about intent, structure and dynamics of program behavior.

Acknowledgments for this Chapter

The probabilistic version of CAPA and the underlying statistical topic model (described in Chapter 6) was the outcome of a joint collaboration [17] with Mrinal Kanti Das, K. Gopinath and Chiranjib Bhattacharyya. We thank Mangala Gowri Nanda for the underlying static analysis and program slicing infrastructure that we used to build our solution. We also thank K. V. Raghavan, Aditya Kanade, Rupesh Nasre and K Vasanta Laxmi for feedback and discussions.
Chapter 6

Modeling Statement Context to Surface Diffused Concerns Automatically for Probabilistic CAPA

In the absence of any apriori concern information, we use a statistical topic model to automatically discover latent concerns from source code statements and thus enable probabilistic CAPA to correlate latent concerns with program properties that vary at statement granularity.

6.1 Introduction

Understanding performance properties of framework based software in order to diagnose runtime bloat requires significant expertise and effort, even with state-of-the-art tools. With concern augmented program analysis (CAPA) in the previous chapter (Chapter 5), we introduced a structured approach to exploit information about program intent in terms of software concern assignments for tackling difficult bloat analysis problems that require higher level insight. However, such concern information may not be available in practice when using unfamiliar

---

1 After this thesis was sent for review, further work resulted in a more advanced topic model, due to appear in the following publication: Mrinal Kanti Das, Suparna Bhattacharya, Chiranjib Bhattacharyya and K. Gopinath. Subtle Topic Models and Discovering Subtly Manifested Software Concerns Automatically, ICML 2013.
A promising alternative is to devise new performance summarization techniques that automatically discover latent program concerns (in source code) and analyze their runtime resource usage. Latent concerns reflect underlying intent and are not tied to assumptions about the specific nature of concerns. These could be features, non-functional requirements, design idioms, implementation mechanisms or other conceptual considerations that can impact the implementation of a program [106].

Summarizing dynamic properties such as runtime resource usage in terms of latent concerns can provide a novel perspective for performance understanding. Awareness of high level functional intent can provide more insight than analysis tools that report costs in terms of low level artifacts (such as methods and components). In Section 5.6, we presented one such application – a probabilistic concern augmented dynamic analysis of object churn as an early diagnostic aid for estimating bloat. In this chapter, we explore how to discover the concern information needed for probabilistic CAPA.

**Statistical topic models:** One way to automatically summarize concerns in source code for approximate analysis is to use statistical topic models, such as Latent Dirichlet Allocation (LDA). The technique relies on the observation that method names, class names, variable names, strings, annotations and comments in source code text contain natural language words that provide clues about functional intent. Hence, it has been argued [11, 76] that latent topics discovered by analyzing source code as textual documents can provide a sense of what a given source module is about - i.e. they reflect underlying program concerns. The advantage of this approach over other concern identification and location techniques (whether generative or query-based [75]) is that it works without any additional input, knowledge about concern domains or other assumptions about concern characteristics. This makes it possible to extract a somewhat representative set of latent concerns from a large unfamiliar software code base in a fully automatic manner unbiased by apriori notions about the specific nature of concerns. The concerns discovered could include both high-level and low-level concepts whose implementation may be scattered across low level artifacts such as methods and components.

**Technical challenge and scope:** Despite the possibilities that it can open up, we find that the use of statistical topic models as an input for applications of this nature is problematic as it
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

138

raises demanding requirements that challenge the current state of the art in modeling concerns using LDA and its variants. For instance:

1. Interesting cross cutting concerns (in terms of resource usage) may not always have a prominent presence in any source module. This is especially true in framework based code where all underlying module sources may not be available. Existing models cannot usually surface these diffused concerns as topics, as the statistical contribution of modular content typically dominates over statement level information (Section 6.4).

2. Accurate statement level granularity of concern assignment is required for concern-wise attribution of dynamic properties such as runtime resource usage and bloat (because different statements contribute differently to resource usage). LDA is known to behave poorly on small documents with statistically insignificant textual content, such as individual source code statements seen in isolation.

While point 2 can be covered to a certain extent by suitably engineering a solution that combines existing methods from topic modeling research, addressing point 1 requires new model extensions. We find that even applying specialized models such as MG-LDA [119] to software fails to address these challenges (Section 6.4). Hence we propose using a single extended model as developed in [17] to jointly address challenges 1 and 2. Our key contribution is to explicitly control the trade-off between modular and contextual contribution of concerns so that the model is more sensitive to contextual information contained in the neighborhood of a statement; this helps surface even diffused concerns at statement granularity. The model was called CSTM (context sensitive topic model) in [17] to highlight its general applicability to topics in documents. However, to emphasize our adoption of the model for finding statement level concerns, we will henceforth refer to it as a Context Sensitive Concern Model, or CSCM.

Evaluation methodology challenge: To evaluate the effectiveness of our model, we collect measures of the diversity of concerns found for four Java applications and design synthetic experiments using a well-understood application, BerkeleyDB, to expose differences in the sensitivity of alternative models to diffused concerns. This is not enough, as we also need to confirm that the increased sensitivity obtained using our extensions does not hurt accuracy.
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

However, concerns are subjective and represent human interpreted concepts; thus there is no one correct assignment for a given program (Section 6.2). This makes it difficult to quantitatively measure the accuracy of the models we investigate. Further, our main goal is not to find specific concerns or perform a specific predictive task, but to surface a representation or summarization that serves an exploratory purpose (such as performance understanding). Thus, neither standard information retrieval/concern location metrics such as precision and recall for a specific concern nor intrinsic measures such held-out likelihood, are suitable for evaluating the effectiveness of the models for this purpose. Instead, we opt for a human (programmer\(^2\)) evaluation approach along the lines recommended by topic modeling researchers [33] for quantitatively judging interpretability of concern topics and their statement level assignments.

**Contributions:**

- Using a probabilistic model called CSCM (Context sensitive Software Concern Model) [17], we show that it is possible to automatically discover both prominent and diffused concerns at statement level granularity without any human input or apriori knowledge (Section 6.5). The model assigns a mixture of concerns to each statement via a probabilistic inference procedure that is sensitive to both the surrounding statements (local context) in which the statement occurs and the containing module.

- We conduct a systematic evaluation of the sensitivity of diffused concern detection, interpretability of statement level concern assignments and the diversity of concerns found by CSCM and LDA-CS, an adaptation of LDA with an inference procedure for statement level assignment of concerns. Our detailed evaluation (Section 6.7) includes a program-mer interpretability study where we compile 540 responses on word intrusion and topic relevance tasks by 35 programmers from different organizations.

- We illustrate a novel application of the model: computing cumulative byte-code profile summaries in terms of latent concerns (Section 6.6). This paves the way for a new class of automated analyses (e.g. probabilistic CAPA) correlating latent concerns with program properties that vary at statement granularity.

\(^2\)familiar with Java
6.2 Problem Definition

6.2.1 Concerns

To support the exploratory context of performance understanding, we take a very broad view of what constitutes a software concern (please see Common Terms). Concerns can exist at many conceptual levels and do not fit neatly within a single dominant decomposition.

**Concern Localization**: The process of locating the structural or syntactic program units (modules or statements) that implement a given concern. Some program concerns may be *modular* (i.e. implemented by a single file or module) while others may be *cross-cutting*, i.e. dispersed (scattered) across several code modules and interspersed (tangled) with other concerns.

**Diffused Concern**: A cross-cutting concern that does not have a prominent presence in any available source module.

6.2.2 Problem statement

In this chapter, we are interested in the automatic discovery and statement level localization of latent concerns from unfamiliar source code, with the ability to distinguish statements that implement different concerns even if they appear in consecutive lines of code within the same module. The model must work without any apriori knowledge or human input and should be able to surface diverse concerns including diffused concerns.

Let $\mathcal{P}$ be a software project which consists of $M$ source modules, $\mathcal{P} = \{D_1, \ldots, D_M\}$, where a module consists of $N$ statements, $D_i = \{S_{i1}, \ldots, S_{iN}\}$. Using this notation, we precisely define our problem below.

**Objective**: Find $f$ such that,

$$f : \mathcal{P} \rightarrow (C^m, C^c, y)$$

where $C^m$ is the set of modular concerns, $C^c$ is the set of cross-cutting concerns and $y = \{y_{11}, \ldots, y_{MN}\}$ where $y_{ij} \in \text{the power set of } C^m \cup C^c$, i.e. $y_{ij}$ is the mixture of concerns assigned to $S_{ij}$, statement $j$ in module $i$. This captures the observation that a statement can
reflect multiple concerns.

### 6.2.3 Evaluation criteria

Given the highly subjective nature of concerns, there is no one true representation of concern assignments for an application. We select the following criteria for evaluating the effectiveness of different models:

**Criteria:** Even if a model generates several incoherent concern topics, we will consider it to be effective (for an exploratory purpose) as long as it can surface diverse concerns (including diffused ones) and their relevant statements as interpreted by a human programmer. More precisely,

Let $I$ be the set of valid interpretations $f_A$ of the representation of actual program concerns $A^m$ and $A^c$ and their statement-wise assignment $y_A = \{y_A^{11}, \ldots, y_A^{MN}\}$, according to human judgment.

$$I = \{f_A : P \rightarrow (A^m, A^c, y_A)\}$$

Then, the effectiveness of the model can be assessed in terms of the following criteria.

1. $\exists f_A \in I$ for which $|C^m \cap A^m|$ and $|C^c \cap A^c|$ is high (Interpretability of $C^m$ and $C^c$)

2. $\exists f_A \in I$ for which divergence among the concerns in $C^m \cap A^m$ and $C^c \cap A^c$ is high (Diversity of $C^m$ and $C^c$)

3. $\exists f_A \in I$ for which overlap between $y$ and $y_A$ is high (Interpretability of $y$)

It is, however, unrealistic that $I$ can be determined in practice. Hence it is difficult to create test datasets for evaluating the above criteria. However, a systematic evaluation is still possible, if we make the more reasonable assumption that humans with programming or application domain knowledge can judge whether a known concern $c \in C^m \cup C^c$ or whether a particular concern assignment $y_{ij}$ is likely to be consistent with some valid interpretation $f_A^A \in I$. In this scenario, we propose the following evaluation methodology:

- Create a list of concerns $\{a\}$ which are known to be present under some particular interpretation $f_A^A$ for a test project, and check if $a \in C^m \cup C^c$ (criterion 1).
• Measure divergence of \(C^m \cup C^c\) (criterion 2).

• Design a programmer interpretability study to quantify whether samples from \(C^m, C^c\) and \(y\) are consistent with some \(f^A \in \mathbf{I}\) as judged by several humans with programming domain knowledge (criteria 1 and 3).

In order to test for detection of diffused concerns we design two more tests:

• Inject a diffused external concern \(w\) into the test project, and check if \(w \in C^c\).

• Prune source files in the test project where a known concern \(w\) is prominent, so that it becomes diffused, and check if \(w \in C^c\).

### 6.3 Background: Probabilistic Topic Modeling

Topic modeling is an unsupervised machine learning technique which uses a statistical analysis of words in a collection of text documents to automatically discover topical themes. The key idea is to formulate a statistical model of document collections based on the intuition that documents are mixtures of topics, where a **topic is defined as probability distribution over words from a fixed vocabulary**. For example, a document about quantum computing could reflect three topics in high proportion \{quantum, uncertainty, entanglement \ldots\} (quantum mechanics), \{algorithms, computing, complexity \ldots\} (computer science) and \{bit, entropy, information \ldots\} (information theory). A single word could figure in multiple topics. For example, the word *play* could occur in both sports and music related topics.

A topic model is often described by its generative process, a random process that encodes these intuitive assumptions as latent (random) variables based on which the model expects the documents were generated \([30, 117]\). Statistical inference algorithms are then used to reverse engineer the process to fit an observed collection of text and estimate the model parameters or latent variables: i.e. the topics that characterize the collection and the proportions in which they are reflected in any given document. In other words, the model treats the data as if it arose from the generative process in order to infer latent topics that best explain the data.
Topic modeling algorithms have also been adapted to discover themes in non-textual data such as images, genetic information, software code and execution traces by redefining the vocabulary and the notion of words and documents to entities and units appropriate for the category of data modeled.

### 6.3.1 Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) [29] is a popular generative probabilistic topic model. It treats documents as random mixtures of topics, where each document exhibits topics in different proportions. Topics represent pre-existing distributions over words for a collection. Documents are assumed to be generated according to a two-stage generative process. First for each document, we randomly choose a multinomial distribution over topics by sampling a Dirichlet prior distribution. Second we generate each word in the document by first randomly choosing a topic by sampling the corresponding *per-document distribution* over topics and then randomly choosing a word from the topic by sampling the corresponding *per-topic distribution* over words.

Thus the LDA model for a corpus (collection of documents) is characterized by the hyper-parameter $\alpha$ of the Dirichlet prior distribution of document topic proportions and by its topics $\beta$, where the $k^{th}$ topic is characterized by $\beta_k$, a probability distribution of words\(^3\) over a fixed vocabulary. The number of topics $K$ (reflected in the dimensionality of the Dirichlet distribution) is assumed to be known and fixed. In practice it is chosen based on intuition and experimentation, although several approaches have been proposed that empirically determine a suitable value of $K$ based on the data or use non-parametric models.

**Significance of the Dirichlet hyper-parameter $\alpha$:** The use of a Dirichlet prior on the topic distribution $\theta$ helps smoothen the distribution of topics based on the hyper-parameter $\alpha$ which controls the mean shape and sparsity of the document-wise topic proportions. [117]. The Dirichlet distribution is a convenient choice as it is the conjugate prior of the multinomial distribution – the posterior distribution is of the same form as the prior, enabling parameter

\(^3\)i.e. topics are latent multinomial variables
estimates to be updated by successively incorporating new observations during inference. The $K$ dimensional Dirichlet random variable $\theta$ takes values in a simplex representing all possible probability distributions over the $K$ topics (i.e. $\theta_i \geq 0, \sum_{i=1}^{K} \theta_i = 1$) with the following probability density:

$$p(\theta | \alpha) = \frac{\Gamma\left(\sum_{i=1}^{K} \alpha_i\right)}{\prod_{i=1}^{K} \Gamma(\alpha_i)} \theta_1^{\alpha_1-1} \cdots \theta_K^{\alpha_K-1}$$

With symmetric parameters, $\alpha_1 = \alpha_2 = \ldots = \alpha_K$, higher values of $\alpha >> 1$ favour points towards the center of the simplex with a high probability, i.e. this choice assumes all topics are likely to appear in equal proportion in any given document. Lower values of $\alpha << 1$ favour points towards the corners (all corners are favoured equally because of the symmetry), where only a small number of topics occur in significant proportion in any given document (i.e. it favours sparse distributions). Usually $\alpha$ is set to a small value as the latter choice better reflects common intuition.

Generative models such as LDA are also conveniently described using a probabilistic graphical model representation (Fig 6.1).

**Generative process for LDA:** Each document $d$ in the corpus of $M$ documents is generated as follows

- Sample a distribution of topics $\theta_d \sim Dir(\alpha)$

- For each of the $N$ words, $w_{d,n}$ in the document $d$
  - Sample a topic $z_{d,n} \sim mult(\theta_d)$
  - Sample a word $w_{d,n} \sim mult(\beta_{z_{d,n}})$

LDA represents documents as a *bag-of-words*, i.e. the generative process ignores the order of words in the documents – only the number times a word is produced is relevant after filtering out frequently occurring (stop) words such as “the, and, to, of” that would not contribute to meaningful topics.
Figure 6.1: Graphical model representation of LDA. Each node is a random variable. Edges represent conditional dependence, where incoming arrows at a node depict the variables it depends on. \( w \) is the only observed variable here (shaded node) representing words in a document. The hidden nodes (latent variables) are unshaded. This includes the document wise topic proportions \( \theta \) (and its Dirichlet prior parameter \( \alpha \)), the topic assignments \( z \) and the topics \( \beta \), where topics are distributions over words in the vocabulary. The rectangles are “plates”, a concise notation for replication – the outer and inner plates marked \( M \) and \( N \) imply iteration over \( M \) documents and then over \( N \) words in each document.

### 6.3.1.1 LDA Inference

The generative process for LDA corresponds to the following joint distribution of the latent and observed variables:

\[
p(\beta_{1:K}, \theta_{1:M}, z_{1:M,1:N}, w_{1:M,1:N}) = \prod_{i=1}^{K} p(\beta_i) \prod_{d=1}^{M} p(\theta_d | \alpha) \left( \prod_{n=1}^{N} p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)
\]

LDA statistical inference algorithms invert this process to obtain an estimate of the latent variables i.e. the topic structure (model parameters \( \beta \) and the document-wise topic proportions \( \theta \)), given an observed data set (document collection). The conditional distribution (posterior) of the latent variables given the observed documents is:

\[
p(\beta_{1:K}, \theta_{1:M}, z_{1:M,1:N} | w_{1:M,1:N}) = \frac{p(\beta_{1:K}, \theta_{1:M}, z_{1:M,1:N}, w_{1:M,1:N})}{\int_{\beta_{1:K}} \int_{\theta_{1:M}} \sum_{z_{1:M,1:N}} p(\beta_{1:K}, \theta_{1:M}, z_{1:M,1:N}, w_{1:M,1:N})}
\]

\[ (6.1) \]
The maximum likelihood method can be applied to infer the values of the parameters. However, the problem of computing the posterior (in particular, the denominator in the above equation) is intractable. There are two popular alternative methods available in the literature to address such problems: variational inference methods\(^4\) and sampling based methods [117]. We adopt the variational expectation maximization (EM) algorithm from Blei et al. [29] in our models (for details we refer the reader to [29, 124]).

The E-step of the EM algorithm iterates over all documents applying variational inference to estimate the document-wise topic proportions based on previously inferred estimates of the topics, \(\beta_{l,K}\), and the words observed in the documents. The M-step then updates the inferred posterior estimates of the topics, \(\beta_{l,K}\), based on the aggregated per-document parameters obtained from the E-step. The two steps are repeated iteratively until convergence (in the relative improvement in likelihood attained over successive iterations). This means that the latent topics found are purely outcomes of a stochastic process inferred from a specific dataset – even so, in practice, they have been observed to reflect human intuition of word-clustering around semantic themes. The \(k^{th}\) topic is typically summarized in terms of the most likely words and their probabilities in the multinomial distribution described by \(\beta_k\). For example, the latent topic \{bat 0.25, wicket 0.12, ball 0.11, runs 0.10 \ldots\} may be found in a given document collection which contains some articles related to cricket.

\section{6.4 Assessing applicability of state-of-the-art topic models to the problem of statement level concern discovery}

Several researchers have observed that the notion of a concern in software is very similar to that of a topic in documents – this has led to the adoption of topic models such as LDA for mining software concerns from source code [71, 11, 76, 110]. Baldi et al. proposed that the concept of concerns and topics should be unified by definition – according to them, “a concern

\(^4\)Variational methods approximate the posterior by placing a parameterized family of (variational) distributions on the latent structure, optimizing these variational parameters to find a member of the family that is the closest to the true posterior.
is a latent topic” [11]. Hence, concerns are modeled exactly as topics, i.e. as multinomial distributions over words present in source code text, which can be estimated by running the LDA inference algorithm on a collection of (suitably pre-processed) source code files. The document-wise (concern) topic proportions inferred by the model reflect the concern assignments, the proportions in which concerns are manifested in each source file (or method).

In this section, we discuss our experiences in applying state-of-the-art statistical topic models to our problem, automatic latent concern discovery at statement granularity, a much finer granularity than has been attempted by prior work.

### 6.4.1 FINDING 1: LDA cannot detect diffused concerns

Although LDA enables us to discover concerns automatically from software code [11], we find that LDA based models have two major drawbacks when it comes to finding concerns in individual code statements.

First, due to the bag-of-words approach, these models do not localize the concerns at a very fine level of granularity such as code statements. One can try to treat each statement as a single entity (or document) and apply LDA based topic modeling but it is difficult to understand the usage of a statement in isolation without looking at the surrounding statements and containing module for context. We mitigate this issue by adopting a two step approach in LDA-CS, our adaptation of the prevalent LDA based methodology. We estimate concern topics by treating each source file as a document and just modify the subsequent inference procedure to treat each statement as a separate document to assign concerns to statements.

Second, we find that LDA works well in locating prominent concerns, where a prominent concern refers to a concern (either modular or cross-cutting) that has a prominent presence in at least one source module. However LDA is unable to detect diffused concerns. We confirmed this by conducting a controlled experiment that introduces a diffused concern by injecting few statements corresponding to a foreign concern at random into multiple source files belonging to a particular software application (We have discussed the experiments in detail in [17] and summarize them in section 6.7.2). We observed that LDA could not detect the foreign concern even when five lines each from the concern were introduced at random positions in all source
files of size greater than 100 lines (which covered approximately 50% of the total number of files).

To address this issue, we explored the use of an alternative model that can estimate topics at a finer location granularity than an entire document.

6.4.2 FINDING 2: MG-LDA is ineffective for source code

Similar challenges have been considered by a specialized model called MG-LDA\(^5\) [119], originally developed for extracting ratable aspects\(^6\) from online user reviews. In addition to topics which have a global presence in some files (called global topics), MG-LDA models topics which only occur across small text fragments in many files (called local topics). Hence we decided to investigate whether MG-LDA could be applied to detect diffused concerns in software.

However, our controlled experiment [17] showed that even MG-LDA fails to detect the diffused foreign concern. This happens even when the concern is present in 50% of the files, i.e. five lines each from the concern occur in all files with more than 100 lines. On a closer analysis, our experience with MG-LDA on software datasets uncovered two major issues which make MG-LDA unsuitable for our problem:

First, we find that a concern can only be detected as a local topic by MG-LDA, if it is present in a very large percentage of files. Unlike ratable aspects in online review data which are present widely across reviews and hence appropriately modeled as local topics by MG-LDA, cross-cutting concerns are restricted to a smaller percentage of files, which makes it difficult for MG-LDA to detect these concerns.

Second, many concerns in software packages have a typical dual presence - these concerns are used across several files (a cross-cutting presence) and defined in a separate file (a prominent modular presence). As the cross-cutting presence of these concerns is very weak statistically, the modular presence makes these concerns appear as global topics in MG-LDA instead of local topics. As they are detected as global topics, MG-LDA now fails to localize

---

\(^5\)multi-grained LDA
\(^6\)e.g. location, comfort, food, cleanliness and pricing could be ratable aspects in reviews of hotels
these concerns effectively at the statement level.

In the next section we describe a statistical topic model specifically suited to address these challenges.

### 6.5 A statistical software concern model that is sensitive to the context of individual source code statements

![Figure 6.2: Example of 3 contexts, each as a set of 3 consecutive statements in a sliding window mechanism. Statement 8 or statement 9 alone may not be clearly identified as a copy or buffering concern, but along with statement 7 it becomes more apparent.](image)

#### 6.5.1 Context Sensitive Concern Modeling (CSCM)

We now discuss a particular statistical topic model described in [17], a context sensitive concern model (CSCM)\(^7\), which (without assuming any human input) is not only able to discover concerns automatically, but is also able to localize them at a statement level.

The model assumes that spatially co-located statements may give us a context to understand the underlying concern of a statement. Hence, CSCM does not treat a statement alone, but

---

\(^7\)The model was the outcome of a joint research collaboration with Mrinal K. Das, Chiranjib Bhattacharyya and K. Gopinath. In particular, the generative process and the variational inference algorithm were developed by Mrinal K. Das but we reproduce all the relevant details here for completeness. In [17] we have called the model CSTM, or context sensitive topic model, to reflect general applicability to topics in documents; however, in this chapter, we refer to it as CSCM to emphasize its application to finding software concerns.
Chapter 6. Surfing Diffused Concerns Automatically for Probabilistic CAPA

models it in a context of neighborhood statements, where a context is defined as a set of $T$ contiguous statements in a file. As there is no physical boundary between contexts, we build a sliding window of contexts (Figure 6.2), where the first context window in a file contains only the first line of the file and each line belongs to multiple overlapping context windows. Thus, if there are $S$ number of statements in a file, there are $T + S - 1$ number of context windows in that file. Each such context is then modeled as a distribution over concerns.

CSCM incorporates two levels of abstraction, one at the file level and the other at the context level. Thus, in addition to the concerns that manifest at the context level (as described above), which we call contextual concerns, there are file level concerns, which we call modular concerns. A statement contributes to both the modular concern and the contextual concern in some proportion. CSCM provides a configuration parameter to allow control over this proportion to adjust the context sensitivity of the model. If a concern is weak enough that it does not appear at a file level, it will remain undetected by models like LDA but can be detected by CSCM, even if it is confined to the locality of few statements and present in only a relatively small proportion of files.

The details of the model are described in terms of a graphical model representation (Figure 6.3) as well as a generative process (Figure 6.4).

**Concern topics:** $\beta^m$ and $\beta^c$ are parameter matrices of sizes $K^m \times V$ and $K^c \times V$ respectively; the matrices represent the concern topic-word distributions of the $K^m$ modular and $K^c$ contextual concerns $C^m$ and $C^c$ found by the model (where $V$ is the total number of distinct words and $K^m$ and $K^c$ are specified by the user). $\beta^m_{ij}$ is probability of picking word $j$ given that $z$ is the $i^{th}$ modular concern and $\beta^c_{ij}$ is probability of picking word $j$ given that $z$ is the $i^{th}$ contextual concern. A concern can be described by the most probable words in the distribution.

**Distribution of concerns:** The parameters $\alpha^m$, $\alpha^c$ specify the Dirichlet prior distributions corresponding to the per-module and per-context concern topic proportions, respectively (i.e. $\theta^m$ and $\theta^c$). $\alpha^d$ is a 2 dimensional vector parameterizing the Beta distribution of $\pi$, the contribution of modular vs contextual concerns. $\lambda$ is a symmetric Dirichlet prior corresponding to the proportion $\psi$ in which all the contexts (windows) applicable for each statement contribute
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

Figure 6.3: Graphical model representation of CSCM. The shaded circles represent inputs and outputs of the model. The rectangles marked with M, C, S, N imply iteration over M modules, C contexts, S statements and N words. A directed arrow means that the “from” node influences the “to” node. w is the only observed variable here representing words in software code, while z represents the concerns. The suffixes “m” and “c” in the parameters distinguish modules and contexts respectively.

to the statement’s contextual concern distribution.

**Modeling context sensitivity:** To model the preference between contextual concerns and modular concerns, we use an asymmetric prior for the beta distribution $\pi$. However, as software code is highly modular in nature, we find experimentally that an asymmetric prior over $\pi$ alone is not sufficient to detect contextual concerns because the posterior inference of $\pi$ strongly depends on the data and little on the prior emphasis. The model is equipped with an external influence through $t$, which can help increase bias towards contextual concerns more effectively. $\zeta$ is the Bernoulli parameter of this external control variable $t$.

Using the configuration parameter $\zeta$ it is possible to introduce external control over the level of intensity towards discovering concerns at the context level. $\delta$ from data and $t$ from the external configuration setting together control the level of context sensitivity of the model.
For each module $f$

- Sample modular concern proportion $\theta_{df}^m \sim symmDir(\alpha^m)$
- For each statement $s$ in module $f$, pick $\psi_{fs} \sim symmDir(\lambda)$
- For each sliding context $t$ in module $f$
  - sample contextual concern proportion $\theta_{ft}^c \sim symmDir(\alpha^c)$
  - proportion of modular or contextual $\pi_{ft} \sim Beta(\alpha_1^t, \alpha_2^t)$
- For each token $w_n$ in statement $s$ of module $f$
  - sample $z_{fn} \sim Bernoulli(\zeta)$
  - select context $v_{fn} \sim multi(\psi_{fs})$
  - pick $\delta_{fn} \sim Bernoulli(\pi_{fn})$
  - if $\delta_{fn} \vee \pi_{fn} = 0$, use modular concern $z_{fn} \sim multi(\theta_{df}^m)$
  - if $\delta_{fn} \vee \pi_{fn} = 1$, get contextual concern $z_{fn} \sim multi(\theta_{df}^c)$
  - sample token $w_{fn} \sim multi(\theta_{df}^{m+c})$

Figure 6.4: Generative process of CSCM model for each module $f$ in a software application.

<table>
<thead>
<tr>
<th>$\mathcal{F}$</th>
<th>$\zeta$</th>
<th>Concern</th>
<th>Existing model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta \text{ OR } t$</td>
<td>0</td>
<td>ignores t (external influence)</td>
<td>MG-LDA</td>
</tr>
<tr>
<td>$\delta \text{ OR } t$</td>
<td>&gt; 0.5</td>
<td>more contextual</td>
<td>-</td>
</tr>
<tr>
<td>$\delta \text{ OR } t$</td>
<td>1</td>
<td>always contextual</td>
<td>-</td>
</tr>
<tr>
<td>$\delta \text{ AND } t$</td>
<td>1</td>
<td>ignores t (external influence)</td>
<td>MG-LDA</td>
</tr>
<tr>
<td>$\delta \text{ AND } t$</td>
<td>&lt; 0.5</td>
<td>more modular</td>
<td>-</td>
</tr>
<tr>
<td>$\delta \text{ AND } t$</td>
<td>0</td>
<td>always modular</td>
<td>LDA</td>
</tr>
</tbody>
</table>

Table 6.1: Choice of $\mathcal{F}, \zeta$ and its implications. If $\mathcal{F}$ is 1, contextual concerns are selected, else modular concerns are selected.

One can design a function $\mathcal{F}(\delta, t) \in \{0, 1\}$ such that if $\mathcal{F}$ is 1 then CSCM will choose contextual concerns, otherwise modular concerns. The ability to vary $\zeta$, and use various possible definitions of $\mathcal{F}$, introduces a lot of flexibility in the model. We highlight some interesting special cases in Table 6.1.

**Inferring concerns:** We have used the *variational inference* EM method detailed in [17] to infer the concern topics and document wise concern proportions based on the joint distribution of the model described by the generative process in Fig 6.4.
Localizing Concerns to Statements: We could assign concerns to statements in two ways. The naive way is to treat each statement as a module and infer the posterior distribution over the concerns for each statement. Instead, CSCM enables us to utilize the context and obtain concern proportions at both module and context level. Using posterior estimation of $\theta^m$ and $\theta^c$ we can deduce $u_{ijk}^c$ and $u_{ijk}^m$, the proportions of the $k^{th}$ cross-cutting concern or modular concern respectively for statement $S_{ij}$. We concatenate the modular and cross-cutting concern proportions to get $u_{ij}$, and then from $u_{ij}$, we assign concerns which have a high contribution to the statement as follows:

$$y_{ij} = \{k \mid u_{ijk} > \text{threshold}\} \quad (6.2)$$

### 6.5.2 Boosting capability to detect weak concerns

We have observed that if the specified number of concerns in the model ($K^m$ or $K^c$) is increased to a large number with the intent of locating weaker concerns, in many cases, instead of detecting new concerns, CSCM (as well as LDA and MG-LDA) repeats nearly the same concerns with slight variations, while many weak concerns remain un-surfaced.

Hence, following [125], an asymmetric Dirichlet prior on statement-concern distribution is introduced in CSCM to help detect weak concerns. In addition, the model tries to increase the gap between the discovered concerns using the following mechanism. After the estimation of the concerns are done in a given iteration of the inference procedure, CSCM updates the concerns as follows.

$$\beta_{ij}^c = \beta_{ij}^c \prod_{l \neq i} (1 - \beta_{lj}^c)$$

If a word has high probability in any concern, then this will reduce its probability in other concerns, whereas if a word has low probability in almost all the concerns, it will increase the probability in one of the concerns. This increases the diversity in detecting concerns which in turn helps surface relatively weaker concerns more effectively. However, the divergence is achieved at the price of coherence of concern topics.
6.6 Attributing resource usage to latent concerns

In this section, we illustrate an example of a novel application enabled by statement level concern discovery: the ability to correlate program properties that vary at statement granularity, such as its runtime resource usage, with automatically discovered latent concerns.

By jointly post-processing the output of an existing profiler and the results of our model, we can estimate the relative runtime resource consumption of latent concerns for an application or automatically discover concerns that are resource intensive (and hence potential candidates for optimization). We combine statement level resource consumption statistics with concern proportions assigned by our model to generate a concern-wise performance summary instead of summaries in terms of syntactic source code modules (e.g. methods, component packages etc). This can provide an interesting view of program performance behavior in terms of underlying functional intent, as opposed to low level implementation modules.

Since concerns can be implemented using other concerns, a richer form of summarization than a flat profile is useful. For example traditional method-wise profiling often incorporates calling context information. A calling context tree (CCT) profile (as generated by a bytecode profiler like JP2 [24]) can be converted into a concern context tree profile by mapping each level in the CCT to the concern proportions assigned to the corresponding statement. This can then be used to generate various summary views. For example, the cumulative bytecode execution cost attributed to a concern includes the cumulative resource usage of statements belonging to the concern and the methods invoked by those statements.

Let $R(S_{ij})$ be the resource usage (e.g. bytecodes executed) of statement $S_{ij}$ ($j^{th}$ statement of $i^{th}$ module).

Attributing costs in accordance with concern proportions when computing flat profiles is relatively straightforward.

Estimated flat resource usage (bytecode execution) cost of the $k^{th}$ concern

$$R_k = \sum_{ij} u_{ijk} \times R(S_{ij})$$  (6.3)
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

Accounting for statement-wise concern proportions when computing the cumulative resource usage of a concern is more tricky. A CCT node and each of its descendants may be assigned different concern proportions. If a CCT node is assigned only to the \(k^{th}\) concern, then the entire cumulative cost of the node should be attributed to the \(k^{th}\) concern. On the other hand, if the node is not assigned the \(k^{th}\) concern, then we should recursively proceed to apply the same logic to its child nodes. Thus, with probability \(u_{ijk}\), we assign the cumulative cost of \(S_{ij}\) to the \(k^{th}\) concern and with probability \(1 - u_{ijk}\) we examine its child nodes. The formula is more simply expressed in a bottom up fashion in terms of ancestor relationships by noting that the cost of a node \(S_{ij}\) should only be attributed to the \(k^{th}\) concern if \(S_{ij}\) or any ancestor node in its call chain is assigned to the \(k^{th}\) concern.

Estimated cumulative resource usage (bytecode execution) cost of the \(k^{th}\) concern

\[
R_k^{\text{cum}} = \sum_{ij} R(S_{ij})(1 - \prod_{S_{pq} \in \hat{S}_{ij}} (1 - u_{pqk}))
\]

(6.4)

where \(\hat{S}_{ij} = (S_{ij} \cup \text{ancestor}(S_{ij}))\)

These estimates are approximate, given the statistical nature of the model and statement level concern assignments.

Table 6.2 shows results from computing such a concern-wise bytecode execution summary for two benchmarks from the DaCapo suite [27], lusearch and luindex, which are both based on Apache lucene. It reports the cumulative bytecode execution cost attributed to sample concerns discovered in Apache lucene by our model CSCM. We list the top 5 words of a concern topic and assign a label to the concern for ease of interpretation. The entire process of generating the summary is fully automatic (except the choice of labels for concern topics of interest).

Note the differences in the profile for the two benchmarks.

The SEARCH and QUERY related concerns, including EXPLAIN, have a high bytecode execution cost when running lusearch, but are hardly exercised when running luindex. On the other hand the WRITE concern contributes to a significant percentage of bytecodes executed when running luindex. Some other concerns such as READER affect the execution
Table 6.2: Byte-code execution summaries computed for sample Apache lucene concerns found by CSCM. Results are shown for two benchmarks DaCapo lusearch and DaCapo luindex. It reports cumulative bytecode execution cost attributed to a concern as a percentage of total bytecodes executed by the program. Shaded rows are examples of *diffused concerns* and they are undetected by LDA-CS.

The cost of both benchmarks. As per the DaCapo benchmark descriptions, *luindex* uses lucene to index a set of documents while *lusearch* uses lucene to perform a text search of keywords over a corpus of data. Thus the results are intuitive. The *TIMING* concern is used for timing search queries (e.g. to timeout queries that might be taking too long), hence relevant for *lusearch*. The *STEMMING* concern is used when indexing words and also when parsing queries.

Some *key concerns that account for the resource usage differences*, *e.g.* *WRITE* and *TOKEN BUFFER* are *diffused concerns*. LDA was unable to detect these concerns. This highlights the importance of modeling statement context.
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Files</th>
<th>Lines</th>
<th>Vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BerkeleyDB</td>
<td>Embedded database engine</td>
<td>238</td>
<td>38954</td>
<td>2733</td>
</tr>
<tr>
<td>Apache lucene</td>
<td>Text search</td>
<td>958</td>
<td>114228</td>
<td>7869</td>
</tr>
<tr>
<td>SPECjbb2005</td>
<td>Server-side Java benchmark</td>
<td>63</td>
<td>9723</td>
<td>1444</td>
</tr>
<tr>
<td>DaCapo BLOAT</td>
<td>Bytecode-Level optimizer</td>
<td>188</td>
<td>36843</td>
<td>2553</td>
</tr>
</tbody>
</table>

Table 6.3: Four Java applications of various scales and domains used in our experiments.

6.7 Empirical Evaluation

In this section, we present findings from an empirical evaluation of the models. We use the models CSCM and LDA-CS to analyze four different Java applications (Table 6.3). We evaluate the results using experiments that expose differences in concern detection sensitivity and coverage (diversity) of the models, and a human programmer evaluation study of the interpretability of statement level concern assignments.

Parameter settings: The model parameters are selected uniformly for all the tests and models. In the experiments reported here we specified 100 concerns for each application. When using CSCM these were divided into 50 modular concerns and 50 cross-cutting contextual concerns. We have used 3 as the size of the context window, that is a sentence can belong to 3 context windows (previous, current, next). We set $\zeta = 0.9$ and specified $F$ as $\delta$ OR $t$. Further details on the settings used for other parameters and their selection is available in [17].

Pre-processing: We consider only the textual part of the source code, any syntactical elements have been removed. We have not used any linguistic tools like stemmer or parts-of-speech tagger, but only removed a set of standard English stop words\(^8\) excluding a few Java specific words such as get, set etc. We have also removed Java key-words\(^9\). Tokens like StringCopy have been split into two words String and Copy based on the position of a capital face inside a token, and all uses of capital face have been converted to small face.

\(^8\)http://en.wikipedia.org/wiki/Stop_words
\(^9\)http://en.wikipedia.org/wiki/List_of_Java_keywords
6.7.1 Key evaluation criteria

Our evaluation is designed to assess the selected models according to the following criteria (per Sec 6.2.3):

1. **Concern detection sensitivity**: *Can the method surface diffused concerns?* (Section 6.7.2).

2. **Concern coverage diversity**: *Does the method surface a diverse set of concerns?* (Section 6.7.3).

3. **Interpretability of concern assignments**: *Does the method assign concerns to relevant statements with a meaningful interpretation?* (Section 6.7.4).

6.7.2 Concern detection sensitivity

<table>
<thead>
<tr>
<th>Criterion: Can the method surface diffused concerns?</th>
</tr>
</thead>
</table>

In these experiments we use a single application, BerkeleyDB for which a set of known concerns are available from a published manual analysis [60]. Most of these identified concerns have both a modular component (e.g. a key class or interface) and cross-cutting statements, i.e. very few of these well-known concerns are diffused. To expose differences in model sensitivity to diffused concerns we design the following tests:

6.7.2.1 Inject a foreign diffused concern

In this test we insert a few (five) statements corresponding to a foreign concern (we used graphics/color related statements from JHotDraw as the foreign concern) at random positions into some randomly chosen BerkeleyDB source files.

We run our models on this modified source dataset and check whether the foreign concern is detected. We observe that CSCM is able to detect the concern when the number of altered files is only 10% of total number of files, whereas LDA fails to detect the concern even when the number of modified files includes all files with more than 100 lines (which covers around 50% of the total number of files) (Table 6.4).

---

10 Additional details about the concern injection logic used in our experiments are available in a technical report [17]
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

<table>
<thead>
<tr>
<th>Statements of foreign concern “graphics”</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class HSVColorSpace extends ColorSpace</td>
</tr>
<tr>
<td>public static HSVColorSpace getInstance()</td>
</tr>
<tr>
<td>instance = new HSVColorSpace();</td>
</tr>
<tr>
<td>super(ColorSpace.TYPEHSV, 3);</td>
</tr>
<tr>
<td>public class HSVHarmonicColorWheelImageProducer</td>
</tr>
<tr>
<td>extends PolarColorWheelImageProducer</td>
</tr>
</tbody>
</table>

Table 6.4: Example statements corresponding to a foreign concern “graphics” injected into BerkeleyDB (top). The concern is detected by CSCM, but LDA fails to detect it due to its weak presence (bottom).

### 6.7.2.2 Prune modules where a known concern is prominent

In this test, we choose a couple of known BerkeleyDB cross-cutting concerns and remove the main source files where these concerns are prominent. Now only the diffused cross-cutting statements corresponding to these concerns remain in source tree. We run our models on this pruned source dataset and check whether these two concerns are reflected in the concerns found by these models. The two concerns we chose for this experiment are “Trace” and “Memory Budget”.

<table>
<thead>
<tr>
<th>Concern</th>
<th>CSCM topic</th>
<th>LDA topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>param level util cleaner trace</td>
<td>NONE</td>
</tr>
<tr>
<td>Memory Budget</td>
<td>memory delete ret match budget</td>
<td>NONE</td>
</tr>
</tbody>
</table>

Table 6.5: Context sensitivity experiment results: Pruned BerkeleyDB test concern topics found (5 most likely words of relevant topics are listed)

Table 6.5 lists the most likely words of the relevant concerns found by the models on running these experiments. LDA on the pruned tree does not find any topics which reflect the test concerns. CSCM is able to detect both these concerns, despite the weakened and diffused
presence.

6.7.3 Concern coverage diversity

**Criterion:** Does the method surface a diverse set of concerns?

The previous experiments focused on the sensitivity of the model with respect to surfacing potentially non-trivial and interesting concerns with a diffused presence in the source. In the next set of experiments we compare models in terms of diversity of concerns found (a representative summary should cover a broad set of concerns).

6.7.3.1 Topic diversity measurement

![Figure 6.5: Jenson Shannon Divergence among concerns detected by CSCM and LDA-CS.](image)

We measure concern topic diversity quantitatively for both LDA-CS and CSCM in terms of the Jenson Shannon divergence (JSD) [70] of the concern topic-word distributions of the 100 concerns found. We used the generalized definition of JSD for more than two distributions, which computes the total divergence to the average of these distributions.

\[
JSD(\beta_1, \beta_2, \ldots, \beta_K) = H\left(\sum_i^K \pi_i \beta_i\right) - \sum_i^K \pi_i H(\beta_i)
\]

where \(H(\beta_i)\) is the Shannon entropy for distribution \(\beta_i\) and we choose the weights \(\pi_1 = \pi_2 = \ldots = \pi_K = \frac{1}{K}\). As the number of topics specified for both models is the same, a higher
value of JSD indicates a more diverse set of topics. From our results, we observe that CSCM outperforms LDA-CS in all the four applications in terms of topic diversity (Figure 6.5).

### 6.7.3.2 Coverage of known BerkeleyDB concerns

![Figure 6.6](image.png)

Figure 6.6: Out of 23 known concerns in BerkeleyDB, CSCM detects 19, and LDA-CS detects 17. All concerns detected by LDA-CS are detected by CSCM.

<table>
<thead>
<tr>
<th>Concern name</th>
<th>CSCM Concern topic (most probable words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evictor</td>
<td>evict nodes scan target bytes evictor renewed iter scanned eviction</td>
</tr>
<tr>
<td>Transactions</td>
<td>txn xid transaction active txns nxa prepare aborts commits commit</td>
</tr>
<tr>
<td>Latch</td>
<td>latch thread shared owner waiters held access exclusive stats latches</td>
</tr>
<tr>
<td>Statistics</td>
<td>bin count stats obsolete progress removed notify accumulator names dcl</td>
</tr>
<tr>
<td>ChunkedNIO</td>
<td>closed nio channel channels libraries communications job chunked log lock</td>
</tr>
<tr>
<td>Checksum</td>
<td>checksum user pre future adler implementation cksum validator anticipate assume</td>
</tr>
</tbody>
</table>

Table 6.6: Examples of BerkeleyDB concerns found by CSCM with 10 most likely words

In order to obtain a qualitative confidence in the ability of the model(s) to surface representative concerns, we also assess whether topics found by the models cover known concerns in BerkeleyDB. We make this assessment based on whether words from a known concern (feature) name appear in the top 10 words of one or more topics. Both LDA-CS and CSCM exhibit a good coverage of these concerns (Figure 6.6). As most of these known concerns have a modular component, LDA-CS is able to detect them. We observe that CSCM surfaces all
the known concerns found by LDA-CS plus a few additional concerns, e.g. ChunkedNIO (a diffused concern). Table 6.6 lists examples of the concern topics surfaced by CSCM.

### 6.7.4 Programmer interpretability (human evaluation) study:

**Criterion:** Does the method assign concerns to relevant statements with a meaningful interpretation?

Assessing the quality of the latent concern structure and the accuracy of fine-grained assignments surfaced by these models involves a subjective judgment that requires expertise possessed by experienced programmers. We conducted a human evaluation study by designing multiple choice questions to quantify “programmer” interpretability of concern assignments, in terms of metrics for word intrusion and statement topic mapping relevance, using a suitably modified version of the methodology recommended in [33].

In order to explain the rationale behind our chosen methodology for evaluating this criteria, we next discuss some of the alternatives we considered from the state-of-the-art in topic model evaluation.

#### 6.7.4.1 Evaluating Topic Models: Choice of Methodology

Devising a suitable evaluation methodology to assess the outcome of topic modeling is non-trivial. The latent variables represent distributions over words – they are called “topics” only to reflect anecdotal experience that words co-occurring in a “topic” with a high probability are semantically connected to a common theme. This internal representation is difficult to validate directly because of the lack of ground truth to compare against. Instead, different evaluation criteria have been adopted in topic modeling literature [29, 33, 126, 93, 118], depending on the intended purpose, such as:

- Measures based on held-out likelihood (or perplexity\(^{11}\)) that quantify how well the model learned from a corpus predicts the statistical characteristics of unseen documents or unseen parts of partly seen documents (document completion).

\(^{11}\)the perplexity of a held-out test set \(exp \left\{ - \frac{\sum_{d=1}^{M} \sum_{n=1}^{N_d} \log(p(w_{d,n}))}{\sum_{d=1}^{M} N_d} \right\} \) is monotonically decreasing in the likelihood of the test data; a lower perplexity score indicates better generalization performance [29].
• Secondary measures that evaluate the use of the model for an external task independent of the topic space, such as information retrieval. For example, one such measure could be the performance of a classifier that uses the topics as features with topic proportions representing feature vectors of sampled documents. Here, the latent space inferred by the topic model represents a dimensionality reduction of the feature space of the document collection.

• Qualitative assessment that illustrates whether topics inferred are semantically meaningful. For example, many topic modeling papers present samples of topics found by the model – usually the ten most likely words in each topic are displayed to enable readers to judge the quality for themselves. The semantic content attributed to topics is important when the output is intended for human understanding, e.g. consider the automatic categorization, summarization and annotation of a large corpus of articles by their themes.

• Measures based on human evaluation that quantify the interpretability of the internal representation of the model, i.e. the extent to which both the topics themselves and the document-wise topic assignments are semantically meaningful. For example, Chang et al. introduced the use of word intrusion and topic intrusion tests [33] as a human evaluation methodology – a practical approach that goes beyond a purely qualitative assessment to obtain a quantitative comparison of the interpretability of different models.

For the work described in this chapter, we use the fourth approach above. We employ a variation of the methodology proposed by Chang et al. [33] for assessing model interpretability. Interpretability of the latent structure of topics learnt is important when topic models are employed for an exploratory purpose as in our intended application – performance understanding of software concerns. Although objective evaluation measures based on held-out likelihood provide the most easily generalizable techniques to assess topic models, it has been observed that models that perform better on these measures may result in less interpretable topics [33]. There has been some effort to identify alternate objective metrics that could be used as a better proxy for judging topic interpretability e.g. point-wise mutual information [93] or other indicators of semantic coherence [83] of words in a topic. When this works well, it could save
the need for a human evaluation. However, in our experimental trials we had mixed success in using or devising semantic coherence metrics as a consistent predictor of interpretability software code topics, especially across topics identified by different models and over different document collections.

6.7.4.2 Word intrusion and Topic intrusion measures

Chang et al. [33] describe two types of human evaluation tests which can be used to measure interpretability of the latent space discovered by a topic model.

The word intrusion test measures the conceptual coherence of the inferred topics according to human interpretation. The experiment involves inserting an intruder word at a random position in the list of top five words of a topic and testing whether participants are able to detect the odd word out (i.e. whether subjects agree with the model on the intruder word). The intruder word is selected so that it has a low probability in the topic being evaluated, but has a high probability in some other topic (so that it is not ruled out purely because it is a rare word in the corpus).

The topic intrusion test measures whether the topics assigned to a document by the model agree with human judgement. The experiment involves inserting an irrelevant intruder topic at a random position in the list of the three highest probability topics assigned a document and testing whether participants are able to detect the least relevant topic from the list when shown snippets from the document. Each topic in this list is shown as a set of the most likely words corresponding to that topic. The intruder topic is selected randomly from the other topics in the model which have a low probability in the document.

6.7.4.3 Adapted methodology

We make the following main adaptations to the methodology to make it suitable for assessing statement level concern assignments:

- We use statement topic mapping relevance instead of topic intrusion for assessing concern assignments. Unlike a text document, a code statement is unlikely to be assigned
to more than one or two topics. Hence, asking respondents to choose the most relevant topic is more appropriate than detecting the least relevant topic. The percentage of responses that agree with the model provides a measure of statement topic mapping relevance.

- Instead of judging the concern assignment of a statement in isolation, we provide a snippet of about 5 code statements that are assigned to the same concern (preferably from the same file, but potentially non-contiguous lines) so that the participant has some context.

- We add optional fields in the questionnaire for users to fill in a label or assign a name to topic word groups and statement groups. Unlike natural language text, topics for concerns can include terse or obscure words and program or domain specific terms that are difficult to interpret without application knowledge. Hence detecting intruder words can be a problem even for experienced programmers. In such cases, the labels of a topic and its matching statement group act as a secondary indication of the topic’s interpretability. When the labels assigned by different programmers are consistent we can conclude that the topic is interpretable even when an intruder word cannot be detected.

We illustrate a few sample questions from the study:

**Sample questions:**

There are 2 sections A and B. Please answer both sections.

This will be evaluated automatically using a script, kindly write your answer after the ":" symbol only

**SECTION A: Find the odd one out.**

Each word set in the questions below contains terms or words observed in code statements (or comments) related to a particular purpose.

There is one word that does not belong to the set. Can you guess which one it is? Can you guess the purpose or assign a name to the set?
**Hint:**

The words in a set are typically sorted in order of relevance to the purpose, except for the odd word that is introduced at a random position index.

---

Instruction: Pick the index of the odd word, and give a name to the set.

---

Words set 1

1. stats
2. accumulator
3. walker
4. tree
5. acc
6. parse

ANSWER [give the index]: 6
LABEL [give a name to the set]: stats

---

Words set 2

1. cleanup
2. time
3. millis
4. now
5. completed
6. match

ANSWER [give the index]: 1
LABEL [give a name to the set]: time

---

SECTION B: Match the statement set provided with the most relevant word set and the least relevant word set from the 4 choices given

Each statement set in the questions below contains code statements (or comments) that reflect similar or related purpose, but sampled from multiple files in a single application. Can you guess the purpose or assign a name to the set?
Each word set listed in the choices contains terms or words observed in code statements (or comments) related to a particular purpose (the words are sorted in order of importance to the purpose).

Can you guess which word set from the choices is the most relevant match for the statement set? Can you identify the least relevant match?

Instruction: Pick the index of the most relevant and least relevant choices, and give a name to the statement set.

Statements set 1

* Proxy to Cursor.getCursorImpl()
* Proxy to EnvironmentConfig.setTxnReadCommitted()
* Proxy to EnvironmentConfig.cloneConfig()
* Proxy to EnvironmentMutableConfig.validateParams.
* Proxy to DatabaseConfig.match(DatabaseConfig())

Choices
-------
1. stats accumulator walker tree acc
2. parse error enabled the get
3. pool methods versions large buffers
4. environment config txn env properties

Most relevant [give the index]: 4
Least relevant [give the index]: 1
LABEL [give a name to the set]: proxy config

Statements set 2

exactSearch, lockType, bin.getLsn(index));
if (lockResult.getLockGrant() != LockGrantType.DENIED) {
    return lockResult;
* Try a non-blocking lock first, to avoid unlatching. If the default
lockResult = locker.nonBlockingLock
6.7.4.4 Study results

35 programmers participated in our study, including experienced Java programmers from multiple software development organizations as well as 12 computer science research students with a strong programming background. The questions were divided into questionnaires containing a set of simple tasks for determining word intrusion and statement topic mapping relevance. Different questionnaires were created covering samples of topics surfaced across the four applications on running the two model variations\(^{12}\), LDA-CS and CSCM. The question sets were generated using automated scripts applied to the model outputs. We circulated a larger number of copies of the questionnaires created based on CSCM output to potential respondents in order to focus more attention on evaluating the newer model.

<table>
<thead>
<tr>
<th></th>
<th>CSCM</th>
<th>LDA-CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of responses</td>
<td>171</td>
<td>85</td>
</tr>
<tr>
<td>No. of responses matching model</td>
<td>124</td>
<td>58</td>
</tr>
<tr>
<td>% matching responses</td>
<td>72.5%</td>
<td>68.2%</td>
</tr>
</tbody>
</table>

Table 6.7: Interpretability of statement level concern assignments (most relevant topic)

A total of 540 individual task responses were collected (a single programmer had the option of answering one or two orthogonal questionnaires, a total of 20 questions at the most), 345

\(^{12}\) to contain the scale of expert effort required, we limit the number of concern topics sampled for evaluation to 40 per model, i.e. 10 from each application
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

<table>
<thead>
<tr>
<th></th>
<th>CSCM</th>
<th>LDA-CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of responses</td>
<td>174</td>
<td>110</td>
</tr>
<tr>
<td>No. of detections of intruding word</td>
<td>71</td>
<td>33</td>
</tr>
<tr>
<td>% agreement with model on intruding word</td>
<td>40.8%</td>
<td>30%</td>
</tr>
<tr>
<td><strong>Topic interpretability</strong> (% responses that indicate concern topics to be interpretable)</td>
<td>72.8%</td>
<td>63%</td>
</tr>
</tbody>
</table>

Table 6.8: Concern topic interpretability results: ability to detect intrusion words or assign consistent labels to concern topic words or its relevant statements

responses for CSCM and 195 responses for LDA-CS.

Table 6.7 and Table 6.8 summarize the consolidated results. We observe that statement level concern assignments are interpretable to a similar extent for both LDA-CS and CSCM, with about 70% responses that match the model. The word intrusion score, i.e. percentage agreement with the model in word intrusion detection, is comparatively low for the same topics even though we find that in many cases programmers were able to assign labels to the topic words or to the corresponding statement sets. To compute concern topic interpretability, we use the labeling consistency score (percentage of consistent labels) for a concern topic when the topic has a word intrusion score lower than 60% and at least 50% of its labels are consistent. Otherwise we rely on the topic’s word intrusion score. We observe that LDA-CS and CSCM exhibit 60-70% topic interpretability for the sampled concern topics used in the study.

Researchers have previously pointed out that not all the topics inferred for a given document collection are interpretable [83]. Rather topic models produce a mix of topics, some of which are coherent and others less so – the higher the number of topics specified for a given collection, the wider the mix. The notched boxplots in Fig 6.7 complement Tables 6.7 and 6.8, by providing a perspective of how the interpretability results vary across the sampled topics for LDA-CS and CSCM. The overlap between the notches indicates that differences in the medians may not be statistically significant. We also notice a lower variation in the CSCM results compared to LDA-CS for the sampled topics.

Increasing the number of topics increases the resolution of the model to help unearth specialized topics.
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

6.7.5 Summary

Table 6.9 summarizes our evaluation findings along the dimensions of concern detection sensitivity, concern diversity and interpretability of statement level concern assignments. CSCM exhibits better concern detection sensitivity and diversity and both LDA-CS and CSCM show 60-70% agreement with programmer interpretation in their statement level concern assignments.

Figure 6.7: Notched boxplots of the interpretability of statement level concern (topic) assignments (left) and concern (topic) interpretability (right) for CSCM and LDA-CS. These plots illustrate the variation in interpretability across topics, complementing the results in Tables 6.7 and 6.8 which showed the averages aggregated over all responses. The interpretability value (Y-axis) computed for each topic represents the fraction of subjects whose responses indicate that they perceive that topic to be interpretable.

Table 6.9 summarizes our evaluation findings along the dimensions of concern detection sensitivity, concern diversity and interpretability of statement level concern assignments. CSCM exhibits better concern detection sensitivity and diversity and both LDA-CS and CSCM show 60-70% agreement with programmer interpretation in their statement level concern assignments.
Chapter 6. Surfacing Diffused Concerns Automatically for Probabilistic CAPA

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Evaluation question</th>
<th>LDA-CS</th>
<th>CSCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern detection sensitivity (Table 6.4, 6.5)</td>
<td>Can the method surface diffused concerns?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concern coverage diversity (Figure 6.5, 6.6)</td>
<td>Does the method surface a diverse set of concerns?</td>
<td>Good, but only for prominent concerns</td>
<td>Better than LDA-CS</td>
</tr>
<tr>
<td>Interpretability of concern assignments (Table 6.7, 6.8, Fig 6.7)</td>
<td>Does the method assign concerns to relevant statements with a meaningful interpretation?</td>
<td>About 60-70% of the time</td>
<td>As good as LDA-CS</td>
</tr>
</tbody>
</table>

Table 6.9: Evaluation summary

```java
if (PreviousMaxWarehouses == 0)
    MaxWarehouses = numberofwarehouses;
else
    ++MaxWarehouses;
String msg = Loading Warehouse MaxWarehouses
System.out.println(msg);
JBButil.getLog().info(msg);
// Item Table must be loaded first since
if (PreviousMaxWarehouses == 0) {
    loadItemTable();
}
```

Table 6.10: Lines from SPECjbb2005/spec/jbb/Company.java, colored based on concerns

### 6.8 Discussion

A major challenge in statistical modeling is that there is no control over the concerns being detected. If we merely increase the number of topics, the same concerns can repeat multiple times, whereas many important concerns remain undetected. We have focused on this issue by using CSCM to introduce external control and boosting the divergence among concerns detected. This clearly shows an improvement over state-of-the-art statistical models in detecting statement level concerns (Table 6.10).

**Analysis time and scalability** CSCM is a more complex model than LDA, and hence requires an increased analysis time for large applications. However, CSCM can be made scalable using online mechanisms [55].
Semantic Coherence Although CSCM can find meaningful concerns, it is well-known [83] that not all topics found by a statistical model may be meaningful or coherent. This can be addressed to an extent by filtering out topics which exhibit a relatively low semantic coherence [83].

Future work This chapter barely scratches the surface in terms of potential applications combining information on latent concerns with program properties. For example, in the previous chapter, we have experimented with using the model to help highlight concerns responsible for high object churn, a common form of runtime bloat in Java applications. Concerns and their resource usage proportions could be included as features in models for estimating performance / power consumption and for mining resource intensive concern usage patterns.

6.9 Related Work

Statistical topic models: Latent Dirichlet Allocation (LDA) [29] is a well known topic model which has been successfully applied in various fields. [71, 11] introduced the use of LDA for analyzing and mining software code to discover and model program concerns as latent topics without any apriori knowledge or expert input. Several variations of these techniques [7, 74], delta-LDA [3] and relational topic models [48] have been used for addressing software maintenance tasks such as computing conceptual coupling metrics, statistical debugging and software evolution, besides program comprehension and reverse engineering problems. To the best of our knowledge, we are the first to explore the possibility of using such models for performing automated summarization of runtime resource usage or other program properties in terms of latent concerns.

There has been some recent work on improving LDA to handle sparseness of short documents, by aggregating short documents into a larger text [56, 129] or incorporating large scale external data [100]. [144] proposed a model for sentence based summarization. However, none of these address detection of diffused concerns.

Other concern analysis techniques: There is a large body of existing literature on concern identification and location besides topic models. A variety of techniques ranging from
formal concept analysis, exploiting program topology [106], information retrieval, graph mining, program slicing [53, 25] and dynamic analysis [39] have been employed in this context. Solutions that combine multiple approaches [104, 146, 109] and exploit multiple sources of information have also been used to improve the quality of results. However, unlike statistical topic models, most of these techniques assume that some information about the desired concern is provided to begin with, such as feature names, structural attributes, search patterns, bug reports, testcases or execution traces. Hence we find them less suitable for purposes of unsupervised automated performance summarization. Also, a statement may contribute to multiple concerns, in which case we need to attribute properties such as statement execution cost proportionately to these concerns. A statistical topic model provides a natural probabilistic framework to infer these proportions in terms of the probabilities assigned to different concern topics.

6.10 Conclusions

The ability to automatically discover representative latent concerns at the level of individual code statements can enable a new class of automated analyses using the CAPA framework described in Chapter 5. Statistical topic models such as LDA have emerged as a popular tools for automatic concern discovery. However, we find that LDA and even specialized models such as MG-LDA, which have been applied successfully in other domains, fail to detect diffused concerns that occur only at a statement level without a prominent presence in any module. Based on insights gained from these experiences, we adopt a different statistical model, called CSCM, that addresses these challenges and present an application of the model in automated summarization of bytecode execution profiles. We observe that diffused concerns can indeed account for a significant differences in resource usage under different execution scenarios. We assess CSCM using a systematic evaluation methodology which confirms the effectiveness of the model in comparison with LDA. The work presented in this chapter is also of independent interest as an important step towards the invention of sophisticated analysis tools by combining information about underlying intent (as represented by latent concerns) with dynamic or static
properties of programs.

**Acknowledgments for this Chapter**

We thank all the respondents of the programmer interpretability study for their contribution to this research. In particular we thank Albee Jhoney, Jojo Joseph, Mangala Gowri Nanda, Nayna Jain, Sreerupa Sen, Prateek Sarkar and Debashish Chakraborty for also helping us reach out to experienced (Java) programmers from diverse groups in different organizations for their inputs. We also thank M. Narasimha Murthy and Girish Maskeri for their feedback and discussions.
Chapter 7

Putting It All Together - A Systems View

We integrate our findings from previous chapters to guide bloat mitigation efforts for power-performance optimization. To enable a principled design approach to reduce bloat, we introduce the notion of resource proportionality of software features as a measure of bloat propensity.

Energy-proportional computing [14] has been proposed as a principled design approach to achieve significant energy savings in server systems [122, 121]. Is there a similar principle that is applicable to the problem of bloat reduction?

The key observation behind the notion of energy proportional design is that inefficiencies arise because data centers must often be provisioned to support very high peak demands, when they actually require only a small fraction of the available computing capacity in typical situations. Such over-provisioning of resources need not necessarily have caused an energy wastage if only the hardware components (servers, cooling equipment, power supplies etc) were truly energy proportional - i.e. if the components consumed power only to the extent they are actually utilized. Thus, designing components to be more energy proportional can enable large energy savings with simpler power management logic [14]. As it is very hard to achieve this in general for each component, techniques have been proposed to achieve effective aggregate energy proportional behaviour from ensembles [121] of non energy proportional components.

Developing a systematic scheme to address energy inefficiencies due to software bloat requires a new perspective that goes beyond these principles. However, we notice a parallel
between energy waste due to hardware overprovisioning in a non-energy proportional system and overheads arising from the functional overprovisioning of software. Enterprise applications must support extremely demanding levels of variability and interoperability, but they actually exploit only a small fraction of this versatility in a typical deployment situation. Ideally, if software features could be designed so that the runtime resource usage of an application is only a function of the exploited fraction of features\(^1\), efficiency would be achievable without losing flexibility.

With this view, we propose a notion called *resource proportionality* of software features to enable a principled design approach to reduce energy wastage due to software bloat. Resource proportional features/concerns allow software to be provisioned to support a very high level of versatility in terms of features (or concerns) but still incur minimum bloat by consuming computing resources in proportion to what would have been required to support only the features (concerns) actually exploited in a given deployment scenario.

Minimizing the runtime overhead expended on unused built-in generality is critical to achieving resource proportional software features. Though we don’t yet fully understand how to model all such costs, the findings in this thesis can provide some guidance to help address some of these overheads. In the previous chapters we independently explored the following intermediate links from the origin of bloat to its energy efficiency implications.

- the extent to which sources of bloat may arise from excess concerns
- how resource bloat can be mitigated for repeatedly executed sources of bloat
- the extent to which power-performance is impacted by resource bloat

We can now combine the insights into systematic approach to reason about bloat and the different levels at which it could be mitigated.
### 7.1 Towards systematic strategies for bloat mitigation

As illustrated in Figure 7.1, the following are some of the factors\(^2\) that jointly determine how runtime bloat arises from overprovisioned features and impacts power-performance.

**Software design/code characteristics**

- Structural interactions between optional features and the rest of the code, which represent code statements or fields that are potential bloat contributors

- Structural recurrence granularity of these interactions (e.g. nesting depth of loops containing these potential contributors) compared to the rest of the code

---

\(^1\) i.e. the remaining fraction does not induce a runtime overhead  
\(^2\) designated by labelled arrows in the Figure
Software deployment or runtime environment characteristics

- Features unexploited by the deployment scenario (excess features)
- Dynamic recurrence frequency of the relevant structural interactions corresponding to these unexploited features
- Hardware system characteristics (resource bottlenecks and energy proportionality)

<table>
<thead>
<tr>
<th>Control</th>
<th>De-bloating optimization</th>
<th>Applicable techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reduce overprovisioned features</td>
<td>Build/refactor purpose specific variants (e.g. software product line, make config)</td>
</tr>
<tr>
<td>2</td>
<td>Reduce structural interactions from optional features and optimize incidental concerns</td>
<td>Use bloat diagnostic tools to aid manual optimization ([138, 140], Chapter 5)</td>
</tr>
<tr>
<td>3</td>
<td>Amortize recurring overheads</td>
<td>Automated compiler/runtime optimizations e.g. object reuse (Chapter 4), tool aided invariant hoisting [142], memoization and layer folding [66], partial evaluation</td>
</tr>
<tr>
<td>4</td>
<td>Enable high overhead features on-demand</td>
<td>Runtime feature adaptation, lazy evaluation (also see Chapter 8)</td>
</tr>
<tr>
<td>5</td>
<td>Tune system power-performance</td>
<td>Power manage non-bloated resources (if they are power hungry) and reduce bottleneck strain for bloated resources (Chapter 3)</td>
</tr>
</tbody>
</table>

Table 7.1: Different levels of intervention to contain energy wastage due to bloat

Thus, energy efficiency loss due to bloat can be contained in many different ways by applying suitable interventions at any of the several potential control points marked by the numbered circles in Fig 7.1 and elaborated in Table 7.1. Only the first involves sacrificing flexibility and/or productivity. The second can affect development productivity but the effort can be reduced with the aid of tools. The fourth can affect deployment productivity as it requires some
way to determine which features are likely to be actually used at runtime. As an area of future work, further automation and software modeling enhancements could be explored to help alleviate the tradeoffs in each approach.

For a given deployment, the largest gains from de-bloating are expected for those structural interactions which are due to unexploited features, have a high dynamic recurrence frequency and either affect a steeply energy proportional power-hungry resource or affect a bottleneck resource in a system where non-bottleneck resources are power-hungry and non energy proportional. A systematic de-bloating strategy would focus on these opportunities first, via automated optimization and then using bloat diagnosis tools to guide manual code changes if still worthwhile. As detailed information about unexploited features or concerns is difficult to obtain automatically, an alternative strategy would be to first identify expensive inter-method structural interactions which have significant power-performance impact and then manually determine if methods that induce those interactions correspond to unexploited features.

### 7.2 Resource proportional features to reduce bloat propensity of software

In order to enable a principled approach to assess the propensity for bloat of a given software implementation, we now outline an approach for quantifying bloat propensity and develop the notion of resource proportionality that we introduced earlier in this chapter. This provides a foundation for performing a simple “what-if” analysis to compare alternative strategies for bloat reduction.

In general, the term resource proportionality reflects the degree to which resources used by a system scale in proportion to function accomplished. In performance (scale up) and energy efficiency (scale down) studies, the function accomplished is typically interpreted in terms of load, i.e. volume of transactions (work) processed. In these cases, non-proportional behavior may arise due to the presence of bottlenecks and startup overheads (which hurt scale up) or unreleased resources (which hurt scale down).

In our study of bloat, function accomplished is interpreted in terms of features exploited
(utilized). Thus, the resources used by the system should be in-proportion with features that are actually utilized (rather than all the features that the system is configured to support). In this case, as illustrated in Fig 7.1, non-resource proportional behavior can arise when the structural interactions due to an unexploited feature occur at a recurrence granularity that is significant compared to the rest of the code.

However, at the outset, it is not clear how this intuitive notion of resource proportionality in terms of feature utilization should be quantified. Features may not only be of different sizes but are also qualitatively different. Thus, unlike load based measures, the number of features exploited cannot be used as a uniform scale against which resource proportionality can be measured quantitatively. In the following subsection, we describe how we address this issue to arrive at suitable measures for feature utilization and propensity for bloat.

7.2.1 Quantifying feature utilization and bloat propensity

Feature utilization scale: To construct a scale of feature utilization, we first compute the resource usage of a specialized version of software that is configured only with the utilized features and then normalize it with respect to the resource usage of the fully configured version when all features are utilized. We can now assess how resource proportional the fully configured version of the software is by plotting against this scale the actual resource usage of the fully configured version as its utilized features are increased. For convenience, we also normalize this resource usage with respect to the value at 100% feature utilization, i.e. the resource usage of the fully configured version when all features are utilized.

We introduce the following notation:

$\Omega_{X}(Y)$ denotes a given deployment scenario for a software component that is configured to support a set of available features $X$, where a subset of these features $Y \subseteq X$ are actually exploited (utilized).

$R(\Omega)$ denotes the runtime resource cost of a deployment scenario $\Omega$, e.g. $R(\Omega_{X}(Y))$ is the runtime resource cost of scenario $\Omega_{X}(Y)$.

$\Delta R(Z|\Omega)$ denotes the runtime resource cost attributed to a sub-set of available features $Z$ under a deployment scenario $\Omega$. 
We assign $\Delta R(Y|\Omega_X(Y)) = R(\Omega_Y(Y))$ as the runtime resource cost attributed solely to the utilized features $Y$, where $\Omega_Y(Y)$ denotes a deployment scenario for a specialized configuration of the software component which supports only the utilized features $Y$.

Now, we can define the feature utilization $\mu$ of the deployment scenario $\Omega_X(Y)$ as follows:

$$\mu_X(Y) = \frac{\Delta R(Y|\Omega_X(Y))}{\Delta R(X|\Omega_X(X))} = \frac{R(\Omega_Y(Y))}{R(\Omega_X(X))} \quad (7.1)$$

**Bloat due to unexploited features:** Using the above notation, the runtime resource overhead (bloat) incurred due to the presence of a set of unexploited features $Z \subseteq X$ under the execution scenario $\Omega_X(Y)$ is $\Delta R(Z|\Omega_X(Y))$.

Note: In the special case where $X = Y \cup Z$, we have

$$\Delta R(Z|\Omega_X(Y)) = \Delta R(Z|\Omega_{Y\cup Z}(Y))$$

$$= R(\Omega_{Y\cup Z}(Y)) - R(\Omega_Y(Y))$$

$$= R(\Omega_X(Y)) - R(\Omega_Y(Y))$$

Let $F$ be a subset of features (belonging to a program or component) which are unexploited in some deployment scenario where the remaining features $\overline{F}$ are exploited.

Let $R_{\text{specialized}}(F) = R(\Omega_{\overline{F}}(F))$ be the resource cost incurred by features $\overline{F}$ alone excluding any overheads due to the rest of the features, i.e. the resource cost that would have been incurred by a specialized variant that only supports features $\overline{F}$.

Let $R_{\text{overhead}}(F|\overline{F}) = \Delta R(F|\Omega_{F\cup \overline{F}}(F))$ be the resource overhead (bloat) incurred due to the presence of unexploited features $F$ when only features $\overline{F}$ are used.

Let $R_{\text{actual}}(F) = R(\Omega_{F\cup \overline{F}}(F))$ be the actual resource cost incurred in this deployment scenario. Thus,

$$R_{\text{actual}}(F) = R_{\text{overhead}}(F|\overline{F}) + R_{\text{specialized}}(F)$$

We define the percentage bloat $\beta$ incurred due to features $F$ as the fraction of resource
consumption attributed to the overhead induced by $F$:

$$
\beta(F|\overline{F}) = \frac{R_{\text{overhead}}(F|\overline{F})}{R_{\text{actual}}(F)} \quad (7.2)
$$

$$
= \frac{\Delta R(F|\Omega_{F\cup\overline{F}}(F))}{R(\Omega_{F\cup\overline{F}}(F))} \quad (7.3)
$$

**Bloat propensity:** We define the resource bloat propensity factor $b$ incurred due to features $F$ as the relative increase in resource consumption attributed to the overhead induced by $F$ over that required to support only $\overline{F}$:

$$
b(F|\overline{F}) = \frac{R_{\text{overhead}}(F|\overline{F})}{R_{\text{specialized}}(F)} \quad (7.4)
$$

$$
= \frac{\Delta R(F|\Omega_{F\cup\overline{F}}(F))}{R(\Omega_{F\cup\overline{F}}(F))} \quad (7.5)
$$

If the unbloated version of the software which exploits features $\overline{F}$ consumes $r$ amount of resources, then the bloated version that is overprovisioned with additional features $F$, consumes $r(1 + b)$ amount of resources.

**Computing bloat propensity when $R_{\text{specialized}}$ is not directly available:** One difficulty with adopting the above measures in practice is the requirement for a specialized version of software that is configured only with the utilized features in order to compute $R_{\text{specialized}}$. This is impractical for applications which are not created using a product line or feature-oriented programming approach (such as the use of pre-processor directives or externally maintained information that can be used to unconfigure features at compile time).

However, it is possible obtain an estimate of bloat propensity without imposing this requirement by using the analysis techniques developed in the previous chapters.

Let $r(s)$ be the resource usage directly attributed to a statement $s$ during program execution and let $r_{\text{cum}}(\psi)$ be the cumulative resource usage collectively attributed to a set of statements $\psi$, i.e. to all statements $s \in \psi$ and the methods called by these statements. Using a reasoning
similar to that discussed in Sec 6.6:

$$r^{cum}(\psi) = \sum_{\{s | s \in \psi \text{ OR ancestor}(s) \in \psi\}} r(s)$$

If we can locate the structural interaction statements which are potential bloat contributors due to a set of optional concerns/features $F$ (e.g. using the technique in Chapter 5) then the resource bloat (overhead) contributed by these concerns when they are not needed can be computed as follows:

$$\beta(F | F) = \frac{r^{cum}\left(\{s | \text{potentialbloat}(s)\}\right)}{R^{\text{actual}}(F)}$$  \hspace{1cm} (7.6)

$$b(F | F) = \frac{r^{cum}\left(\{s | \text{potentialbloat}(s)\}\right)}{R^{\text{actual}}(F) - r^{cum}\left(\{s | \text{potentialbloat}(s)\}\right)}$$  \hspace{1cm} (7.7)

where $\text{potentialbloat}(s)$ is true when a statement $s$ directly contributes to bloat (as per the rules in Sec 5.5).

Thus, we can lower the bloat propensity factor of a given software implementation in two ways:

- by reducing the number of potential bloat contributing statements (control point 2)
- by reducing their cumulative resource utilization $r^{cum}(s)$ overhead (control point 3) (e.g. by optimizing recurring allocation costs induced by $s$ as in Chapter 4).

### 7.2.2 Tradeoff between feature exploitation and provisioning overhead

**Resource proportionality:** Let $R^{\text{full}} = R(\Omega_{F \cup F}(F \cup F))$ be the resource usage that would have been incurred in a deployment scenario where all features supported by the configuration are utilized. As feature utilization $\mu(F) = \frac{R^{\text{specialized}}(F)}{R^{\text{full}}}$, the actual resource usage varies as:

$$R^{\text{actual}}(F) = R^{\text{overhead}}(F | F) + \mu(F) \ast R^{\text{full}}$$
i.e. \( R_{\text{overhead}} \) determines the extent to which resource usage deviates from being proportional with respect to feature utilization.

Now, consider a change in the deployment scenario, after which features \( F \) are exploited in addition to features \( \overline{F} \).

\[
R_{\text{actual}}(F \cup \overline{F}) = R_{\text{exploit}}(F|\overline{F}) + R_{\text{actual}}(\overline{F})
\]

\[
= R_{\text{exploit}}(F|\overline{F}) + R_{\text{overhead}}(F|\overline{F}) + R_{\text{specialized}}(\overline{F})
\]

where \( R_{\text{exploit}}(F|\overline{F}) \) is the additional resource cost incurred when features \( F \) are exploited in addition to features \( \overline{F} \).

When a feature is not exploited, its contribution to resource consumption is determined by its provisioning overhead; when it is exploited, then its contribution to resource consumption is determined by its exploitation cost in addition to its provisioning overhead. With dynamically reconfigurable features, for example, the provisioning overhead may be lower \((R_{\text{overhead}} < R_{\text{overhead}})\), reducing the bloat propensity factor due to unused features. However, the exploitation cost may include a feature activation overhead in addition to base in-use resource demand \((R_{\text{exploit}} > R_{\text{exploit}})\).

\[
R_{\text{actual}}^{\text{dyn}}(F \cup \overline{F}) = R_{\text{dyn}}^{\text{exploit}}(F|\overline{F}) + R_{\text{dyn}}^{\text{overhead}}(F|\overline{F}) + R_{\text{specialized}}^{\text{dyn}}(\overline{F})
\]

Thus \( R_{\text{dyn}}^{\text{actual}}(F \cup \overline{F}) \) could be higher than \( R_{\text{specialized}}(F \cup \overline{F}) \).

**Resource proportionality characteristics** Fig 7.2 illustrates examples of different types of resource proportionality characteristics which may be exhibited by software implementations. On the X-axis we plot feature utilization \( \mu \) for an increasing sub-set of features and on the Y-axis we plot the corresponding resource usage \( R_{\text{actual}}^{\text{actual}} \) normalized wrt to \( R_{\text{full}}^{\text{full}} \).

S1 depicts an implementation whose features are completely non-resource proportional, i.e. the resource usage cannot be scaled down even when only a small fraction of its features are exploited. S2 depicts a typical implementation with some amount of bloat, where some overhead is incurred due to unexploited features, but resource usage can still be scaled down...
Figure 7.2: Different types of resource proportionality characteristics. The curves plot the normalized actual resource usage of a fully generalized implementation as the feature utilization $\mu$ increases. The resource usage (Y-axis) is normalized wrt to the resource usage when all the features are utilized.

to an extent when less features are used, e.g. according to a sub-linear resource proportionality characteristic. S3 depicts an implementation with a steep resource proportionality characteristic, e.g. using dynamically reconfigurable features, where the resource consumption increases sharply as more features are used because of the feature activation overhead. Such an implementation incurs the lowest bloat but is resource efficient only as long as the fraction of features exploited is small.

7.3 Related Work

7.3.1 Lessons from architecture research

The operational perspective developed in this chapter is inspired by related work in hardware or system architecture research on the tradeoff between energy efficiency and flexibility.
Power-awareness metric  Bharadwaj et al. [15] introduced a power-awareness metric for evaluating and constructing power-aware VLSI designs, where power-awareness $\phi$ is a measure of how close a system is to the most power efficient system that could be designed specifically for a given operating point, summarized across all operating points in the desired range of scenarios supported, weighted by their likelihoods. According to their definition, “a system $H$ is perfectly power aware iff its energy dissipation in scenario $s_i$ is no greater than that of a dedicated system $H_{s_i}$ constructed to execute scenario $s_i$ as efficiently as possible.”

$$\phi = \frac{\sum_{\text{scenarios}} E(H_{\text{perfect}}, s_i) d_i}{\sum_{\text{scenarios}} E(H, s_i) d_i}$$

where $d_i$ is the likelihood (distribution) of scenario $s_i$.

This metric ensures that a higher weight is given to most common (typical) scenarios. Thus it is possible that the most energy proportional system may not necessary be the most power-aware alternative.

Our characterization of resource proportionality in software features and the resource bloat propensity factor are modeled using similar principles. However, basing measures of bloat on the notion of a dedicated software system $H_{s_i}$ constructed to execute scenario $s_i$ as efficiently as possible is impractical (e.g. consider Blum’s speedup theorem). Further, we are mainly interested in systematic approaches for avoiding overheads that arise in connection with overprovisioning of features as opposed to algorithmic efficiency issues. Hence, we chose to constrain the baselines for desired scenarios to a well-defined set of specialized variants that implement the exact subset of features required to execute each scenario $s_i$.

Requirements aware energy scaledown  Energy scale down studies [77, 40] have demonstrated how the power consumption of similar operations can vary significantly across different systems and device. For example, Mayo and Ranganathan [77] report results of an email reply benchmark which consumes 16W on a laptop, 1.44W on a handheld and 0.473W on a cellphone. Implementing certain requirements-aware energy scaledown optimizations (e.g. in the display component) enabled significant reductions (more than factor of 2 improvement) in their experience. Such techniques would apply to control point 1 marked in Fig 7.1.
Optimizing data and instruction supply overheads in supporting generality  In contrast with approaches based on scaling down requirements to save energy, the Stanford ELM project [40] explored the extent to which efficiency gains could be attained without sacrificing generality and programmability. On analyzing sources of the difference in energy/operation between hardwired media ASICs (approx 5pJ/op) vs programmable embedded processors (approx. 250pJ/op), Dally et al. found that data and instruction supply overheads alone account for 70% of this difference. By devising techniques to optimize instruction and data supply energy costs (e.g. using a deeper cache hierarchy with explicit control) in their design of the Stanford ELM microprocessor, they report a 23X improvement in energy efficiency, closing the gap with ASICs to within 3X.

In the case of software, structural interactions due to excess concerns are likely to include data transformations, parsing, book-keeping, method lookups, indirection chains, condition checks and other overheads incurred in accessing actual data and core logic implemented. Thus de-bloating techniques applicable for software at control points 2 and 3 in Fig 7.1 can be viewed as similar in spirit to some of the architecture optimizations adopted by ELM.

7.3.2 Approximate computing

A different approach that has been proposed to reduce energy spent on avoidable computation is the idea of approximate computing [105, 10, 91]. This approach is based on the key observation that there are several problems (e.g. in search, analytics and media applications) where approximate results are sufficient and appropriate but computational resources are needlessly expended on ensuring exact and accurate answers. A variety of mechanisms to support acceptable accuracy vs efficiency tradeoffs have been devised at different levels of the hardware, architecture and software stack [115, 108, 44]. These include compiler transformations for relaxing accuracy while provably maintaining suitable acceptability tolerance bounds [84, 34].

Such approaches can be viewed as exploring a complementary dimension of resource proportionality compared to our work, i.e. with respect to overprovisioned accuracy instead of overprovisioned flexibility. An interesting question to explore as an area of future work is whether approximate computing techniques, such as approximate memoization [34] or code
perforation [115] could potentially be used for mitigating bloat as well. Perhaps, bloat contributing statements are likely to have a low enough importance in typical executions for some loss in accuracy to be acceptable if sufficiently strong reasoning tools are available to bound or characterize the loss.

7.4 Conclusions

We highlighted multiple control points that could potentially be used to devise systematic strategies for mitigating bloat to achieve energy savings without necessarily sacrificing flexibility or productivity. The underlying principle is that if software features were resource proportional, energy wastage due to bloat need not be an inevitable consequence of overprovisioning flexibility. Our insights from previous chapters show how the propensity for bloat of a software implementation and its power performance effects depends on characteristics of the program structure and the underlying hardware system, such as the recurrence granularity of structural interactions between features, system bottlenecks and the energy proportionality of hardware resources. By controlling these characteristics, software features could, in principle, be made more resource proportional. However, we note that it may not be straightforward to achieve this in practice for existing software implementations. We propose quantitative measures to assess the propensity for bloat of an implementation as a foundation for performing a simple “what-if” comparison of the tradeoffs and implications of alternate strategies for bloat reduction.

Acknowledgements for this Chapter

We thank Parthasarathy Ranganathan for insights on requirements aware energy scaledown and for feedback on our early ideas about resource proportionality of software features.
Chapter 8

New Research Directions: *Exploring alternate software data models and hardware support to enable explicit management of propensity for bloat*\(^1\)

“The root cause of many of the environmental issues that we have in front of us is that there’s no line of sight between behavior and the environmental consequences of that behavior”

– Gordon Lambert, Suncor Energy Inc., Canada

(IBM Global Innovation Outlook, 2005)

We suggest a new research direction - exploring architectural innovations that provide line of sight into runtime overheads caused by software bloat and enable systems to control resource proportionality characteristics of features to optimize efficiency.

While feature bloat appears to be an unavoidable consequence of well established software development trends, the previous chapters show that the presence of excess features need not

\(^1\)Some of the ideas discussed in this chapter have appeared in the following publication: Suparna Bhattacharya and K. Gopinath. Virtually Cool Ternary Content Addressable Memory. *HotOS 2011*
Chapter 8. New directions: Alternate software models and hardware support

necessarily result in significant runtime bloat and energy inefficiencies. However, detecting and mitigating runtime bloat in existing software is difficult, in particular because bloat is often indistinguishable from essential function without some additional information.

In addition, avoiding bloat by construction without limiting flexibility is also non-trivial under current software models. We can now refer to Fig 7.1 in the previous chapter to understand why this is so. Firstly, we note that control point 1 is not applicable as it would constrain features (extent of flexibility) supported. Secondly, bloat could be avoided (using control point 2) if it were possible to code every fine-grained feature in a way that avoids structural interactions due to any optional feature on other features which can be used independently of the optional feature. However, even if it were feasible to do this, it is impractical to generalize such an approach in a large scale rapid development environment without compromising development productivity.

For example, when there is a hierarchy of features such as a base feature and series of extensions which build on previous extensions, the structural interactions due to the extensions can be avoided by making the base feature and the extensions independently accessible. This ensures that code or data corresponding to an extension can only consume resources during execution if the extension is explicitly invoked. Indeed this kind of a minimalist and incremental coding approach tends to be used in Linux kernel development where efficiency, flexibility and maintainability of the code are all critical considerations. However, a lot of careful thinking is often required in order to systematically build features in this way. In particular, deciding what the minimal core should be is not always immediately obvious, e.g. a linear hierarchy among features may not be easily apparent. In such situations a transformation in perspective may be required to logically restructure code into a well engineered sequence of feature increments.

Instead of attempting to construct software in a way that avoids all structural interactions that are potential bloat contributors, it might be simpler to focus only on avoiding those interactions that are likely to be expensive in terms of resource (and power) consumption. However, these may not be known or easy to determine at the time of software construction as the resource consumption characteristics depend on runtime parameters. This leads us to raise the following question as an open problem to explore in continuation of the line of investigation
adopted in this dissertation.

**An open question:** *Would it instead be easier to design software in a way that exposes bloat and to co-design the underlying (hardware and runtime) system in a way that enables it to detect and manage this bloat to mitigate its effects on energy efficiency?*

There is much uncharted space to cover as a part of future research before this particular question can be addressed. However, in order to provide a flavour of what might be possible, we consider a few radical ideas towards enabling:

- Software to expose bloat by making the overheads of flexibility explicit, preferably in a way that enables bloat to be moved out of line if desired (Section 8.1).

- Systems to reduce bloat automatically for optimizing resource efficiency, preferably using mechanisms that can adapt to changes in deployment scenarios (Section 8.2).

As such ideas are currently impractical to realize, we assess them at an abstract level in terms of the notion of bloat propensity formalized in the previous chapter.

### 8.1 Radical data representation schemes for managing Java data structure bloat

Much work in enterprise Java workloads is oriented towards referencing, transforming and organizing data or logical content dynamically and flexibly [89]. This motivates a need for alternate software data models that directly expose and minimize systemic data path inefficiencies (instead of focusing on inefficiencies in computational logic). The evolution of large ternary content addressable memories opens up possibilities of one such class of solutions for bloat - the use of alternate memory addressing schemes to obviate some of the causes of these inefficiencies.

Consider the following typical sources of memory bloat for long lived Java objects highlighted by previous researchers [87, 86]:

Chapter 8. New directions: Alternate software models and hardware support

1. choosing a wrong collection data structure (e.g. using a Java Treemap <Double, Double> with 80% memory bloat in a situation where parallel double arrays could have sufficed for deployment access pattern)

2. the per-collection overhead of maintaining many small collections,

Availability of efficient and fast content-based lookup and ordering can help create bulk storage representations for many Java collections that bypass much indexing overhead and avoid the need for pessimistic sizing of entries which is often a source of inefficient memory utilization.

Another major category of runtime data bloat occurs in long sequences of data transformations due to the integration complexities of composing solutions from disparate frameworks, systems and data sources involving high overhead data exchange standards and protocols. In actual deployment scenarios, only a fraction of intended flexibility is utilized in servicing any given request and many intermediate transformations are potentially irrelevant for the eventual result. It would be interesting to explore whether a combination of content-addressability and zero-copy primitives can enable a virtual representation of these data transformations that enables a deferred extraction of transformed values when actually required, potentially across address space and system boundaries and multiple storage levels.

8.1.1 Common patterns of Java memory data structure bloat

The problem of data structure bloat in Java is interesting not only because of the direct impact on memory consumption, but also indirect and compounded implications of associated processing and data transfer costs. Some common patterns of memory data bloat in real world industrial Java applications described in the literature [86] include:

1. The use of fine grained modeling resulting in highly delegated data structure designs, where multiple objects are used to represent a single entity, e.g. anecdote 1 in [86] illustrates an example of a single request structure modelled using 34 objects. This incurs a heavy representation cost due to high per object overheads (JVM object headers) and pointer costs of several levels of delegation.
2. Choosing the wrong collection data structure or oversizing collections because of inability to foresee the most suitable option for the actual usage context. The TreeMap example cited in the previous section is a typical case of overestimating the level of flexibility demanded by the real usage scenario and a lack of visibility into costs incurred (five pointers per entry plus the overhead of boxing the primitive type double into a Double object). Creating a large number of small or empty collections results in high overheads because of oversized pre-allocation defaults (e.g. 16 entries for an ArrayList) and per-collection infrastructure costs.

3. Data duplication and transformations across frameworks, layers or products e.g. Java and non-Java code, databases and object caching frameworks, SOAP/XML and Java objects [86]. Duplication of data amongst multiple caches and pools separated by insulation abstractions at different layers results in space wastage and complex configuration choices.

4. Inefficient implementations of dynamic types, inclusion of structural information (such as field names) with data leading to bloated representations and interpretation costs. Anecdote 15 in [86] incurs significant dynamic dispatch overheads even for types that are never expected to change across requests.

Many of these cases and other similar patterns are symptoms of designing without context and optimizing for the uncommon case without awareness of costs incurred.

8.1.2 Alternate addressing schemes for reducing memory overheads

An ideal solution to the problem of data structure bloat should retain all the advantages of flexibility offered by modeling principles adopted by Java developers while automatically obviating the source of bloat. As the safest and least time consuming norm to follow during development is to overestimate the extent of flexibility required, the presence of a huge efficiency gap between the most efficient choices and the most flexible ones biases decisions towards inefficient solutions. What if we could, instead, make the most commonly used models most efficient?
One way to approach this is to try to understand deeper systemic constraints that aggravate the problem. For example, an interesting question to explore is as follows: given that inefficient space usage is closely connected with data structure layout constraints, are there alternate addressing schemes which have a lower propensity for bloat?

### 8.1.2.1 Spatial locality constraints: an underlying cause of bloat?

A closer look at some of the space inefficient structures described in the previous section indicates one kind of systemic constraint which encourages memory bloat: the emphasis on spatial locality for grouping related data and the pointer cost of indirections for correlation of data which cannot be implicitly related arithmetically by spatial location.

In general, representations based on address contiguity (e.g. arrays) are often used in software applications because they involve less indirections and tend to be simpler. Spatial locality of reference has thus become an inherent consideration for efficient software implementations. Originally a reflection of the way programmers naturally wrote software, conscious efforts to increase locality of reference is now a common optimization strategy. This is usually very effective, but in some situations could introduce undue constraints and overheads in memory usage.

#### Examples

- Arrays and hash maps need to be oversized because extendability is constrained by need for contiguity of entries or spatial redistribution.

- Object headers are spatially co-located with objects. This can contribute to high per object costs for small objects.

- Delegation costs are heavy due to a combination of pointer and object header overheads (pointer indirections being the price paid for lack of assurance of spatial contiguity).

- Pointer based indexing and linkage overheads contribute to per-entry overheads in collections.
• Need for contiguous space for entries in nested collections much of which is wasted given sparse occupancy of the effective multilevel index space.

• Association of structural information with data through spatial inclusion results in bloated representations.

8.1.2.2 Effect of bloat on data structure memory layout

As actual data is interspersed with bloat within every object, bloated data spreads out the region of used memory, thus reducing opportunities for memory power management by powering down memory ranks. It also affects cache footprint and memory bandwidth. Since overhead bytes may also be accessed frequently, even clever reference pattern based optimizations are not sufficient for countering these overheads.

A key challenge that this problem presents is that the distinction between actual data and bloat is a matter of logical interpretation (unlike holes of unused data in memory), not something that the operating system or underlying architecture can determine and optimize for transparently.

Addressing schemes that can enable layouts that clearly separate out actual data and bloat into different memory regions would be attractive as they would enable these areas to be managed differently. Of course, it would be even better to avoid the bloated data altogether and bring down the overall memory requirements.

8.1.2.3 Desirable characteristics of a space efficient addressing scheme

Memory hierarchy and virtual memory models used in most systems today employ addressing schemes where memory content is accessed by location in physical / virtual address spaces, i.e. data is typically referenced (uniquely) by where it is placed. While virtual addressing abstracts away the actual underlying physical location/address, it only does so in large units of contiguous memory locations, i.e. pages. Thus spatial contiguity cannot be quite abstracted away to the extent required for reducing data structure bloat using these kinds of addressing schemes.
Here are some desirable characteristics of an addressing scheme for implementing space efficient structures to reduce run-time bloat in large framework based applications.

1. Freedom from spatial locality constraints, i.e. location independent addressability at fine granularity (object or field level)

2. Efficient representation of associations between related data, including sparse multimaps

3. Compatibility support for co-existence with existing addressing schemes, including the ability to take advantage of spatial locality where suitable

4. Efficient support for ordered relations

5. Simple to manage

6. Low overhead implementation

### 8.1.3 Content addressible memory data model

An existing alternate addressing scheme which inherently supports some these desirable characteristics if exploited to its full potential, is the model of content addressability.

**TCAM hardware**  Content addressable memory (CAM/TCAM) is an associative memory array equipped with a fast dedicated parallel search circuitry implemented in hardware. This enables memory to be accessed by specifying a search key on the memory content instead of specifying the address of a memory location [97]. In Ternary CAMs (TCAMs) entries in the array are stored so that any bit position can optionally be set to “X”, a don’t care (wildcard) bit instead of a 0 or a 1, to allow compact and flexible data representation schemes.

**Software constructs for content addressability**  Besides hardware implementations, content addressable access is also exposed in software constructs such as associative arrays used in many programming environments (these are typically implemented completely in software).
8.1.3.1 An abstract model of content addressability

Basic model Our abstract model of a content addressable store maintains unordered, non-unique ternary content word to data associations with support for multiple matches. It supports the interface described below, i.e. the commands: Create, Read (=Search), Update, Delete and Locate.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>[Content Key Word, Loc, Mask]</td>
<td>Status</td>
</tr>
<tr>
<td>Read</td>
<td>[Content Key Word] [Offset]</td>
<td>Data</td>
</tr>
<tr>
<td>Update</td>
<td>[Content Key Word] [Offset] [Data]</td>
<td>Status</td>
</tr>
<tr>
<td>Delete</td>
<td>[Content Key Word]</td>
<td>Status</td>
</tr>
<tr>
<td>Locate</td>
<td>[Content Key Word]</td>
<td>Loc</td>
</tr>
</tbody>
</table>

Content blocking, grouping and scatter-gather As an extension to the basic model above, we propose the following interpretation modes which could be incorporated in implementations involving an associative RAM [12] which stores location addresses corresponding to TCAM matchlines.

- **BLK**: A block of content mapped to contiguous location addresses can be stored compactly using a single ternary content word (single entry interpreted as a sequence of multiple 1-1 mappings).

- **GRP**: A common descriptor for a range of content (single entry pointing to data common across multiple mappings)

- **SG**: Spatially discontiguous content unified virtually using a single search word (multiple entries interpreted as a single data structure)

- **INL**: Inlined data instead of a location reference

A mask indicating how to interpret access requests may be used to distinguish the portion of the content word which maps the block or group and to compute the bit shift to apply to the
offset in the block to account for the size of the location units. This mask and interpretation modes are transparent to users of the CAM, except at the time of entry creation.

### 8.1.4 Applications of CAM based schemes for reducing run-time data bloat

We describe how CAM based representations could be devised for some common data structure modeling situations. The underlying methodology is similar in most of these cases:

1. Pick each major spatially modeled association
2. Replace it with a CAM mapping scheme
3. Combine entries where possible (ternary compaction as BLK/GRP entries)
4. Determine if there is a net reduction in bloat (CAM overhead vs space saving)

A tool like Yeti [85] may be used to identify structures that are particularly space inefficient (bloated) as candidates for such conversion. The remaining data may continue to be maintained in location based form. Location based compatibility is represented as a location range directly mapped by a ternary CAM entry (BLK mode).

#### 8.1.4.1 Representation of key-value mapping structures

An alternative to hash tables and other indexing structures for key-value lookup, would be to use a TCAM based underlying storage representation. A fairly straightforward mapping, this, after all is what CAMs are well suited for. Instead of allocating dedicated CAM space for every map (which would be inefficient for lots of small collections and fraught with spatial locality constraints), the map id (collection identifier) is also included in the content key space. The entire collection can be retrieved (unordered) using a ternary search on the map id with key bits set to don’t care.

The map id and the key would need to be appropriately compacted to fit both within the content key width (64/128 bits). This sort of compact integer representation of the key is needed anyway for most key-value lookup structures where the key can be any object or string.
8.1.4.2 Bulk typing of collections

The above representation scheme for collections containing lots of small objects can be extended to associate a bulk type for all objects in the collection. This will require an additional type bit to be included in the content word, so that a ternary match on the map-id, with key bits set to don’t care can be associated with the default type for all objects in the collection. The type for any object can be looked up by setting the type bit in the search word for accessing that object (in addition to the map id + key).

In case an object of a different type is included in the collection at run-time, then an additional TCAM entry for that object can be entered to override the default type for that particular element. (assuming that preference is given to the longest match).

8.1.4.3 Extensible arrays: reducing overhead due to unused entries

Use of ternary content words enables associations for a range of keys with contiguous value entries to be encoded compactly if there is a ternary word that can cover that range using an appropriate number of don’t care bits. This works for power of two sizes in typical hardware CAMs, and may be improved with advanced range matching techniques and circuitry.

Thus an array can be represented as such an association, allowing for extensibility without spatial reallocation by the addition of a content word entry for the extended range.

For example, consider an array with 8 consecutive entries that are referenced through 8 consecutive content words. These content words can be combined into a single mapping, with
the least significant 3 bits being marked as don’t care, and association being labelled as a block (BLK) with location units of size that match the array entry size. Suppose the 5th element of the array is being referenced by content, then the search word would map the ternary block mapping, and the relative offset of the entry to retrieve \((5 \times \text{unitsize})\) can be calculated from the search word and the beginning of the ternary block. Now, if the array is extended to 8 more elements, then instead of a realloc, the new elements can reside in a different location area with a new content word mapping created to cover the range of those 8 elements. Thus if the 10th element is referenced by content, the search word would rightaway match this new entry without any additional indirections.

<table>
<thead>
<tr>
<th>Content key word</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[array id, 00000XXX]</td>
<td>1st element</td>
</tr>
<tr>
<td>[array id, 00001XXX]</td>
<td>9th element</td>
</tr>
</tbody>
</table>

8.1.4.4 Discontiguous objects: separating less commonly used fields

A similar scheme could also be applied at a finer grained level, i.e. for separating groups of fields within large objects. This can enable interesting features such as deferred allocation of fields, copy-on-write sharing of fields across objects and differently managed regions for fields of different importance and reference frequency.

<table>
<thead>
<tr>
<th>Content key word</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[object id, field range]</td>
<td>fields in the range</td>
</tr>
</tbody>
</table>

8.1.4.5 Compact representation of highly delegated entities

The above approach can be taken a little further to enable highly delegated entities to be represented as a single large (group) object. The leaf fields in the tree of delegation chains could be enumerated in a way that uniquely identifies the delegation path they correspond to. Likewise the associated leaf methods and field types could be included in the group object class.
In case the delegation tree is altered dynamically to include a reference to an existing object, then additional TCAM entries can be created to override the default mapping for corresponding fields and their methods and types.

Implementation wise this will also require semantic expansion [134, 8] of delegated method invocations to operate directly on the group object fields and use the group object methods.

### 8.1.4.6 Multi-level indexes: efficient nested collections

Any practical application is likely to involve nested collections possibly containing highly delegated objects. Thus, efficient representations for nested combinations of the above CAM schemes need to be developed in order to realize significant savings.

In many cases this is equivalent to implementing a sparse multimap without spatial constraints. This would typically require a wider CAM word, which can technically be simulated using multiple CAMs chained together or by using some kind of state machine for pattern matching (e.g. as used in genomic data processing). There is a tradeoff between this approach and the alternative of explicitly encoding multiple levels of nesting with CAM entries which act as indirections to other CAM entries (a denormalization vs normalization tradeoff). Collections or objects that are shared by multiple data structures incur a greater overhead if wider CAM mappings are used.

<table>
<thead>
<tr>
<th>Content key word</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[map id, key1, key2, ..]</td>
<td>key - value entry</td>
</tr>
<tr>
<td>[map id, key, irange, framerate]</td>
<td>fields of values</td>
</tr>
</tbody>
</table>

### 8.1.4.7 Extensions to support ordered keys

Typical hardware CAMs lack any inherent notion of ordering. Thus while it is possible to access a range of keys using multi-match capability, priority encoders that arbitrate between multiple matches operate on the matchlines and not the content key space. Hence they cannot impose ordering of keys unless the entries are physically stored in order in the CAM. Addition of comparison circuitry, clever range encoding schemes or state machines logic to CAMs...
are possible techniques that could be explored for providing content based ordering through hardware. A simple alternative in the absence of such hardware support is to implement an intermediate content ordered memory area which is used to cache sorted groups (ranges) of CAM words as needed. Such caches may in turn be CAM indexed by map id and key ranges. We call these content ordered memory (COM) caches.

## Analysis of space savings and overheads

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Content key word</th>
<th>Data</th>
<th>Mode</th>
<th>Overhead</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Val Map</td>
<td>[map id, key]</td>
<td>Entry</td>
<td>BLK</td>
<td>$sz(mapid, key) \ast entries$</td>
<td>$ptrs + unusedslots$</td>
</tr>
<tr>
<td>Bulk Type</td>
<td>[grp id, isTyp]</td>
<td>Type</td>
<td>GRP</td>
<td>$(sz(grpid) + 1) \ast (1 + overrides) + entries$</td>
<td>typeptr \ast elements</td>
</tr>
<tr>
<td>Ext Array</td>
<td>[arr id, index]</td>
<td>Elem</td>
<td>BLK</td>
<td>$sz(arrid, idx) \ast entries$</td>
<td>unused elements</td>
</tr>
<tr>
<td>Discont Obj</td>
<td>[obj id, foff]</td>
<td>Field</td>
<td>BLK</td>
<td>$sz(objid, foff) \ast pieces$</td>
<td>unused fields</td>
</tr>
<tr>
<td>Deleg Comp</td>
<td>[obj id, leafoff]</td>
<td>Field</td>
<td>BLK</td>
<td>$sz(objid, leafoff) \ast (1 + overrides)$</td>
<td>$ptrs + hdrs$</td>
</tr>
<tr>
<td>Nested Ent</td>
<td>[root, keys..]</td>
<td>Data</td>
<td>BLK</td>
<td>$sz(key) \ast (depth - 1) \ast entries$</td>
<td>$ptrs + sz(grpid, key) \ast (depth - 1)$</td>
</tr>
</tbody>
</table>

Table 8.1: Computing space savings vs overheads of CAM based representation schemes. $sz()$ is a function that computes the size of the portion of the CAM word that represents the fields specified as parameters. entries is the number of data element entries, pieces is the number of pieces in a discontiguous object, $ptrs$ refers to pointers, $hdrs$ refers to object headers, typeptr refers to the type information part of an object header.

Table 8.1 may be used to calculate the corresponding overheads and space savings of the
above CAM based representation schemes:

### 8.1.5.1 Example: HashSet of short strings implemented using a HashMap

We consider the illustrative example from [87], of a HashMap whose keys and values are Java Strings, and the key and value point to the same object. There are 3 strings in this collection, each with a 2 entry character array. This is a somewhat extreme example, which occupies a total of 384 bytes, of which the only 12 bytes constitutes actual data in the Strings’ character arrays. Table 8.2 shows a possible CAM based mapping scheme for this case.

<table>
<thead>
<tr>
<th>Map</th>
<th>Key</th>
<th>isVal</th>
<th>isTyp</th>
<th>field</th>
<th>index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b</td>
<td>32b</td>
<td>2b</td>
<td>2b</td>
<td>12b</td>
<td>16b</td>
<td>variable</td>
</tr>
<tr>
<td>m1</td>
<td>h1</td>
<td>X</td>
<td>0</td>
<td>0X</td>
<td>0X</td>
<td>2 a1 a2</td>
</tr>
<tr>
<td>m1</td>
<td>h2</td>
<td>X</td>
<td>0</td>
<td>0X</td>
<td>0X</td>
<td>2 b1 b2</td>
</tr>
<tr>
<td>m1</td>
<td>h1</td>
<td>X</td>
<td>0</td>
<td>0X</td>
<td>0X</td>
<td>2 c1 c2</td>
</tr>
<tr>
<td>m1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>strtype</td>
</tr>
</tbody>
</table>

Table 8.2: Example: CAM based representation of hash set with 3 short strings

The space overheads for this mapping can be computed as follows:

Cost per entry = 12 bytes CAM (96 bits) + 8 bytes in associative RAM (4 bytes location addr + 4 bytes mask)

As the above mapping requires 4 entries, the total CAM related overhead = 20 * 4 = 80 bytes.
Also, the string data includes a length field in addition to the characters, hence the total non-CAM overhead = 4 * 3 = 12 bytes

Thus the combined total overhead is 92 bytes, while actual data is 12 bytes

While this overhead is still very high, it is a 75% reduction from the overhead (of 372 bytes) under the original data structure representation, achieved without losing any of the flexibility afforded. For example, strings of longer sizes can be added to the table as needed and even entries of a new type can be part of the table. Also the layout can be optimized to reduce the space consumption. For example, as the strings are short, they may be inlined in the associative RAM, saving about 6 bytes per entry, bringing down the total overhead to 68 bytes. Further,
the field and index bits are not essential given that fields in a Java string are stored contiguously and an extensible array is not required since strings are immutable. This can save about 3 bytes per entry, reducing the overhead further to 56 bytes (which is 15% of the original, i.e. an 85% reduction, or less than one-sixth of the cost).

8.1.5.2 Example: TreeMap[Double, Double]

Consider the example (described by Mitchell and Sevitsky [86]) of an application which uses a TreeMap to represent a map from double to double, containing over a million entries. The application does not use the sorted property of the TreeMap until the map is fully populated, yet this structure incurs a high asymptotic memory bloat factor. A more suitable alternative would have been to represent the data with parallel double arrays, which has less than 2% memory bloat [86]. This, however, assumes that the developer has advance knowledge of the usage pattern and an awareness of the costs, which may not always be the case in framework driven application components.

Using a CAM based map representation can afford more flexibility while saving space. Let us consider a scheme where the double is directly used as a key in the CAM entries (instead of a hashcode), and the value is inlined in the associative RAM. The net space savings may be calculated as follows and summarized in Table 8.3 (where $s$ = size of the content ordered cache). In this case bulk typing saves entire object header costs and not just typeptr costs.

**Space Overheads Computation**

CAM mapping: \( \langle \text{Mapid, double} \rangle \text{[double]} \)

Assume a Content-ordered memory (COM) cache is included, where $f$ = fraction of keys cached \((0 < f < 1)\)

Let $n$ = no of elements in the collection

Let the cumulative overhead incurred when using the CAM based representation be $J'$ bytes

Let the actual content (i.e. the double keys and values stored in the map) be $D'$ bytes, and

Let the scaling formula that describes the amplification factor in space consumption due to the overheads be $S' = 1 + \frac{J'}{D'}$ (as defined in [87])

$D' = 16n$
\[ J' = 4n + 12 + 8fn \]
\[ S' = 1.25 + \frac{f}{2} + \frac{0.75}{n} \]

With the original data representation, the cumulative overhead \( J \) and scaling formula \( S \) were:
\[ J = 40 + 36n + 24n \]
\[ S = 4.75 + \frac{2.5}{n} \]

Relative improvement achieved using the CAM representation:
\[ \frac{J'}{J} = \frac{4+8f}{60} \text{ (for large } n) \]
\[ \frac{1}{15} < \frac{J'}{J} < \frac{1}{5} \]
\[ 2.7 < \frac{S'}{S} < 3.8 \]

Thus, the CAM based representation lowers the overheads to less than one-fifth that of the original representation and improves the space usage by more than 2.7 times (i.e. it requires about a third of the space occupied by the original representation).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Overhead</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Val Map</td>
<td>4MB</td>
<td>20MB</td>
</tr>
<tr>
<td>(Ordered)</td>
<td>+ 8s</td>
<td></td>
</tr>
<tr>
<td>Bulk Type</td>
<td>0.13MB</td>
<td>24MB</td>
</tr>
</tbody>
</table>

Table 8.3: Example: Space overheads vs savings of using a CAM based representation of a TreeMap (from \textit{double} to \textit{double}) with 1 million entries

Table 8.3 illustrates savings achieved for a TreeMap with 1 million entries:

Net savings = \( 44 - 4 - 0.13 - 8s >= 31MB \)

Actual data = \( 16MB \), Overhead <= \( 13MB \) (between \( 4 - 13MB \))

i.e. percentage bloat between 20\% - 45\%

We note a significant space-time tradeoff in the chosen size of the content ordered cache. What makes this interesting is that it is an optimization decision that can be taken at the system or runtime level (instead of application level). If we choose \( s \) such that bloat is 30\%, then the overhead is 7MB, less than one-sixth of that with the original layout.
8.1.6 Caveat: Power-performance tradeoffs with hardware TCAMs

Although a CAM based data representation can result in significant reduction in space overhead, a hardware TCAM can consume an order of magnitude more energy per search than reads in traditional memory. With resistive memory based TCAMs, the energy costs are expected to be lower [50, 103]. However, it would still be impractical to maintain CAM based data structure mappings in an actual hardware physical TCAM. For example, consider 1GB of application data structures with 50% representation overhead, which can be brought down to one fourth using a CAM based representation. To support this in hardware TCAM, about 128 MB of CAM space would be required (i.e. a 1Gbit CAM), if we assume the majority of overhead after transformation is in a CAM. Instead, it is worth exploring whether a viable and energy efficient alternative could be simulated by a software based implementation that uses a hardware TCAM just for acceleration purposes. In this case, the main benefit of a CAM based data representation is that it cleanly separates the representation overhead (bloat) from the actual data and enables optimization for efficiency to be handled in a common application agnostic manner so that tradeoffs can be managed at a system level.

8.2 Hardware support for resource proportional primitives by virtualizing bloat?

Such an alternate (software) data model that directly exposes bloat by making the overheads of flexibility explicit is desirable for multiple reasons. For one, it provides the ability to automatically measure the effects of runtime bloat and guide de-bloating efforts by providing an accurate assessment of cost-benefit tradeoffs involved in optimization decisions. A second and potentially more significant benefit of making the overheads of excess flexibility visible is the possibility that novel system level support mechanisms could be devised to optimize resource proportionality by minimizing these overheads through runtime, operating system and architecture enhancements.

To motivate future research on the second possibility, we now illustrate an example of how
an approach of this nature might work under a specific application setting involving the use of ternary content addressable memory to accelerate associative lookups. Instead of focusing on the specific problem of Java data structure bloat for which the viability of using TCAMs remains to be researched, in the rest of this chapter, we choose a more general application setting where the suitability of TCAMs is already established. As associative lookup structures lie at the heart of many computing problems, fast content addressable data access mechanisms have several other compelling applications in today’s systems. Many of these exploit the powerful wildcard matching capabilities provided by TCAMs.

However, large hardware TCAMs are still prohibitively expensive in terms of power consumption and cost per bit. This has been a barrier to extending their exploitation beyond niche and special purpose systems. To overcome this barrier, we propose an approach for extending the traditional virtual memory hierarchy to scale up the user visible capacity of TCAMs while mitigating the power consumption overhead. In this setting, the overhead of excess flexibility (bloat) corresponds to increased TCAM energy/power consumed to support a large number of (over)provisioned association mappings that are unlikely to be used very often, i.e. entries in the TCAM that rarely have any search hits. Virtualizing the TCAM space ensures that the power consumption overhead is incurred only when the association mappings are actually used, reflecting resource proportional behavior.

By exploiting the notion of content locality (as opposed to spatial locality), we outline a novel combination of software and hardware techniques to provide an abstraction of a large virtual ternary content addressable space. Cost vs performance tradeoffs at different levels of a content addressable hierarchy are automatically managed by the operating system. In the long run, such abstractions could enable new applications that disassociate considerations of spatial locality and contiguity from the way data is referenced. This could also have interesting implications in optimization of applications which exhibit poor intrinsic spatial locality, e.g. dynamically composed software and framework based Java programs.

Before we proceed to describe our solution in detail, in the next subsection, we introduce our application setting by providing some examples of well-known and emerging applications of TCAMs.
8.2.1 Established and emerging applications of TCAMs

The most widespread exploitation of hardware TCAMs occurs in high performance routers, for route lookup, access control and packet classification. Examples of other applications include database acceleration [12], frequent items in data streams [13] and several algorithms that use TCAM as an underlying primitive. TCAM based implementations of fundamental techniques in pattern matching, machine learning and data mining, such as regular expression matching [82], nearest neighbor search [113] and subset queries using ternary bloom filters [49], are a few examples that have been developed in recent years. These techniques have diverse real world applications in areas such as information retrieval, image search, genomics, proteomics, intrusion detection, and fraud surveillance. What makes the TCAM abstraction such a powerful primitive for many of these applications is the ability to simultaneously search through a large number of subspaces of a higher dimensional space in one shot. For example, each subspace can be compactly represented as one (or a few) TCAM entries using the don’t care bits to cover ranges that constitute it.

Similarity search and nearest neighbor search are widely used in many algorithms. Locality sensitive hashing is an important technique that maps high dimension feature vectors to lower dimension ones while keeping similar content together. This can be done in a pre-processing step where data points are hashed to a number of buckets. To perform a similarity search, a query is hashed using the same locality sensitive hashing scheme and the similarity search is performed on the data points retrieved from the bucket corresponding to the query hash. However, streaming algorithms that are becoming common still find the “non-parallel” similarity search in the last part slow. A recent technique [113] uses a modified version of locality sensitive hashing to hash data to ternary values, enabling compact TCAM representations and quick similarity searches for various classification problems.

TCAMs may also be useful in many state space exploration problems (such as those encountered in verification) where many states can be combined into a single TCAM entry using Bloom filters, enabling a fast search for previously visited states or error states.

The usage of content based lookup and similarity matching in systems infrastructure is also growing. For example, de-duplication techniques for cache [26], memory [6], IO [65] and
storage data all exploit some form of content lookup or comparison scheme. [49] shows how ternary bloom filters can be used to achieve an order of magnitude throughput improvement over current techniques in high speed multiple string matching (MSM) problems, a key component in data-deduplication, sequence alignment and intrusion detection techniques. Hardware based range caches [120] have been proposed for efficient state tracking to make intensive dynamic analysis of programs viable.

8.2.2 Virtually cool TCAM - a possible way to minimize energy wastage due to unused associations?

Despite all of these developments, hardware TCAMs have not made their way into mainstream computing\(^2\). This is mainly because the power of TCAMs comes at the price of high cost and energy consumption. A TCAM uses about 20x more dynamic power per bit than an SRAM [1, 49] (the overhead of parallel lookups). As a result practical applications have been mostly limited to niche areas where the tradeoff can be justified for a TCAM size which fits the requirements, e.g. in high speed packet classification (with 50x speedup). Both the delay and energy consumed per access increase with the size (width and number of entries) of a TCAM [1]. This restricts the extent to which the use of TCAMs can be scaled so as to be viable in broader setting.

We think that there may be a way to break this barrier if we recognize that unused association mappings are the unused features that lead to bloat in this setting. Most of the power consumed by a TCAM is effectively wasted in mismatches\(^3\). While this observation has prompted many TCAM power optimizations [97], hardware based techniques tend to have limited flexibility in adapting to actual usage scenarios. Significant energy savings could be achieved if we could adopt the principle of resource proportional design, where ideally, only the association mappings (features) that are actively searched should contribute to power consumption. Perhaps, this is an area where operating systems can help (with architectural support). There

---

\(^2\) even though they have been integrated with NPUs for years

\(^3\) all matchlines are pre-charged before a search; lines that do not match the search word are discharged, leaving only the lines that match in high state
is a well-established precedent for solving such problems - consider the invention of the memory hierarchy and virtual memory management (VMM). Can such mechanisms be extended to scale up the applicability of content addressable primitives?

### 8.2.3 Content addressable VMM (CAVMM)

Let us see how the basic concept of ternary content addressable memories may be extended to a generalized content based memory hierarchy by combining the benefits of TCAMs and VMM principles. The key idea is to devise a mechanism that only uses hardware TCAM space to store association mappings that are actually exploited (likely hits) while maintaining the remaining association mappings (unused features or bloat) out of line in a memory area which has a lower cost or power consumption per bit. This enables (one or more) applications to efficiently exploit the power of the ternary search abstraction at a larger scale than that achievable with hardware TCAM alone.

The proposed hierarchy includes a hardware TCAM based cache and multiple levels of ternary content addressable stores (TCASs). These stores may be implemented in hardware or software with different performance vs efficiency tradeoffs, e.g. high performance at levels closer to the processor and high capacity at levels that are further away. Content (search key) words present in these stores are associated with references to data in a traditional (hierarchical) location addressable store (LAS). This data is returned as the result of a content addressed access (search) along with the key.

One of the novel features of this architecture is that traditional notions of pages and blocks are replaced by alternate notions like content subspace pages and content blocks which operate on a content key space (i.e. the domain of the content word) instead of a location based address space. The hardware support required may be implemented using a content addressable memory management unit (CAMMU).

The design must be capable of exploiting the benefits of spatial locality and location based addressing where preferable, while enabling the full power of content addressability at a system level. This is achieved using content mapping schemes that preserve location based addressing where desired (e.g. as a default compatibility mode or where it is more efficient).
Fig 8.1 illustrates how a content addressable virtual memory hierarchy might be organized. We focus on one possible implementation approach to make this example concrete and highlight a few essential details. Many potential variations or extensions may be explored using similar ideas. Fig 8.2 depicts a sample view from a snapshot of the virtual content addressable space and its representation in the CAVMM hierarchy. We assume an implementation with two-levels of TCAS (in addition to the content based cache) where the Level 1 store is implemented using hardware TCAM and the Level 2 store (described in more detail later) uses a software based implementation with DRAM as the underlying physical store.

Figure 8.1: Content addressable VMM example

**Location Addressable Hierarchical Store:** Traditional memory store where data is referenced by its memory address location. Addressing could be physical or virtual, and the hierarchy could span multiple levels of memory and secondary storage.

**Location Based Cache:** Caches data from the LAS.

**Content Addressable Store:** Associates ternary content words with data references in a LAS\(^4\). When

\(^4\)other interpretations are possible, e.g. inlined data
Chapter 8. New directions: Alternate software models and hardware support

8.2.3.1 Content Paging and Content Blocks

The mapping from a content key to a physical location can be as fine grained as a single memory word, effectively dissociating spatial contiguity from content locality. This breaks the traditional concept of pages as used in virtual memory implementations.

A Content Subspace Page is the result of a search matching a lower dimensional subspace of the content key space, i.e. a collection of entries (in a TCAS) whose content key word has a value that falls within the subspace. For example, the content key space may be broken up

Figure 8.2: Sample content addressable space

presented with a ternary search word, matching content words and the corresponding data referenced are retrieved. Since multiple entries can match, a stream of multiple results may be returned.

**Content Based Cache:** Transparently caches content word to data associations. The corresponding data is cached in the location based cache. Since multiple matches are possible there could be multiple entries for the same content word. Cache prefetching is content locality based rather than address locality based.
into uniform subspaces of size $2^k$ formed by setting the least significant $k$ bits of the search word as don’t care when retrieving a content subspace page. The entries belonging to a content subspace page could be distributed across the TCAS with no physical contiguity or ordering implied. They form a logical representation of a page rather than a real memory page. Notice that a content subspace page typically has holes within it (i.e. it may be sparse). As a result, the physical size (number of TCAM entries) is usually smaller than a real memory page. On the other hand, since multiple entries may match the same content word, it is even possible for the physical size to be larger than a real memory page. In general, it is not necessary to use only the least significant bits or even contiguous bits when defining a content subspace page, i.e. the subspace could range over any specified dimension(s). Further, it is even possible for a single ternary entry to straddle more than one content subspace page.

A **Content Block** is a group of content words in a TCAS that contain consecutive values in the content key space and reference data at consecutive location units in the location based address space. These entries can be compressed into a single content block entry if the range of content words can be represented as a ternary word. This feature also enables *location based addressing to be trivially supported* with minimal overhead by using a single content word entry (cached in the content cache) that represents a large ternary content block covering the entire location address space.

### 8.2.3.2 Level 2 TCAS

How might a level 2 TCAS be implemented by an OS using an underlying DRAM store? A single ternary content word is represented as a combination of a binary content word and a binary wildcard mask. For each ternion in the original content word that is set to “X” (or don’t care), the corresponding bit in the wildcard mask is set to 1 and other bits are set to 0. If the unit of transfer between the level 1 and level 2 store is a content subspace page, it is sufficient to track these content words at the granularity of such a content subspace. Regular memory based data structures e.g. hash tables or integer radix trees may be used to maintain key-value and range-value mappings in DRAM. Instead of creating these structures, however, we devise a simpler scheme that takes advantage of the hardware TCAM at Level 1 (making physical
locality or size of content pages irrelevant for paging complexity). This works as follows:

When all entries corresponding to a content subspace page are collected and paged out\(^5\) from Level 1 to Level 2, a single special ternary content word entry is created in the Level 1 TCAM to refer to the location of the content page in the Level 2 store. The same principle may be extended to create a content page container subspace (by paging out content subspace pages that fall within a content page container subspace). Further, as we noted earlier, the notion of a page need not be limited to the range covered by least significant bits in content space - any subset of bits in content words could be defined as a subspace page using wildcards. Different content page subspace masks may be used by different applications (depending on the structure of content locality expected).

### 8.2.4 Content locality classification

If “locality of reference breeds the memory hierarchy” [58], then locality of content would determine the potential value of a content addressable memory hierarchy. While a quantitative characterization of content locality in candidate applications requires further research, we can attempt a qualitative assessment to obtain a sense of the implications. We classify application workloads into different categories based on the expected pattern of matches in content addressable space.

1. **Rare Hits**: e.g. intrusion detection. In this case most entries can be moved to level 2 and are brought in when there is malicious traffic or input pattern that is close to a malicious pattern

2. **Frequent Same Item Hits**: e.g. finding frequent items in data streams. In this case, items above the frequency threshold would be in Level 1 or even in the content cache, while others may be moved in and out on-demand based on available capacity

3. **Clustered or Nearby Item Hits**: In this case, content subspace paging will help as it brings in the items mostly likely to be required into level 1 while bulk of the entries can reside at level 2

\(^5\)using a ternary subspace search and one bit in the content space set aside to detect free entries for reclamation
4. **(Uniformly) Random Item Hits**: In this case performance will depend on the ratio of available level 1 capacity and total number of entries.

In many cases content locality characteristics depend on the input distribution, e.g. similarity search, regular expression matching, packet classification and de-duplication. As a starting assumption, we might expect a few frequently hit clusters and potentially many rare hit clusters. In program analysis, dynamic analysis associations exhibit a high range locality [120]. For database join, it depends on the join selectivity and cardinality.

**Other potential implications**  In traditional location based addressing, associations are modeled through spatial relationships (e.g. spatial contiguity, index arithmetic, pointers, hashing). Using content based addressing, these can be expressed directly to the underlying system. This can free the application from spatial constraints and enable a lower level optimizer to move data around at a fine granularity without breaking any dependencies. With search driven execution becoming a common paradigm, data and operational associations are heavily used in general purpose middleware and application software. In a given deployment context, many conditions change rarely. Thus a small subset of associations are likely to be used most often. Content based caching might be very effective in reducing overheads in these situations.

### 8.2.5 Impact on resource proportionality

We can assess the resource proportionality characteristics of a virtual TCAM in terms of the energy consumed as the percentage of association mappings exploited (feature utilization $\mu$) increases. For ease of analysis, let us assume a simplified model of TCAM power consumption where the energy per access is linearly proportional to TCAM size i.e the number of entries (rows)$^6$. Thus the energy per access under a perfectly resource proportional implementation as normalized with respect to the full capacity TCAM would be proportional to $\mu$.

Consider a virtual TCAM where the first level supports a fraction $\frac{1}{K}$ of the full capacity required to store all the association mappings and the second level is sized at the full capacity.

---

$^6$A more accurate model is available in [1]
When $\mu < \frac{1}{K}$ (the working set fits within the first level CAM), all searches result in first level TCAM matches, and the energy per access is proportional to $\frac{1}{K}$. However, as $\mu$ increases, misses would occur and searches would need to be serviced from the second level CAM (and also brought into the first level CAM). The miss probabilities would depend on the actual distribution of accesses – let us consider an example where misses are proportional to $\sqrt{\mu - \frac{1}{K}}$.

$$\frac{R_{\text{actual}}}{R_{\text{full}}} = \frac{1}{K} + (1 + \frac{1}{K})\sqrt{\frac{K}{C_K}(\mu - \frac{1}{K})}, \mu > \frac{1}{K}$$

If $K = C_K = 16$, ($C_K$ is a factor that determines the miss probability at full utilization for a given value of $K$)

$$\frac{R_{\text{actual}}}{R_{\text{full}}} = 0.0625 + 1.0625\sqrt{\mu - 0.0625}$$

This configuration has a sublinear resource proportionality and consumes less energy per access than the full capacity TCAM if (dynamic) utilization $\mu$ (fraction of stored association mappings that are actively searched or used) is less that 80%. When utilization is 25%, its energy per access is half that of the full capacity TCAM, i.e. a 50% energy saving. When utilization is 6.25% or less, the energy per access is 6.25% that of the full capacity TCAM, resulting in more than 90% energy saving.

### 8.2.6 Implementation challenges

Characterizing content locality and content key working sets of existing workloads is an important first step in determining the design space parameters for feasibility of this approach. Early implementations of a CAVMM may be built without requiring any extra architecture support in order to evaluate minimal hardware system mechanisms that are essential. Besides this, there are many design issues that need to be researched, such as policies for allocation and reclamation of TCAS (and LAS), sharing of space across processes, and ternary compaction optimizations that might be applied by the OS (e.g. at the time of pageout) to minimize the number ternary word entries. Furthermore, mechanisms for concurrent access to the CAS by independent threads needs to be explored in depth along with a study of how transactional consistency can be achieved, in the concurrent context, when multiple entries/locations are
updated on certain “elementary” CAS operations.

**TCAM Extensions** Currently TCAMs are usually configured to return the first match in the event of multiple matches (using a priority encoder). This can be very inefficient in many situations e.g. database operations, content page retrieval. Support for efficient bulk transfer for multiple matches is therefore an important requirement, e.g. using a TCAM functional unit along the lines proposed in [54]. TCAMs need not be the only hardware content addressability mechanism used in a CAVMM hierarchy. For example, hardware range caches or E-TCAMs which allow non-power of two ranges to be represented efficiently and other pattern matching accelerators may also be worth consideration.

### 8.2.7 Conclusions

This chapter motivates a new research question that is relevant to the problem of software runtime bloat – are there alternate data models and architectural mechanisms that could enable software to be designed in a way that makes it easier to detect and mitigate bloat?

As a radical alternative to illustrate the existence of such possibilities, we showed how abstractions based on a ternary content addressable memory (TCAM) based software data model could reduce space overheads of Java data structure bloat without losing the flexibility that underlies existing data representation approaches. However, the practical realization of such abstractions in systems is currently unviable because of the high power consumption incurred by hardware TCAMs. At the same time, one advantage of the TCAM model is that the energy consumption overheads of unused flexibility can be automatically characterized in terms of rarely used associations. We proposed the novel idea of a virtual TCAM to demonstrate how this characteristic enables an operating system to automatically save energy by optimizing the resource proportionality characteristics of TCAM based applications. The discussion we presented here has provided a flavor of such ideas, how they may be implemented and the difficulties involved, but we believe that it has only scratched the surface of what appears to be a promising new direction of research.
Acknowledgments for this Chapter

Our thanks to Bob Montoye, Richard Freitas, Viji Srinivasan, Bipin Rajendran, C. Mohan, John Karidis and Jai Menon for their inputs, particularly a perspective of technological trends, tradeoffs and challenges in building large scale hardware TCAM solutions.
Chapter 9

Conclusions

Software bloat is an issue of growing importance in the face of rising operational costs in computing systems. A free ride on improvements in hardware efficiency can no longer be taken for granted when developing software solutions. This thesis offers a fresh approach to the problem of runtime bloat by viewing it from a high level systems perspective. Our work is the first\(^1\) to explore the relation between bloat and energy efficient design.

Table 9.1 summarizes how the chapters in the dissertation are related to the research questions raised in the introduction and the key techniques contributed towards addressing these questions. Through the work presented in chapters 3, 4 and 5 we have systematically investigated the connection between sources of bloat and its power-performance implications. Bloat is a side-effect of practices that favour an overprovisioning of software concerns (or features) in applications, in particular, the reuse of components that support more considerations than strictly required in a given deployment scenario. While these trends are unavoidable, our findings show that they need not lead to energy wastage from runtime bloat.

Firstly, from Chapter 5, we know that an unexploited (excess) software concern only impacts runtime resource consumption through statements or fields that intrude in the code or data path of exploited software concerns. We presented a novel technique to analyze the connection between optional software concerns and sources of execution bloat induced by their structural intrusions on essential concerns. Identifying these bloat contributing statements is

\(^{1}\)to the best of our knowledge
Table 9.1: Research questions addressed by different chapters and techniques contributed

<table>
<thead>
<tr>
<th>Research question</th>
<th>Where addressed</th>
<th>Techniques contributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>When do excess features lead to runtime bloat?</td>
<td>Chapter 5</td>
<td>CAPA</td>
</tr>
<tr>
<td>How can the extent of bloat attributed to a given source be determined?</td>
<td></td>
<td>Micro-slicing</td>
</tr>
<tr>
<td>When is the resource overhead due to bloat most pronounced?</td>
<td>Chapter 4</td>
<td>Object reuse transformation</td>
</tr>
<tr>
<td>What can be done to mitigate bloat once we have identified it?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>How much does bloat matter for power-performance?</td>
<td>Chapter 3</td>
<td>Analytical model of power-perf</td>
</tr>
<tr>
<td>What information is necessary to automatically estimate the extent of bloat?</td>
<td>Chapter 5, 6</td>
<td>Probabilistic CAPA</td>
</tr>
<tr>
<td>How do we assess the propensity for bloat of a given software implementation?</td>
<td>Chapter 7</td>
<td>Resource proportionality</td>
</tr>
<tr>
<td>What information is necessary to automatically de-bloat software?</td>
<td></td>
<td>Fig 7.1</td>
</tr>
<tr>
<td>Can systems be redesigned to enable software to avoid propensity for bloat</td>
<td>Chapter 8</td>
<td>Virtual TCAM</td>
</tr>
<tr>
<td>for bloat without losing flexibility or productivity?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Research questions addressed by different chapters and techniques contributed

an example of an analysis task that cannot be tackled solely using static and dynamic program analysis. We addressed this problem by introducing the idea of concern augmented program analysis (CAPA). CAPA extends traditional program analysis with abstractions that exploit higher level information about optional software concerns, where available.

Secondly, from Chapter 4, we know that bloat contributing statements only cause a heavy execution cost when the corresponding overheads are incurred frequently. Thus, automated code transformations could be effective in mitigating bloat if they are targeted at optimizing recurring runtime overheads induced by sources of bloat. We presented one such transformation, a novel static analysis algorithm for reusing objects that would otherwise be repeatedly allocated in a loop, thus reducing the volume of temporary objects generated due to bloat. Our solution is effective because it works where a conservative analysis would have missed most opportunities for reuse, e.g. for objects reusable in loops several call levels above the allocating
method or objects reusable only along certain paths and not others.

Thirdly, from Chapter 3, we know that runtime resource bloat only has a substantial impact on system power-performance under certain conditions. The extent of energy savings achievable through bloat reduction may not always be obvious due to a curious interplay between bloat, system bottlenecks and hardware energy proportionality. Hence, we presented an analytical model that enables a generalized “what-if” analysis of the power-performance impact of bloat from a whole systems perspective. We showed that bloat reduction matters most for energy efficiency when the underlying hardware resource whose usage is bloated is super energy proportional, the non-bloated resources are non energy proportional and bloat affects a bottleneck resource.

In contrast to these deterministic analysis approaches, Chapter 6 employs unsupervised machine learning – a statistical topic model inferred from source code text – applied in a novel manner to discover potential latent concern topics and assess their relative contribution to runtime resource usage such as objects generated or bytecodes executed\(^2\). This serves as an early diagnostic aid in situations when no information about concerns is available otherwise. A human could review the analysis results to identify concern topics that have a disproportionately high contribution to resource usage compared to their perceived utility and mark these as potential candidates for de-bloating.

By integrating all these insights in Chapter 7, we created a foundation for developing a principled design approach to minimize propensity for bloat by exercising multiple control points from a systems perspective. We find a parallel between engineering principles for energy efficient design in hardware systems and those for avoiding bloat in software. There is an analogy between the prevalent design practice of overprovisioning hardware capacity and the prevalent development practice of overprovisioning software features. Just as energy proportionality of hardware components avoids energy wastage due to excess hardware capacity, our proposed principle of resource proportionality of software features would avoid runtime resource bloat due to excess software features.

The techniques that we have developed can help developers estimate and tackle sources

\(^2\)using a probabilistic form of the CAPA approach introduced in Chapter 5
of non-resource proportional behavior in software components. Just as reducing data and instruction supply overheads has been found to lower the energy efficiency gap between general purpose embedded processors and special purpose ASICs [40], we have proposed how mechanisms that reduce recurring overheads of structural interactions due to optional features in general purpose software components would lower propensity for bloat in applications built using these components.

This may not always be straightforward to achieve under existing hardware and software models where these overheads are not explicitly visible. To gather further insight, Chapter 8 proposes a new research direction: exploring whether there are alternate software models and hardware mechanisms that could enable software to be designed in a way that makes it easier to detect and mitigate bloat. We have presented some radical ideas to illustrate this possibility and inspire future research on architectural mechanisms for virtualizing bloat but leave further investigation as an open problem for future research.

Through this dissertation, we have identified several challenging aspects of the multi-dimensional nature of research needed to develop a deeper understanding of problem of bloat and its energy efficiency implications. While we have initiated an exploration into these aspects, our work has opened up deeper questions for future research in each of these directions:

- How to extend the scope of program analysis to tackle difficult problems that require reasoning across intent, structure and dynamics of software programs, e.g. discovering intent from natural language cues in source text or human input, structure from static program analysis, dynamics from runtime measurements and analytical models.

- How to adopt a full systems perspective across hardware and software in properly evaluating and deploying solutions for bloat, e.g. the impact of hardware energy proportionality and bottlenecks in traditional systems on the one hand and the use of radical architectural mechanisms or alternate memory organizations (e.g. a virtual TCAM) on the other.

- How to use the abstract concept of resource proportional software as a guiding framework for developing a systematic approach to bloat mitigation or building software and
systems that minimize the impact of bloat.

Thus, although software bloat is a consequence of technological trends that have favored development productivity over run-time efficiency, our work shows that addressing the above aspects would make it possible to save energy lost due to bloat without sacrificing flexibility or ease of development. We hope that the findings in this dissertation would be helpful in adopting an integrated analysis of software bloat and hardware platforms towards realizing flexible software that’s also green.
Bibliography


