Compiler-Directed Frequency and Voltage Scaling for a Multiple Clock Domain Microarchitecture

Arun Rangasamy, Rahul Nagpal and Y.N.Srikant
Dept. of Computer Science and Automation, Indian Institute of Science
Bangalore, Karnataka, India
arunr@csa.iisc.ernet.in, rahul@csa.iisc.ernet.in, srikant@csa.iisc.ernet.in

ABSTRACT

Multiple Clock Domain processors provide an attractive solution to the increasingly challenging problems of clock distribution and power dissipation. They allow their chips to be partitioned into different clock domains, and each domain's frequency (voltage) to be independently configured. This flexibility adds new dimensions to the Dynamic Voltage and Frequency Scaling problem, while providing better scope for saving energy and meeting performance demands.

In this paper, we propose a compiler directed approach for MCD-DVFS. We build a formal petri net based program performance model, parameterized by settings of microarchitectural components and resource configurations, and integrate it with our compiler passes for frequency selection. Our model estimates the performance impact of a frequency setting, unlike the existing best techniques which rely on weaker indicators of domain performance such as queue occupancies (used by online methods) and slack manifestation for a particular frequency setting (software based methods).

We evaluate our method with subsets of SPECFP2000, Mediabench and Mibench benchmarks. Our mean energy savings is 60.39% (versus 33.91% of the best software technique) in a memory constrained system for cache miss dominated benchmarks, and we meet the performance demands. Our ED^2 improves by 22.11% (versus 18.34%) for other benchmarks. For a CPU with restricted frequency settings, our energy consumption is within 4.69% of the optimal.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors —*Compilers*

General Terms

Algorithms, Measurement, Performance

Keywords

Energy, Dynamic Energy, DVS, Multiple Clock Domains

CF'08, May 5–7, 2008, Ischia, Italy.

1. INTRODUCTION

Multiple Clock Domain (MCD) processors provide an attractive solution to the increasingly challenging problems of clock distribution and power dissipation [1]. They allow their chips to be partitioned into different clock domains, and each domain's frequency and voltage setting to be independently configured. The most popular MCD proposal partitions the Alpha 21264 CPU into four domains [7]. This extra flexibility adds new dimensions to the Dynamic Voltage and Frequency Selection problem, while providing much better opportunities for reducing power dissipation and meeting performance demands.

The MCD-DVFS problem (for a program and a processor system) is to assign and effect frequency (voltage) settings for each domain of the processor, with possibly different settings for different intervals of execution time, so as to minimize the energy consumption for program execution, while maintaining any performance degradation below a specified limit. This problem has been solved in different ways, in hardware (online) and in software.

Most online methods [2, 3, 15] use local queue occupancies to select frequency (voltage) settings independently for each domain. They always run the fetch unit at the highest speed. As a result, a high queue occupancy in a domain is not always an indicator of the domain being a performance bottleneck. For instance, a cache miss in a system with a finite number of memory ports, coupled with an aggressive fetch domain and a floating point domain which cannot remove operations as fast as the fetch domain provides, could lead to a high FP queue occupancy, while the real bottleneck could be the memory system.

The profile based software method [6] (henceforth called PDFVS) constructs a detailed precedence graph of microarchitectural events, when all domains are run at the highest frequency, to identify slack. However, graphs can not accurately model resource constraints, and the performance of PDFVS is dependent on the way slack is manifested for the simulation at the highest frequency.

The existing online and profile based methods rely on weak indicators of performance (queue occupancy or slack manifestation for one particular frequency setting) for their decisions of frequency scaling. As a result, their decisions could be conservative.

We have developed a compiler directed approach for frequency (voltage) selection (CDFVS). We build a formal petri net based program performance model, parameterized by settings of microarchitectural components and resource configurations, and integrate it with our compiler passes for fre-

quency selection. Using petrinets, we can capture resource constraints characterizing a microarchitecture, and data dependencies characterizing a program, and hence can model performance better. Unlike the existing methods, CDFVS directly estimates the impact of a frequency setting on the execution time for a program region (using petrinet simulation), instead of relying on weak indicators of performance, and is independent of how slack is manifested.

Our main contributions are as follows:

1. We propose a formal petrinet based program performance model, parameterized by settings of microarchitectural components and resource configurations.
2. We propose a new compiler directed approach for voltage and frequency scaling for MCD architectures using this model. To the best of our knowledge, this is the first approach that formally deals with the MCD-DVFS problem at the compiler level.
3. We have developed a complete framework, integrating the model with the new machsuif compiler passes [14] for MCD-DVFS.

We evaluate the proposed approach with subsets of SPEC FP2000, Mibench and Mediabench benchmarks in the MCD simulator. The experimental results show that our approach is indeed effective.

The rest of the paper is organized as follows. We present the motivation for our approach with working examples demonstrating CDFVS' unique strengths in Section 2. Section 3 describes our approach. Section 4 presents our experimental setup, performance results, comparison with PDFVS, and the optimal setting for a few media benchmarks. Section 5 summarizes related work. We conclude in Section 6 with future directions.

2. MOTIVATION

In this paper, we use the MCD architecure proposed in [1]. The proposal partitions the Alpha 21264 CPU into the FrontEnd (ICache, Fetch, Rename, Dispatch and Commit Units), Integer (Integer ALU's and Register Files), FloatingPoint (FP ALU's and Register Files) and LoadStore (L1 Data Cache and Unified L2 Cache) domains.

In CDFVS, we first build a program performance model, and use it to estimate the impact of a frequency setting on program performance. A timed petrinet model, derived from a detailed precedence graph and resource availability serves as the performance model. The expressiveness of petrinets for modeling resource constraints and data dependencies is unique (among the existing queue utilization based methods [2, 3, 15] and the profile based method [6]). We then choose the lowest frequency setting that does not degrade the performance beyond a specified threshold.

A petrinet is a three tuple (P,T,A), where P is the non-empty set $\{p_1, p_2, \ldots p_n\}$ of places, T is the non-empty set $\{t_1, t_2, \ldots t_m\}$ of transitions, such that $P \cap T = \emptyset$, and $A \subseteq PXT \cup TXP$. A marking is a function $M : P- > I$, where I is the set of non-negative integers, indicating the number of *tokens*, a place has. A transition can *fire* if all places incident into it have a non-zero token count. When a transition fires, the token count of each place incident into it is decremented by one, and the token count of each place incident out of it is incremented by one. A timed petrinet is a tuple (PN, Ω) where PN is a petrinet and $\Omega : T- > \Re^+$ assigns the time required by each transition t_i in PN to fire ($\Re^+$ is the set of positive reals).

The non-zero token count condition for transition firing and the firing time can be used to model the conditions for an instruction's entry to a pipeline stage and the latency of the pipeline stage [9].

The nodes of the precedence graph (similar to [17]), used for deriving the petrinet, model pipeline stages of instructions. An edge from node i to node j in the graph means that the instruction at stage i must leave that stage before the one at stage j can enter. Edges abstract precedence constraints (eg. data dependencies, commit to dispatch edges due to Finite Re-Order Buffer (ROB) entries).

From the precedence graph and the microarchitectural resource specification, we derive the petrinet by

- creating a transition t_i for each node i of the graph
- creating a place $p_{i,j}$ for edge (node i, node j) of the graph; the place has t_i as an input transition and t_j as an output transition
- creating a place for each resource type; a transition using that resource is both an input and output transition for that place; the token count is initialized to the number of instances of that resource

Frequency Settings determine how long it takes for a transition to fire, once all its input places have sufficient tokens. For a given frequency setting, we estimate the execution time of a region by finding the average time taken for the last transition to fire, by simulating the petrinet.

Our frequency selection procedure orders domains in the decreasing order of power consumption, and carries out a binary search on the frequency settings, using petrinet simulation for estimating the performance for a setting. Binary Search heuristic helps reduce the number of frequency settings to consider. Our method then chooses the lowest frequency setting that meets the performance constraints.

We now present two examples, demonstrating the working of the CDFVS technique and showing that the frequency of a domain can be scaled down considerably even if its queue occupancy is high. In the interest of space, our explanations are brief. Detailed explanations could be found in [5].

2.1 Looking beyond a Single Schedule

Figure 1 presents a loop for which an energy efficient frequency setting that does not degrade performance has to be found. The target processor has a branch unit, a saturating adder, an ordinary adder, a logic unit, and a 24 entry ROB. It can fetch, issue and commit 6 instructions per cycle. Only the frequency setting of the saturating adder is configurable: it can operate at either the highest frequency, or half of it. The processor resolves resource conflicts by giving priority to the earliest instruction instance.

Figure 1 also shows a highly simplified version of the precedence graph, with only Execute (odd numbered nodes) and Commit (even nodes) stage for each instruction, our method constructs. Each column represents one instruction.

Each edge has two labels - a latency (l) and a distance (d). The latency of edge (m,n) specifies the minimum time l after which an instruction of an iteration $k + d$ can enter the pipeline stage n, once the instruction of iteration k has entered the stage m. For instance, the ROB size induced

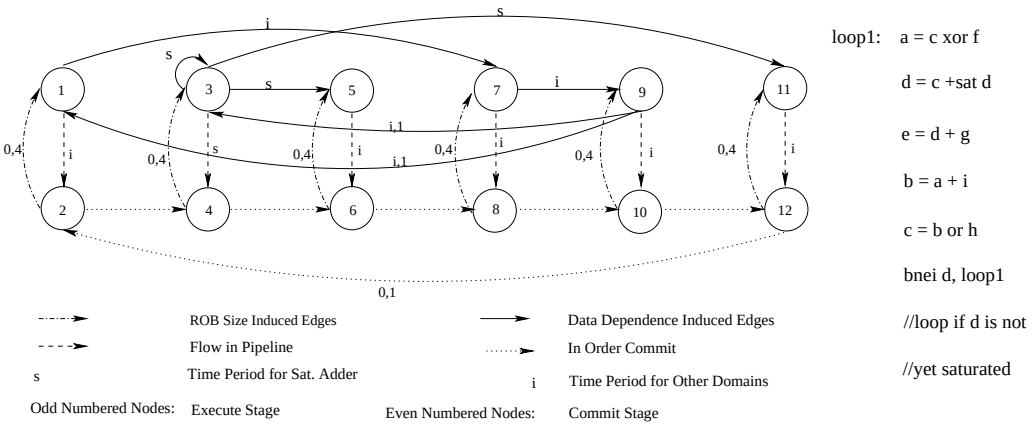


Figure 1: A loop and its Simplified Precedence Graph

Time	s = 1		s = 2	
	Transitions Fired	Comments	Transitions Fired	Comments
0	T1, T3	None else is ready	T1, T3	None else is ready
1	T5, T11, T2, T4	T5 given priority over T7	T7, T2	T3 - not over; T5 not ready
2	T7, T6	T9 depends on T7	T5, T9, T11, T4	No resource conflict
3	T9, T8	T1, T3 depend on T9	T1, T3, T6, T8, T10, T12	Iter 1 ends; 2 begins
4	T1, T3, T10, T12	Iter 1 ends; 2 begins	T7, T2	T3 - not finished; T5 not ready
5	T5, T11, T2, T4	T5 given priority over T7	T5, T9, T11, T4	No resource conflict
6	T7, T6	T9 depends on T7	T1, T3, T6, T8, T10, T12	Iter 2 ends; 3 begins
7	T9, T8	T1, T3 not ready yet	...	...
8	T1, T3, T10, T12	Iter 2 ends; Iter 3 begins	...	...

Table 1: Firing Sequences

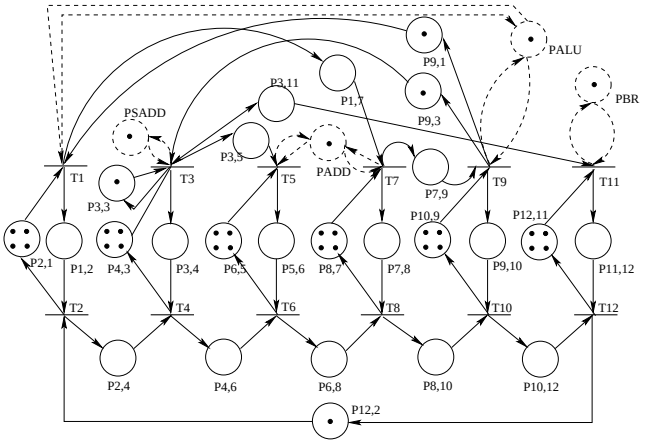


Figure 2: Petrinet based on Figure 1

edge from node 2 to node 1 indicates that the fifth instance of instruction 1 can enter the Execute stage only after the first instance enters Commit stage. The variable label i is always 1, and the value of s can be 1 or 2, depending on whether the saturating adder is operated at the highest frequency or half frequency.

Figure 2 shows the initial marking for the petrinet based on Figure 1, and the resource constraints. The token counts of all the places corresponding to loop carried dependences are initialized to the corresponding distances [9].

The left and right halves Table 1 show the firing of transitions, for s = 1 and s = 2, respectively. CDFVS estimates the loop throughput by averaging the time taken for firing transition T12.

At the end of the binary search (with s = 1 and s = 2), CDFVS chooses the half frequency setting for the saturating adder, since the throughput of the loop is maintained (in fact, improved to three cycles per iteration, from four, when the saturating adder is run at the highest frequency) with that setting.

The important point to note here is that CDFVS does not create an edge from node 5 to node 7 in Figure 1 just because they use the same resource, and they happen to execute in that order when the saturating adder was run at the maximum frequency. Adding such an edge (as PDFVS does [1,6]) creates a critical cycle with length 4 (nodes $3, 5, 7, 9$), and obscures the discovery of any better frequency setting [5].

2.2 Issue Queue Monitoring

Table 2 gives the assembly code for a simple vector reduction (sum) program, generated by machsuif. The problem is to find an energy efficient setting without degrading the performance beyond 5%. We will examine two different MCD CPU system configurations, very similar to the one in [1], except that there is one memory port, 120 ROB entries; In configuration CFG1, there are 112 integer and FP registers each, and in CFG2, there are 80 of them each.

There is an L2 miss every 8 iterations (loading d[i]). Therefore it takes atleast 10 cycles on an average, for an iteration (80 ns memory access time) to complete. This bound will be enforced by stalls. ROB fills and insufficient physical integer registers are the major stall contributors for configurations CFG1 and CFG2 respectively. The average Fetch, Integer, FP and LS queue occupancies are 99.3% (99.5%), 11.4% (11%), 70.9% (44.3%), and 51.5% (32.8%) respectively, for

```
main.L1:
mull $1,8,$3          /* r3 <= i * 8 */
lda $2,d              /* r2 <= d = base of vector */
addq $2,$3,$2         /* r2 <= EA(d[i]) */
ldt $f16,0($2)        /* f16 <= d[i] */
ldt $f31,64($2)       /* prefetch d[i+8] */
addt $f17,$f16,$f17   /* f17 <= partial sum + d[i] */
ldil $2,1
addl $1,$2,$1
ldil $2,20000         /* r2 = 20000 = vector size */
cmplt $1,$2,$2        /* Are we done? */
bne $2,main.L1        /* If not, next */
```

Table 2: Assembly Code Generated for Vector Reduction

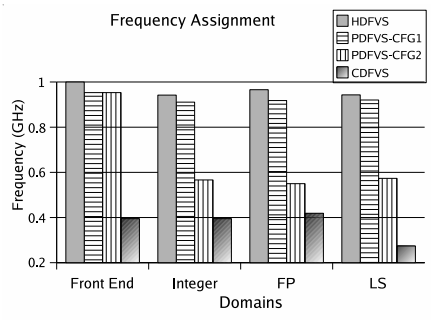


Figure 3: Frequency Assignments

CFG1 (CFG2), when all domains are operated at the highest frequency.

CDFVS selects the same low frequency settings for CFG1 and CFG2, (Figure 3) saving about 81.2% energy for 1.6% degradation [5]. PDFVS saves 18.24% and 62% respectively, while the heuristic online controller (HDFVS [3]) saves atmost 4.5% energy, for no performance loss.

These experiments demonstrate that queue lengths are not strong indicators of performance bottlenecks for memory constrained systems with an aggressive fetch unit. When memory is the bottleneck, sometimes, even an unlimited number of ROB entries may not help.

Unlike the existing online and profile based methods which rely on how the slack is manifested, CDFVS directly estimates program performance for a frequency setting, and can potentially find more energy efficient frequency settings, while meeting the performance requirements.

3. THE CDFVS METHOD

Figure 4 gives a high level flow of the CDFVS compiler pass. CDFVS first identifies the set of frequency reconfiguration points (Region Identification), and then assigns a frequency setting for each point. It uses path and cache miss profiles for determining an energy efficient frequency setting. The steps in Figure 4 are explained in detail in the rest of this section.

3.1 Identifying Reconfiguration Points

Like PDFVS, CDFVS considers loop and function boundaries as potential reconfiguration points. For each function, a program region hierarchy tree whose nodes are regions of the function's flow graph is constructed. A region is either a natural loop [19] or an acyclic graph whose nodes are regions or basic blocks. In Figure 5, there are 3 regions: the inner loop R1, the outer loop R2 and the function main R3. There is an edge from node i to node j in the tree if j is nested

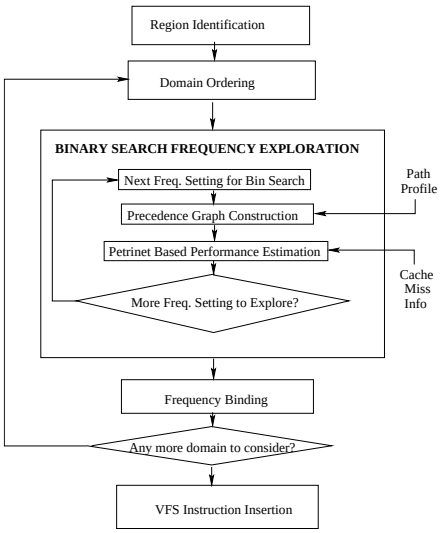
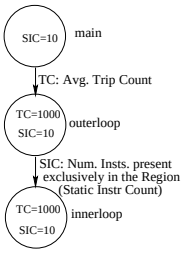


Figure 4: The CDFVS Compiler Pass




```
main() {
/* Region R3 */
  int count1 = 0, count2 = 0;
  for (i = 0; i < 10000; i++) {
    /* Region R2 */
    count2++;
    for (j = 0; j < 10000; j++) {
      /* Region R1 */
      count1++;
    }
  }
}
```

Figure 5: Regions: An Example

Figure 6: Reconfiguration Points: An Example

within i. The program region hierarchy tree for the code in Figure 5 will be isomorphic to that in Figure 6. CDFVS requires that the flow graph be reducible; It does not handle recursive procedures, currently.

Table 3 defines some properties of regions. CDFVS uses them in the following rules for identifying reconfiguration points. Since it takes some time for a new frequency setting to take effect [1], CDFVS chooses the boundaries of those regions in which atleast 10000 instructions are likely to be executed before a transition to a new setting, to initiate frequency reconfigurations [6].

If a region does not have any *LongRunning* descendant, and its *AIC* is less than 10000, then it is not a *LongRunning* region; its *EICForOuterRegion* is the sum of the *EICForOuterRegion* of its children and the instruction counts of the basic blocks exclusively present in it, for each entry into the region. If a region has a *LongRunning* descendant, it is *LongRunning*, if the ratio of the above sum and *ATC* is more than 10000. In this case, *EICForOuterRegion* is 0. Otherwise, it is not *LongRunning* and its *EICForOuterRegion* is set to the ratio.

For Figure 6, we do not want to consider the outer loop as a reconfiguration point, even though its *AIC*, without the contribution from the innerloop is 10000. This is because the outer loop's frequency setting will be effective for atmost 10 instructions: the inner loop's setting will override it.

Property	Definition
NumEntries	Number of entries to the region from a basic block outside the region
AIC	Average number of instructions executed per entry
ATC	Average tripcount for natural loops, 1 for acyclic regions
LongRunning	True iff the region entry and exit are frequency configuration points. If the region does not have a *LongRunning* descendant and its $AIC >= 10000$, it is *LongRunning*
EICForOuterRegion	A measure of the average number of instructions executed per entry, *before a reconfiguration*. For *LongRunning* regions, this is set to 0, since there is an immediate reconfiguration after entry

Table 3: Properties of Regions

Applying these rules, $AIC(R1) = 10000$, and it is *LongRunning*. $AIC(R2) = 10010000$, but it has a *LongRunning* child, so the ratio is $(0 + 10000)/1000 = 10$, and CDFVS will not classify the outerloop as a reconfiguration point (in contrast to PDFVS [6]).

3.2 Frequency Assignment

For each region, CDFVS carries out a binary search to find a low frequency setting meeting the performance constraints. It uses petrinet simulation at each setting for performance estimation. The petrinet simulation is implemented in two layers: the Basic Petri Net module, which handles innermost loops without function calls and instruction sequences, and the Petri Net Wrapper module which decomposes an arbitrary region into a sequence of innermost loops and instruction sequences, based on path profile.

The rest of this section briefly describes the details involved in the frequency assignment for a region.

3.2.1 Domain Ordering

Since there are a large number of all possible frequency settings, CDFVS considers the Load Store, FP, Integer and Front End domains in this decreasing order of power consumption (like PDFVS). It finds a setting for one domain, and fixes it before considering another domain.

3.2.2 Path Profiling

We use a variant of the Ball Laurus Path Profiling (BLPP) algorithm [10], called Hierarchical Path Profiling (HPP) [20] to find the most frequently executed paths from the entry node to the exit node of a region. This is a context insensitive profile, but PDFVS has already established that context sensitivity does not add much value for the SPEC and MediaBench benchmarks [6].

3.2.3 Cache Hit/Miss Profiling

Misses to memory provide excellent opportunities for frequency scaling. However, it is critical to get an accurate estimate of misses, because a lower estimate affects competence whereas a higher estimate could inflict performance loss. Currently CDFVS exploits cache misses with regular strides (atleast 70% of all strides must be contributed by a single stride).

How frequently a memory access instruction results in a miss depends on many factors (eg. access stride). In general, during program execution, the period of occurence of miss could vary. We consider only the members of the smallest set S of periods which totally contribute to atleast 80% of the miss periods. For each miss period $p \in S$, we account for $p-1$ partial hits, and we treat all partial hits as misses. We classify an access as a frequent miss, if the number of misses computed this way is atleast 70% of the total number of accesses. We conservatively set the period of occurence of miss as the largest member in S. This period will be used in petrinet simulation for modeling cache misses (Please refer Table 6).

For instance, if for a load instruction, the miss period of 8 occurs twice, and there are 14 hits, then this heuristic will decide that there are $2 * (8 - 1) = 14$ overlapped hits. Therefore there are 16 misses (including overlapped hits), and CDFVS classifies this load instruction as a miss, with period 8. We found that this heuristic identifies frequently missing instructions with regular strides and miss periods reasonably well (Please refer [5] for details).

3.2.4 Binary Search Frequency Exploration

For each domain, binary search is carried out to find the least frequency setting that does not degrade the performance beyond a specified fraction of the estimated base performance (when all domains run at the highest frequency). In practice, an attempt is made to reduce the frequency of all domains to 700 MHz, then to 450MHz, and 250 MHz [5]. We will see that this heuristic is effective, in Section 4. The entire procedure takes only a few minutes for most benchmarks, although for some (whose *LongRunning* regions have several subregions), it takes about 8 hours.

For every frequency setting explored in the Binary Search, a precedence graph is constructed. From this, a Petri Net is derived, and it is simulated to estimate performance.

Precedence Graph Construction:

As described in Section 2, the nodes of the precedence graph (From now onwards, we will use the terms nodes and transitions interchangably) (based on [17]) correspond to the pipeline stages an instruction goes through. An edge from node i to node j in the graph means that the instruction at stage i must leave that stage before the instruction at stage j can enter that stage. Table 4 lists the nodes of the precedence graph and their meaning; Table 5 specifies the edges (Pipeline stage flow edges are not listed). More information about the precedence graph can be found in [17] and [8]. Unlike [17] and [8], resource constraints are not modeled in the precedence graph, and are modeled during petrinet simulation.

Petri Net Simulation:

Petri Net Simulation is implemented in two layers: The Basic Petri Net, and The Petri Net Wrapper. The wrapper module streams a sequence of innermost loops and instruction sequences to the basic petri net module, which returns control back to the wrapper once it is done with the simulation. In the interest of space, only an outline of these modules is presented here. More details can be found in [5].

The Basic Petri Net:

As described in Section 2, it is derived from the precedence graph and the microarchitectural resource specification.

Node	Microarchitectual Event
Fetch (F)	Fetched from ICache to Fetch Queue
Dispatch (D)	Register Renamed and routed to the corresponding issue queues; An issue queue entry is consumed
Ready (R)	Wait in the queue, for the operands to be available
Execute (E)	A functional unit is reserved and the instruction gets executed in the corresponding functional unit; The issue queue entry is freed immediately after the first cycle; If the functional unit is pipelined, the reserved unit is released immediately after the first cycle; Else released after latency cycles; For loads and stores, EA computation is modeled in this node type
Mem (M)	This node is present only for memory access instructions. Cache access take place in this stage
Write Back (W)	Write Results to registers and wait in ROB till it is time to commit
Commit (C)	Update architectural state; Free up rename register, ROB entry

Table 4: Description of Precedence Graph Nodes

Edge Name	Constraint Modeled	Edge	Comments
IF	In Order Fetch	$F_{i-1} \rightarrow F_i$	Latency is one fetch period, either if $i-1$ corresponds to last inst. of loop or i is a multiple of 4 (fetch bandwidth); Else 0
FBS	Fetch Buffer Size	$D_{i-fbs} \rightarrow F_i$	fbs = maximum number of insts. that can be in Fetch Queue
ID	In Order Dispatch	$D_{i-1} \rightarrow D_i$	lat. is 0
DBW	Dispatch Bandwidth	$D_{i-dbw} \rightarrow D_i$	dbw = maximum number of insts. that can be routed, renamed and dispatched per cycle; lat. = fetch period
IC	In-order Commit	$C_{i-1} \rightarrow C_i$	lat. is 0
CBW	Commit Bandwidth	$C_{i-cbw} \rightarrow C_i$	cbw = commit bandwidth
ROB	Finite ROB Size	$C_{i-w} \rightarrow D_i$	w = #ROB entries
DD	Data Dependences	$W_j \rightarrow R_i$	Added if inst. j produces source for inst. i, and i is not a store depending on j for the value to be stored; If j produces a value to be stored by i, then edge is from W_j to C_i
RR	Finite Rename Registers	$C_i \rightarrow Dj$	there is rrnum number of new integer results from instruction i to instruction j; rrnum = number of extra rename registers present
TSO	Ordering of Stores	$R_i \rightarrow M_j$	i is a store and j is a memory operation
BL	Latency of Branch Resolution	$W_i \rightarrow F_{i+1}$	For all but loop closing branches to fetch targets

Table 5: Description of Precedence Graph Edges

The petrinet module uses a priority queue (based on firing times of transitions) to simulate the firing of transitions. Each time a transition fires, it updates the token counts of all its outgoing places, and sets a lower bound on the firing times of its successor transitions (transitions corresponding to the successor nodes in the precedence graph). If all the incoming places of a transition have nonzero token counts, it will be enqueued in the priority queue, ready to be removed at the earliest possible time (determined by the predecessors). Its firing will then be determined by resource availability (steps 4-7 in Table 6).

The petrinet module supports simulation of innermost loops with a specified trip count. It maintains three pairs of queues to honour the precedence constraints among the dynamic transitions (eg. multiple transitions corresponding to different iterations of the same (static) transition in a loop) : one pair each, for the list of pending instructions (not yet simulated for trip count times) and the pending instructions which will write to the integer and the floating point registers. One queue in each pair (called the TraceQueue) keeps track of static instructions and the other (called the ROBQueue) keeps track of dynamic instructions. Using these queues, it is possible to track the set of instructions (and hence, transitions) on which an instruction (a transition) depends.

Clearly, the TraceQueue will be updated when the wrapper module streams an instruction sequence or a loop, or when the last instance of a static instruction commits. The ROBQueue will be updated when a dynamic instruction enters the fetch stage, or when it commits (Steps 8-10 in Table 6). The petrinet module also supports operations Flush, which fires all pending transitions, and Reset, which resets the time maintained by the module to 0.

```
Enque the first transition in Priority Queue PQ;
Set its FiringTime to CurrentTime.
While (∃t : t is a fetch transition and
  has not been fired for TripCount times) {
  1. t = PQ.Deque(); // Remove a transition from PQ
  2. CurrentTime = t.FiringTime; // Advance Time
  3. Update Resource Usage;
  4. If ResourcesAvailable(t) {
    5. Update Token Count of elements of P;
       P = {p|p ∈ OutPlaces(t)};
    6. Update Firing Time of elements of T;
       T = {s|s ∈ Succ(t)};
    // Cache misses are modeled in Step 6. The transition
    // for the MemAccess stage of a frequently missing
    // memory instruction will have a firing time which will
    // reflect when memory will complete the request.
    7. Enque elements of R in PQ; R = {r|r ∈ Succ(t) and
       ∀p ∈ InPlaces(r), TokenCount(p) > 0} ;
    8. Enque Fetch Transitions of R in ROBQueue;
    9. If t is a Commit Transition, remove t
       from ROBQueue;
    10. If t is a Commit Transition and has been fired
        TripCount times, remove t from TraceQueue;
  } else {
    11. Update FiringTime(t) to Earliest
        Resource Availability Time;
    12. PQ.Enque(t);
  }
}
```

Table 6: Petrinet Simulation

```
EstimatePerformance(⟨ProcID, RegionID⟩) {
//Petrinet Wrapper
  if (⟨ProcID, RegionID⟩ is an innermost loop without calls) {
    Use Petrinet Simulation to Estimate Performance;
    Update Execution Time;
  } else { // Not an innermost loop without function calls
    for each HotPath p from Entry to Exit of
      ⟨ProcID, RegionID⟩ {
      for TripCount(⟨ProcID, RegionID⟩) * Freq.(p) times {
        for each node n in p {
          if (n is a loop and n is not LongRunning) {
            EstimatePerformance(⟨ProcID, n.RegionID⟩);
            Update Execution Time;
          } else if (n is a function call and
            not LongRunning) {
            EstimatePerformance(⟨n.ProcID,
              n.OutermostRegionID⟩);
            Update Execution Time;
          } else if (n is not a region) {
            Simulate the Basic Block n in Petrinet;
            Update Execution Time;
          } else { // n is LongRunning
            Flush Petrinet and Update Execution Time;
            Reset Petrinet;
          } // n is LongRunning
        } //end for each node
      } // end for TripCount
    } // end for each hotpath
  } // not an innermost loop
} // end EstimatePerformance
```

Table 7: Petrinet Wrapper

The Petrinet Wrapper:

The petrinet wrapper generates the decomposition of an arbitrary program region by calling itself recursively on encountering a subregion (a loop or a function call). It ignores *LongRunning* regions (which will be simulated separately). For other regions, it invokes the petrinet module to estimate performance for each unit of decomposition. It abstracts conditional control flow by distributing the simulations of a region over different paths based on path profile.

The high level flow for the wrapper is shown in Table 7. It takes as input, a procedure id and a region id and increments the overall execution time with the estimated execution time for the region.

An instruction to effect voltage-frequency changes is implemented in the MCD simulator [1], and this instruction is used by the CDFVS pass to effect voltage/frequency changes.

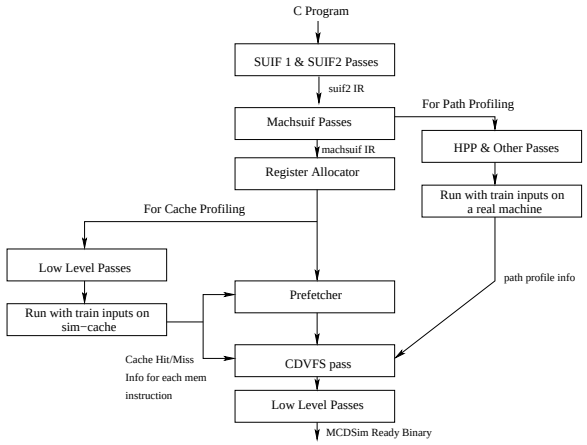


Figure 7: Our Compilation Framework

Parameter	Value
MCD	
Frequency Range	250MHz - 1GHz
Domain Voltage Range	1.083V - 2.0V
Domain Clock Jitter	± 110 ps, norm. distrn.
Inter-domain Synch Window	300 ps
Branch Predictor	
Branch Predictor	Hybrid (bimodal and 2 level)
Bimodal Table Size	2048 entries
2Level Config	1024 (L1), 1024 (L2), 12 bit hist.
Combining Predictor	4096 entries
BTB Size and Associavity	4096, 2
Branch Mispred. Penalty	7
Memory System	
L1 DCache	64K, 2 Way SA 64B, 2 cycle hit
L1 ICache	64K, 2 Way SA, 64B, 3 cycle hit
L2 UCache	1M, DM, 64B, 12 cycle hit
Mem. Access latency	80 cycles
Num. Mem. Ports	1
Num. Pending Mem. req.	8
Other Resources	
Decode/Issue/Retire Width	4/6/11
Integer ALUs	4; 1 multiplier
FP ALUs	2; 1 FPMult, 1 FPDiv, 1 FPSqrt
Issue Queue Size	20 Int, 15 FP, 64 LS
ROB Size	80
Extra Rename Registers	41 Int, 41 FP

Table 8: Simulator Configuration

4. EXPERIMENTAL EVALUATION

We have implemented our (cross) compiler passes in the machsuif [14] and suif [13] framework. Figure 7 shows the overall flow for compilation, indicating the flow for path and cache statistics profiling. In addition to the CDFVS pass, we have also implemented a profile based prefetching pass. It is shown in [12] that profile guided prefetching, in general, improves performance. Except for the instructions for frequency scaling, the binaries for CDFVS, PDFVS and HDFVS are the same, for a fair evaluation. We use sim-cache [4] for profiling cache statistics. More information about the working of these passes can be found in [5].

We have slightly modified the base MCD simulator (based on [4] and [22]) to implement a single memory port with 8 pending misses to memory and a prefetch instruction. Table 8 specifies the CPU configuration, similar to [7] and [6].

We have used subsets of SPECFP2000, Mediabench and Mibench for evaluation. We could compile and run only three SPEC benchmarks in our cross compiler (Redhat/IA32 for alpha assembly code) setup. We use SimPoint [11] to skip the initialization phase of SPECFP benchmarks.

4.1 Performance Statistics

Figure 8 shows the number of cycles, energy consumption, EDP and ED^2P achieved by HDFVS [3], PDFVS (with L+F definition - no context sensitivity, potential reconfiguration at boundaries of loops and function calls - as suggested in [6]) and CDFVS, as ratios with respect to the baseline where all domains run at full speed. The maximum allowed performance degradation is set to 5% [6]. Both HDFVS and PDFVS are packaged with the MCD simulator.

4.2 Analysis of Results

On an average, CDFVS outperforms PDFVS and HDFVS in all metrics - it suffers less performance degradation, saves more energy, and has better EDP and ED^2P improvement.

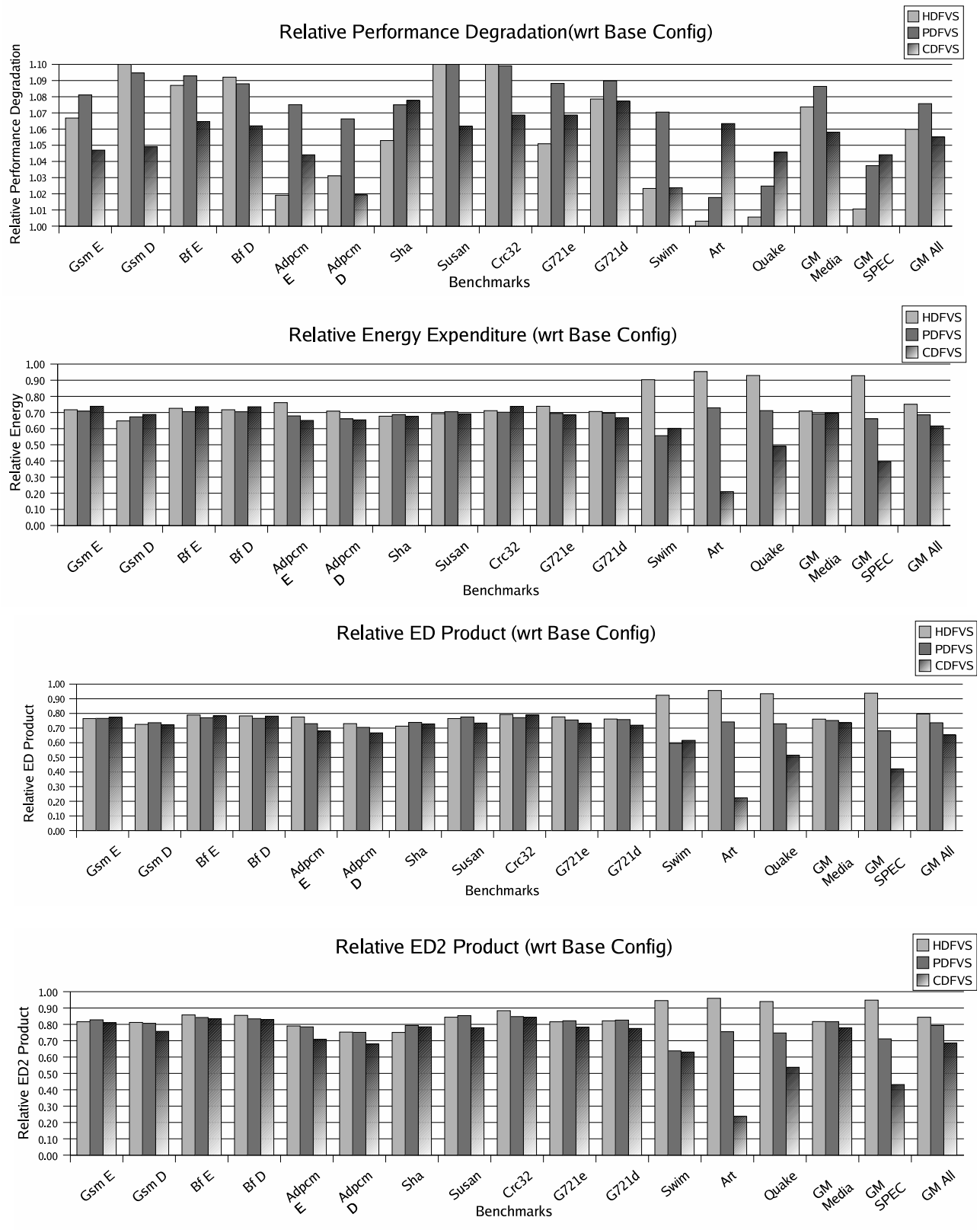


Figure 8: Results for the Heuristic Online, Profile based Method and Compiler based Method

The SPEC FP and the media benchmarks have strikingly different cache miss statistics. The SPEC FP benchmarks have a large number of L2 misses, whereas the media benchmarks have almost negligible L1 (both instruction and data) misses. This is because the media benchmarks have a very small working set which fits in cache, and of the abstraction of file reads by simplescalar. Naturally, the SPEC benchmarks offer more opportunities for energy savings.

For SPEC benchmarks, CDFVS saves significantly higher energy on an average than PDFVS (60.39% savings of CDFVS vs. 33.91% of PDFVS), while keeping the average performance degradation below 5%. The performance degradation and ED and ED^2 improvements of CDFVS (PDFVS) over the baseline are 4.41% (3.74%), 57.89% (31.88%) and 56.82% (28.87%) respectively.

For media benchmarks, the difference in energy savings is negligible (PDFVS saves slightly higher energy of 30.81% compared to 30.43% of CDFVS), but PDFVS degrades performance much more - 8.64% vs. 5.81% of CDFVS. Hence CDFVS achieves better ED and ED^2 improvements : 26.4% (24.83%) and 22.11% (18.34%), for CDFVS (PDFVS) respectively. For both classes of benchmarks, CDFVS does better than HDFVS. CDFVS achieves better ED^2 product than PDFVS consistently across *all* benchmarks.

Among the SPEC benchmarks, PDFVS saves more energy than CDFVS in swim. One distinct feature of swim is the siginificant fraction of stalls due to the non-availability of the Miss State Holding Registers, which hold the state of outstanding misses to memory (27%, among stalls due to insufficient ROB entries, MSHRs, Integer and FP physical registers, Integer and FP issue queue entries) - For others it is less than 3%. CDFVS currently assumes that there can be an unlimited number of outstanding misses. However, even for swim, the ED^2 improvement of CDFVS is better.

HDFVS saves less energy in both classes of benchmarks. It fares relatively better in the media benchmarks, where the average queue occupancies for the FP and Load/Store domains are very low (0% and 5.57% respectively). It reduces the frequencies of both the domains. In SPECFP benchmarks, the average queue occupancies are high (37.33%, 27.84% and 28.59% for the Integer, FP and Load/Store domains). It does not scale down the frequencies much, and saves less energy, even though there are better opportunities.

Because [6] uses alpha compiler generated code and unlimited memory ports with no bank conflicts, and we use machsuif generated code and a single memory port, we do not get the exact numbers in [6]. Yet, the relative trends are the same: For SPEC benchmarks (with relatively less ILP due to the cache misses), PDFVS saves more energy and suffers less performance degradation, whereas for media benchmarks, PDFVS saves less energy and suffers more performance degradation; CDFVS too follows the trend, though it saves significantly more energy in SPEC benchmarks, and suffers relatively less degradation in the media benchmarks.

To conclude, CDFVS can exploit performance constraints imposed by the memory system for high energy savings, if they can be characterized well with a small set of inputs. This is because the petrinet model captures resource constraints much better than a static precedence graph built based on the event trace obtained during the execution for one frequency setting. Consequently, we can expect the petrinet based model to be significantly more effective for a resource constrained system (as the results indicate).

Even when the memory system is not the bottleneck, CDFVS achieves competetive energy savings, for a much lower performance degradation. CDFVS achieves these, without the need for a detailed simulation.

4.3 CDFVS vs Optimal Frequency Setting

To quantify how far CDFVS is away from an optimal frequency setting (A frequency setting that saves most energy while meeting the 5% performance degradation constraint), we have conducted some experiments on a configuration with a restricted number of frequency settings : 5 settings for each domain, equally spaced from 700MHz to 1000 MHz. Otherwise, the configuration is the same as in Table 8.

We consider only those media benchmarks for which there is only one *LongRunning* region - The number of simulations needed to find the optimal setting is exponential in the number of *LongRunning* regions [5]. For these benchmarks, FP units are unused, so we manually run the FP domain at the lowest frequency, when enumerating all possible frequency settings [5]. CDFVS chooses the least frequency setting for FP domain [5]. Hence we enumerated 125 different frequency settings (against 32^4 settings in the original configuration); We run the benchmarks for the first 100 million instructions.

In Table 9 we show the ratio of energy consumption of the CDFVS and that of the optimal setting, and the performance degradation of the optimal setting and CDFVS with respect to the baseline. We find that CDFVS indeed achieves energy savings close to the optimal setting, deviating from the optimal by only 4.69% on an average, and 8.56% in the worst case.

Benchmark	$Energy_{CDFVS}$ / $Energy_{Opt}$	$PerfDeg$ for Opt	$PerfDeg$ for $CDFVS$
G721e	1.0000	1.0451	1.0451
G721d	1.0000	1.0441	1.0441
Crc32	1.0727	1.0463	1.0294
Adpcme	1.0801	1.0457	1.0129
Adpcmd	1.0856	1.0338	1.0071
Mean	1.0469	1.0430	1.0276

Table 9: CDFVS vs Optimal Setting: Comparison of Energy and Performance

5. RELATED WORK

The MCD microarchitecture that we use in CDVFS is proposed by Semararo et al. [1]; They also propose the shaker algorithm, which has been used by Magklis et al. [6]. A few enhancements to the base MCD microarchitecture have been proposed by Zhu et al. in [18]. Recently, different GALS microarchitectures have been proposed by Magklis et al. [24] with front-end scaling by the hardware.

Most hardware based online algorithms use local queue occupancies for frequency/voltage scaling decisions. The controller proposed by Semeraro et al. [3] uses a set of heuristic rules, while that of Wu et al. [2] is based on control theory. Wu et al. also propose a controller [15] whose reaction time is adaptive to application workload variations. Both [2] and [15] perform much better than [3].

Magklis et al. propose the profile guided approach [6] for MCD-DVFS (PDFVS). PDFVS makes a detailed precedence graph of microarchitectural events based on a cycle accurate simulation. Their method uses this graph to identify and exploit the available slack for frequency scaling.

[26] and [25], compiler based frequency (voltage) scaling for single clock domain systems, achieve impressive results.

We have heavily borrowed the precedence graph concepts, from the work of Fields, Rubin and Bodik [8] and Fields, Bodik, Hill and Newburn [17]. However, we model resources in the petrinet. The work of Karkhanis and Smith [21], and that of Joseph, Kapil and Mathew Jacob [23] are some of the more recent performance models for superscalar processors. Karkhanis' work inspired us to just do cache simulation for gathering cache hit and miss information.

Petrinets have been used to identify kernels for software pipelining [9]. Yang, Govindarajan, Gao, Cai and Hu [16] find rate optimal, energy efficient software pipeline schedules for inorder processors.

6. CONCLUSIONS AND FUTURE DIRECTIONS

We have proposed a new compiler based MCD-DVFS solution, CDFVS. CDFVS uses a formal petrinet based program performance model to find an energy efficient frequency setting. Our experimental results show that CDFVS is indeed effective. We would like to enhance CDFVS to handle recursive programs and reduce the number of reconfigurations. We want to model aggressive memory systems, and are looking at the possibility of scheduling to create more opportunities for energy savings for MCD processors.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We are grateful to Prof. Albonesi and his students for their MCD simulator; to Qiang Wu, Ram and Eswaran for facilitating its acquisition. Arun and Rahul are supported by Infosys and Philips fellowships, respectively. Kapil's feedback and suggestions throughout this work have been extremely useful. Prachee is one of the reviewers of the draft copy of this paper. We express our gratitude to all those who developed and maintain suif and machsuif.

7. REFERENCES

[1] G.Semeraro, G.Magklis, R.Balasubramonian, D.H.Albonesi, S.Dwarkadas, M.L.Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *HPCA*, pages 29-40, 2002

[2] Q.Wu, P.Juang, M.Martonosi, D.W.Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *ASPLOS*, pages 248-259, 2004

[3] G.Semeraro, D.H.Albonesi, S.G.Dropsho, G.Magklis, S.Dwarkadas, M.L.Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *MICRO*, pages 356-367, 2002

[4] D.Burger, T.M.Austin. The Simplescalar Toolset, Version 2.0. CS-TR-97-1342. Technical Report, Computer Science Department, Univ. of Wisconsin, June 1997

[5] R.Arun, R.Nagpal, Y.N.Srikant. Compiler-Directed Frequency and Voltage Scaling for a Multiple Clock Domain Microarchitecture. IISc-CSA-TR-2007-13. Tech. Report, Dept. of Comp. Sci. & Automation, Indian Institute of Science, Bangalore, Dec 2007

[6] G.Magklis, M.L.Scott, G.Semeraro, D.H.Albonesi, S.Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *ISCA*, pages 14-25, 2003

[7] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24-36, 1999

[8] B.Fields, S.Rubin, R.Bodik. Focusing Processor Policies via Critical-Path Prediction. In *ISCA*, pages 74-85, 2001

[9] M.Rajagopalan, V.H.Allan. Specification of Software Pipelining using Petri Nets. *Int. J. of Parallel Program.*, 22(3):273–301, 1994

[10] T.Ball, J.R.Larus. Efficient Path Profiling. In *MICRO*, pages 46-57, 1996

[11] G.Hamerly, E.Perelman, J.Lau, B.Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *In Workshop on Modeling, Benchmarking and Simulation*, 2005

[12] C.K.Luk, R.Muth, H.Patil, R.Weiss, P.G.Lowney, R.Cohn. Profile-Guided post-link Stride Prefetching. In *ICS*, pages 167-178, 2002

[13] http://suif.stanford.edu

[14] http://www.eecs.harvard.edu/hube/software

[15] Q.Wu, P.Juang, M.Martonosi, D.W.Clark. Voltage and Frequency Control with Adaptive Reaction Time in Multiple Clock Domain Processors. In *HPCA*, pages 178-189, 2005

[16] H.Yang, R.Govindarajan, G.R.Gao, G.Cai, Z.Hu. Exploiting Schedule Slacks for Rate-Optimal Power-Minimum Software Pipelining. In *COLP*, 2002

[17] B.A.Fields, R.Bodik, M.D.Hill, C.J. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In *MICRO*, pages 228-239, 2003

[18] Y.Zhu, D.H.Albonesi, A.Buyuktosunoglu. A High Performance, Energy Efficient GALS Processor Microarchitecture with Reduced Implementation Complexity. In *ISPASS*, 2005

[19] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. *Compilers: Principles, Techniques and Tools*, Chapter 9. Machine Independent Optimizations. Addison-Wesley, Second Edition, 2007

[20] Y.Wu, A.A.Tabatabai, D.A.Berson, J.Fang, R.Gupta. Hierarchical Software Path Profiling. US Patent 6848100, Jan 2005

[21] T.S.Karkhanis, J.E.Smith. A First-Order Superscalar Processor Model. In *ISCA*, pages 338-349, 2004

[22] D.Brooks, V.Tiwari, M.Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In *ISCA*, pages 83-94, 2000

[23] P.J.Joseph, K.Vaswani, M.J.Thazhuthaveetil. A Predictive Performance Model for Superscalar Processors. In *MICRO*, pages 161-170, 2006

[24] G.Magklis, P.Chaparro, J.Gonzalez, A.Gonzalez. Independent Front-end and Back-end Dynamic Voltage Scaling for a GALS Microarchitecture. In *ISLPED*, pages 49-54, 2006

[25] Q.Wu, M.Martonosi, D.W.Clark, V.J.Reddi, D.Connors, Y.Wu, J.Lee, D.Brooks. Dynamic Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE MICRO*, 26(1):119-129, 2006

[26] C.H.Hsu, U.Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *PLDI*, pages 38-48, 2003.