Executable Analysis using Abstract Interpretation with Circular Linear Progressions

Rathijit Sen Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore
E-mail: {rathi,srikant}@csa.iisc.ernet.in

Abstract

We propose a new abstract domain for static analysis of executable code. Concrete states are abstracted using Circular Linear Progressions (CLPs). CLPs model computations using a finite word length as is seen in any real life processor. The finite abstraction allows handling overflow scenarios in a natural and straight-forward manner. Abstract transfer functions have been defined for a wide range of operations which makes this domain easily applicable for analyzing code for a wide range of ISAs. CLPs combine the scalability of interval domains with the discreteness of linear congruence domains. We also present a novel, lightweight method to track linear equality relations between static objects that is used by the analysis to improve precision. The analysis is efficient, the total space and time overhead being quadratic in the number of static objects being tracked.

1 Introduction

A wide selection of problems require statically analyzing programs for deducing or verifying properties that hold at the time of execution. These include detection of memory aliases with direct impact on compiler optimizations, timing verification for real-time systems, detecting possibilities for stack overflow, checking program assertions for correctness and many more. Often, source code may not be available. Analysis of arbitrary executable code in the absence of source level information has important uses such as detecting malicious content and vulnerabilities, performing code comparisons, and others. In all of these, we are interested in knowing about properties that hold over all possible program inputs and execution sequences.

A landmark paper by Cousot and Cousot [4] introduced the concept of Abstract Interpretation. It provides a formal framework for dealing with abstract representations of program states and is a well established technique for static analysis of programs. The idea is to define an abstract domain and operations on elements of that domain consistent with concrete execution semantics. At any program point, the set of abstract values holding at that point is an over-approximation of the possible set of concrete values that can hold at that point over all possible execution sequences.

In order to analyze arbitrary code, abstract elements must be composable for a wide range of operations. Computations that involve a finite word length introduce additional challenges in guaranteeing safety for abstract transfer functions. For example, the result of adding two positive values may not be positive. It is also desired that the analysis is not computationally intensive.

We introduce a new abstract domain particularly suited for statically analyzing arbitrary executable code, with abstract elements represented by Circular Linear Progressions (CLPs). CLPs model computations in the concrete domain that use a finite word length. CLPs ensure safety even if computations incur overflow. CLPs track discrete sets of values and are easily composable for a wide range of operations. Space overhead for our analysis is $O(N^2)$ per basic block, where N is the total number of static objects being tracked. Each static object represents a register or a statically identifiable memory partition. The abstract function results are refined for better precision using linear equality relations between static objects. Such relations are derived using an $O(N^2)$ algorithm, resulting in a computational complexity of $O(N^2)$ for analyzing each instruction and join point.

As an example, consider the source representation of a computation in Figure 1. The computation for y results in an overflow for $x = 3$ but not for $x = 7$. The CLPs for y and z as computed by our analysis are shown alongside as 3-tuples (lower bound, upper bound, step) as described in §3. They indicate that y is either -4 or 0 and z is either -1 or 3 during actual execution. For this example, the analyzer has been able to compute a safe and tight approximation of the actual runtime values.

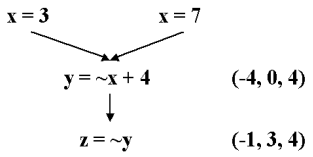

Figure 1. Sample computation and CLP

The rest of this document is organized as follows: Existing state of the art and our contributions are presented in §2. The CLP abstract domain is introduced in §3. Abstract transfer functions for various arithmetic, logical and set operations are described in §4. The handling does not take into account circularity; this limitation is overcome in §5. Convergence to a fix-point is discussed in §6. A quadratic time procedure is formulated in §7, which helps to track linear and partially linear relations between static objects. A best-of-two selection strategy is introduced in §8. Our experimental setup is described in §9. We conclude with a discussion on the strengths and limitations of this model, some application scenarios, and possibilities for future extension in §10.

2 Related Work

A significant amount of research has been made in developing numerical abstract domains, varying in expressiveness and computational complexity, for statically analyzing program properties. One of the simplest domains is the interval domain. The interval abstraction though simple is largely imprecise as it results in a lot of over-approximation. One of the sources of imprecision is the loss of information regarding relations between objects being tracked. Inspection of two abstract sets representing the set of concrete values which the corresponding objects may take at that point may not reveal such relations. Karr [8] developed linear algebraic techniques to automatically infer affine relations between variables at any program point. The time complexity is $O(N^4)$ and has been subsequently improved to $O(N^3)$ in [9]. The work has been extended for precise interprocedural analysis in [10] with a time complexity of $O(N^8)$. Halbwachs et al. [7] introduce detection of linear inequalities using convex polyhedra. Several domains have been proposed that reduce computational overheads by tracking particular classes of inequalities. A comparison of various domains that are used to track linear inequalities between variables is available in [3].

A second source of imprecision arises from not exploiting discrete structural properties of the concrete value sets. Value Set Analysis [2] uses the concept of Reduced Interval Congruences(RICs) to model discrete sets of values. Each RIC can be viewed as the intersection of a congruence domain and an interval domain. The congruence domain serves to restrict the possible values of the interval element. Each RIC is represented as a 4-tuple (a, b, c, d) representing the set $a[b, c] + d = \{a * (b + i) + d | 0 \leq i \leq c - b, i \in \mathbb{Z}\}$. However, it is not clear from [2] how RICs may be composed efficiently, as for instance, how to get a third set if one set is shifted by another. While composing an RIC with a constant is straight-forward for most operations, it may not be trivial to extend this to three-operand arithmetic, which is very common in modern ISAs. Further, the RIC representation is not unique; multiple combinations of (a, b, c, d) can represent the same set of elements.

Conventional methods for pointer analysis may not readily work for executable code. Alias analysis for executable code has been explored in [5] by modeling each address descriptor as a tuple consisting of an instruction context and the lower k-bits of the set of addresses for some fixed k. Value Set Analysis [2] improves on this by tracking statically identifiable memory partitions in addition to registers.

Our work extends RICs in three directions – (a) *Finiteness*: CLPs abstract finite word length computations. Apart from ease in ensuring safety for overflow situations, the finite abstraction also offers opportunities for efficient set union that we exploit; (b) *Composability*: we show how multiple CLPs can be efficiently composed for a large number of operations, thereby rendering our techniques applicable to a wide range of ISAs; (c) *Uniqueness*: every concrete set has exactly one abstract representation. We also propose a light-weight $O(N^2)$ algorithm to improve precision of the analysis by utilizing linear equality relations between static objects. CLPs combine the power of the interval domain in that they can be easily extended to abstract complex computations and of congruence domains in that they track and maintain inherent discreteness in the abstracted sets.

3 Circular Linear Progressions

We model discrete value sets as circular linear progressions of values. CLPs fit a first degree polynomial to the possible set of concrete values. This model is an exact fit for induction variables and linear computations. Each CLP is represented as a 3-tuple (l, u, δ), where $l, u \in \mathbb{Z}(n), \delta \in \mathbb{N}(n) \cup \{0\}$, and the parameter n denotes n-bit representation. The components denote the starting point, ending point and positive step increment respectively. Each CLP requires $3n$ bits for representation. Let $MAX\_P = 2^{(n-1)} - 1$, $MAX\_N = -2^{(n-1)}$, $MAX\_D = 2^n - 1$. Since we are considering finite representation using n bits, the set of all CLPs is finite.

The finite set of values abstracted by the CLP $C(l, u, \delta)$ is computed by the concretization function

$$conc(C) = \{a_i = l +_n i\delta | 0 \leq i \leq s, i \in \mathbb{Z} \text{ and } s \text{ is the } \textit{smallest} \text{ non-negative integer such that } a_s = u\}$$

$+_n$ denotes addition in n-bits, which is addition modulo 2^n. We can visualize this computation by considering a

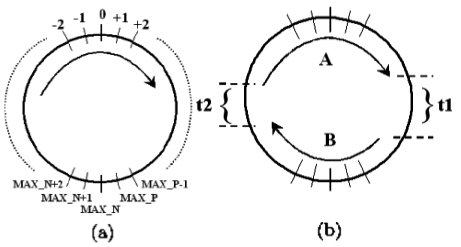

Figure 2. (a)CLP visualization (b)Set union

circular disc as in Figure 2(a) marked in sequence with $0...MAX\_P,\ MAX\_N, ...0$. Mark the points l and u on this disc. Then proceed *from* l along the periphery in a *clockwise* direction, reading off values in increments of δ till the point u is reached. The set of values chosen is precisely the set of values abstracted by this CLP. The top element, $\top$, is characterized by the constraints: $\top.l = \top.u+1, \top.\delta = 1$.

This modeling implicitly captures the effect of wraparound in computations. A second advantage is in the ability to build tighter abstract sets by taking advantage of circularity. For example, consider the concrete set $\{1, 2^n - 1\}$. A non-circular unsigned interval abstraction would result in the abstract representation of $[1, 2^n - 1]$ including $2^n - 1$ values. The over-approximation is huge. This problem may be remedied by considering signed abstractions that result in the tight set $[-1, 1]$. However, the problem now appears for sets like $\{-2^{(n-1)}, 2^{(n-1)} - 1\}$ for which abstractions using unsigned representation are more efficient. Circularity allows us to efficiently represent both cases with tight abstractions. For the former example, it is the CLP$(MAX_D, 1, 2)$ and for the latter it is $(MAX\_P, MAX\_N, 1)$. In both cases, exactly 2 elements are included.

4 Transfer Functions

The following abstract transfer functions define the composition of a CLP with another CLP or an integer, k. A proof sketch is provided for Shift. More proofs and examples can be found in [12]. Let $A = (l_1, u_1, \delta_1)$ and $B = (l_2, u_2, \delta_2)$ denote respectively, the first and second CLP being composed. For the moment, assume that for all operations other than $\cup$, $MAX\_N \leq l_1, u_1, l_2, u_2 \leq MAX\_P$, $l_1 \leq u_1$, $l_2 \leq u_2$ and no operation results in overflow. We shall remove these restrictions in §5.

The function $size(x)$ returns the number of elements in CLP x and is defined as follows under the current restriction of no wraparound.

$$size(x) = \begin{cases} 0 & \text{if } x \text{ is empty} \\ MAX\_D + 1 & \text{if } x \text{ is } \top \\ \frac{(x.u - x.l)}{x.d} + 1 & \text{otherwise} \end{cases}$$

4.1 Set Operations

UNION

- $k_1 \cup k_2 = (a, b, diff)$
- $(l_1, u_1, \delta_1) \cup k \subseteq (a, b, gcd(diff, \delta_1))$
- $(l_1, u_1, \delta_1) \cup (l_2, u_2, \delta_2) \subseteq (a, b, gcd(diff, \delta_1, \delta_2))$

Let δ denote the step of the result CLP as computed above. $diff$ is chosen by considering two alternatives as shown in Figure 2(b). Out of $t1$ and $t2$, it is the one that results in a *smaller* value for $(\frac{diff}{\delta})$. This choice results in minimum over-approximation. a is either l_1 or l_2 and is set according to the choice taken in the above step. Similarly, b is one of the upper bounds. The result set always has $1 \leq \delta \leq MAX\_P$. This follows since for the union of two constants,

$$t1 + t2 + 2 = MAX\_D + 1$$
$$\Rightarrow\ \delta = min(t1, t2) \leq \frac{t1 + t2}{2} \leq \frac{MAX\_D - 1}{2} \leq MAX\_P$$

In case the two input sets overlap, there is only one choice.

INTERSECTION

The definitions follow from the equations $l_1 + i * \delta_1 = k$ and $l_1 + i * \delta_1 = l_2 + j * \delta_2$.

- $(l_1, u_1, \delta_1) \cap k = k$ iff $l_1 \leq k \leq u_1$ and $\frac{k - l_1}{\delta_1}$ is an integer. Otherwise, the intersection is empty.
- $(l_1, u_1, \delta_1) \cap (l_2, u_2, \delta_2) = (\theta, \phi, \delta)$, where
  - $\delta = lcm(\delta_1, \delta_2)$
  - $\theta = l_2 + j' * \delta_2$, where j' is the smallest value such that $j = \frac{(l_2 - l_1) + j' * \delta_2}{\delta_1}$ is an integer and $j \geq 0$, $j' \geq 0$.
  - $\phi = \theta + \delta * n$, where $n = [\frac{min(u_1, u_2) - \theta}{\delta}]$

  provided that $0 \leq j' < \delta_1$ and $\theta \leq min(u_1, u_2)$. Otherwise, the intersection is empty. It is assumed that $l_1 \leq l_2$. If this does not hold, the sets are to be renamed appropriately. This is possible to do as set intersection is commutative.

DIFFERENCE

- Let $C(l_c, u_c, \delta_c) = A \cap B$. If $C = \phi$ or A, then $A \setminus B = A$ or ϕ respectively. If $size(A) - size(C) = 1$, then $A \setminus B$ is a constant and it is the unique element not present in C. This element is either l_1, u_1, or $l_1 + \delta_1$.
- Otherwise, the step is the same as that of A. The bounds may change. $A \setminus B \subseteq (l, u, \delta_1)$, where the parameters are given by the following conditions:
  - If $l_c \neq l_1$, then $l = l_1$
  - If $l_c = l_1, \delta_c \neq \delta_1$ then $l = l_1 + \delta_1$
  - If $l_c = l_1, \delta_c = \delta_1$ then $l = u_c + \delta_1$
  - If $u_c \neq u_1$, then $u = u_1$
  - If $u_c = u_1, \delta_c \neq \delta_1$ then $u = u_1 - \delta_1$
  - If $u_c = u_1, \delta_c = \delta_1$ then $u = l_c - \delta_1$

Table 1. end points for division

$s(l_1)$	$s(u_1)$	$s(l_2)$	$s(u_2)$	a	b
$-$	$-$	$-$	$-$	$\frac{u_1}{l_2}$	$\frac{l_1}{u_2}$
$-$	$-$	$-$	$+$	$\frac{l_1}{\alpha}$	$\frac{l_1}{\beta}$
$-$	$-$	$+$	$+$	$\frac{l_1}{l_2}$	$\frac{u_1}{u_2}$
$-$	$+$	$-$	$-$	$\frac{u_1}{u_2}$	$\frac{l_1}{u_2}$
$-$	$+$	$-$	$+$	$min(\frac{u_1}{\beta}, \frac{l_1}{\alpha})$	$max(\frac{l_1}{\beta}, \frac{u_1}{\alpha})$
$+$	$+$	$-$	$-$	$\frac{u_1}{u_2}$	$\frac{l_1}{l_2}$
$+$	$+$	$-$	$+$	$\frac{u_1}{\beta}$	$\frac{u_1}{\alpha}$
$-$	$+$	$+$	$+$	$\frac{l_1}{l_2}$	$\frac{u_1}{l_2}$
$+$	$+$	$+$	$+$	$\frac{l_1}{u_2}$	$\frac{u_1}{l_2}$

4.2 Arithmetic Operations

ADDITION

- $(l_1, u_1, \delta_1) + k = (l_1 + k, u_1 + k, \delta_1)$
- $(l_1, u_1, \delta_1) + (l_2, u_2, \delta_2) \subseteq (l_1 + l_2, u_1 + u_2, gcd(\delta_1, \delta_2))$

SUBTRACTION

- $(l_1, u_1, \delta_1) - k = (l_1 - k, u_1 - k, \delta_1)$
- $(l_1, u_1, \delta_1) - (l_2, u_2, \delta_2) \subseteq (l_1 - u_2, u_1 - l_2, gcd(\delta_1, \delta_2))$

MULTIPLICATION

- $(l_1, u_1, \delta_1) * k = (min(l_1 * k, u_1 * k), max(l_1 * k, u_1 * k), |\delta_1 * k|)$
- $(l_1, u_1, \delta_1) * (l_2, u_2, \delta_2) \subseteq (min(l_1 * l_2, u_1 * u_2, l_1 * u_2, u_1 * l_2), max(l_1 * l_2, u_1 * u_2, l_1 * u_2, u_1 * l_2), gcd(|l_1 * \delta_2|, |l_2 * \delta_1|, \delta_1 * \delta_2))$

In the above, min and max take care of negative values in the operands.

DIVISION (signed)

- $(l_1, u_1, \delta_1)/k = (min(l_1/k, u_1/k), max(l_1/k, u_1/k), |\delta_1/k|)$
- $(l_1, u_1, \delta_1)/(l_2, u_2, \delta_2) \subseteq (a, b, 1)$ where a, b depend on the signs of l_1, u_1, l_2, u_2 and are defined in Table 1.

Here, $s(x)$ denotes the sign of x. If any of the bounds of the first series is 0, its sign is considered to be the same as that of the other bound. α and β denote respectively the smallest positive and largest negative numbers of the second series. The second series is assumed not to include 0.

4.3 Shift Operations

LEFT SHIFT

- $(l_1, u_1, \delta_1) << k = (l_1 << k, u_1 << k, \delta_1 << k)$. It is assumed that $k \geq 0$.
- $(l_1, u_1, \delta_1) << (l_2, u_2, \delta_2) \subseteq (l_1 << l_2, u_1 << u_2, gcd(|l_1|, \delta_1) << l_2)$. The second series is assumed to contain only non-negative values.

Table 2. end points for shift

$s(l_1)$	$s(u_1)$	a	b
$-$	$-$	$l_1 >> l_2$	$u_1 >> u_2$
$-$	$+$	$l_1 >> l_2$	$u_1 >> l_2$
$+$	$+$	$l_1 >> u_2$	$u_1 >> l_2$

Proof. Shifting of an arbitrary value of the first set by an arbitrary value of the second set produces the result $(l_1 + i * \delta_1) * 2^{l_2 + j*\delta_2} = 2^{l_2} * (l_1 + i * \delta_1) * 2^{j*\delta_2}$. The difference from the lower bound is $2^{l_2} * (l_1 + i * \delta_1) * 2^{j*\delta_2} - 2^{l_2} * l_1 = 2^{l_2} * l_1 * (2^{j*\delta_2} - 1) + i * 2^{l_2} * 2^{j*\delta_2} * \delta_1$. The first term in the sum is a multiple of $2^{l_2} * l_1$ and the second term is a multiple of $2^{l_2} * \delta_1$. The gcd of these two quantities=$2^{l_2} * gcd(l_1, \delta_1) = gcd(l_1, \delta_1) << l_2$ and is a safe approximation for the step. Further, as we are considering the gcd and the other quantities are unrelated, the approximation is tight. □

RIGHT SHIFT

- $(l_1, u_1, \delta_1) >> k = (l_1 >> k, u_1 >> k, \delta_1 >> k)$. It is assumed that $k \geq 0$.
- $(l_1, u_1, \delta_1) >> (l_2, u_2, \delta_2) \subseteq (a, b, 1)$ where a, b depend on the signs of l_1, u_1 and are defined in Table 2. The second series is assumed to contain only non-negative values.

Right shift can be considered as a special case of division by a power of 2 and the entries of Table 2 are derived from the relevant entries of Table 1.

4.4 Bit Operations

BITWISE COMPLEMENT

- $\sim (l_1, u_1, \delta_1) = (\sim u_1, \sim l_1, \delta_1)$

BITWISE AND

The bitwise AND operation is not boundary-preserving in general. For example, $\{3, 7, 11\}\&5 = \{1, 5\}$ but $11\&5 = 1$, $3\&5 = 1$. Thus, the bounds of the resulting CLP may not be obtained, in all cases, by a straightforward ANDing of the bounds of the operands.

- $(l_1, u_1, \delta_1)\&k \subseteq (\beta, \gamma, \delta)$ where the parameters are given by the following algorithm:
  - $\delta = 2^{max(\alpha_1, \alpha_2)}$ where α_1, α_2 denote respectively the number of trailing 0-bits of δ_1 and k.
  - Let θ_1 and θ_2 denote respectively the positions of the leading non-zero bit in l_1 and u_1.
  - If all bits in k from bit α_2 to bit θ_2 are 1, then $\beta = (l_1\&k)$ and $\gamma = (u_1\&k)$. Otherwise,
  - safe upper bound $(s.u.b.) = min(u_1, k)$.
  - If bits θ_3 to θ_2 in k are all 1, $\theta_3 \leq \theta_1$, then safe lower bound $(s.l.b.) = (a'\&k)$ where a' has the same bits as l_1 for bit ranges $[\theta_3, \theta_1]$ and $[0, \alpha_1 - 1]$ and the rest of the bits are zero. Otherwise,

– $(s.l.b.) = (a'\&k)$ where a' has the same bits as l_1 for bit range $[0, \alpha_1 - 1]$ and the rest of the bits are zero.
– $\beta = (l_1\&k) + \delta * [\frac{(s.l.b. - l_1\&k)}{\delta}]$, $\gamma = (l_1\&k) + \delta * [\frac{(s.u.b. - l_1\&k)}{\delta}]$

The algorithm first finds out the number of lower bits of the result that are fixed. This step determines δ. Next, runs of 0's and 1's are considered in computing bounds. For the second case, the result cannot be less than slb or greater than sub. The final computation of β and γ fine tunes slb and sub so that they can be reached in steps of δ from values known to be present in the result.

- The algorithm for computing $(l_1, u_1, \delta_1)\&(l_2, u_2, \delta_2)$ can be found in [12].

Definitions for BITWISE OR and BITWISE XOR can be found in [12].

4.5 Comparison Operations

The comparison A relop B is analyzed by computing $C = A - B$ and determining the relation of 0 with C. The result is 1 or 0 iff the relation can be definitely evaluated to true or false, respectively. Otherwise, it evaluates to $(0, 1, 1)$.

- A is definitely equal to B iff $C = 0$.
- A is definitely not equal to B iff $C \cap \{0\} = \phi$.
- A is definitely greater than B iff $C.l > 0$.
- A is definitely not greater than B iff $C.u < 0$.
- Definitely less than and definitely Not less than are defined similarly.

4.6 Operand Repetitions

Before applying the transfer functions for computing the abstract result for any concrete instruction I, we need to check whether a source operand is *definitely* repeated in I. Such repetitions, if present, present opportunities for computing tighter abstract results than that obtained by a straightforward application of the function. For example, consider $I : r1 = r2 * r2$. Assume that the current value for $r2$ is abstracted by $(2, 3, 1)$. Straightforward application of the abstract multiplication function results in $(4, 9, 1)$. This is safe, but a tighter representation is $(4, 9, 5)$. The problem appears because we are allowing for the possibility of combination 2×3 to exist, which is spurious. Similar examples can be constructed for other operations. If repetition is recognized, spurious cross-combinations can be eliminated, resulting in tighter approximations. An instruction may have syntactically different operands, yet there may be implicit repetitions. For example, in the sequence $m1 = r2, r1 = m1 * r2$, $r2$ is implicitly repeated. This can be handled by tracking relations among static objects.

5 Circularity and Overflow

In §4 we had imposed the restrictions $MAX\_N \leq l_1, u_1, l_2, u_2 \leq MAX\_P$, $l_1 \leq u_1$, $l_2 \leq u_2$. We relax these restrictions now, and construct two disjoint sets P, Q given any general CLP C(l, u, δ) as follows:

- If$(l \leq u)$, then $P = C, Q = \phi$
- Otherwise, $P = (l, l + \delta \left[\frac{MAX\_P - l}{\delta}\right], \delta)$, $Q = (u - \delta \left[\frac{u - MAX\_N}{\delta}\right], u, \delta)$.

Each of P and Q individually satisfy the restrictions of §4. Further, $P \cap Q = \phi$ and $P \cup Q = C$. All the operations that we have considered other than set difference distribute over union of mutually disjoint sets. Thus, for two CLPs A and B, any operation $\otimes$ other than $\cup$ and $\setminus$ is defined as

$$\begin{aligned} A \otimes B &= (P_A \cup Q_A) \otimes (P_B \cup Q_B) \\ &= (P_A \otimes P_B) \cup (P_A \otimes Q_B) \cup (Q_A \otimes P_B) \cup \\ &\quad (Q_A \otimes Q_B) \end{aligned}$$

All operations on individual components remain as defined in §4. For $\setminus$, no change is required other than $\cap$ taking circularity into account.

Overflows are handled by computing the result $C(l, u, \delta)$ of arithmetic and left shift operations in $2n$ bits and converting it back to an n-bit representation. If l and u are both representable in n-bits, then no processing is required. Otherwise, if $l > u$, the result is approximated to $\top$. For the other cases, C is first decomposed into two sets P and Q as follows:

- If l is representable in n-bits, but u requires more than n-bits, then it is necessarily the case that $u \geq 0$. This follows from the fact that $l \leq u$. $P = (l, l + \delta \left[\frac{MAX\_P - l}{\delta}\right], \delta)$. Let $first = P.u + \delta$.
  – if $(u - first)$ can be represented in n bits, $Q = (MAX\_N + first - MAX\_P - 1, MAX\_N + u - MAX\_P - 1, \delta)$.
  – Otherwise, $Q = (t, t + \delta' \left[\frac{MAX\_D - t}{\delta'}\right], \delta')$, where $t = first\&MAX\_D\&(\delta' - 1)$, $\delta' = 2^\alpha$, $\alpha =$ the number of trailing zero bits of δ. The over-approximation for Q is required as the bitwise AND operation is not boundary-preserving.
- The case where u is representable in n-bits, but l requires more than n-bits, is handled similarly.
- If both l and u require more than n-bits, then $P = \phi$, $Q = (t, t + \delta' \left(\frac{MAX\_D - t}{\delta'}\right), \delta')$, where $t = l\&MAX\_D\&(\delta' - 1)$.

In the above, P is basically the part that can be represented in n bits whereas Q is an approximation of the lower n bits of the rest of the result. The final n-bit result is computed as $P \cup Q$.

Table 3. Filter Specification: path indicates whether b is on the true path after exiting p.

relop	path	$K(p,b,y)$	
		$y.l \leq y.u$	$y.l > y.u$
$<$ $\geq$	true false	$(MAX\_N, y.l-1, 1)$	$\top$
$<$ $\geq$	false true	$(y.u, MAX\_P, 1)$	$\top$
$>$ $\leq$	true false	$(y.u+1, MAX\_P, 1)$	$\top$
$>$ $\leq$	false true	$(MAX\_N, y.l, 1)$	$\top$
$=$ $\neq$	true false	y	y
$=$ $\neq$	false true	$\top \setminus y$	$\top \setminus y$

6 Convergence

Convergence to a fix-point requires several important considerations – (a) how information flowing along different paths is collected at program join points (b) how branch conditions can be utilized in limiting precision loss and (c) how a fix-point can be quickly reached.

At program join points, CLPs are composed through the operation of Set Union. The CLP value of static object s_i ($s_i.val$) for the IN state of basic block b is computed as

$$b(IN).s_i.val = \bigcup_{p \in pred(b)} p(OUT).s_i.val$$

However, the OUT CLP for s_i of predecessor p may be constrained by a branch condition at the exit point of p. Exploiting these constraints will often lead to tighter approximations for s_i. The effect of the constraint can be viewed as the application of a *filter* operation which takes into account the predecessor branch condition, the current value for s_i and whether b lies on the true or false path as a result of executing the branch. Assume that the evaluation of the branch condition is essentially a CLP comparison operation of the form s_i relop y. We implement the filtering operation by intersecting $s_i.val$ with a filter CLP computed by the function $F : B \times B \times S \rightarrow L$, where B, S, L denote respectively the set of basic blocks, static objects and CLPs.

$$F(p,b,s_i) = \begin{cases} \top & \text{if exit from } p \text{ is unconditional or if } s_i \text{ is not the left operand of the branch condition,} \\ K(p,b,y) & \text{otherwise.} \end{cases}$$

The function K is defined in Table 3. The join computation can now be re-written as

$$b(IN).s_i.val = \bigcup_{p \in pred(b)} (p(OUT).s_i.val \cap F(p,b,s_i))$$

As an example, suppose the branch condition at the end of p is $r2 < r3$ and the current values for $r2$ and $r3$ are described by the CLPs $(2, 126, 4)$ and $(7, 21, 2)$ respectively. Then, the OUT value for $r2$ is computed as $(2, 126, 4) \cap (MAX\_N, 6, 1) = (2, 6, 4)$. Without filtering, the OUT value would be $(2, 126, 4)$.

```
l1:  mov  r2, #2          l1:  add  r5, r3,r6
     b    l3                    mov  r1, r2,lsl r7
l2:  mov  r2,#3                 b    l4
l3:  add  r4, r1,r2       l2:  add  r1, r3,r6
     mov  r5, r4                mov  r5, r2,lsl r7
     sub  r5, r5,r2       l4:
l4:
        (a)                        (b)
```

Figure 3. At $l4$, is always r5=r1? (a)Yes (b)No

Although the height of the CLP lattice is finite, we need widening to quickly converge to a fix-point. For a candidate CLP, widening is considered separately for its l, u and δ components. Further details are available in [12].

7 Linear Dependence

The transfer functions of §4, by themselves, are not adequate in capturing fine-grained relations between static objects. Consider for example, the code snippets in Figure 3. Assume that control cannot reach $l4$ before reaching either $l1$ or $l2$. Suppose that we now want to know if *always* $r5 = r1$ at $l4$. Such questions are important in improving the precision of the computed results and also in propagating branch filters, with implications in widening. The answer should be in the affirmative for example (a) but not necessarily for (b).

For example (a), assume that the initial value of $r1$ is 5. At $l3$, the CLP for $r4$ is $5 + (2, 3, 1) = (7, 8, 1)$. Finally, the CLP for $r5$ is $(7, 8, 1) - (2, 3, 1) = (4, 6, 1)$. However, a comparison between the corresponding CLPs, $(4, 6, 1)$ and 5 results in $(0, 1, 1)$ indicating that it cannot be definitely said if these two objects will hold the same value at execution time. This loss of precision in (a) is due to the fact that we are not tracking relations between static objects as they are being modified.

To address this shortcoming, we track linear equality relations between static objects *in addition* to value computations. At each point of the program, we track combinations of CLPs of the form

$$\lambda_i x_i = \sum_{j \neq i} \lambda_j x_j + c_i$$

where x_i denotes the CLP representing the value for static object s_i at that point. Figures 4(a) and (b) show a sample code and relations that hold at the end point as computed by our tool. Internal representation is discussed in §9.

These equations hold *simultaneously*. They track the relations that hold between the objects at that point. Thus, while analyzing a set of equations, it is not required to know about execution order. The ordering is implicitly captured in the equations themselves. The simultaneity is ensured by modifying the equations after processing each instruction so that we get a new set of equations reflecting the updated value as a result of instruction execution. The transformations are detailed in §7.1.

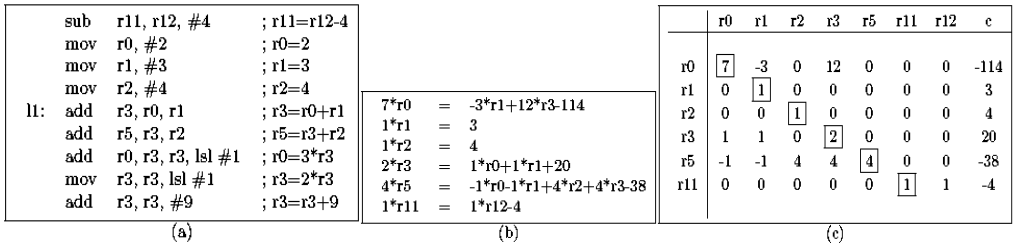

Figure 4. (a) sample ARM7 code (b) relations generated after analysis (c) internal representation

7.1 Update

Let, x_j denote the CLP for static object s_j before execution of instruction I and x'_j denote the CLP after execution of I. I is of the form

$$x_k = \rho x_k + \sum_{j \neq k} \sigma_j x_j + c$$

Suppose that the current value of static object s_i depends on the current value of static object s_k. Let the old values of static objects s_i and s_k be denoted by the equations $\theta x_i = \sum_{j \neq i} \lambda_j x_j + c_i$ and $\phi x_k = \sum_{j \neq k} \delta_j x_j + c_k$ with $\lambda_k \neq 0$. The new value of s_k is given by

$$\begin{aligned} x'_k &= (\rho/\phi) \sum_{j \neq k} \delta_j x_j + (\rho/\phi) c_k + \sum_{j \neq k} \sigma_j x_j + c \\ &= \sum_{j \neq k} ((\rho/\phi)\delta_j + \sigma_j) x_j + (\rho/\phi) c_k + c \end{aligned}$$

Equivalently, $\phi x'_k = \sum_{j \neq k} \gamma_j x_j + c'$, where $\gamma_j = \rho\delta_j + \phi\sigma_j \ \forall j \neq k$, $\gamma_k = \delta_k = 0$, $c' = \rho c_k + \phi c$. As x_i has not changed, the equation for x_i needs to be updated to compensate for the change in x_k. Let x'_i denote the incorrect value for x_i that results if the equation is not updated. Then

$$\begin{aligned} \theta\phi x_i &= \phi \sum_{j \neq i, j \neq k} \lambda_j x_j + \lambda_k \sum_{j \neq k} \delta_j x_j + (\lambda_k c_k + \phi c_i) \\ \theta\phi x'_i &= \phi \sum_{j \neq i, j \neq k} \lambda_j x_j + \lambda_k \sum_{j \neq k} \gamma_j x_j + (\lambda_k c' + \phi c_i) \\ &= \theta\phi x_i + \lambda_k \left(\sum_{j \neq k} (\gamma_j - \delta_j) x_j + (c' - c_k) \right) \\ \theta\phi x_i &= \theta\phi x'_i - \lambda_k \left(\sum_{j \neq k} (\gamma_j - \delta_j) x_j + (c' - c_k) \right) \\ &= \sum_j (\phi\lambda_j - \lambda_k(\gamma_j - \delta_j)) x_j + \phi c_i - \lambda_k (c' - c_k) \end{aligned}$$

Thus, the transformations for s_i are:

θ	$\rightarrow$	$\theta\phi$
λ_j	$\rightarrow$	$\phi\lambda_j - \lambda_k(\gamma_j - \delta_j)$
c_i	$\rightarrow$	$\phi c_i - \lambda_k(c' - c_k)$

The transformations for s_k are:

δ_j	$\rightarrow$	γ_j,	c_k	$\rightarrow$	c'

Relations for other objects are not modified.

The modified equations may not be in the form $\theta' x_i = \sum_{j \neq i} \lambda'_j x_j + c'_i$ since λ'_i may not be zero after update. θ',λ',c'_i are as defined by the above transformations. The previous step results in a value of $-\lambda_k(\gamma_i - \delta_i)$ for λ'_i. This is corrected by adjusting the coefficient of x_i in the LHS so that the equation is now in the form

$$(\theta\phi + \lambda_k(\gamma_i - \delta_i)) x_i = \sum_{j \neq i} \lambda'_j x_j + c'_i$$

Further, if θ',c'_i and all coefficients evaluate to constants, they are divided by the common gcd. The time complexity per update and subsequent normalization is $O(N^2)$.

To illustrate the computations with an example, we again consider the code snippet in Figure 4(a) and the point when the instruction $r0 = 3r3$ is being analyzed. The relations holding at this point before analyzing this instruction are: $r0 = 2$ and $r3 = r0 + r1$. For this instruction, we have $\rho = 0, \sigma = [0,0,0,3,0], c = 0$. Also, $\delta = [0,0,0,0,0], c_0 = 2, \phi = 1$. To consider the impact on $r3$ as a result of this update, we have $\lambda = [1,1,0,0,0], c_3 = 0, \theta = 1, \lambda_0 = 1$. Thus, $\gamma = [0,0,0,3,0], c' = 0$ and the transformations for $r3$ are:

$$\theta \rightarrow \theta\phi = 1,\ c_3 \rightarrow 0 - 1(0 - 2) = 2,$$
$$\lambda \rightarrow [1,1,0,0,0] - 1([0,0,0,3,0] - [0,0,0,0,0]) = [1,1,0,-3,0].$$

This implies $r3 = r0 + r1 - 3r3 + 2$. Normalizing, we get $4r3 = r0 + r1 + 2$. Also, $r0 = 3r3$ holds. Figure 4(b) shows the relations after processing all the instructions in the code given in Figure 4(a).

An instruction I may contain both linear and non-linear components, as for example, $r1 = r2 + r3 << r7$. Such instructions are handled by evaluating the non-linear part using the current values and substituting the result. For this example, assume that the current abstract values for $r3$ and $r7$ are $(5, 20, 5)$ and $(2, 3, 1)$ respectively. After evaluation and substitution, I takes the form $r1 = r2 + (20, 160, 20)$. Such transformations help in tracking and utilizing partial linear relations between static objects.

7.2 Join

At program join points, the relation for each s_i in the IN state of a basic block is computed by performing CLP set

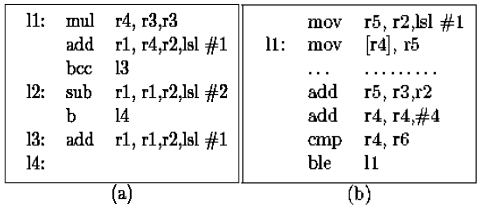

Figure 5. Selection: (a)Coeff-best (b)Val-best

union for each coefficient of the corresponding relations in the OUT states of predecessor blocks.

$$b(IN).s_i.coeff[j] = \bigcup_{p \in pred(b)} p(OUT).s_i.coeff[j], 0 \leq j \leq N$$

In the above, $s_i.coeff[j]$ denotes the j^{th} coefficient of the linear relation for s_i.

Additional spurious relations may be generated as a result of the join. Consider the example in Figure 5(b). At $l1$, the relation for $r5$ is the union of the two relations $r5 = 2r2$ and $r5 = r2 + r3$, resulting in $r5 = (1, 2, 1)r2 + (0, 1, 1)r3$. However, this representation gives rise to two additional spurious relations: $r5 = r2$ and $r5 = 2r2 + r3$. In general, we can view the coeff matrix as representing two sets of simultaneous equations – (a) the set of valid relations, V and (b) the set of spurious relations, Q. Q may include inconsistent equation sets. The abstract value of any static object at a point must satisfy equations in V only.

Widening is performed by widening each coefficient CLP that has not yet stabilized. If any coefficient is set to $\top$, the entire relation is collapsed into the form $x_i = \top$.

8 Selection

The preceding discussions have described two separate abstractions for the same concrete state – one computes $s_i.val$ using the CLP transfer functions, the other expresses constraints in the form of linear equalities using $s_i.coeff$. The first one over-approximates the concrete state due to loss of information regarding relations between static objects and fitting a linear model to non-linear sets. The second abstraction over-approximates due to spurious relations being generated at join points and in the presence of non-linearity. They present two alternate but safe abstractions of the same concrete state. However, neither is clearly better than the other in all cases in producing a tighter abstraction. We follow a best-of-two selection strategy between the result computed by the CLP transfer function and that implied by the relation matrix to obtain a tighter approximation. The following example clarifies this point.

First, consider Figure 5(a). Assume that the abstract values for $r2$ and $r3$ before control reaches $l1$ are $(-10, 5, 15)$ and $(2, 3, 1)$ respectively. First, let us compute for the val abstraction. $r4$ evaluates to $(4, 9, 5)$. The first add results in $r1 = 2r2 + r4 = 2(-10, 5, 15) + (4, 9, 5) = (-16, 19, 5)$. val for $r1$ evaluates to $(-16, 19, 5) - 4(-10, 5, 15) = (-36, 59, 5)$ after the sub in $l2$ and $(-16, 19, 5) + 2(-10, 5, 15) = (-36, 29, 5)$ after the add in $l3$. At the join point $l4$, $r1 = (-36, 59, 5) \cup (-36, 29, 5) = (-36, 59, 5)$. On the other hand, the $coeff$ entry for $r1$ indicates $r1 = (-2, 4, 6)r2 + r4$. Substituting the current abstract values for $r2$ and $r4$ we get $r1 = (-2, 4, 6)(-10, 5, 15) + (4, 9, 5) = (-40, 20, 30) + (4, 9, 5) = (-36, 29, 5)$. The $coeff$ abstraction is better than val in this case.

In Figure 5(b), assume that the abstract value of $r2$ before control reaches this code is $(6, 8, 2)$ and that $r3$ assumes the value $(1, 9, 1)$ in the loop. We wish to find out the abstract value of $r5$ used in the store operation at $l1$. Using the computed relation $r5 = (1, 2, 1)r2 + (0, 1, 1)r3$ we get $r5 = (1, 2, 1)(6, 8, 2) + (0, 1, 1)(1, 9, 1) = (6, 16, 2) + (0, 9, 1) = (6, 25, 1)$. On the other hand, the val abstraction computes $2(6, 8, 2) = (12, 16, 4)$ for $r5$ before entering the loop and $(6, 8, 2) + (1, 9, 1) = (7, 17, 1)$ within the loop. Taking union at $l1$ we get $r5 = (12, 16, 4) \cup (7, 17, 1) = (7, 17, 1)$. The val abstraction is better than $coeff$ in this case.

The selection is implemented in a separate pass after the two abstractions have been built up. For any static object s_i, $s_i.val$ is re-evaluated to store the current best approximation:

$$s_i.val = s_i.val \cap eval\_coeff(s_i)$$

where $eval\_coeff$ computes a CLP for s_i using the $coeff$ matrix and current best values for other static objects appearing in the relation. The correctness of this step follows from the fact that the intersection of two safe sets representing the same concrete set is safe.

9 Experimental Setup

The analysis framework is applicable to a wide range of ISAs. Porting to a new ISA requires coding the abstract instruction semantics for each instruction and implementing modules that identify memory partitions and basic block boundaries. We have implemented the framework for the ARM7TDMI [1]. The ARM7TDMI is a 32-bit RISC processor and has applications in audio equipments, wireless devices, printers, digital still cameras, etc.

Analyzer Architecture: Figure 6 shows a block diagram of our tool. Call graph construction is complicated by the fact that the ARM7TDMI does not have a RET instruction. Currently we treat any PC write as a potential return point. We also assume that all basic blocks of a procedure are grouped together in memory. Procedure entry points are identified by targets of the BL instruction and the entry point for execution. Basic blocks are sorted by start address and all basic blocks between the start of the current procedure and the next procedure are assigned to the current

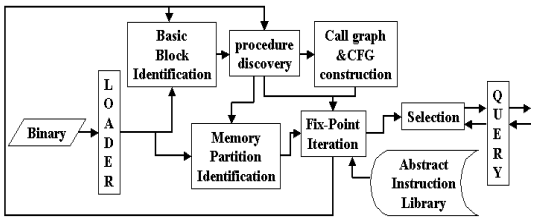

Figure 6. Analyzer Architecture

procedure. This step may be revisited during fix-point iteration when function pointers get resolved. The query engine accepts a register number or memory address and returns the corresponding CLP. The abstract execution semantics for any instruction is obtained from an executable specification of the concrete semantics and modifying it to use CLP operators and operands. The analysis is flow-sensitive, context-insensitive. As in [2], we follow the approach of simultaneous numeric and pointer analyses.

Memory Partitions: Memory partitions are determined by scanning the global data section and program code for numeric offsets and stack operations. A unique identifier is associated with each partition based on the address range [start,end] and defining procedure; $M : A \times A \times P \rightarrow \mathbb{N}$ defines this map, where A and P denote respectively the address space and set of procedures. A special value for P is assumed for global data.

Data Structures: Each static object is represented by a structure with the following fields:

- val: a CLP that represents the current value of the static object.
- $coeff$: a CLP array of size $N + 1$ to represent linear equality relations with other static objects. Figure 4(c) shows the matrix representing relations in Figure 4(b). The boxed entries indicate the self-coefficient(λ_i) for each s_i. For example, the fourth row in Figure 4(c) denotes the relation $2r3 = r0 + r1 + 20$.
- $status$: a 1 byte field containing flags which indicate if a fix-point for this static object has been reached.

The abstract state of the program at any point is given by the abstract states of all static objects at that point. The space required for representing N static objects is $O(N(N+2)) = O(N^2)$. Two copies of the abstract program state are maintained per basic block – the IN copy represents program state when control reaches the current basic block, whereas the OUT copy represents the state at exit. Apart from this a working state, $CURR$, is maintained while processing any basic block. The total space overhead is $O(N^2(2|B| + 1))$.

State Lookup and Update: When all operands of an instruction are registers or memory locations mapping fully to a single memory partition, reads and writes are straightforward – the val field for the object corresponding to each source is read and that corresponding to the destination

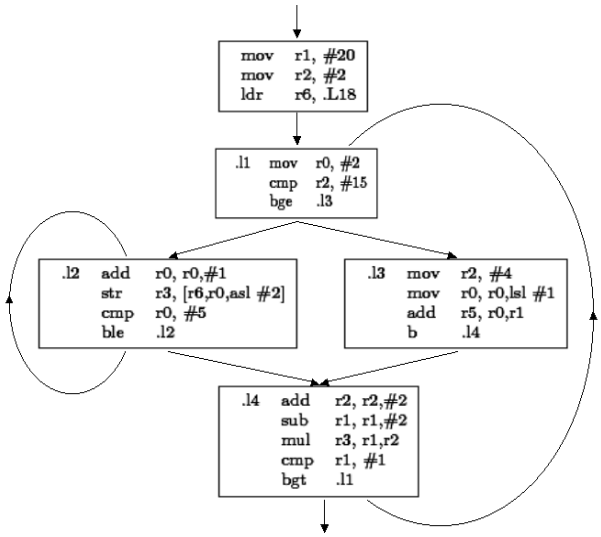

Figure 7. Sample program and CFG

operand is updated with the result computed. In case of other full and non-strided memory accesses, a number of objects corresponding to multiple memory partitions may be read/updated. For reads, the source CLP is taken as the union of the values from the individual partitions. For writes, val for each destination partition is unioned with the data to be written. Static objects corresponding to registers are atomic, but this is not the case with those corresponding to memory partitions. This is because in case of memory, the partition size may be larger than the smallest access size allowed by the processor. Additionally, non-aligned accesses may stride multiple partitions. For partial and/or strided accesses, a read results in the top element, $\top$, of the CLP lattice to be returned, while a write results in all affected memory partitions to be abstracted by $\top$ till a subsequent update.

Sample Analysis: The control flow graph for a sample input program is shown in Figure 7. $.L18$ is the label for the starting position (0x20080e0) of an integer array. Our analyzer determines the following CLPs as safe *after* update by the corresponding instruction: $.l4$: $r2 \rightarrow (4, 16, 2), r1 \rightarrow (0, 18, 2), r3 \rightarrow (0, 288, 4)$; $.l3$: $r5 \rightarrow (6, 24, 2)$; $.l2$: $r0 \rightarrow (3, 6, 1)$, Mem access $\rightarrow$ (0x20080ec,0x20080f8,4). The CLPs for $r2$, $r1$, $r0$ and memory access addresses tightly abstract the corresponding concrete sets. For this example, the filters at the exit points of blocks $.l1$, $.l4$ and $.l2$ play a critical role in ensuring tightness for $r2$, $r1$, $r0$ respectively. For $r3$, the concrete set is $\{0, 16, 24, 72, 96, 112, 120\}$ and for $r5$ it is $\{10\}$. The over-approximation for $r3$ happens because we do not specially handle the case of multiplication where the operands change in step. For $r5$, the analyzer is unable to recognize that block $.l3$ will be executed only once and at a particular value of $r1$.

10 Conclusions

This paper introduces the CLP abstract domain for static program analysis. Key features are abstraction of finite word length computations, scalability in the range of operations supported, analysis efficiency, and utilization of circularity, discreteness of data, linear equality relations and branch filters to tighten the abstractions. A limitation in our relation analysis is that the set of equations abstracting the concrete state at a program point is not unique.

We now outline several potential applications of our tool.
Address Analysis: In this problem we are interested in computing a safe approximation of the set of addresses accessed by a memory reference instruction. This information can be used in at least two ways. The first application is in points-to and alias analysis. This can be handled by obtaining the respective CLPs at the program point and checking whether or not their intersection is empty. The second application is in estimating the Worst-Case Execution Time (WCET) of programs in the presence of data caches. This has applications in schedulability analysis of real time systems. Static analysis using Abstract Interpretation for caches has been explored in [6]. In case of data caches, an additional problem to be solved is determining the memory locations accessed by any memory reference instruction. We are currently building a WCET analyzer that supports data cache analysis by constructing CLPs for each memory reference. The current status of the work can be found in [11].
Range Query: In this problem, given a register/memory location whose initial contents are unknown, and a program point, we are interested in finding out a range of initial values for the unknown quantity so that some property holds at the given program point. For example, determining a range of values for which a particular condition evaluates to true, with applications in code coverage. Another example could be finding a range for which an assertion is violated, with applications in static debugging. Such problems can be handled by starting with the unknown quantity set to $\top$ and then performing several runs, each time adjusting l or u so that the size of the initial abstract set is halved every run in a manner similar to binary search. Continue till tight bounds are obtained. Then successively refine d starting from 1 and doubling every run similar to binary search. The total number of runs required will be $O(log(2^n)) = O(n)$.
Trace Reduction: In this problem we are interested in reducing the size of a simulation trace. A trace records the values of selected registers and memory locations at predetermined program points. Instead of storing absolute values, we propose to store offsets into the CLPs. For example, suppose that the value of register $r0$ at some point is 1006 and the CLP is $(1000, 1008, 2)$. The offset of 1006 into this set is 4 which can be stored instead of 1006.

Precision of the analysis can be improved for certain non-linear computations with modifications targeting specific cases. For example, the division of one set by a value that is not a factor of the common difference can be handled better by using progressions with rational numbers, maintained as integer 6-tuples. Another area of improvement could be in the identification of memory partitions which is currently based on statically determinable offsets in the code. Often fine-grained offsets are absent. A code fragment that initializes array elements in a loop and subsequently processes them, may result in having a single partition for the whole array. A better technique could be to deduce the sizes of data structures used and track individual memory locations depending on the size of the structure. We plan to explore this further.

Acknowledgments

This research was supported in part by the Defence Research and Development Organisation, India.

References

[1] ARM7TDMI. Technical Reference Manual. "http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf".
[2] G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *CC*, pages 5–23, 2004.
[3] R. Clarisó and J. Cortadella. The Octahedron Abstract Domain. In *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, Aug. 2004.
[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, New York, NY, USA, 1977. ACM Press.
[5] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In *POPL*, pages 12–24, New York, NY, USA, 1998. ACM Press.
[6] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17((2/3)), 1999.
[7] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Form. Methods Syst. Des.*, 11(2):157–185, 1997.
[8] M. Karr. Affine Relationships Among Variables of a Program. *Acta Inf.*, 6:133–151, 1976.
[9] M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *ICALP*, pages 1016–1028, 2004.
[10] M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *POPL*, pages 330–341, New York, NY, USA, 2004. ACM Press.
[11] R. Sen and Y. N. Srikant. Estimating WCET in the presence of Data Caches. Technical Report. "http://archive.csa.iisc.ernet.in/TR/2007/5".
[12] R. Sen and Y. N. Srikant. Executable Analysis with Circular Linear Progressions. Technical Report. "http://archive.csa.iisc.ernet.in/TR/2007/3".