WCET Estimation for Executables in the Presence of Data Caches

Rathijit Sen
rathi@csa.iisc.ernet.in

Y. N. Srikant
srikant@csa.iisc.ernet.in

Department of Computer Science and Automation
Indian Institute of Science
Bangalore-560012

ABSTRACT

This paper describes techniques to estimate the worst case execution time of executable code on architectures with data caches. The underlying mechanism is Abstract Interpretation, which is used for the dual purposes of tracking address computations and cache behavior. A simultaneous numeric and pointer analysis using an abstraction for discrete sets of values computes safe approximations of access addresses which are then used to predict cache behavior using Must Analysis. A heuristic is also proposed which generates likely worst case estimates. It can be used in soft real time systems and also for reasoning about the tightness of the safe estimate. The analysis methods can handle programs with non-affine access patterns, for which conventional Presburger Arithmetic formulations or Cache Miss Equations do not apply. The precision of the estimates is user-controlled and can be traded off against analysis time. Executables are analyzed directly, which, apart from enhancing precision, renders the method language independent.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Performance, Verification

1. INTRODUCTION

Estimation of worst case execution time of programs is extremely important in the context of real time systems where the correctness of the system depends not only on the computations performed, but also on the timing of such computations. For task scheduling on such systems, it is necessary to know whether the task can execute to completion within a predetermined time interval. Thus, given a program and a target architecture, the WCET problem is to estimate a bound on the maximum execution time taken by the program for any input data set. A related problem is the determination of worst case energy consumption by the program to ensure that the battery does not drain out before the completion of the task. WCET estimation is again a central problem to be solved in this context.

A simple approach could be to assume worst case latency for every instruction, determine the maximum execution time of each basic block and maximize the execution time over all paths. This approach, although valid, may overestimate the WCET by a large amount as it fails to recognize the presence of performance enhancing features such as caches and pipelines in the architecture. Our techniques are directed towards improving the estimates in the presence of data caches and can easily be integrated with analyses targeting other features.

In the context of hard real time systems, the WCET estimate of a program must be safe, that is, it cannot exceed the actual execution time for any input data set. It is also desired that the estimate be tight to reduce resource allocation costs. The stringent requirement of safety may be relaxed in the case of soft real time systems where deadlines may occasionally be missed without having a significant impact on the quality of service offered. In this work, we explore a static analysis technique that gives safe estimates for WCET and a slight modification that results in most likely values for WCET. The safe and probable estimates can be used for hard and soft real time systems respectively.

Our techniques are based on the analysis of executable code. Analysis techniques that are based purely on source code may not be very reliable as compiler transformations may have radically changed the underlying code structure. At the very least, some variables may be allocated to registers and may not cause a memory access when used, while others might be retrieved from memory, used and possibly stored back again. Additionally, there can be memory spills at unknown points which can collide in the cache with other memory accesses.

2. RELATED WORK

A significant amount of research has been made in estimating data cache behavior in the presence of memory accesses which are affine in the loop induction variables. Two major techniques are the use of Cache Miss Equations(CME)

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

[9] and Presburger Arithmetic formulations [5]. The CME approach has been used in [11] for analyzing worst case data cache behavior with enhancements to the original framework. The Presburger Arithmetic method is exact, but the analysis time can be super-exponential in the length of the formulae in the worst case. Both of these approaches are limited by their dependence on affine access patterns. The techniques cannot be applied in the presence of data dependent conditionals and indirection even if the array indices are affine in the loop induction variables. Further, it is not clear how how they integrate with analyses that target other architectural features such as pipelines and instruction caches.

The work in [14] uses information from the compiler and successive substitution to calculate relative addresses of memory accesses. Control flow information is then used to convert these addresses to virtual addresses. Next, static cache simulation is performed to produce access categorizations. The approach is similar to an interprocedural data flow analysis. The algorithm is quite complex and it may be difficult to check for safety.

Abstract Interpretation [6] is a well established technique for static analysis of programs. A major advantage with Abstract Interpretation is that it guarantees safety. Static estimation of cache behavior for scalar accesses has been extensively studied in [8] using Abstract Interpretation with abstract cache domains. An extension has been proposed in [7] for data caches. However, two critical subproblems – address computation and access sequencing have not been discussed. Performance numbers are also not known. Simultaneous numeric and pointer analysis for tracking memory accesses in executable code for x86 has been studied in [4] using Abstract Interpretation with Reduced Interval Congruences. Contents of registers and statically known memory partitions are tracked.

Our approach combines automatic executable analysis for address determination and Must Analysis for predicting cache behavior, both using Abstract Interpretation. The abstract domain in [4] has been extended to support finite width computations and a wide range of operations, thereby allowing a larger set of programs that can be analyzed. The original Must Analysis is extended to support sets of memory addresses instead of singleton sets. Partial access sequencing is an integral part of our analysis. We also propose a heuristic that gives tighter estimates but may not be safe. It can be useful in the context of soft real time systems and also for reasoning about the tightness of the safe estimate. Programs with both affine and non-affine access patterns are analyzable by our tool.

Our analysis is flow sensitive, context insensitive. CFGs for individual procedures are linked together at call and return points to form a SuperGraph before analysis starts.

3. FOUR SUBPROBLEMS

In our view, the WCET problem for data caches can be decomposed into four main parts – address analysis, cache analysis, access sequencing and worstcase path analysis. We adopt a modular approach and separate the concerns from each other. This enables each of these to independently evolve using increasingly sophisticated methods.

(a)Address Analysis: This subproblem is concerned with determining the set of memory locations that are accessed at any point in the program. This information will be used to determine the cache effect (hit/miss) and the cache state update due to the corresponding memory accesses. For immediate addressing, address determination is straightforward. However, in most cases, addresses are computed by the program instructions before the access is made.

We use Abstract Interpretation to compute a safe approximation of the set of memory addresses being accessed by any memory reference. We apply the technique of simultaneous numeric and pointer analyses as developed in [4]. The abstract domain that we use is more compact in representation and have their semantics defined for a very wide range of operations. The domain is further described in §5.

(b)Cache Analysis: This subproblem is concerned with determining the cache behavior (hit/miss) for any access and the subsequent cache state update.

We use Abstract Interpretation to compute a safe approximation of the cache state at a given program point using the information computed by Address Analysis. We reuse and extend techniques developed in [8] to support access streams where the memory access at each point may be over-approximated by a set of addresses instead of a single address. §6 details the analysis.

(c)Access Sequencing: This subproblem is concerned with determining frequency and ordering of accesses to distinct memory locations generated by memory references during execution. This information is very important in determining the existence of spatial and temporal reuse. For example, consider a direct mapped cache and a memory reference accessing the locations x, y, z during execution. Assume that x and y conflict in cache and that z lies in the same block as x. The access order (x, y, z) results in all misses whereas (x, z, y) results in a hit. In both cases, the set of locations accessed is $\{x, y, z\}$ but the results are different due to access order. Similar observations hold for address set $\{x, y\}$ and access sequences x, x, x, y and x, y, x, y.

We deal with this sequencing problem through both partial physical and virtual unrolling of loops. Partial physical unrolling of the outer loop partitions the iteration space into regions with sequencing across regions. We alternately select a region to be analyzed in either *expansion* or *summary* mode. The expansion mode can be visualized as a virtual unrolling of the loop nest over the region and performing the analysis over this virtually unrolled code. It is different from simulation since at any point it takes into account all possible scenarios whereas simulation is concerned only with the current input data set. Expansion mode maintains sequencing within the region. Additionally, it also helps to prime the abstract cache. The summary mode performs analysis over the region without unrolling it. It does not consider sequencing within the region but is faster. Selection between these two modes is governed by a tradeoff between tightness of analysis and analysis time and can be controlled by the user. Section 7 deals with this process.

(c)Worstcase Path Analysis: This subproblem is concerned with determining the path corresponding to the worst execution time.

We handle this by first computing worst case costs for each basic block and then solving an integer linear program (ILP) to maximize the overall execution cost subject to structural constraints. Use of ILP to determine the worstcase path and hence the WCET is an established technique. As our analysis is context insensitive, interprocedural constraints need to be added to avoid unbounded cyclic constraints in the SuperGraph due to false paths. §9 describes this method.

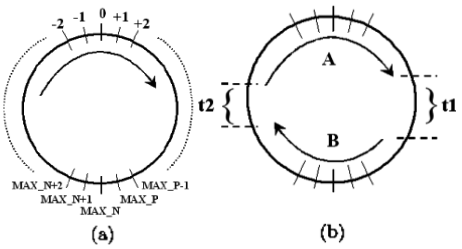

Figure 1: (a)CLP visualization (b)Set union

4. ASSUMPTIONS

We assume that the input executables have reducible flow graphs and that all loops are counter-based with the loop continuation condition being a comparison between the contents of a register/memory location and an expression that evaluates to a constant after constant propagation. Any two loops are either disjoint or one is contained in the other. Any loop has a single entry point but can have multiple exit points. Currently we do not handle recursion, either self or mutual, and the call graph is assumed to be acyclic. We also assume that the target architecture employs an in-order execution model. An underlying assumption in our analysis is that misses correspond to worst case scenarios. The WCET problem for data caches then reduces to estimating a lower bound on the number of hits. The assumption may not hold for architectures employing out-of-order execution semantics [10]. Further, we assume that caches are virtually addressed and the replacement policy is perfect LRU.

5. ADDRESS ANALYSIS

The objective is to compute a safe approximation of the set of memory locations that can be accessed by any memory reference. The approach is similar to that in [4], but the abstract domain is enhanced to support finite width computations and composability for a wide range of operations. Our domain is extensively described in [13].

CLP Abstract Domain: We model discrete value sets as circular linear progressions (CLPs) 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 circular disc as in Figure 1(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: $\top.l = \top.u + 1, \top.\delta = 1$.

Static Objects: Our analysis tracks contents in registers and statically identifiable memory partitions, each of which is represented by a static object. Two copies (IN,OUT) of the abstract state per object are maintained per basic block. 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. 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.

Abstract Transfer Functions: These functions define the composition of a CLP $A = (l_1, u_1, \delta_1)$ with another CLP $B = (l_2, u_2, \delta_2)$ or an integer, k. Transfer functions have been defined for a wide range of arithmetic, logical, shift, bitwise and set operations. Here we present a few sample compositions.

UNION

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

$diff$ is chosen by considering two alternatives as shown in Figure 1(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.

For the following transfer functions, first assume that $MAX\_N \leq l_1, u_1, l_2, u_2 \leq MAX\_P$, $l_1 \leq u_1$, $l_2 \leq u_2$ and no operation results in overflow.

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

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

```
int t[12];
int x[100];

int main()
{
    int i,j,k,p;
    int *d;
    d=x;
    for(k=1;k<=8;k+=7) {
        for (i=0;i<8;i++) {
            for (j=0;j<4;j++) {
                t[j] = d[k * j] + d[k * (7 - j)];
                t[7 - j] = d[k * j] - d[k * (7 - j)];
            }
        }
    }
    return 0;
}
```

Figure 2: Sample computation

LEFT SHIFT

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

To remove the restriction $l < u$ for the above compositions, given any general CLP $C(l, u, \delta)$, we construct two disjoint sets P, Q as follows:

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

Now, any operation $\otimes$ other than $\cup$ and $\setminus$ is defined as

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

Overflows are handled by computing the result in $2n$ bits and converting it back to an n-bit representation. This is done by decomposing the result into 2 CLPs, X containing the first n-bit part and Y containing the lower n-bits of the rest of the result obtained by ANDing the rest part with MAX_D. The final n-bit result is computed as $X \cup Y$.

Linear Dependence: Precision can be improved by considering linear relations between static objects. For example, in the sequence $r0 = m1 + r2, r3 = r0 - r2$, the fact that $r3$ is equal to $m1$ can be used to compute a possibly tighter abstraction for $r3$. We use a best-of-two selection strategy between the result computed by the transfer function and that implied by the relation matrix. Space and time complexity is quadratic in the number of static objects tracked.

Sample analysis: Figure 2 shows the source for a sample computation which is similar to that appearing in a JPEG DCT algorithm. The corresponding ARM7 assembly code could not be included here due to space constraints, but is available in [12]. $d[k * j]$, $d[k * (7 - j)]$ are read by two memory reference instructions and $t[j]$, $t[7 - j]$ are modified by another two. x starts at virtual address 0x2008120 and t at 0x20080f0. Our analysis automatically computes the following CLPs to describe the sets of memory locations being accessed by the 4 instructions mentioned in that order: (0x2008120,0x2008180,4), (0x2008130,0x2008200,4), (0x20080f0,0x20080fc,4) and (0x2008100,0x200810c,4). By simulation, we find that the approximations for the writes are exact. For the reads, the actual address sets are proper subsets of the abstract sets computed. For the first read, the simulation address set corresponds to 7 array indices: 0,1,2,3,8,16,24 whereas our approximation corresponds to 25 indices: 0 through 24, and is the closest linear approximation to the actual set. For the second read, the actual set corresponds to 8 indices: 4,5,6,7,32,40,48,56 whereas our approximation includes 53 indices: 4 through 56. In all cases, the analysis correctly detects the inherent granularity of 4 bytes in the access stream.

6. CACHE ANALYSIS

An abstraction of the state and transformation semantics of physical caches is useful for tracking the set of all possible cache states that may hold at a program point during execution. The methodology adopted here is an extension of the Abstract Cache model and Must Analysis technique developed in [8]. Must analysis tracks the set of memory blocks definitely residing in the cache at any program point. This is useful for tracking memory accesses that will always result in cache hits regardless of program input. Currently, only set associative caches with perfect LRU replacement policy are supported.

Cache State: The abstract cache state at any point in the program is a safe approximation of all possible concrete cache states that can hold at that point over various execution sequences. Two copies (IN,OUT) of the abstract cache state are maintained per basic block. The organization of the abstract cache is similar to the concrete cache, but with two notable differences:

1. The abstract cache blocks in any set are arranged in increasing order of access age.
2. Each abstract cache block can hold data corresponding to a set of memory blocks instead of only 1 block in the concrete case.

Consider a cache organization with associativity A, number of sets S and block size B. For abstract set $\hat{s}$, $\hat{s}(0)$ denotes the set of memory blocks most recently accessed whereas $\hat{s}(A-1)$ denotes the set of least recently accessed ones. Memory blocks in $\hat{s}(A-1)$ are candidates for replacement on the next miss mapping to $\hat{s}$. Memory blocks mapping to the same abstract set and determined to have the same age map to the same abstract cache block. Maintaining accurate information for a large number of memory blocks mapping to the same abstract block not only consumes space, but also increases processing time as each block needs to be individually checked to determine reference classification. We maintain another parameter, η, which serves as a threshold. When the number of memory blocks mapping to the same abstract cache block exceeds η, the abstract cache block is cleared. This decision is safe with respect to our analysis.

Reference Classification: A memory reference with address set described by the CLP x is classified as *always hit*(ah), iff every address $a \in conc(x)$ is present in the abstract cache state at that point. Otherwise, it is marked as *non classified*(nc). Performing lookup for every $a \in x$ is time consuming if $|x|$ is large. We maintain a parameter ξ which serves as a CLP unroll limit. If $|x| \geq \xi$, we treat it as $\top$ and classify the reference as nc. Access latency for nc references is equal to the miss latency, while for ah it is the hit latency.

Cache Update: A key difference between instruction and data references is that the address set for the latter may not be a singleton set, as for example, array references. Further, when the address set is not singleton, we cannot say which particular subset of addresses will be definitely accessed during actual execution. This is because Address Analysis computes an over-approximation of the actual set and in the absence of other information, it is not possible to say whether the abstraction is tight or not. Thus, in such cases, no new element can be brought into the abstract cache as that element may never be accessed during any concrete execution. This is to ensure safety since Must Analysis computes upper bounds on ages of memory blocks. However, if the address set is singleton, we know that the abstraction is tight and the addressed memory block will always be brought into the cache.

Let x denote the CLP describing the address set of a memory reference at a point. For any $a \in x$, let $\phi(a) = (a/B)$ denote the block address of a and $\rho(a) = \phi(a)\%S$ denote the cache set that a maps to. The abstract cache update function $\hat{U}(x, \hat{s})$ defines how the state of abstract set $\hat{s}$ needs to be updated, as a result of (multiple) accesses to addresses in x, to ensure safety for Must Analysis. In the following, the LHS of $\mapsto$ describes state *after* update whereas the RHS is computed using state *before* update.

First suppose $|x| = 1$ and let $\hat{s} = \rho(x.l)$. If $\exists h : \phi(x.l) \in \hat{s}(h)$ (hit), then,

$$\hat{U}(x, \hat{s}) = \begin{cases} \hat{s}(i) \mapsto \hat{s}(i-1), 1 \leq i < h \\ \hat{s}(0) \mapsto \{\phi(x.l)\} \\ \hat{s}(h) \mapsto (\hat{s}(h) \setminus \{\phi(x.l)\}) \cup \hat{s}(h-1), \text{if } h > 0 \end{cases}$$

otherwise, (miss)

$$\hat{U}(x, \hat{s}) = \begin{cases} \hat{s}(i) \mapsto \hat{s}(i-1), 1 \leq i < A \\ \hat{s}(0) \mapsto \{\phi(x.l)\} \end{cases}$$

Next, consider $|x| > 1$. For any abstract set $\hat{s}$, define $shift\_ctr(\hat{s}, j), 0 \leq j < A$, as the number of positions that the contents of $\hat{s}(j)$ need to be shifted towards the older age blocks to make way for memory blocks that can potentially occupy the lower age blocks. To compute this, we determine all addresses in $conc(x)$ mapping to $\hat{s}$ and then the number of distinct memory blocks present in abstract blocks of age greater than j or not present at all. Formally,

$shift\_ctr(\hat{s}, j) = |\{a \in conc(x) | (\rho(a) = \hat{s}) \wedge ((\phi(a) \in \hat{s}(k), j < k < A) \vee (\phi(a) \notin \hat{s}(k), 0 \leq k < A))\}|$

$shift\_ctr$ is computed using state *before* update. The update function is now defined as:

$$\hat{U}(x, \hat{s}) = \begin{cases} \hat{s}(i) \mapsto \{k | k \in \hat{s}(j) \wedge ((j + shift\_ctr(\hat{s}, j)) = i), \\ \qquad\qquad 0 \leq i < A\} \end{cases}$$

If $(i + shift\_ctr(\hat{s}, i)) \geq A$, the old contents of $\hat{s}(i)$ are shifted out and do not appear in the new state. The above definition ensures the following two conditions if $|x| > 1$:

1. No new memory block can be brought into the abstract cache
2. No memory block already in the abstract cache can decrease in age

The following example illustrates the update function. The first line shows the state of abstract set $\hat{s}$ of a 4-way associative cache. Each of the following lines show the resulting state after $\hat{U}(x, \hat{s})$ has been applied to the state before. Assume that all memory blocks m_i shown have $\rho(m_i) = \hat{s}$.

	Age →			
$\hat{s}$		$\{m_1\}$	$\{m_2\}$	$\{m_3, m_4\}$
$\hat{U}(\{m_3\}, \hat{s})$	$\{m_3\}$		$\{m_1\}$	$\{m_2, m_4\}$
$\hat{U}(\{m_5\}, \hat{s})$	$\{m_5\}$	$\{m_3\}$		$\{m_1\}$
$\hat{U}(\{m_2, m_3\}, \hat{s})$			$\{m_3, m_5\}$	

Cache Join: In Must Analysis, the join function for an abstract cache set results in associating each memory block with the maximum age, equivalently, maximum abstract block number in the abstract sets being joined. Formally, the join function of two abstract sets $\hat{s1}$ and $\hat{s2}$ is an abstract set $\hat{s}$ with $\hat{s}(j)$, $0 \leq j < A$, defined as follows:
$\hat{s}(j) = \{m | \exists a, \exists b, 0 \leq a, b < A : (m \in \hat{s1}(a)) \wedge (m \in \hat{s2}(b)) \wedge (j = max(a, b))\}$

7. ANALYSIS MODES

As mentioned earlier, access sequencing is handled through partial physical and virtual unrolling of loops. Partial physical unrolling divides the program into regions and regions are alternately assigned to be analyzed in either Expansion or Summary mode. The size and number of regions and hence the relative proportion of time spent by the analysis in each of these modes can be controlled by the user through the following environment settings:

- $frac\_exp$: approximate fraction of the total iteration space to be analyzed in expansion mode
- $samples$: number of regions over which the fraction to be analyzed in expansion mode is to be distributed.

Each outer loop is partially unrolled to form $2 \times samples$ regions. Let α denote the number of program iterations that a single region in expansion mode corresponds to and let β denote the number for summary mode. Let N denote the total number of iterations of the outer loop. Then,

$$samples \times (\alpha + \beta) = N, \qquad samples \times (\alpha) = frac\_exp \times N$$

$$\Rightarrow \alpha = \frac{frac\_exp \times N}{samples}, \qquad \beta = \frac{N \times (1 - frac\_exp)}{samples}$$

Unrolling is performed by disconnecting the loop blocks from the rest of the SuperGraph, replicating the blocks, changing the loop continuation constants and finally connecting all the blocks (old and created) back to the SuperGraph. For each region created, the iteration space is constrained by setting the loop continuation condition appropriately. Consider for example, a simple dot-product application as shown in Figure 3. Let $frac\_exp = 0.1$ and $samples = 4$, that is, we want to analyze 10% in expansion mode spread over 4 samples. The partially unrolled CFG is shown in Figure 3(d) along with the modified loop continuation conditions for each region. Regions 1,3,5,7 are set to expansion mode(E) and each corresponds to 1 iteration of dotp. The rest are in summary mode(S) and correspond to 9 iterations each.

Expansion mode: The analysis in this mode can be visualized as a virtual unrolling of the loop nest over a portion of the iteration space. The region can correspond to more than 1 program iteration depending on $frac\_exp$ and $samples$. The expansion mode helps to prime the abstract cache as more often, references have singleton address sets in this mode. It also maintains sequencing within the region.

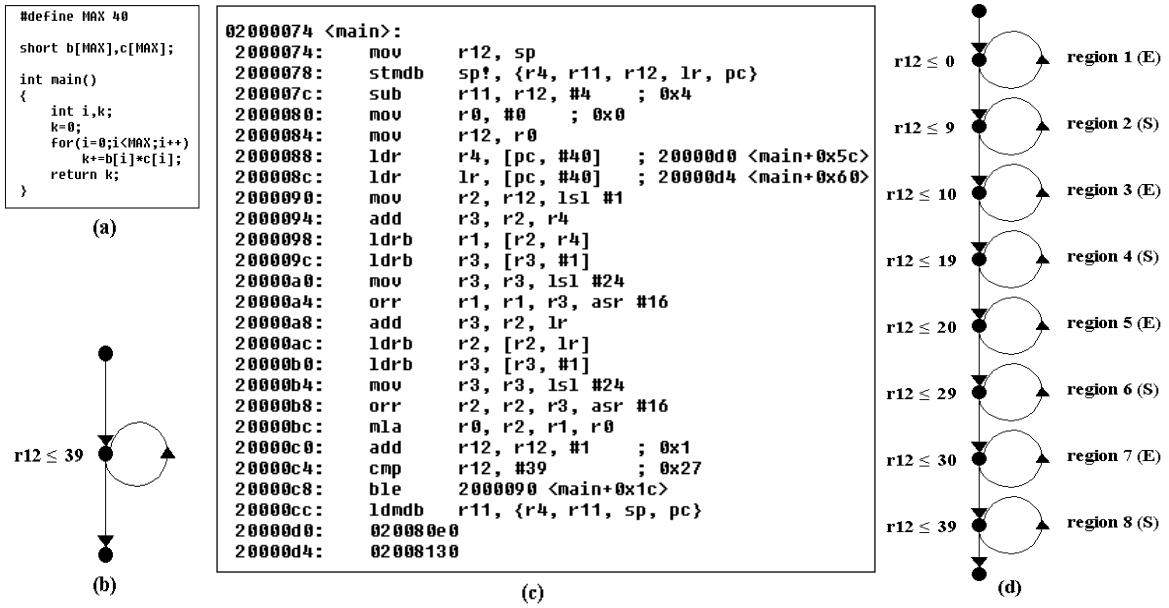

Figure 3: (a)dotp source (b)original CFG (c) ARM7 assembly code (d) partially unrolled CFG (10%,4)

Blocks in the region are analyzed in any reverse postorder sequence (produced by a topological sort ignoring back edges). At every branch, the evaluation of the condition (if any) at that point determines whether the true path or the false path needs to be taken. The successor block on the path that is not to be taken and all the blocks that it dominates are not processed. In case the condition cannot be decisively resolved, both paths will be activated for further analysis. A boolean flag indicating activation is maintained with each basic block data structure and is set to true or false according to the above decision. In every pass, all blocks in the region will be considered, but a block will be processed iff this flag is set to true. Iteration over the list of blocks in that region continues as long as the basic block corresponding to the loop header for that region is activated. Deactivation can happen only when the loop continuation condition evaluates to false. In that case, the loop header is not on the path to be taken and all blocks in the region that it dominates (which is all the blocks in that region) are deactivated and iteration terminates. The analysis is equivalent to performing Abstract Interpretation over the virtually unrolled loop. As the virtually unrolled part is acyclic, no widening is required.

Both Address and Cache Analyses are carried out simultaneously. The address computed for any reference in any pass is immediately used for abstract cache lookup and update. Depending on the result of cache lookup, the appropriate latency is added to the local worstcase cost for that block. Worst case estimates over the whole region can be efficiently computed without solving integer linear programs. The key observation is that in this mode, the virtually unrolled loop resembles a directed acyclic graph, so the problem reduces to the following well studied graph-theoretic problem:

"Given a DAG with nodes in topologically sorted order and costs on vertices, find the longest path from the first to the last vertex"

The above problem can be solved by considering each vertex in turn, considering the maximum values propagated to it by its predecessors, adding its own cost and propagating the new value to its successors. The expansion mode takes time proportional to the static size of the relevant portion of the SuperGraph multiplied by the number of iterations unrolled. No extra space is required as unrolling is virtual.

Summary mode: In this mode of analysis, abstract interpretation for only address computation is carried out till a fix-point is reached. At the end of the fix-point iteration, the abstract cache analysis is carried out using the fix-point CLP for each memory reference till a fix-point is reached. Reference classification happens using the fix-point states for the cache. The summary data loses intra-region sequencing information, but analysis is faster. The analysis time taken in this mode depends only on the static size of the relevant portion of the SuperGraph.

The distinction between expansion and summary modes only applies if a block is part of a loop. Otherwise, for non-loop blocks, both analyses are equivalent.

Sample analysis: Consider the dot-product example of Figure 3. In 3(c), we see that two reads happen for each element of b. Similarly for c. There are a total of 4 reads in every loop iteration and the result variable is allocated a register. These facts are not obvious from the source. The CLP sets for the 4 memory reference instructions at PCs (in hex) 2000098, 200009c, 20000ac, 20000b0 computed by Address analysis are (in hex) (20080e0,200812e,2), (20080e1,200812f, 2), (2008130,200817e,2), and (2008131,200817f,2), each including 40 2-byte elements. With $frac\_exp = 0$, analysis happens on the CFG of Figure 3(b). None of the memory blocks can be brought into the abstract cache and every reference is categorized as *nc*.

With $frac\_exp = 0.1$, $samples = 4$, analysis happens on the CFG in Figure 3(d). In regions 1,3,5,7, references for both b and c have singleton address sets. Memory blocks corresponding to those addresses are brought into the abstract cache. For other references, memory blocks are not brought into the abstract cache. Assume a cache configuration with associativity=4, block size=32 bytes, sets=256. Table 1 gives the breakup of the address sets computed along with reference classification.

Table 1: Partial access sequencing for dotp (10%,4)

Region	Address(hex)	Block(hex)	Set	Class
1	20080e0	100407	7	nc
	20080e1	100407	7	**ah**
	2008130	100409	9	nc
	2008131	100409	9	**ah**
2	(20080e2,20080f2,2)	100407	7	**ah**
	(20080e3,20080f3,2)	100407	7	**ah**
	(2008132,2008142,2)	100409,10040a	9,10	nc
	(2008133,2008143,2)	100409,10040a	9,10	nc
3	20080f4	100407	7	**ah**
	20080f5	100407	7	**ah**
	2008144	10040a	10	nc
	2008145	10040a	10	**ah**
4	(20080f6,2008106,2)	100407,100408	7,8	nc
	(20080f7,2008107,2)	100407,100408	7,8	nc
	(2008146,2008156,2)	10040a	10	**ah**
	(2008147,2008157,2)	10040a	10	**ah**
5	2008108	100408	8	nc
	2008109	100408	8	**ah**
	2008158	10040a	10	**ah**
	2008159	10040a	10	**ah**
6	(200810a,200811a,2)	100408	8	**ah**
	(200810b,200811b,2)	100408	8	**ah**
	(200815a,200816a,2)	10040a,10040b	10,11	nc
	(200815b,200816b,2)	10040a,10040b	10,11	nc
7	200811c	100408	8	**ah**
	200811d	100408	8	**ah**
	200816c	10040b	11	nc
	200816d	10040b	11	**ah**
8	(200811e,200812e,2)	100408,100409	8,9	**ah**
	(200811f,200812f,2)	100408,100409	8,9	**ah**
	(200816e,200817e,2)	10040b	11	**ah**
	(200816f,200817f,2)	10040b	11	**ah**

Table 2: frac_hit for nc references of dotp

Region	Address(hex)	Block(hex)	frac_hit
2	(2008132,2008142,2)	100409,10040a	7/9
	(2008133,2008143,2)	100409,10040a	7/9
4	(20080f6,2008106,2)	100407,100408	5/9
	(20080f7,2008107,2)	100407,100408	5/9
6	(200815a,200816a,2)	10040a,10040b	3/9
	(200815b,200816b,2)	10040a,10040b	3/9

8. AN ESTIMATION HEURISTIC

In general, for a reference with a non-singleton address set, we may find some fraction $0 \leq frac\_hit \leq 1$ of the set of memory blocks that can be possibly accessed to be in the cache. If $frac\_hit < 1$, we cannot classify the reference as *ah* since all blocks are not present. In the last example, although reference (0x2008132,0x2008142,2) of region 2 has 7 out of 9 elements in block 0x100409 that are in the abstract cache, it cannot be classified as *ah*. Table 2 shows $frac\_hit$ for all *nc* references with non-singleton address sets of the last example.

The probable estimate considers $frac\_hit$ as an indicator of the potential reuse that the memory reference will experience in that region during actual execution. The heuristic takes $frac\_hit$ as a likely estimate of the hit ratio for that particular reference in that region and the corresponding access latency as a weighted average of the hit and miss latencies.

$$access\_latency = frac\_hit \times hit\_latency + (1 - frac\_hit) \times miss\_latency$$

For references with a singleton address set, the access latency is the same as that for the safe case. Thus, the probable estimate is always $\leq$ the safe estimate. Typically, a large difference between safe and probable values indicates that the safe estimate may be tightened significantly by varying $frac\_exp$ or *samples* as potential reuses have not been tracked. The converse does not hold as the abstract cache may not have been adequately primed before analyzing that region and hence $frac\_hit$ is small although reuse exists. The probable estimate may not be safe as the CLP representing the address set may not be a tight approximation of the actual set.

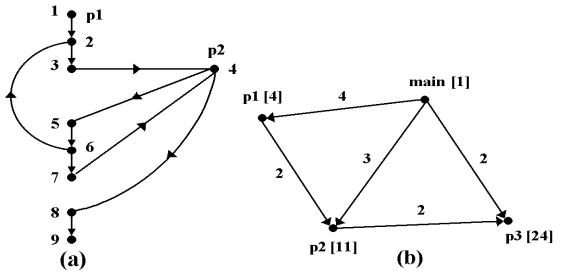

Figure 4: (a)False cycle (b)Instance calculation

9. ILP FORMULATION

After the worst case execution costs for each basic block has been individually computed, an approximation of the overall worst case cost and corresponding path is obtained by solving an ILP. The ILP seeks to maximize the overall cost subject to structural constraints. To every basic block, and every edge, a variable is assigned that indicates the number of times the corresponding block or edge needs to be executed to maximize the overall cost. The values of these variables in the final solution indicates the worst case path and the objective function value gives the overall cost.

Objective function: Let x_i be the variable associated with basic block i and let w_i denote the individual worst case cost of block i. Then the ILP seeks to maximize the value of the expression $\sum_{i=1}^{B} w_i \times x_i$ subject to the following constraints.

Flow constraints: Let $e(i,j)$ be the variable associated with the edge from block i to block j. Then, the flow constraints ensure that the total number of times each block is executed is equal to both the total number of times control can reach this block and also the total number of times control can leave this block. Formally,

$$x_i = \sum_{j \in pred(i)} e(j,i) = \sum_{j \in succ(i)} e(i,j)$$

Loop constraints: In case of loops, the execution count of blocks in the loop need to be bounded by the maximum number of times the loop can execute. We constrain the execution count of the loop back edges as follows.

$$e(i,j) \leq \begin{cases} loop\_cnt - loop\_parent\_cnt & \text{if the loop is not an outer loop,} \\ loop\_cnt - 1 & \text{otherwise.} \end{cases}$$

loop_parent refers to the closest enclosing loop of an inner loop. The loop counts are taken as absolute values. Thus, the loop count of an inner loop is the product of its local count and the local counts of all its enclosing loops. Note that loop constraints are considered only if the particular regions have been analyzed in summary mode. For expansion mode, the worst case for that region has already been computed and will just appear in the objective function.

Table 3: Benchmarks

Name	Characteristics	WCET bounds (cycles)	
		simulation	All-Miss
bsort100	array access,conditions in loops,premature loop exist,imperfect nesting	81091	315897
cnt	array access,conditions in loops,function calls in loops,perfect nesting	4410	9895
edn_fir	array access,imperfect nesting	52990	103133
edn_fir_no_red_ld	array access,imperfect nesting	40328	73114
edn_iir	array and pointer access	2274	5032
edn_latsynth	array access	3281	5875
edn_mac	array and pointer access	3139	6104
jfdctint	array access,loop counter loaded from memory in every iteration	2275	3481
matmult	array access,multiple calls to same function,function calls in loops,imperfect nesting	147413	298022

Interprocedural constraints: Although the call graph is acyclic, cycles may be created in the SuperGraph due to false paths as our analysis is context-insensitive. Consider the example in Figure 4(a). There are 2 locations of calls to procedure $p2$ from procedure $p1$ – from blocks 3 and 7. In the SuperGraph, we see that a cycle has been created with blocks 4 through 7. Unless additional constraints are imposed, the ILP solution will be unbounded. We handle this by imposing upper bounds on blocks which end in function calls as follows:

$$x_i \leq \begin{cases} instances(proc(x_i)) & \text{if } x_i \text{ is not in a loop,} \\ instances(proc(x_i)) \times loop\_cnt & \text{otherwise.} \end{cases}$$

$instances(p)$ of procedure p is the number of instances of p that would have been present if every procedure were inlined at the point of call. To compute this, we first scan every basic block of p to get the maximum number of invocations of every successor procedure, r, of p relative to one invocation of p. Let us denote this count as $call\_cnt(p, r)$. The edges of the call graph are annotated with this count. Figure 4(b) shows an example. $instances(p)$ is now computed by processing each procedure in reverse postorder sequence and using the following recurrence relation.

$$instances(p) = \sum_{q \in pred(p)} instances(q) \times call\_cnt(q, p)$$

For this example, $instances(p)$ for every p are shown in brackets in Figure 4(b).

10. EXPERIMENTAL SETUP

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. We assume the existence of a set associative cache with perfect LRU replacement policy.

We have used the benchmarks listed in Table 3 for testing our setup. Source code for these benchmarks have been taken from [3]. The edn_ programs are subroutines in the edn benchmark. All sources have been compiled with gcc to create ARM7 executables. Complete listing of the sources and assembly can be found in [12]. The actual WCET numbers are obtained by running the executables on the sim-safe Simplescalar/ARM simulator [2] with a configurable cache model added. All-Miss WCET cycles corresponds to our analysis results with every memory reference classified as *nc*. These two numbers provide lower and upper bounds for other WCET estimates. The numbers in the table correspond to the following settings: Associativity=4, Block size=32 bytes, Total Cache size=32 KB, read/write Hit latency=1 cycle, read Miss latency=6 cycles, write Miss latency=4 cycles. Every instruction has a base cost of 1 cycle. Worst case numbers for other configurations appear in [12].

Following are the major processing steps performed by the tool on loading an executable:

CFG construction: The text section is scanned, starting from the program entry point, one word at a time to detect control transfer instructions. Such an instruction may be a Branch or Branch-and-link (function call), or a write to PC (returns and indirect jumps). During processing, a Transfer Queue is maintained. Whenever it is possible to determine the target of a control transfer, the target is enqueued in the Transfer Queue. Processing continues along the current path until a previously visited address is reached again. At that point, an entry from the Transfer Queue is dequeued and processing continues along that path. The process stops when the Transfer Queue is empty and the current path has been exhausted. CFG construction may be revisited later when targets of indirect control transfers get resolved.

SuperGraph construction: The entry point and targets of BL instructions are considered as procedure start addresses. We assume that all basic blocks of a procedure are placed together in memory. The list of blocks is sorted by start address and partitioned according to the procedure start addresses. All blocks falling in a partition are assigned to the corresponding procedure. Blocks ending in function calls are linked with the entry block of the called procedure. The immediately following block in the calling procedure is linked with all return points in the called procedure.

Memory Partitioning: The data section is scanned for identifying global memory locations to be tracked. Next, each procedure is scanned for local memory locations. These are identified through stack push and pop operations. In ARM7 code, $sp(r13)$ is usually copied to $r11$ which is used to access the stack. The contents of the memory partitions are tracked during Address Analysis as described in §5.

Loop detection: Loops are detected with the standard algorithm using dominance relationships. While considering back edges for natural loops we check that the end points of the back edge lie in the same procedure. Registers and memory locations holding loop induction variables for every loop are identified.

Constant Propagation: Constant propagation over the SuperGraph is needed to simplify loop continuation conditions of the form $cmp\ reg1, reg2$ with $cmp\ reg, const$.

Unrolling: As described in §7.

Tables 4 and 5 show the analysis results for the benchmarks listed and the cache configuration mentioned earlier. The missing WCET entries in Table 5 are for the cases where $frac\_exp$ and $samples$ are such that $\alpha < 1$ for that partic-

ular benchmark. This becomes equivalent to the case when $frac\_exp = 0$ since our tool can currently partition only in units of loop iterations. Results for other cache configurations can be found in [12]. The following estimates are provided:

- Safe WCET: estimate not including the heuristic of §8.
- Probable WCET: estimate including the heuristic.

We observe that without any expansion(0%), Must Analysis is not very useful for programs with array accesses. *cnt* at 0% gives good results due to a larger number of scalar references in loops. Scalar references always have singleton address sets. For fir programs, 10% expansion itself shows significant improvement. We also see the large difference between safe and probable estimates indicating possibilities for further improvement of the safe value. The minimum expansion for $jfdct$ is 20% as the outer loop is only 8 iterations. $jfdct$ shows most improvement at minimum expansion. *bsort* does not show good results as our tool is unable to precisely track a condition embedded within the inner loop that effectively makes the loop nest triangular. Without this condition, simulation takes almost twice its current cycles which is close to our estimates. Other programs show progressive improvement as expansion is increased. Increasing *samples* affects results in two ways – it tends to improve as more regions are chosen for expansion ($\propto$ *samples*), and it tends to degrade as each such region gets smaller ($\propto 1/$*samples*). Usually improvement prevails as the regions get more spread out, but there are exceptions. The probable estimate is significantly better than the safe one in most cases and appears to be safe for the given cache configuration and executables.

11. CONCLUSIONS

This paper presents techniques for predicting WCET of executables taking the effects of data caches on performance into account. Executables are analyzed since only source based analyses may not be reliable due to compiler transformations during code generation. Further, the ability to directly analyze executables makes the tool language independent. Another strength of the tool is in the ability to analyze programs with both affine and non-affine memory access patterns. Current limitations include requirement of an inorder execution model, program loops being counter based with the termination condition being a comparison between the contents of a register/memory location and an expression evaluating to a constant, and caches with a perfect LRU replacement policy.

The WCET estimation problem in the presence of data caches is subdivided into four subproblems – address analysis, cache analysis, access sequencing and worstcase path analysis. Abstract Interpretation, with its guarantees of safety, is the basic mechanism for both address and cache analyses. The CLP numerical abstract domain is used for computing a strided linear approximation to the set of addresses that can be generated by any memory reference instruction. This information is used by Must Analysis for determining lower bounds on access ages of memory blocks in the abstract cache states and for classifying references as always hit or otherwise. Sets of access addresses do not have ordering or frequency information. This shortcoming is partially alleviated through partial unrolling of loops. Partial physical unrolling establishes inter-region sequencing which allows separate classifications of the same set of references over disjoint sections of the iteration space. Virtual unrolling establishes intra-region sequencing and helps to prime the abstract cache. Expansion mode results in tighter estimates whereas summary mode reduces analysis time. Together they render Must Analysis useful for data caches. The tool offers a range of analyses by allowing the relative proportion of the two processing modes to be chosen by the user. Our future work will focus on extending the tool to support a wider range of cache organizations.

ACKNOWLEDGMENTS

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

12. REFERENCES

[1] ARM7TDMI. Technical Reference Manual. "http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf".

[2] Simplescalar/ARM. "http://www.simplescalar.com/v4test.html".

[3] WCET Project/Benchmarks. "http://www.mrtc.mdh.se/projects/wcet/benchmarks.html".

[4] G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *CC*, pages 5–23, 2004.

[5] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, pages 286–297, New York, NY, USA, 2001. ACM Press.

[6] 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.

[7] C. Ferdinand and R. Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *LCTES*, pages 16–30, London, UK, 1998. Springer-Verlag.

[8] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17((2/3)), 1999.

[9] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *ICS*, pages 317–324, New York, NY, USA, 1997. ACM Press.

[10] T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.

[11] H. Ramaprasad and F. Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *RTAS*, pages 148–157, Washington, DC, USA, 2005. IEEE Computer Society.

[12] 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".

[13] R. Sen and Y. N. Srikant. Executable Analysis using Abstract Interpretation with Circular Linear Progressions. In *MEMOCODE*, pages 39–48, 2007.

[14] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *RTAS*, page 192, Washington, DC, USA, 1997. IEEE Computer Society.

Table 4: WCET estimates (cycles) for bsort100, edn_fir, edn_fir_no_red_ld, edn_iir

frac_exp	samples	bsort100		edn_fir		edn_fir_no_red_ld		edn_iir	
		Safe	Probable	Safe	Probable	Safe	Probable	Safe	Probable
0	-	315793	308650	103077	103077	73037	73037	4956	4956
0.1	1	301945	274555	75605	60519.2	55393	51010	4696	4528
0.1	2	301945	274564	76116	58998.7	55750	47556.6	4752	4584
0.1	3	301903	262984	75616	57992.9	54972	45093	4706	4370
0.1	4	301867	251410	70127	54683.5	55301	44731.3	4772	4212
0.2	1	286463	251879	73093	60044.3	53724	48722.1	4426	4314
0.2	2	286371	241532	73098	56132.5	53737	47145.9	4436	4156
0.2	3	287830	232045	59606	54438.9	53345	44808.4	4426	4370
0.2	4	286279	231194	68112	54504.7	53356	43312	4436	4212
0.3	1	270884	231663	71081	59642.7	52383	48464.9	4212	4100
0.3	2	270745	222546	71089	55888.9	52221	45404.3	4212	4156
0.3	3	273712	214906	53094	53076	52044	44544.3	4222	3942
0.3	4	270461	204312	66603	54362	52052	42176	4222	3998
0.4	1	255007	206011	68071	55910.1	50375	46229.3	3886	3886
0.4	2	255007	206020	68074	55507.9	50091	43655	3896	3728
0.4	3	254815	198130	53079	53055	50099	43078	3896	3728
0.4	4	254623	190246	60585	53589.1	50102	41917.4	3916	3356
0.5	1	239175	191945	65559	55574.8	48703	44180	3626	3458
0.5	2	238933	185288	59567	53681	48787	43426.4	3682	3514
0.5	3	240262	178999	53070	53013	47189	40976.6	3636	3300
0.5	4	238939	185306	59573	53527.1	48804	41733.6	3692	3356

Table 5: WCET estimates (cycles) for edn_latsynth, edn_mac, matmult, cnt, jfdctint

frac_exp	samples	edn_latsynth		edn_mac		matmult		cnt		jfdctint	
		Safe	Probable	Safe	Probable	Safe	Probable	Safe	Probable	Safe	Probable
0	-	5806	5594	6046	6046	283688	283688	6173	6121	3265	3265
0.1	1	5551	5439	5761	5661	232520	232520	5551	5525	-	-
0.1	2	5540	5108	5791	5491	232567	230439	-	-	-	-
0.1	3	5529	4803	5781	5321	-	-	-	-	-	-
0.1	4	5513	4505	5811	5131	-	-	-	-	-	-
0.2	1	5301	5033	5471	5351	223097	223097	5424	5424	2713	2665
0.2	2	5285	4857	5481	5181	223135	223135	5427	5341	-	-
0.2	3	5295	4803	5491	5011	223182	221718	-	-	-	-
0.2	4	5258	4350	5496	4976	223229	220301	-	-	-	-
0.3	1	5046	4782	5201	5041	218390	217326	5424	5424	2641	2601
0.3	2	5030	4606	5211	4871	223135	223135	5427	5341	2542	2478
0.3	3	5066	4552	5256	4856	218475	215947	-	-	-	-
0.3	4	4998	4350	5226	4666	223229	220301	-	-	-	-
0.4	1	4791	4627	4891	4731	204251	204251	5173	5121	2569	2537
0.4	2	4780	4296	4901	4561	204289	204289	5176	5038	2542	2478
0.4	3	4764	4050	4906	4546	204336	202872	5176	5026	2286	2286
0.4	4	4379	3961	4916	4356	204383	201455	5179	4943	-	-
0.5	1	4536	4376	4601	4421	194828	194828	5046	5020	2569	2537
0.5	2	4520	4296	4626	4406	194875	192747	5176	5038	2542	2478
0.5	3	4530	4050	4621	4081	194913	193449	5049	4925	2286	2286
0.5	4	3885	3619	4646	4046	204383	201455	5179	4943	2286	2286