# Tighter WCET Analysis of Input Dependent Programs with Classified-Cache Memory Architecture

Yanhui Li, Shakith Devinda Fernando, Heng Yu, Xiaolei Chen, Yajun Ha and Teng Tiow Tay
Department of Electrical and Computer Engineering
National Univeristy of Singapore, Singapore 119260
Email: {liyanhui, shakith, h.yu27, g0600118, elehy, eletaytt}@nus.edu.sg

***Abstract*— Caches in Embedded Systems improve average case performance, but they are a source of unpredictability, especially in the worst case software timing analysis with the consideration of data caches. This is a critical problem in real-time systems, where tight Worst Case Execution Time (WCET) is required for their schedulability analysis. Several works have studied the data cache impacts on the WCET of programs, but they can only handle programs with no input dependent data accesses. To solve this problem, we have developed a novel architecture and a WCET analysis framework for this architecture. Our work classifies predictable and unpredictable accesses and allocates them into predictable caches and unpredictable caches respectively, using the CME (Cache Miss Equations) and reuse-distance based algorithms accordingly. The analysis framework produces a very good WCET tightness compared with simulations, and our architecture creates almost no hardware overhead or performance degradation.**

## I. INTRODUCTION

The Worst Case Execution Time (WCET) analysis is to estimate a priori (before execution) the WCET of a given program on a given architecture. It is important for the schedulability analysis of real-time systems, which requires the timing correctness on top of the functional correctness.

Cache memories have created issues in real-time systems, because of their unpredictable nature of cache hits or misses. A conservative approach is to assume all memory accesses miss in the cache, which is obviously overly pessimistic, because some memory accesses are predictable. Whether or not they would be cache misses or hits, it could be determined even before the program is run in the absence of input data [1].

For a single task execution, the timing analysis for instruction caches has been extensively researched [2,3]. But for data cache analysis, the majority of work has been focused on predictable memory access patterns in [4,5,6].

The unpredictable memory access and predictable memory access of a program have been defined in [1], *"An unpredictable memory access is a load or store access whose reference address is unknown during the estimation of the WCET. Conversely, a predictable memory access is a load or store access whose reference address is known during the estimation of the WCET".* An example program exchangesort is given in figure 1, which illustrates unpredictable memory accesses. The array (int a [ ]) to be sorted is the input to the program. Exchange of array elements for sorting is based on pos_min (line 14 and 15) variable which depends on input array a. (line 9 and 10). Array accesses to a[pos_min] in line 14 and line 15 become unpredictable at analysis time. But array accesses to a[i] in line 13 and line 14 are predictable at analysis time.

```
1 int main()
2 { int MAX 1215;
3   int a [1215];
4   int i,j,c, temp,pos_min;
5   for (i = 0; i < MAX; i++)
6   {         pos_min=i;
7             for (j = i+1; j < MAX; j++)
8             {
9                if (a[pos_min] > a[j])
10               {          pos_min = j;
11               }
12            }
13            temp = a[i];
14            a[i] = a[pos_min];
15            a[pos_min] = temp;
16  }
17 }
```

Figure 1. Example of input-dependent access (C source for Exchangesort)

We proposed a new architecture and an analysis framework, where data access is classified and analyzed separately so that the data cache timing behavior for programs with input dependent accesses can be accurately analyzed. In other words, our work classifies predictable and unpredictable accesses and allocates them into predictable caches and unpredictable caches respectively. For predictable accesses, we employ the CME framework [2] for the WCET analysis. For unpredictable accesses, we calculate array element reuse distance to examine the detailed unpredictable data cache behaviors, thus make the estimated WCET more accurate under this in-depth exploration. Our experimental results show that our data timing analysis in presence of unpredictable data accesses is very accurate with ignorable extra hardware complexity.

978-1-4244-2182-4/08/$25.00 ©2008 IEEE. 

The remainder of the paper is organized as follows. Section II introduces the related work on software timing analysis with cache modeling. Section III describes the principle behind the new architecture. Section IV describes the theory and workflow of our analysis framework. Section V presents our experimental results and analysis. Finally section VI shows conclusions and future research work.

## II. RELATED WORKS

To obtain an accurate WCET, past efforts have been made in two directions. One direction tries to develop new architectures that are more predictable. The other tries to develop new analysis approaches to obtain more accurate WCET results.

### *A. Predictable Cache Architectures*

Cache partitioning [8] is a mechanism developed to reserve blocks of cache for individual tasks such that cache hits becomes predictable. But this can significantly affect cache performance because of its fragmented address space.

Another approach—cache locking is to lock frequently used cache blocks [11]. Selected data is loaded into cache and locked in place so that it may not be replaced until the cache is explicitly unlocked. There could be performance loss if data is too big for the locked cache, then the whole cache must be unloaded it to be predictable.

Scratch-pad SRAM [12] also has been introduced to hold frequently used cache blocks to make it more predictable. But this is at significant area and power cost. This also requires compiler support for additional address space and context switching.

### *B. WCET Analysis for Data Caches*

Extensive researches have been performed on WCET analysis for programs with only predictable data accesses. Wolfe et al [13] proposed an integer linear problem (ILP) formulation and data flow analysis techniques on data cache analysis. Unknown data references are not considered and array ranges would have to be annotated by the user. Gosh et al [4] proposed CME framework, which computes re-use vectors to calculate cache accesses hit or miss within loops.

Few researches have analyzed input-dependent accesses but they either (1) restrict these input dependent accesses as un-cacheable and eliminate the interface between these accesses and the predictable data accesses or (2) assume the input dependent accesses spanning the whole data caches and analyze them as input independent accesses. These models simplify the analysis at the cost of WCET tightness and accuracy. For example, Lundqvist et al [1] introduced a symbolic simulation technique to classify predictable and unpredictable memory accesses, but unpredictable data structures are tagged as non-cacheable and consequently always require a cache miss. Ferdinand et al [14] introduced an abstract interpretation to predict data cache behavior. For an unpredictable array access, it assumes that all cache blocks of an array are accessed. Staschulat et al [15] introduced a theoretical framework to classify data accesses predictable and unpredictable in a single direct mapped data cache and perform cache analysis on the two patterns using ILP.

## III. CLASSIFIED CACHE ARCHIECTURE

Our novel classified cache architecture is proposed such that the interface between input dependent accesses and input independent accesses is eliminated. With compiler support, the data accesses are mapped into two classified caches: predictable cache and unpredictable cache. And data accesses are fed into the processor from the two caches independently assisted by annotating the load and store instructions. For example, when an annotated unpredictable load instruction 'ff' is executed; the processor would try to fetch the data in this instruction from the unpredictable data cache. The advantage is that there exists no interference between the input dependent data accesses and input independent data accesses because they don't compete each other for the same cache blocks.

This new architecture was implemented by extending the Sim-outorder Model in SimpleScalar to include two parallel level data caches. Unpredictable data accesses are put as annotations in the 64 bit PISA instruction set. Original benchmark source is converted to assembly and then manually annotated with unpredictable data access and converted to SimpleScalar executable to be run on the new architecture.

## IV. OUR ANALYSIS FRAMEWORK

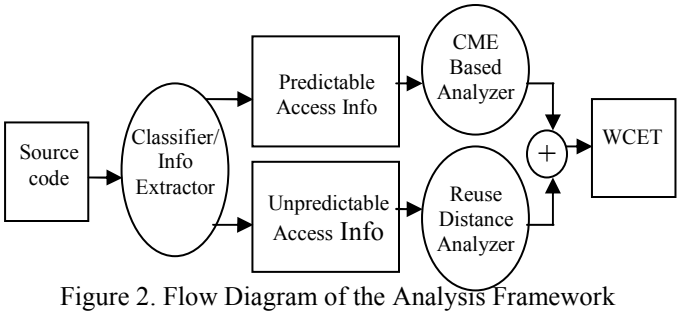


Figure 2. Flow Diagram of the Analysis Framework

Our analysis framework is show in Figure 2. From C source programs, the desired information for further analysis of both unpredictable and predictable memory accesses is extracted using the parser and extractor. Then CME framework [2] is used to analyze our predictable data cache behavior for predictable memory accesses. Our proposed reuse distance algorithm is then employed to deal with the unpredictable accesses for our predictable data cache. Finally, we combine results from these two independent parts to yield the overall cache misses and estimated WCET. The above steps will be introduced in the following subsections.

### *A. Classification and Reuse Distance Extraction*

Figure 3 is our algorithm for memory access classification into predictable or unpredictable access. By determining the input-dependency of array memory accesses, unpredictable memory accesses are identified. Then the unpredictable array element reference order can be extracted by simulating the program once, as though their respective address could not be decided at compile time, the relative order of unpredictable accesses is often determined, which is luckily true for typical programs. Then reuse distances of unpredictable array elements can be calculated for unpredictable data cache behavior analysis. The reuse distance of an array element is the number of unique unpredictable array elements referenced since the last access to that array element. We will elaborate its usage in later sections.

```
1  Memory accesses by scalar variables are predictable (as memory address is constant)
2   Memory accesses by predefined array accesses are classified as predictable.
3  Look at the index expression of other arrays
4  If all the variables in it are input independent
5  {
6    memory accesses by this array is predictable
7  }
```

Figure 3. Memory access classification algorithm

### B. WCET for Input Independent References: CME

CMEs platform [4] develops a miss equation providing the number of misses for each reference in a set of nested loops, and shows how to obtain the WCET for input independent references. To reduce computing complexity, X. Vera et al [6] proposed an analytical method based on CMEs to calculate cache misses efficiently using statistical sampling.

In our predictable memory access modeling, we gather related information about loops, variables and references in loops and build CME as follows. There are three steps to obtain the cache hit or miss information for every reference using CMEs. (1) Compute reuse vector: if a memory reference has the same memory line for two iteration points then the vector deduction of the later iteration point from the earlier one is a reuse vector. (2) Build CME: (a) Cold Miss Equations—build equations for iteration points where the reuse did not hold because reuse from point outside the iteration space or reuse from data that is mapped to a different cache lines (cache-aligned); (b) Replacement Miss Equations—build equations for iteration points where the reuse did not hold because multiple references (include self-interference) map to the same cache set. (3) Solve CMEs: to find the total number of misses of a loop nest by traversing the iteration space and check whether a point is solution or not of the equations.

### C. WCET for Input Dependent References

This section presents the reuse-distance and the algorithm that we use to obtain the cache timing behavior of input dependent memory accesses.

#### 1) Reuse Distance

The reuse distance is a metric for measuring program's cache behavior [5]. Further research explored the method of reducing the reuse distance and cache misses based on reuse distance visualization. It has been observed that in a fully associative LRU cache with $n$ cache lines, a reference will hit only if the reuse distance $d$ is smaller than $n$. Further, reuse distance may help identify the type of a cache miss as listed in table I.

TABLE I. CACHE MISS TYPES AND REUSE DISTANCE

| Miss Type | Relation between d and n |
|---|---|
| Conflict miss | d < n |
| Capacity miss | $n \le d \le \infty$ |
| Cold miss | $d = \infty$ |

#### 2) Timing Analysis Based on Reuse Distance

Based on the above findings in section IV C 1), we developed an algorithm for the timing analysis of input-dependent memory accesses. As illustrated in Figure 4, we keep a record of all the reuse distances of cache lines in the input dependent data cache at the current state. Each cache line's reuse distance is the minimum reuse distance of all data elements in this line. And when a new reference comes, we first check if this reference is already in the current data cache, if so we have a cache hit, otherwise it would be a cache miss and we need to decide if this data item would go into the input dependent data cache. We first check if its reuse distance is smaller than the maximum reuse distance of all the current cache lines. If true, the cache line containing the maximum reuse distance gets replaced out with the cache line that reference lies in., and the reuse distance of the cache is updated accordingly.

```
1  cache_miss=0;
2  while not end_of_file
3  {
4   get array element;
5   get reuse distance (r.d.);
6
7   if array element in cache
8   {
9     if array element r.d. < its cache line r.d.
10    {
11    its cache line r.d. = array element r.d.;
12    }
13  }
14  else
15  {
16  cache_miss++;
17     {
18        //replace out the cache line with largest reuse distance and update information
19       cache line (with largest r.d. in cache) = its cache line;
20       its cache line r.d. = array element r.d.;
21       cache r.d. = max(cache line r.d.);
22       }
23  }
24 }
```

Figure 4. Reuse-distance based algorithm

We illustrate this pseudo code using the following example memory access sequence: a[1], a[2], a[3], a[4], a[1], a[4]. For a fully associative cache with cache line size of 4 bytes, and n=3 cache sets, we compute first each data item's reuse distance and then output the miss condition for each reference table II.

TABLE II. REUSE DISTANCE TABLE EXAMPLE

| Reference | Reuse distance ($d$) | Hit/miss type |
|---|---|---|
| a[1] | ∞ | cold miss (cm) |
| a[2] | ∞ | Cm |
| a[3] | ∞ | Cm |
| a[4] | ∞ | Cm (replace a[1]) |
| a[1] | 3 | replacement miss |
| a[4] | 1 | Hit |

## V. EXPERIMENT AND RESULTS

Table III shows the benchmarks with its number of unpredictable data accesses per iteration and its number of

loop iterations. If the loop time is not easily determined, we describe both the instruction memory and data memory sizes.

To observe the necessary cache misses, the following sets of configurations were set. For exchangesort, small and quart benchmarks, we set the predictable data cache 4KB and the unpredictable data cache 4KB. For cover, the predictable data cache is 512B and the unpredictable data cache is 512B for a 1KB total. For count it is set as predictable: 4KB and unpredictable: 512B for a 4.5KB in total. For FFT and FIRfilter, the predictable and unpredictable caches are direct-mapped and fully associative respectively and each is 512B.

TABLE III. BENCHMARK DESCRIPTION

| Benchmark | No. of instructions with unpredictable accesses per iteration | No. of loop iterations/ Code/data size (bytes) |
|---|---|---|
| Exchangesort | 2 | 737505 |
| Smallexample | 1 | 300 |
| Quart | 1 | 1000 |
| Cover | 1 | 2000 |
| Count | 2 | 1000 |
| FFT | 8 | 1852(code)256(data) |
| FirFilter | 1 | 240(code) 80(data) |

We denote WCET as the time to access the two caches. The core execution time of the processor is not considered for our comparison. We define,

$$overestimation = \frac{t_{analysis} - t_{simulation}}{t_{simulation}} * 100\%$$

Where $t_{analysis}$ means the analysis WCET, $t_{simulation}$ refers the observed WCET from simulation using human observed worst case input data set.

TABLE IV. OUR CACHE ANALYZER RESULTS

| Benchmark | $t_{analysis}$ | $t_{sim}$ | Overestimation% |
|---|---|---|---|
| Exchangesort | 1597536 | 1546740 | 3.28 |
| Small | 8632 | 8074 | 6.91 |
| Quart | 18749 | 18065 | 3.79 |
| Cover | 18654 | 18609 | 0.24 |
| Count | 16045 | 16009 | 0.22 |
| FFT | 254027 | 205136 | 23.83 |
| FirFilter | 60793 | 56028 | 8.50 |

Table IV shows our experimental results, from which we can see that the classified analysis platform renders very tight WCET. Please note that the only exception one FFT, its main overestimation comes from the limitation of CME where the loops should be well designed (bounded iteration time, index expression is affine function of the variables) and it's improved a lot than previous data at around 40% overestimation.

To define our architecture performance, we define the deviation from perfect cache as:

$$\frac{\text{Our platform simulation-Chronos simulation}}{\text{Chronos simulation}} * 100\%$$

For exchangesort, the observed deviation is 0.621% compared to the Chronos architecture (which has no data cache miss) in simulation WCET. Our new architecture is nearly perfectly cache, close to no cache miss case in Chronos (no data cache but main memory access time is 1 cycle). The cache size is reasonably small (1KB instruction cache, 4KB predictable cache, 4KB unpredictable cache).

## VI. EXPERIMENT AND RESULTS

In this paper, we have proposed a new classified-cache architecture and its software timing analysis framework to accurately analyze the input dependent memory accesses. Experimental results show quite small WCET over-estimations ranging from 0.22% to 6.91% compared to our simulation results. In the future work, we would like to extend our framework to analyze the interaction between the data cache and other micro-architectural factors such as pipelining, branch prediction and instruction cache.

## REFERENCES


[1] T. Lundqvist and P. Stenstr¨om. A method to improve the estimated worst-case performance of data caching. In *Intl Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999.

[2] 1955 S. Malik and Y.-T. S. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[3] Xianfeng Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. *40th ACM/IEEE Design Automation Conference* (DAC), June 2003.

[4] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.

[5] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE RTAS*, pages 148–157, 2005.

[6] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2003.

[7] J. Liedtke, H. H¨artig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 9-11 1997.

[8] Mueller, F. 1995. Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools For Real-Time Systems* (La Jolla, California, United States). NY, 125-133

[9] I. Puaut and D. Decotigny. Low-complexity algorihtms for static cache locking in multitasking hard real-time systems. In *IEEE RTSS,* 2002.

[10] B. Lisper and X. Vera. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, 2003.

[11] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.

[12] Marwedel, P., Wehmeyer, L., Verma, M., Steinke, S., and Helmig, U. 2004. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair* (Yokohama, Japan, January 27 - 30, 2004).

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium,* Dec. 1996.

[14] C. Ferdinand and R. Wilhelm. On predicting data cache behaviour for real-time systems. In *ACM SIGPLAN Workshop 1998 on Languages, Compilers, and Tools for Embedded System*, 1998.

[15] Jan Staschulat, Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*. 06.