

Unified Cache Modeling for WCET Analysis and Layout Optimizations

Sudipta Chattopadhyay
National University of Singapore
sudiptac@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Abstract—Presence of instruction and data caches in processors create lack of predictability in execution timings. Hard real-time systems require absolute guarantees about execution time, and hence the timing effects of caches need to be modeled while estimating the Worst-case Execution Time (WCET) of a program. In this work, we consider the modeling of a generic cache architecture which is most common in commercial processors — separate instruction and data caches in the first level and a unified cache in the second level (which houses code as well as data). Our modeling is used to develop a timing analysis method built on top of the Chronos WCET analysis tool. Moreover we use our unified cache modeling to develop WCET-driven code and data layout optimizations — where the code and data layout are optimized *simultaneously* for reducing WCET.

Keywords-WCET Analysis, Cache Memories.

I. INTRODUCTION

Hard real-time systems require absolute guarantees on the execution time of software. For this purpose, accurate analysis methods for estimating the Worst-case Execution Time (WCET) of real-time embedded software have been developed over the past two decades. WCET analysis methods usually involve program path analysis (which determines infeasible paths in the program’s control flow graph), micro-architectural modeling (for accurate estimation of the execution time of the basic blocks) and finally WCET calculation often via Integer Linear Programming.

Micro-architectural modeling involves considering the timing effects of micro-architectural features which are common in modern processors. Over the past few decades, the increasing processor-memory gap in terms of performance have prompted computer architects to develop on-chip caches — where some redundant memory (over and above the main memory) is placed on-chip to capture recently accessed memory blocks. This obviates the need to access main memory for every memory access, thereby improving program performance.

While caches constitute a well-established solution in terms of improving program performance, their presence raises serious questions about program predictability. Caches are not visible to the programmer and cache management is not done at the programming / compiler level. Given a memory block which is accessed many times during a program execution, we need to precisely predict the number of times it will hit or miss in the cache. Indeed imprecise

(but conservative) hit/miss classification of memory blocks can produce a gross over-estimate in terms of the Worst-case Execution Time (WCET).

For safe and tight WCET analysis, precise analysis methods for predicting cache behavior have been developed in the past decade. Abstract interpretation based hit-miss classification methods have been proposed for instruction as well as data cache [1], [2], [3]. Recently, multi-level cache analysis methods have also been proposed [4]. However, each of the works consider only a fragment of the cache architecture — either a single instruction cache, or a single data cache, or two levels of instruction cache.

In real processors (such as Intel x86), the most common cache architecture is a multi level one. In the first level (L1), there are separate instruction and data caches. In the second level (L2), there is a unified cache which houses both instruction and data. Instruction accesses are looked up first in the L1 instruction cache, followed by the L2 unified cache, and finally in the main memory. Similarly, data accesses are looked up first in the L1 data cache, followed by the L2 unified cache and finally in the main memory. Existing cache modeling techniques for WCET analysis consider different fragments of the architecture, but not the entire architecture. In this work, we draw upon and enhance existing works to develop a comprehensive cache modeling framework covering *all* aspects of the two level cache architecture.

Technical Contributions: In terms of technical contributions, ours is the first work to model the timing effects of a L2 unified cache which houses instruction as well as data. Previous works had either considered instruction cache or data cache but not both. In this paper, we integrate the modeling of instruction and data accesses by modeling the timing effects of a unified (instruction + data) cache. Using our cache modeling, we can identify the different sources of WCET over-estimation in a multi-level cache architecture with instruction and data caches. Our cache modeling framework has been integrated into the open-source WCET analyzer Chronos [5]. Using our WCET analysis, we develop heuristics to perform *simultaneous* code and data layout optimizations which can help reduce the WCET estimate. Previous works on WCET-driven compiler optimizations have studied code layout separately without concern for data layout, and this is problematic in the presence of a unified

cache. Thus, we present a cache modeling framework which goes beyond existing approaches, integrate it into a state-of-the-art WCET analyzer, and use the analyzer results to guide novel WCET-driven layout optimizations.

II. RELATED WORK

Research on WCET analysis was initiated two decades ago. The early works [6], [7] analyzed the program source code without considering the timing effects of the underlying micro-architecture. In subsequent works, timing effects of micro-architectural features such as pipelines, caches and branch prediction have been studied.

Cache modeling for WCET analysis has been an active topic of research in the area. Most of the research in cache modeling consider a single level instruction cache. Early works [8] used Integer Linear Programming (ILP) for instruction cache modeling. However ILP-based methods do not scale well in terms of analysis time. Subsequently, abstract interpretation based methods have been developed for instruction cache modeling [1]. These works consider all incoming program flows for categorizing individual memory accesses as always hit, always miss or persistent (only the first access misses). Such an approach is more scalable and has been integrated into various WCET analyzers.

Static analysis of data cache timing effects have also been studied (*e.g.*, see [3], [9]). In particular, [3] adapts the abstract interpretation approach for data cache analysis. One of the major difficulties in data cache analysis is the fact that several executions of an instruction can access different data memory addresses and it is difficult to precisely predict the range of data memory addresses accessed by a particular instruction. A recent paper [2] addresses this concern somewhat by partial unrolling of loops.

Multi-level cache analysis has been studied in [10], and more recently in [4] which discusses timing anomalies permitted by previous approaches.

In summary, all existing works on cache modeling focus on either instruction cache or data cache, but not both. Moreover, for cache hierarchies, in most real processors the second-level cache is a unified cache which contains both instruction and data — an issue not considered in existing works. In our work, we build on existing works [1], [2], [4] to develop a WCET analyzer which considers separate instruction and data caches in the first level and a unified cache in the second level.

Finally, our work on WCET-directed layout optimizations extends the state-of-the-art in the area. WCET directed code layout optimizations have been studied in [11]. Recently [12] studies WCET-aware positioning of procedures. Once again, none of the works consider code and data layout together. Due to the presence of unified (code+data) caches in real-life processors, it is important to simultaneously optimize code and data layout — an issue addressed by this paper from a WCET-centric perspective.

III. ASSUMPTIONS

We consider a memory hierarchy containing L1 instruction cache, L1 data cache and a L2 unified D/I cache. For simplicity, all our examples and evaluation assume that the cache replacement policy is LRU and the write policy is write-through with allocate, although the proposed analysis is not tied to a specific cache replacement or write policy. We also assume the following.

- 1) A piece of information is searched in the level 2 cache if and only if a cache miss occurs in level 1 cache. Cache of level 1 is searched always.
- 2) Every time a cache miss occurs in level L cache, the entire cache line containing the missing piece of information is loaded in cache of level L .
- 3) There is a separation of address space for instruction and data. That is from the memory address alone, it can be verified whether it is the address of an instruction or the address of a data.
- 4) Effects of micro-architectural features such as out of order pipeline, branch prediction (in particular timing anomalies created by the interaction of cache with these other features) are disregarded.

Assumption 1 rules out architectures where cache levels are searched in parallel to speed up the search for a piece of information. Assumption 2 rules out architectures with exclusive caches. These two assumptions also appear in [4].

IV. ISSUES IN CACHE ANALYSIS

Cache modeling for our multi level cache architecture involves solving the following sub-problems. In the following, we describe each of these sub-problems briefly.

- 1) **Instruction cache analysis:** Several works have already proposed techniques for analyzing instruction caches. We use the abstract interpretation based approach as described in [1] for analyzing instruction caches. Instructions are classified as *all-hit*(AH), *all-miss*(AM), *persistence*(PS) or *not-classified*(NC) after carrying out three analysis namely *must*, *may* and *persistence* analysis. If an instruction is categorized as AH, it means that the instruction is always in cache whenever it is accessed. On the other hand if an instruction is categorized as AM, it means that the instruction is never in the cache whenever it is accessed. *Persistence* analysis is carried out for increasing precisions. An instruction is *persistent* if it is never evicted from the cache. If an instruction cannot be categorized as one of AH, AM or PS, it is categorized as NC. For details of the analysis, readers are referred to [1].
- 2) **Address Analysis:** Unlike instructions, data accesses in a program are unpredictable. Thus a separate analysis is needed to determine the range of addresses accessed by a *load/store* instruction.

Value set analysis [13] is a technique which combines both numeric and pointer analysis to get an over approximation of addresses accessed at each memory access site of a given executable. An *a-loc* abstraction is used to track the values in memory and registers. An *a-loc* is represented by a stride interval $s[a, b]$ where $s \in \mathbf{N}$; $a, b \in \mathbf{Z}$ and denotes the progression $[a, a + s, a + 2s, a + 3s, \dots, b_k = a + (n - 1)s]$ where b_k is the *largest* integer of the form $a + (n - 1)s$ such that $b_k \leq b$. This representation allows us to represent a set of addresses with finer granularity than a normal interval $[a, b]$. A normal interval $[a, b]$ includes all integers between a and b whereas a stride-interval includes integers only with a period of s . It is also very useful for data accesses in programs as this nicely represent array accesses inside a loop, the stride representing the width of a single array element. Update is performed on this abstract domain for each possible instruction in the ISA which may change any *a-loc* and a fixed-point is computed. The address analysis handles arrays and pointers. Details of the analysis have been published in [13].

3) **Data cache analysis:** Output of address analysis is used to analyze data cache and get a similar AH, AM, PS and NC categorization on data accesses in a program. So far *must* analysis for instruction cache is extended to handle data caches in [2]. *Persistence* analysis for data cache was introduced in [3] but no experimental results were presented. In our framework we have used both [2] and [3] for *must* and *persistence* data cache analysis respectively and propose an extension of *may* analysis for data caches which eventually leads us to get an analogous AH, AM, PS and NC categorization for data accesses to make instruction and data cache analysis more compatible. We describe our extension in detail in section VI-A.

4) **Cache access classification:** In presence of multi level cache hierarchy, a specified cache level may not be accessed at all. Thus the access categorization of a specified cache hierarchy must also be known. This categorization was first presented in [4]. For a given memory access r , CAC (Cache access classification) of a particular cache level L can be:

- a) *A*: This means that the cache level L will always be accessed. For example for cache level 1 this is always true.
- b) *N*: This means that the cache level L will never be accessed.
- c) *U*: This means the access of this cache level L cannot be determined statically for this memory access.

CAC of cache level L for memory reference r is

determined from the CAC and AH/AM/PS/NC categorization of r in cache level $L - 1$. For example consider a two level hierarchy with L1 instruction cache, L1 data cache and a unified L2 cache. It is clear that AH categorized instructions or data are never brought into unified cache, as it is never accessed. On the other hand AM categorized instructions or data are always brought into unified cache as it is always accessed. For the other two categorizations it is not sure whether the unified cache is accessed or not. Thus all possibilities must be explored for a *safe* solution. For further details readers are referred to [4].

5) **Unified cache analysis:** Unified cache contains both instruction and data. Thus an instruction may get evicted by some data access and vice versa. As instruction is fetched before any data reference made by the same, a unified cache analysis first updates the abstract cache state by bringing the memory block containing the instruction into a cache line. Further, if the instruction is a memory reference instruction (load or store) abstract cache state is updated by applying the corresponding data cache update function depending on the analysis type (*may*, *must* or *persistence*). We describe this analysis in detail in section VI-B.

The above modular approach is useful as the framework can be evolved by the introduction of more sophisticated approach for each module.

V. OVERVIEW OF OUR CACHE ANALYSIS

An overview of our cache analysis is shown in Figure 1. A separate address analysis, which predicts the range of data addresses accessed by each load/store instruction, is needed for data cache analysis. From the result of L1 instruction and data cache analysis, the “*Compute CAC*” block in the diagram computes the access criteria for the unified cache which is used by the final analysis to compute the *hit/miss* classification of data and instruction in the same.

All cache analysis results are used for the final WCET estimation (the hexagon marked with “*WCET computation*” in Figure 1). The WCET estimation follows the methods

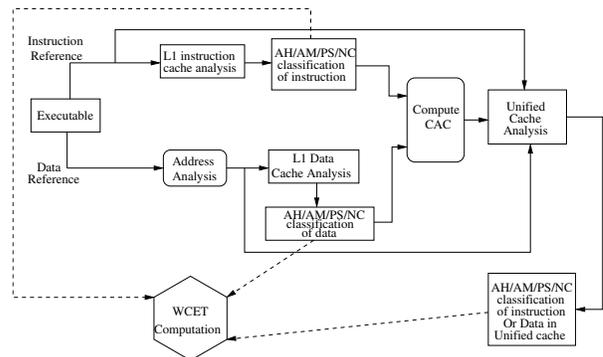


Figure 1. Overview of Cache Modeling Framework

in a state-of-the-art estimation tool like Chronos [5] — performing program flow analysis to find infeasible paths in the assembly level control flow graph, and using micro-architectural modeling (which includes cache analysis) to estimate the WCET of the individual basic blocks. Finally, the WCET of the basic blocks are pieced together to form the program’s WCET estimate via Integer Linear Programming.

To illustrate unified cache analysis consider the following code fragment and its corresponding assembly code targeting Simplescalar Portable Instruction Set Architecture (PISA).

```
int a[4][18];
for(j = 0; j < 4; j++)
  for(i = 0; i < 18; i++)
    a[j][i] = a[j][i] + 10;
```

```
00400208 addu    $4,$0,$0
00400210 addu    $3,$0,$5
00400218 lw     $2,0($3)
00400220 addiu   $4,$4,1
00400228 addiu   $2,$2,10
00400230 sw     $2,0($3)
00400238 addiu   $3,$3,4
00400240 slti   $2,$4,18
00400248 bne    $2,$0,00400218
00400250 addiu   $5,$5,72
00400258 addiu   $6,$6,1
00400260 slti   $2,$6,4
00400268 bne    $2,$0,00400208
```

Assuming that each integer takes 4 bytes to store and cache block size being 32 bytes, array a accesses nine memory blocks where a starts from the memory block boundary. Also assuming that each instruction take 8 bytes to store, the loop accesses four memory blocks for fetching instructions say $\{I_1, I_2, I_3, I_4\}$. Let $\{m_1, \dots, m_9\}$ be the memory blocks accessed by a . Assume a direct mapped L1 data cache and for the sake of illustration let us say m_1 maps to the same cache block as m_9 in L1 data cache. Apart from these, there are no other conflicts in data cache. Since access patterns are not considered, a persistence analysis on data cache cannot classify any of the accesses of the array a in the loop to be *persistent* which leads to adding the cache miss penalty for all $4 \times 18 \times 2 = 144$ array accesses in the source code. But in reality, only 3 memory blocks are accessed per outer loop iteration leading to a total miss count of 12.

Now consider the presence of a L2 unified cache (common in commercial processors such as Intel x86) whose size is four times bigger than the L1 data cache by increasing the associativity from 1 to 4. Increase of associativity is a reasonable assumption as cache associativity generally increases with hierarchy level. In this case, persistence analysis on unified cache will not evict m_1 for accessing the memory block m_9 . The instruction memory blocks $\{I_1, I_2, I_3, I_4\}$ being contiguous map to different cache sets. Moreover for a set-associative cache, one of these instruction memory blocks can co-exist with m_1 in a cache set of the unified cache. Thus, persistence analysis on the unified cache can declare all accesses in the loop to be *persistent* in the same. Since access to unified cache is much faster than accessing memory and access of array a is persistent in unified cache

in this example, overall estimate in WCET will have much tighter result. Persistence analysis of the data cache and unified cache results at the start of the loop are shown in Table I where l_{evict} represents cache blocks which may be evicted from the cache and l_i represents the usual cache blocks. The different rows in Table I represent different cache sets. For the above example, L1 data cache analysis encountered a cache *thrashing* scenario which leads to a much higher WCET than expected. The example also shows a scenario where the presence of unified cache does not make much difference in concrete execution but certainly analyzing the unified cache makes us predict a much tighter WCET estimate.

L1 Data Cache		Unified Cache				
l_0	l_{evict}	l_0	l_1	l_2	l_3	l_{evict}
\perp	$\{m_1, m_9\}$	\perp	$\{m_1, m_9\}$	I_1	\perp	\perp
m_2	ϕ	m_2	I_2	\perp	\perp	\perp
m_3	\perp	\perp	$\{m_3, I_3\}$	\perp	\perp	\perp
m_4	\perp	\perp	$\{m_4, I_4\}$	\perp	\perp	\perp
m_5	\perp	m_5	\perp	\perp	\perp	\perp
m_6	\perp	m_6	\perp	\perp	\perp	\perp
m_7	\perp	m_7	\perp	\perp	\perp	\perp
m_8	\perp	m_8	\perp	\perp	\perp	\perp

Table I
EXAMPLE OF PERSISTENCE ANALYSIS IN UNIFIED CACHE, \perp
REPRESENTS EMPTY CACHE LINE

On the other hand consider a very large loop which cannot entirely fit into the instruction cache. As a result in a concrete execution of the loop, cache *thrashing* will take place in presence of only instruction cache. But in presence of a unified cache in the memory hierarchy this problem may be resolved as unified cache is generally much larger than level 1 caches. For illustration suppose in the example I_1 and I_4 conflicts. Thus every iteration will encounter two instruction cache misses apart from cold misses. But presence of a unified cache will resolve this by getting the relevant instruction block from unified cache. From the analysis result in Table I also we can see that in the presence of a larger unified cache we can resolve this problem since all of the instructions have become *persistent* in unified cache. This constitutes an example where cache thrashing is avoided in *concrete* execution because of unified cache, and this will also be captured in the unified cache analysis.

VI. DETAILS OF CACHE ANALYSIS

For the rest of the discussion we consider a set associative cache with associativity A and with a set of cache lines $L = \{l_1, l_2, \dots, l_n\}$ in a single set. The memory store is considered as a set of memory blocks $S = \{s_1, s_2, \dots, s_m\}$. An abstract cache set is a mapping $\hat{d} : L \Rightarrow 2^S \cup \perp$ where each cache line corresponds to a set of memory blocks and \perp captures the situation where a cache line is empty. Let \hat{D} represents the set of all abstract cache states. To model

the LRU replacement policy it is assumed that the memory blocks in the cache set are ordered by increasing age.

A. Data Cache Analysis

The output of *address analysis* is used in data cache analysis. A key difference between instruction and data references is that the address set for the latter may not be a singleton set (for example consider array references). As long as a data reference accesses a single memory block, the *update* function for any data cache analysis remains same as that of instruction cache analysis; in this case it is definitely known which memory block is accessed and thus it can be brought to the abstract cache set. On the other hand, if number of memory blocks accessed is more than one, it is not definitely known which memory blocks are accessed in concrete execution as address analysis computes an over-approximation of the actual addresses accessed. Thus for our analysis to be safe, the *update* functions for different data cache analysis (*may*, *must* and *persistence*) become different. The *persistence* analysis for data cache is described in [3] although no experimental results were presented. *Must* analysis for data cache is introduced in [2]. For a precise analysis result of the unified cache in our architecture, we also need to perform *may* analysis on data cache. *May* data cache analysis classifies *all-miss* data references of a program. As memory blocks corresponding to *all-miss* data references in one cache level are always searched for in the next cache level, these memory blocks are potential candidates to be brought into the unified cache at level 2. We describe our proposed *may* analysis for data cache next.

May Analysis: As described before, when the accessed memory block is a singleton, the *update* function remains same as in the case of instruction cache analysis. But when number of accessed memory blocks is more than one, a *safe* update function for *may* analysis should satisfy the following two properties:

- 1) All memory blocks possibly accessed by the address set must be brought into the abstract cache set and have lowest possible age in the corresponding set.
- 2) Age of all memory blocks that are already in the abstract cache must be decreased to the lowest possible age.

Thus we use the following generalized *update* function for *may* data cache analysis:

$$\hat{U}_{may}(\hat{d}, M) = \sqcup_{m_i \in M} \hat{U}(\hat{d}, m_i) \quad (1)$$

Here M is the set of memory blocks accessed by the data reference, $\hat{U}_{may} : \hat{D} \times 2^S \rightarrow \hat{D}$ is the update function used for *may* data cache analysis, \hat{U} is the *update* function used in instruction cache analysis, \hat{d} is the current data cache set and \sqcup is the join operation used for *may* analysis. Informally, a *may* join operation is performed for each

possible memory blocks accessed by the reference. It is clear that this update operation satisfies both of the above specified conditions of *may* analysis. Join operation for *may* data cache analysis remains same as that of *may* analysis for instruction cache [1].

A particular data access at some program point p is classified as *all-miss*(AM) if the abstract cache contains none of the memory blocks accessed by it at p . Otherwise the data access is categorized as *not-classified*(NC).

Following example shows the difference of *must*, *may* and *persistence* analysis on data cache. Let an abstract cache set be as shown in the first row of Table II at some program point p . For a particular memory reference r , assume the address analysis module computes a range of addresses which corresponds to a set of memory blocks $M \subseteq S$. Let $\{s_x, s_y, s_z\} \subseteq M$ map to the same cache set whose abstract state is shown at row 1 of Table II. Abstract cache sets for *must*, *may* and *persistence* analysis after memory reference r are shown in subsequent rows.

state	l_1	l_2	l_3	l_4	l_{evict}
<i>Initial</i>	s_x	s_p	\perp	\perp	\perp
<i>Must</i>	\perp	\perp	s_x	s_p	\perp
<i>May</i>	$\{s_x, s_y, s_z\}$	s_p	\perp	\perp	\perp
<i>Persistence</i>	\perp	\perp	$\{s_x, s_y, s_z\}$	s_p	\perp

Table II
ILLUSTRATION OF DATA CACHE ANALYSIS, \perp REPRESENTS EMPTY CACHE LINE

B. Unified Cache Analysis

After *must*, *may* and *persistence* analysis of instruction and data cache we have AH/AM/PS/NC classification of each instruction in the program and additionally if the instruction is a load/store instruction we also have the same classification for its data access. Moreover for each instruction and data access we know whether the unified cache will be accessed or not. Since level 1 caches are always accessed, only the *hit/miss* criteria of instruction and data access will decide the access classifications of unified cache.

Let an abstract cache set of unified cache be a mapping $\hat{u}_1 : L \Rightarrow 2^S \cup \perp$ where each cache line corresponds to a set of memory blocks. Let $CAC_u(i)$ represents the CAC (cache access classification as described in IV) of instruction i in unified cache and additionally if the instruction is a memory load/store $CAC_u(d_i)$ represents the CAC of data access at instruction i in unified cache.

Informally, for any memory block accessed, it is checked whether the corresponding access in unified cache is an A (always) access or not. If the unified cache has a N (never) classification for the same access, no update is performed. In an U (unknown) classification of the access, a join operation is performed on previous two possibilities depending on the

kind of analysis (*may*, *must* or *persistence*) and access type (instruction or data).

It is also worth mentioning that for each instruction in the program, instruction is fetched from memory or cache first and if the instruction is a load/store instruction then the data is fetched from memory or cache subsequently. Thus when updating the unified cache, we always update it first with the memory block representing the instruction and then with all memory blocks representing the data access (if any).

Algorithm 1 describes the operations carried out for each instruction i in unified cache analysis. Given an input abstract cache set \hat{u}_1 it produces the output abstract cache set \hat{u}_f after the execution of instruction i .

Algorithm 1 Unified Cache Analysis. \hat{u}_1 is the input abstract cache state (for a cache set) and \hat{u}_f is the output abstract cache state (for the same set) after executing a given instruction i .

Let m_i be the memory block corresp. to instruction i

if ($CAC_u(i) = A$) **then**

$\hat{u}_m = \hat{U}(\hat{u}_1, m_i)$

else if ($CAC_u(i) = N$) **then**

$\hat{u}_m = \hat{u}_1$

else if ($CAC_u(i) = U$) **then**

$\hat{u}_m = \hat{U}(\hat{u}_1, m_i) \sqcup \hat{u}_1$

end if

if the instruction is not a load/store instruction **then**

$\hat{u}_f = \hat{u}_m$

return

end if

Let M is the set of data memory blocks accessed by instruction i

if ($CAC_u(d_i) = A$) **then**

$\hat{u}_f = \hat{U}_d(\hat{u}_m, M)$

else if ($CAC_u(d_i) = N$) **then**

$\hat{u}_f = \hat{u}_m$

else if ($CAC_u(d_i) = U$) **then**

$\hat{u}_f = \hat{U}_d(\hat{u}_m, M) \sqcup \hat{u}_m$

end if

In Algorithm 1, \hat{U} and \hat{U}_d represents the update functions used for instruction and data cache analysis respectively, and \sqcup denotes the join function. Note that the update and the join function depends on the type of analysis performed (*may*, *must*, *persistence*). For our purposes, we performed both *must* and *persistence* analysis on unified cache. This allows us to categorize certain code/data accesses as AH — Always Hit (via *must* analysis) or PS — Persistent (via *persistence* analysis), thereby tightening our WCET estimate. We now discuss the experimental results obtained from our analysis.

VII. ANALYSIS RESULTS

In this section we evaluate the accuracy and precision of our unified cache analysis. We have implemented the unified cache analysis inside the Chronos WCET analyzer framework [5]. To compare the overestimation of WCET we have taken four different cache configurations whose essential parameters are shown in Figure 2. For the rest of the discussion we shall use the abbreviations for cache configurations as shown in Figure 2. Our implementation has a 5 staged pipeline with in-order execution. Branch prediction is assumed to be perfect in all the experiments. L1 cache hit latency is 1 cycle and L1 cache miss penalty is 2 cycles. L2 cache miss penalty is 4 cycles. If there is no level 2 cache in the configuration, the cache miss penalty is taken to be 6 cycles. In Figure 2 all L1 caches have a block size of 32 bytes whereas all L2 caches have a block size of 64 bytes. For each of the cache configurations shown in Figure 2 we define two metrics, Sim and Est . Here Sim represents the observed WCET (in terms of cpu cycles) of a program and Est represents the WCET computed through static analysis (in terms of cpu cycles).

All experiments are run on a 3 GHz Pentium 4 machine having a 1 GB of RAM and running ubuntu Linux 8.10 operating system.

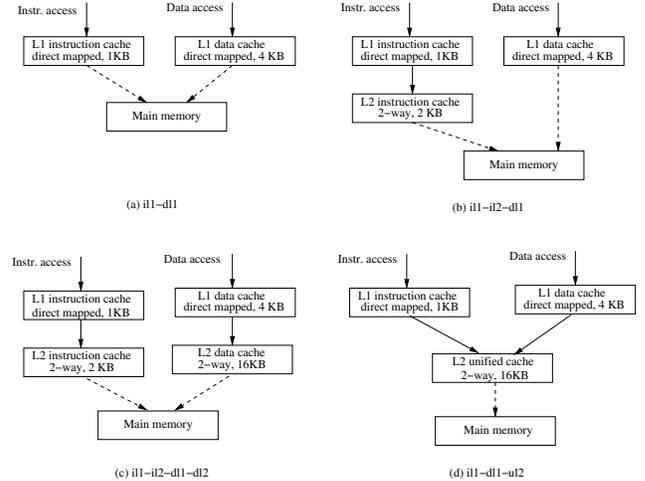


Figure 2. Different cache configurations used in experiments

A. Benchmarks

We have used benchmarks described in table III from [14]. To test the effect of unified cache we have taken benchmarks having different characteristics as follows.

- 1) Benchmarks having small/medium loop (in terms of codesize) but accessing large amount of data (e.g. matmult, cnt, ns).
- 2) Benchmarks having large loop (in terms of codesize), and accessing large amount of data (e.g. fft, edn).
- 3) Benchmarks having large loop (in terms of codesize) but accessing small amount of data (e.g. fdct).

4) Benchmarks having small loop (in terms of codesize) and accessing small amount of data (e.g. qurt, expint). We have benchmarks containing single as well as multiple paths. For example matmult, bsort100 are single path programs, whereas qurt,expint,fft have multiple paths.

B. Comparison of analysis precision

Table IV demonstrates the running time and accuracy of our analysis. Cache analysis time in Table IV corresponds to the total time taken for multi-level cache analysis excluding the time for address analysis (which is presented separately). The WCET overestimation ratio for different cache configurations are shown in Figure 3.

For data intensive programs (e.g. matmult, fir, ns), WCET estimates in presence of unified caches are much tighter than the WCET estimated in presence of only L1 data cache. Presence of a L2 data cache also reduces the WCET. For these data intensive programs, a large number of memory blocks whose *hit-miss* criteria were *not-classified* (NC) in L1 data cache, become *persistent* in unified cache or L2 data cache. We also observe that modeling only an L2 instruction cache together with the L1 instruction cache does not reduce the WCET significantly for these programs. The reason is that all loops of these programs can fit into the L1 instruction cache. Thus no cache *thrashing* happens in L1 instruction cache when executing the loop body. Only reduction in WCET estimation by modeling the L2 instruction cache may come through the higher block size of the same. We also observe that the estimate with unified cache and separated L2 data cache are almost the same. This signifies that there is little interference between instruction and data in the L2 unified cache.

On the other hand, benchmarks which have very large loops in terms of codesize (e.g. edn), modeling only an L2 instruction cache shows significant improvement in WCET. There is one loop in *edn* which cannot fit entirely in a 1 KB instruction cache. Thus in presence of an L2 instruction cache, all instructions in the loop which would have been evicted from L1 cache, become *persistent* in the L2 cache and reduces the overall WCET estimation. However there is a significant amount of data accesses in *edn*; some of which become *persistent* in presence of a unified cache and reducing the WCET estimation even more.

For benchmarks which have very small loop size (in terms of codesize) as well as access very small set of data (e.g. qurt, expint, bsort100), the WCET estimate cannot be reduced much by modeling any type of L2 caches. The reason is, all loops as well as accessed data memory blocks for these benchmarks can fit in L1 instruction and data caches respectively and thus getting no significant reduction in WCET in presence of L2 caches.

Finally, we observe that WCET estimates in presence of separate L2 instruction and data caches are almost same for all of the benchmarks except *edn* and *fft*. For these two

benchmarks, there is a significant amount of interference between instruction and data in the unified cache — an issue which we discuss next.

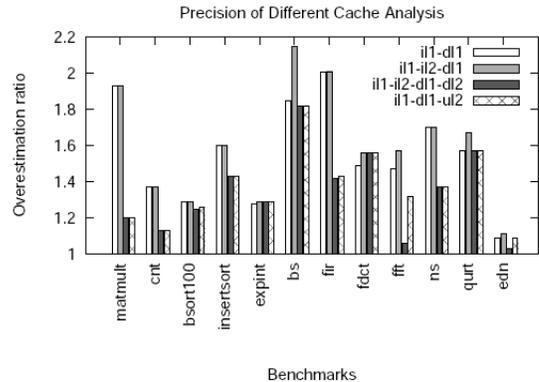


Figure 3. WCET overestimation for the cache configurations in Fig. 2

VIII. WCET-CENTRIC CODE AND DATA LAYOUT

In presence of unified caches, conflict misses may occur between instruction and data. Thus the WCET in presence of a L2 unified cache cannot be better than the WCET in presence of separate L2 data and L2 instruction caches of same size. However a unified cache reduces lot of storage cost. Thus if the code and data layout in the program are placed such that minimal conflict misses occur in unified cache, WCET/ACET of an application can be highly reduced.

Procedure positioning is a well known compiler optimization aiming at the improvement of instruction cache behaviour. A recent paper [12] has proposed procedure positioning optimizations driven by WCET information to effectively minimize the program’s worst case behaviour. However in presence of unified caches, this problem becomes more challenging as instruction may interfere with data and vice versa. Thus there is a need for simultaneous change of code and data layout for WCET reduction in presence of unified caches. We present here a fast, heuristic-based and unified cache aware algorithm for simultaneously changing the code and data layout to effectively reduce the WCET of a program.

A. Issues with WCET-centric procedure positioning in presence of unified cache

Current WCET-centric procedure positioning algorithm may not be helpful in presence of unified caches. For example let us consider the program below:

```
void f1()
{
    .....
    For(i = 0; i < N; i++) {
        if(...) f3();
        else {
            f2(); a[i] = a[i] + 10;
        }
    }
}
```

Table III
DESCRIPTION OF BENCHMARKS USED

Benchmark	Description	Bytes	LOC
matmult	Matrix multiplication of two 20 X 20 matrices	3737	163
cnt	Counts non-negative numbers in a 40 X 40 matrix	2880	267
bsort100	Bubblesort program	2779	128
insertsort	Insert sort a reverse array of size 10	3892	92
expint	Series expansion for computing an exponential integral function.	4288	157
bs	Binary search for the array of 30 integer elements.	4248	114
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample	11965	276
fdct	Fast Discrete Cosine Transform.	8863	239
fft	1024-point Fast Fourier Transform.	6244	135
ns	Search in a multi-dimensional array.	10436	535
qurt	Root computation of quadratic equations.	4998	166
edn	Implements the jpegdct algorithm together with other signal processing algorithms.	10563	285

Table IV
ACCURACY AND RUNNING TIME OF WCET ANALYSIS FOR THE DIFFERENT CACHE CONFIGURATIONS DESCRIBED IN FIG. 2.

Benchmark	i11-d11		i11-il2-d11		i11-il2-d11-dl2		i11-d11-ul2		Analysis time	
	Sim	Est	Sim	Est	Sim	Est	Sim	Est	Address Analysis	Cache Analysis
matmult	187056	360154	187000	360146	186985	224222	186985	224230	2.78	1.6
cnt	75278	103056	75232	103048	74432	83840	74432	83848	1.14	1.3
bsort100	2692	3471	2664	3463	2664	3347	2664	3347	1.01	1.1
insertsort	968	1509	936	1493	936	1341	936	1341	1.01	1.04
expint	2730	3491	2693	3491	2693	3487	2693	3487	1.05	1.22
bs	141	261	121	261	121	221	121	221	1.01	1.01
fir	348411	700311	348374	700303	348168	498027	348192	498051	1.01	1.3
fdct	2893	4300	2744	4276	2744	4276	2744	4276	0.01	1.2
fft	610117	900693	571309	900633	561984	596237	562345	745637	1.29	2.67
ns	7665	13053	7641	13053	7577	10409	7585	10409	1.01	1.08
qurt	1847	2910	1816	2902	1701	2846	1701	2846	0.2	1.6
edn	90730	99574	87945	98054	87945	91216	87945	96216	1.1	3

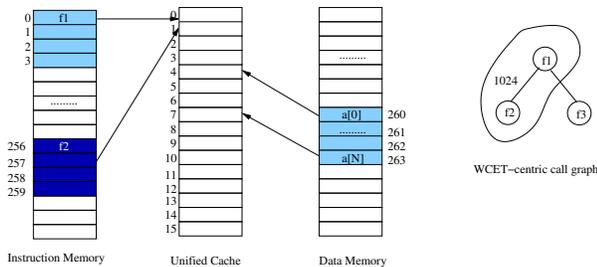


Figure 4. Code and data layout before procedure positioning

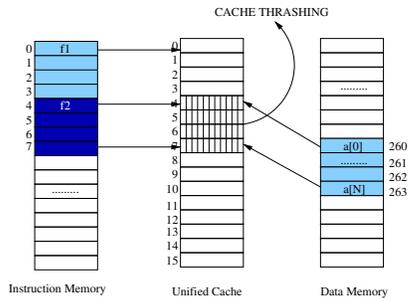


Figure 5. Code and data layout after procedure positioning by [12]

Instruction and data memory layout for procedures $f1$, $f2$ and array a are shown in Figure 4. For sake of illustration assume a direct mapped unified cache with 16 cache lines. Initially procedure $f1$ and $f2$ conflict each other in unified cache as shown in Figure 4. Array a maps to cache lines 4 to 7. Because of the initial layout of $f1$ and $f2$, whenever $f2$ is called memory blocks corresponding to procedure $f1$ got evicted and when control comes back to $f1$ again, memory blocks corresponding to $f2$ are replaced eventually leading to a poor WCET estimate. Thus a WCET-centric procedure positioning algorithm as described in [12] reposition the procedures from a “WCET-centric call graph” so that most frequently called procedures are placed in contiguous memory locations. The WCET-centric call graph of a program¹ is computed by a WCET analyzer and is invariant of all program executions. Call frequency of each edge is computed by the WCET analyzer and not by profiling. The WCET-centric call graph of the example program is shown in Figure 4. The marked portion of the

¹The nodes of a call graph denote procedures, and edges denote calling relationships. The edges are typically weighted with call frequencies.

call graph corresponds to the worst case path and edges corresponding to the worst case path is labeled with call frequencies computed by the WCET analyzer. The layout after procedure positioning is shown in Figure 5. It is clear that without having the knowledge of where array a was mapped, procedure positioning in a unified cache leads to a layout which may encounter cache thrashing scenario as shown in Figure 5. This gives us the motivation to change the layout of instruction and data simultaneously in presence of unified cache.

B. Simultaneous procedure and data positioning

WCET-centric unified graph: We define a unified undirected graph $G_{uni} = (V, E)$ which is an extension to the call graph. We have $V = P \cup R$ where P is the set of nodes corresponding to all procedures and R is the set of nodes corresponding to all data references in the program. There is an edge $e \in E$ between $p_1 \in P$ and $p_2 \in P$ if p_1 calls p_2 . Similarly there is an edge $e \in E$ between $p_1 \in P$ and $r_1 \in R$ if r_1 is inside procedure p_1 . An edge $e \in E$ can be between two data references r_1 and r_2 if and only if $references(r_1) \cap references(r_2) \neq \phi$ i.e. two data references access some common memory blocks. Execution frequencies of all edges of our unified graph are computed from the WCET analyzer. Clearly edges between two data reference nodes do not have any associated frequency, they are drawn only to capture the overlapping memory access behavior. The unified graph for the example in Figure 4 appears in Figure 6 and the worst case path is marked. Edges belonging to the worst case path are labeled with execution frequencies. Data references $r1$ and $r2$ represent two references to array a (one for load and another for store).

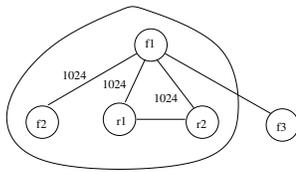


Figure 6. WCET-centric unified graph

Algorithm: Our technique for simultaneous data and procedure positioning is described in Algorithm 2. In the algorithm, the $maxEdge$ function selects an edge $e \in E$ labeled with highest execution frequency at each step. If any node representing the edge (returned by end_1 and end_2 functions) is a data reference, then all overlapping data reference nodes with the current one are first merged together by $collapseData$ function to form a super-node. Subsequently the $collapseEdge$ function collapses the edge selected in that step to form a single node. This function also modifies all related execution frequencies. If none of the ends of a selected edge is a data reference node, only

the $collapseEdge$ function is called to form a super-node as overlapping data references (if any) are already captured in the existing nodes. The algorithm terminates when no edges are left in the WCET-centric unified graph. After the graph has been merged to a single node, the layout is computed assuming the presence of a single unified memory and shifting to other memory such that mapping to the unified cache line is preserved.

Algorithm 2 Simultaneous code and data positioning. G_{uni} is the unified graph and R is the set of nodes in G_{uni} which represent data references.

```

repeat
   $e = maxEdge(G_{uni});$ 
  if ( $e = \phi$ ) then
    return;
  end if
  if ( $end_1(e) \in R$  or  $end_2(e) \in R$ ) then
     $collapseData(e);$ 
  end if
   $collapseEdge(e);$ 
until false

```

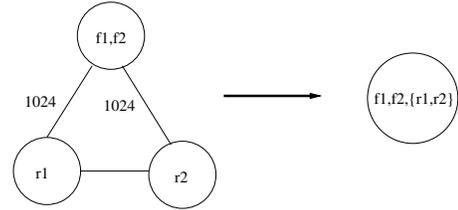


Figure 7. Transforming the unified graph of Figure 6

The collapsing of the unified graph in Figure 6 is depicted in Figure 7. For our example, $f1$ and $f2$ are allocated contiguously in instruction memory. Memory blocks corresponding to $r1$ and $r2$ are allocated in data memory such that they will map to the same cache line as if it would have been allocated contiguously after $f1$ and $f2$ assuming a hypothetical unified memory. The layout produced by our method is shown in Figure 8. As pointed out earlier, our algorithm is unified cache aware (i.e., it assumes the knowledge of unified cache and also assumes the start of instruction and data memory).

C. Experiments

To evaluate our heuristic, we compared the procedure positioning method of [12] with our unified code and data layout method. We chose the two benchmarks in our benchmark-suite which have large codesize as well as manipulate large amounts of data. These two benchmarks are fft and edn . We measure the amount of WCET reduction due to the procedure positioning method of [12] as well as

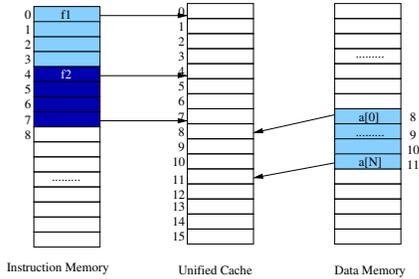


Figure 8. Final layout after our code + data positioning

Table V
REDUCTION IN WCET ESTIMATES VIA CHANGE IN LAYOUT

Benchmark	Est_{ul_2}	Est_p	Est_{p+d}
edn	96216	97240	93108
fft	745637	745638	608418

our unified code and data layout method for these two benchmarks. A multi-level cache architecture with a L2 unified cache is assumed (see Fig. 2(d)). The results appear in Table V. Est_{ul_2} represents the WCET estimate (in presence of a unified L2 cache) assuming a default layout for code/data (e.g. code is laid out as it appears in the program). Est_p denotes the WCET estimate using the procedure positioning of [12] and Est_{p+d} captures the WCET estimate using our simultaneous procedure/data positioning.

As expected, Table V shows that procedure positioning heuristic of [12] alone is not useful in presence of unified caches. But by simultaneous data and procedure positioning we are able to reduce the WCET estimate by 3% for *edn* and almost 18% for *fft*. For *fft* benchmark we observe that using unified caches increases the WCET by almost 0.15 million cycles. *fft* has fairly large loop structures (in terms of codesize) and it is a data intensive program. Improper data and instruction layout causes large number of conflict misses in the unified cache — which we avoid via our unified code and data layout.

IX. CONCLUDING REMARKS

In this paper, we have developed a cache modeling framework for Worst-case Execution Time (WCET) analysis of real-time embedded software. Our framework considers a generic multi-level cache architecture with separated instruction and data caches in the first level and a unified (code+data) cache in the second level. Unified cache is the most common in commercial processors such as Intel x86 and ARM. Existing works on cache modeling have so far considered either instruction or data caches but not both. Our experiments indicate that our analysis of the multi-level unified cache architecture produces tight WCET estimates with low running time overheads.

We also exploit our WCET analysis of the unified cache to build WCET-centric compiler optimizations. In particular,

we develop a joint (code + data) layout heuristic which leads to higher WCET reduction in presence of a unified cache as compared to existing WCET-centric code positioning methods.

ACKNOWLEDGMENTS

This work was partially supported by a NUS University Research Council grant R252-000-321-112.

REFERENCES

- [1] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18:157–179, 2000.
- [2] R. Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.
- [3] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES*, 1998.
- [4] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.
- [5] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007. <http://www.comp.nus.edu.sg/~rembed/chronos>.
- [6] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, April 1989.
- [7] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Transactions on Computers*, 24(5), 1991.
- [8] Y-T.S. Li, S.Malik, and A.Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.
- [9] 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*, 1997.
- [10] F. Mueller. Timing predictions for multi-level caches. In *LCTES*, 1997.
- [11] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *RTSS*, 2004.
- [12] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven cache-based procedure positioning optimizations. In *ECRTS*, 2008.
- [13] G. Balakrishnan and T.W. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
- [14] WCET Benchmarks. <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>.