Timing Predictions for Multi-Level Caches

Frank Mueller
Humboldt-Universität zu Berlin, Institut für Informatik, 10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de *phone: (+49) (30) 20181-276* *fax: -280*

Abstract

This paper describes a framework to extend timing predictions of real-time applications from architectures with primary caches to architectures with multi-level caches. First, general problems of extending timing analysis frameworks to multi-level memory hierarchies are discussed, including a method for dealing with unified caches. Second, an existing data-flow framework for cache analysis, the framework of static cache simulation, is extended to handle arbitrary memory levels. The cache behavior is described for each instruction and loop level by categories. The categorization of one cache level is fed back into another pass of static cache simulation to yield categories for the next cache level. Two data-flow models for implementing such a feed-back framework are introduced and contrasted. In general, the approach of feed-back analysis separates cache simulation for each memory level, which provides a modular approach to timing predictions for multi-level caches and should be quite efficient. Overall, the extensions of timing analysis to unified caches and multi-level memory hierarchies are the first attempt to address these issues, to our knowledge. They provide another step towards bounding execution times for modern architectures and to use these bounds in schedulability analysis for real-time systems.

1 Introduction

Static timing analysis provides a reliable foundation to bound the execution time of real-time tasks. The resulting worst-case and best-case execution times can then be used to determine if a task's deadline can always be met or if a task set is schedulable using a certain scheduling paradigm, *e.g* rate-monotone analysis [11]. Several approaches for timing analysis have been developed to simulate the effect of architectural aspects such as pipelining and caching. In one approach, the caching behavior is determined via a data-flow model. In another approach, a source-level oriented timing schema is extended. In a third approach, constraint-based methods, such as integer linear programming, are employed to model the effect of caching. All these models can be extended to handle multi-level caching as follows.

Once the cache behavior of the primary cache has been determined, it serves as an input to another modeling effort that addresses the secondary cache on the fly (during the analysis of the current path). We call this *integrated analysis* since every memory layer is integrated into the model implemented by data-flow analysis, timing schemas, or constraints. With this approach, additional overhead due to multiple levels of the memory hierarchy is incurred to the analysis.

A *separate analysis* of each memory level can be performed by first analyzing the behavior of primary caches of an entire program. This information is consequently used as an input to the analysis of secondary caches, etc. This approach has several advantages. First, there is a clean separation between the modeling of each memory level, which facilitates the description of an architecture for the user. Second, each memory level is analyzed in isolation and does not interact with other architectural aspects during the analysis, which may reduce the overhead of analysis. Third, existing tools can be reused to analyze each memory level.

The separate analysis is in particular suitable for the approach of static cache simulation. Static cache simulation analyzes the primary cache and produces a description of the cache behavior as an output. This output can be fed back as an input into a second pass of static cache analysis to yield the caching behavior for secondary caches. The feed-back process can be reiterated for each memory level until main memory is being accessed. A schematic description of this approach is depicted in Figure 1.

An optimizing compiler has been modified to emit control-flow information, data information, and the calling structure of functions in addition to regular object code generation. A static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce categorizations, which describe the caching behavior of each memory reference, respectively [3, 12]. The categorizations can be fed back into the static cache simulator for subsequent memory levels. The timing analyzer uses the categorizations of all memory levels and the control-flow information to perform path analysis of the program. This analysis includes the evaluation of architec-

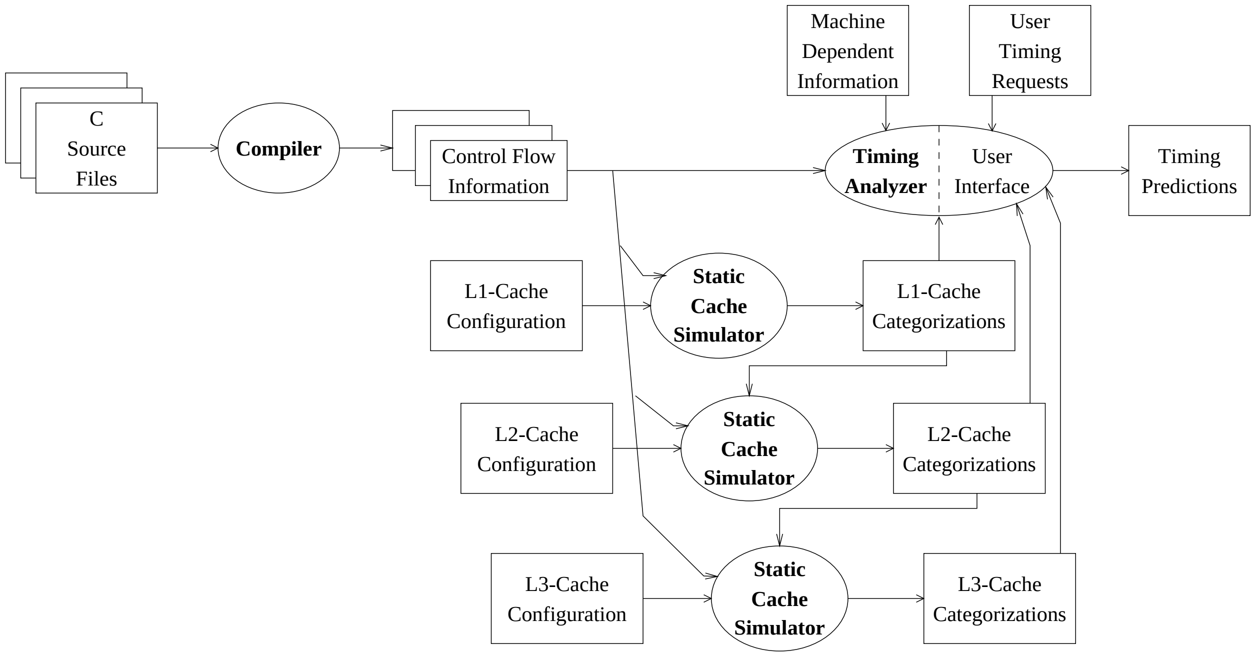

Figure 1: Framework for Timing Predictions

tural characteristics such as pipelining [5]. The description of the caching behavior supplied by the static cache simulator is used by the timing analyzer to predict the temporal effect of cache hits and misses overlapped with the temporal behavior of pipelining. The timing analyzer produces WCET predictions for portions of the program or the entire program, depending on user requests.

2 Unified Caches

References to instructions and data are both subject to caching at various levels. Primary caches are often designed as split caches with a separate cache memory for instruction and data references. Caches beyond the primary level, however, generally follow a unified design where data and instruction references share one cache memory space. The content of a unified cache at any time is thus determined by the temporal ordering of memory references. This ordering may not always be easily determined.

One problem in merging instruction and data references is that memory accesses occur at different times during instruction execution. For a pipelined architecture, instruction references occur at the instruction fetch (IF) stage while data references occur a few cycles later at the memory access (MEM) stage. Another problem is presented by instructions that take multiple cycles to be executed. These instructions prevent us from simply moving data references a fixed number of instructions forward in the control flow. This fixed number would be determined by the difference in cycles between IF and MEM stages. Unfortunately, this difference varies, for example, when an instruction spends multiple cycles in the execution (EX) stage. Finally, structural hazards, such as a limit on the number of memory ports, may cause stalls in the pipeline.

Consider the example given in Table 1 depicting the execution of five instructions for each pipeline stage. Stages ID and WB describe instruction decoding and write back to registers, respectively [6]. A load instruction stalls the pipeline due to a cache miss for three cycles. The loaded value is cached after three more instructions are fetched but before the fourth instruction (OR) will be fetched. Several variations on this example are possible. Had there been only one memory port, the fetch of the AND instruction would have stalled as well. Had the fetch of the SLL instruction been a miss, then the data access would have occurred immediately after the SLL instruction fetch.

LD	IF	ID	EX	MEM	MEM	MEM	MEM	WB	
SLL		IF	ID	EX	–	–	–	MEM	WB
SLR			IF	ID	–	–	–	EX	...
AND				IF	–	–	–	ID	...
OR					–	–	–	IF	...

Table 1: Pipeline Stages with Data Miss

The example illustrates a number of properties for the temporal ordering of data references relative to instruction references.

- Instructions executed prior to a data reference do not affect the temporal ordering.

- In the absence of instruction-level parallelism (ILP), the temporal ordering is bound by the distance δ between instruction fetch and memory access stages within the pipeline.

- The temporal ordering is furthermore bound by the next data reference for simple architectures. If, on the other hand, reads may occur concurrent to writes, the bound would be the next reference of the same kind (read or write, respectively).

- With ILP, instructions may proceed to execute while a memory access is in progress. In this case, the temporal ordering of a data reference is bound by [δ plus the maximum miss penalty of any cache (typically the RAM access time)] times the ILP level, *i.e.* the number of instructions that can potentially be executed in parallel.

- Branches provide a bound. In particular in conjunction with ILP, this bound may be tighter than the bound described in the last item.

- For speculative execution, the depth of the speculative pipeline needs to be included in the bound to extend it past branches.

The influence of data references on the cache behavior can be limited within the minimum of the above bounds, which provides a window within which the data reference may take effect. A conservative estimate of the cache behavior now considers other memory references within this windows. If any of these references conflict with the data reference at a cache level, then both references are considered to be misses for the worst-case estimation. Conversely, they are considered hits for the best case. This approach still allows a separation of cache behavior from the pipeline simulation, even for unified caches and multiple cache hierarchies. The following discussion is based on the assumption that unified caches are dealt with as just described.

3 Data-Flow Analysis

Data-flow analysis (DFA) provides a well-defined framework for analyzing properties of programs statically. Information generally propagates along the control-flow paths of a program. This information may serve a multitude of goals ranging from live variable analysis in optimizing compilers [1] to a description of the cache behavior of programs as outlined in the last section. The latter application allows us to calculate safe upper and lower bounds of a program's execution time, which is the main concern in this paper.

DFA has been discussed as a method to perform static cache simulation for primary caches. Using the information about cache behavior, the worst-case and best-case execution times are calculated via timing analysis. In the following, we discuss two approaches to extend timing analysis to multi-level memory hierarchies, such as secondary caches (or even 3rd level caches) found in today's architectures.

3.1 Cache Behavior by Categorization

The description of the cache behavior for static cache simulation distinguishes the categories in Table 2 for a memory access. For example, an access within a loop that results in a cache miss on the first iteration of the loop but results in cache hits on consecutive iterations is called a first-miss. Each memory access receives a categorization for each nesting level, where functions are treated as loops with one iteration.

Category	First Access	Consecutive Accesses
Always-Hit	Hit	Hit
Always-Miss	Miss	Miss
First-Hit	Hit	Miss
First-Miss	Miss	Hit

Table 2: Cache Behavior for Memory References

For example, a reference to memory location X nested in two loops in `main` may be described as

Location	loop L2	loop L1	main
X:	first-miss,	always-miss,	always-miss

where the inner-most cache behavior is listed first. This means that an access to X results in a miss at the outer loop (the 2nd categorization in the list). For the inner loop (first categorization), there will only be a miss during the first iteration, *i.e.* whenever the inner loop is entered. Consecutive iterations of the inner loop result in hits for X.

When this categorization is produced as an output of the static cache simulator to describe the cache behavior, the question arises how to use this information as an input to the analysis of the next memory level. An always-hit does not need to be analyzed by the next level and can therefore be omitted. An always-miss can simply be handled at the next level as a memory access on each iteration. However, first-hits and first-misses describe a non-uniform cache behavior of an access. There are several options to handle these two categories. Both categories may be interpreted as always-misses for the worst-case analysis (and, conversely, always-hits for the best-case analysis), termed the *conventional* approach. On the other extreme, each loop may be represented as different data-flow information with sets for the first and for consecutive iterations to allow special treatment for first-hits and first-misses. When loops are nested, all DFA information for the inner nesting has to be duplicated to represent first and consecutive iterations at the outer level. We term this the *exponential* approach due to the resulting space overhead.

3.2 The Conventional DFA Approach

In the conventional approach, there is a constant space overhead for DFA with respect to the loop nesting. In particular, each data-flow block has an input state and an output state to support DFA. The cache behavior of each memory reference is determined based on these states. In other words, the input to static cache simulation are the memory references and the output is a categorization of each memory reference describing the cache behavior according to Table 2.

The input to the simulation of the primary cache consists of all memory references within the program (in conjunction with the control flow), where each reference can be regarded as an always-miss (with respect to an imaginary previous memory level). When static cache simulation is performed with feed-back information, consecutive simulations for secondary caches are confronted with any of the four categories (possibly differing for nesting levels) for a memory access. These categories are handled as depicted in Table 3.

Output(N-1)	Best-C. Input(N)	Worst-C. Input(N)
Always-Hit	—	—
Always-Miss	Always-Miss	Always-Miss
First-Hit	—	Always-Miss
First-Miss	—	Always-Miss

Table 3: Conventional Mapping Output(Level N-1) → Input(Level N)

Always-hits as output are never propagated to the next level since such accesses are already resolved at the current level, as discussed before. For the worst case, all other categorizations, regardless of the nesting level, are simply treated as an always-miss. For the best case, only an always-miss is propagated as an always-miss to the next level while first-hit and first-miss are not propagated. The above mapping is conservative in the sense that it does not overestimate the worst case or underestimate the best case.

It may appear that the above mapping can loose information and thus may not guarantee tight timing predictions anymore. This concern is countered by the following protocol. In addition to the above mapping, a categorization of the previous level is adjusted for the next level such that it does not become less tight as depicted in Table 4.

Output(N-1)	Best-C. Output(N)	Worst-C. Output(N)
Always-Hit	—	—
Always-Miss	Any	Any
First-Hit	F-Hit / A-Miss	F-Hit / A-Hit
First-Miss	F-Miss / A-Miss	F-Miss / A-Hit

Table 4: Adjustment Output(Level N-1) → Output(Level N)

For example, a first-miss may either remain a first-miss or become an always-hit for the worst-case analysis. The transition to an always-hit tightens the bound for the worst-case analysis and can be derived from data-flow information since first-misses were handled as always-misses (see Table 3). Conversely, a first-miss may remain a first-miss or become an always-miss for the best-case analysis, where the latter would be derived from the data-flow information.

3.3 The Exponential DFA Approach

The exponential approach takes the categorizations of a previous memory level and simulates the specified access behavior for each loop nesting at the next memory level. The distinction between loop nestings eliminates the necessity for a remapping function required by the conventional approach.

Consider the following example of three nested loops depicted in the left portion of Figure 2. Assume an access to location Y for the previous memory level described as

Location	loop L3	loop L2	loop L1
Y:	first-miss,	first-miss,	always-miss

where any first access in the middle loop is a miss while any other accesses are hits. Then, first and consecutive accesses have to be distinguished for different loop nestings. The adjacent data-flow graph in Figure 2 (on the right side) represents the data-flow sets for loop nestings and iterations. Notice that the resulting data flow would be equivalent to a transitive single unrolling of each loop body. However, instead of modifying the control flow by unrolling the actual code, unrolling is solemnly performed at the analytical (logical) level of the data flow.

In particular, Block C has 8 instances. The odd instances refer to the first iteration of the corresponding loop L3, the even instances to consecutive iterations, respectively. Instances C1 and C2 cover the data flow of L3 within the first iteration of both L1 and L2. Instances C3 and C4 reflect consecutive iterations within L2 that are still within the first iteration of L1. C5 and C6 represent the data flow of L3 for consecutive iterations of L1 within the first invocation of L2. And C7 and C8 cover consecutive iterations for both L1 and L2.

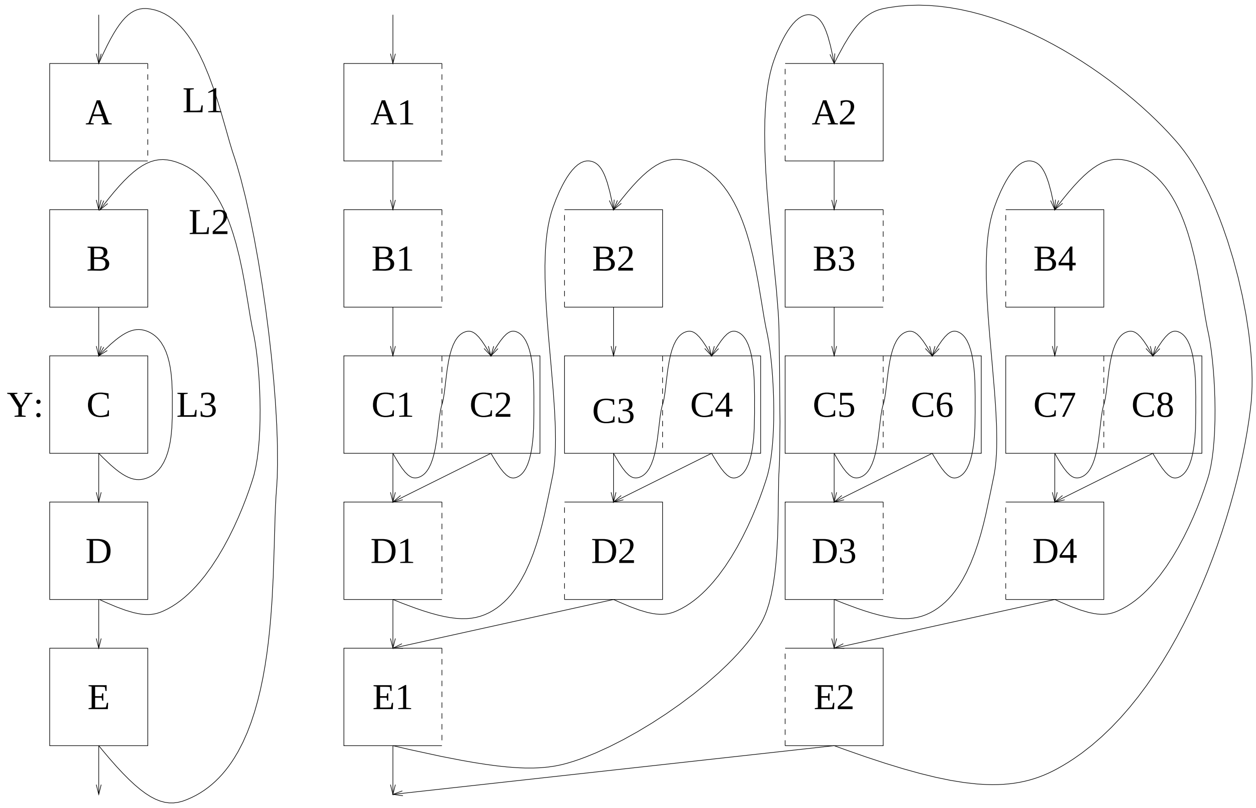

Figure 2: Data-Flow Model for Exponential Approach for 3 Nested Loops

Given the above categorization, we can now simulate the cache accesses as depicted in row 2 of Table 5. (Ignore row 3 for now.) Recall that there was a miss during each iteration of L1 within the first iteration of both L2 and L3 (reflected in C1 and C5) while all other accesses were hits and need not be propagated to the next memory level. During DFA, the access mode in the table indicates which action needs to be performed, *i.e.* only if an access to the memory location is indicated, will the corresponding memory block be cached at this point within the DFA.

Instance	C1	C2	C3	C4	C5	C6	C7	C8
Access	yes	no	no	no	yes	no	no	no
Behavior	miss	hit	hit	hit	hit	hit	hit	hit

Table 5: Cache Behavior interpreted for Location Y

In general, the categorizations of one memory level can be interpreted in terms of accesses at the next level as follows:

- Always-hits do not cause any access at the next level, as seen before.
- Always-misses at all nestings cause an access at the next level for all iterations and nestings.
- A first-miss at the deepest nesting causes an access for the first iteration of the loop hierarchy, *i.e.* for the first iteration of the innermost loop within the first iteration of the next outer loop etc. and the first iteration of the outermost loop. (This corresponds to C1 in the example).
- Given a first-miss for nesting m and an always-miss for nesting n such that $m > n$, an access is caused for the first iteration of the loop at nesting n. (This corresponds to C5 in the example.)
- Given a first-hit at any nesting, an access occurs for consecutive iterations at all levels (all even instances in the example).
- Given an always-miss for nesting m and a first-hit for nesting $m + 1$, an access is caused for the first iteration of loops with nesting $\geq m$, except for first iteration of the innermost loop within the first iteration of all outer loops (odd instances i such that $max.nesting > i \geq m$).

DFA is performed as depicted in the sample control flow in Figure 2 for nested loops. Multiple outgoing arcs, so-called split points, represent the copy operation where an output state is copied into multiple input states of the successor blocks. Multiple incoming arcs, so-call join points, represent the join operation of the predecessors' output states to form an input state. The join operation may depend on the application of DFA. We may apply the union operation at joins as we did with the conventional approach when using the static cache simulation framework. The algorithmic details of the DFA approach for cache analysis are described elsewhere [3, 12]. The exponential approach only uses a different data flow, as described above.

The DFA now yields different data-flow information for each loop level and first / consecutive iterations. This information is now interpreted in five steps to yield categorizations that describe the cache behavior of the next memory level.

1. In the first step, the relevant instances for the first iteration of a loop are collected, which are all instances i, where

$$1 \leq i \leq 2^{\texttt{max\_nesting}} \wedge$$
$$(i-1) \texttt{ mod } 2^{(\texttt{max\_nesting}-\texttt{nesting(loop)}+1)} = 0.$$

2. In the second step, the relevant instances of consecutive iterations are determined to be the instances relevant for the first iteration of the next inner loop, except for the instances determined to be relevant in the first step.

3. In the third step, different categorizations of the relevant instances of step 1 are combined. For the worst case, they are combined to a miss if any relevant instance is a miss and, conversely, they are combined to a hit if any relevant instance is a hit for the best case.

4. In the fourth step, different categorizations of relevant instances of step 2 are combined using the same combinator function as in step 3.

5. In the fifth step, the calculated behavior for the first iterations (from step 3) and consecutive iterations (from step 4) are combined according to Table 2.

Thus, a category is determined for each loop level to describe the cache behavior. For example, the cache behavior of location Y in Figure 2 would be described as follows. For the outermost loop, C1 determines the first and C5 consecutive iterations of the loop, respectively. For the middle loop, C1 and C5 determine the first iteration while C3 and C7 determine consecutive ones. For the inner loop, the odd instances determine the first iterations and the even instances are combined for consecutive iterations. In particular, if we assume the cache behavior for each instance depicted in the third row of Table 5 and interpret the information according to the five steps above, then we get the categorizations of each loop nesting depicted in the Table 6.

Consider loop L2 in the example. The worst-case behavior for the first iteration is the worst behavior of C1 and C5, which is a miss. C3 and C7 describe consecutive iterations and are combined to a hit since both were hits. Thus, L2 has a miss on the first iteration and hits thereafter, which implies a first-miss category for the worst case. Conversely, C1 and C5 are combined to a hit for the best case; C3 and C7 also yield a hit. Thus L2 is an always-hit for the best case.

3.4 Comparison and Discussion

Cache analysis in the context of bounding execution time analysis for real-time system centers around the worst-case estimations used by schedulability analysis. One may at first argue that a first-miss at a higher memory level will most likely cause a first-miss at the next level since the size of caches increases with each memory level. This would imply that the conventional approach suffices to provide a tight enough bound for multi-level cache hierarchies. However, if the primary cache is a split cache and the secondary cache is unified, data and instruction references that are merged for the secondary cache may be in conflict, even if they were first-misses when considered separately for the primary cache. In this case, the exponential approach may be more adequate to infer tighter estimates.

Overall, the advantage of the exponential approach is that categorizations of the previous memory level can be incorporated into the DFA with all their details where the conventional approach looses information at this point. This comes at the expense of a potentially exponential growth of space in terms of the nesting depth of loops for the exponential approach as depicted in Figure 2. This space overhead will also impact the time overhead. Both approaches may have to analyze best-case and worst-case cache behavior in two separate passes since the categories between best-case and worst-case inputs may differ. The conventional approach calculates additional data-flow information whose overhead is in the same order as the DFA for conventional cache behavior.

An interesting result is also that the problem of categorizations for subsequent memory levels exist for the separate analysis approach as well as the integrated approach. Detailed knowledge about cache behavior for loop nestings also requires a mapping into the next memory level. As pointed out before, an integrated analysis would have to deal with all memory levels at once and may thus result in more overhead in practice due to the memory requirements. In addition, separate analysis is more modular and easier to understand.

The conventional and exponential approaches seem to represent two methods that are at the opposite end of a trade-off between simplicity and precision. This raises the issue whether or not there exist methods in

Loop	L1		L2		L3	
Iteration	1st	2-n	1st	2-n	1st	2-n
Relevant Instance	C1	C5	C1,C5	C3,C7	C(odd)	C(even)
Worst Behavior	miss	hit	miss	hit	miss	hit
Worst Category	first-miss		first-miss		first-miss	
Best Behavior	miss	hit	hit	hit	hit	hit
Best Category	first-miss		always-hit		always-hit	

Table 6: Categorization of Location Y for Next Cache Level

between. At a first glance, Figure 2 seems to indicate that the even instances of block C may be all merged into one set. Furthermore, one may at first be tempted to combine sets C3 and C7 so that a linear space overhead was obtained. However, any data-flow paths between the merged instances would no longer allow us to sufficiently distinguish between the different loops. The source of the problem can be reduced to merging C3 and C7 to C37 (in place of C3). Assuming the same data-flow paths, C6 would now obtain information from C37 via C5. Thus, C5 and C6 would no longer represent L3 within the *first* iteration of L2 (and consecutive iterations of L1) but rather within *any* iterations of L2. The same is true for all intermediate instances on data-flow paths between C37 and C6. However, one may use limited logical unrolling of loops, which provides an analytical framework with linear space overhead with regard to loop nesting, such as the callstring-approach [2].

4 Future Work

The exponential approach is being integrated into our framework for static cache analysis. The conventional approach implemented for primary caches needs to be slightly modified to handle remappings between memory levels. A quantitative analysis of both approaches is intended to answer the question which approach should be used to determine the caching behavior for multi-level memory hierarchies in order to yield tight enough results for bounding the best-case and worst-case execution time.

5 Related Work

A number of frameworks for assessing the worst-case and best-case execution time of real-time tasks have been developed. Today, mainly three fundamental models exist. First, a source-level oriented timing schema propagates times through a tree and handles pipelined RISC processors with first-level split caches [13, 10, 7]. Second, a constraint-based method models architectural aspects, including caches, via integer linear programming [8, 9]. Third, our approach of static cache analysis uses DFA to model the cache behavior separate from pipeline simulation, which is handled later in a timing analyzer [3, 5, 12]. The first and second approaches use integrated analysis of caches while our approach uses separate analysis. There has been no discussion on how to deal with multi-level memory hierarchies or unified caches in the context of the above work, to our knowledge.

Another DFA-approach to modeling caching uses the same categorizations as our approach but a different data-flow model [2]. For their conventional approach, termed callstring(0), first and consecutive loop iterations can be distinguished to yield categorizations for each access. However, this does not allow the distinction between categories for different loop nestings since no additional data-flow information is used (as in our conventional analysis). A second approach, termed callstring(n), performs logical unrolling of loops by a fixed number n to distinguish the behavior for the first n iterations. Typically, only the first two iterations differ in their cache behavior. A third DFA model by the same group, termed VIVU, inspired the use of the exponential approach for multi-level caches discussed in this paper. However, their approach handles loops slightly differently, distinct join semantics are used for best-case and worst-case analysis, and the approach is designed to address first-level caches at this time [4].

6 Conclusion

In this paper, general problems of extending timing analysis frameworks to handle unified caches and multi-level memory hierarchies are identified and a number of solutions are contrasted. Unified caches can be addressed by conservatively predicting conflicting references within a certain window as misses in the worst case and hits in the best case, respectively. Many existing approaches can be extended to handle multi-level caches via integrated analysis that simulates each memory level on the fly as execution times are calculated for code portions. Separate analysis provides an alternative framework, where the information of the cache behavior at one memory level is fed back into another analysis pass to yield the cache behavior of the next memory level. This feed-back approach separates cache simulation for each memory level, which provides a modular approach to timing predictions for multi-level caches that should be more efficient than integrated analysis.

To address the feed-back approach, two data-flow models are introduced as an extension to the method of static cache analysis, which is particularly suited to separate analysis. Both models extend cache analysis of primary caches to handle multi-level memory hierarchies. In both models, the cache behavior is described for each instruction and loop level by categories. The categorization of one cache level is fed back into another pass of static cache simulation to yield categories for the next cache level. The first model uses a conventional data-flow model with constant space overhead in terms of the loop nesting depth but looses some information when propagating cache behavior from one memory level to the next one. This model may be useful for caches beyond the second level. The second model employs several distinct data-flow sets for loop nestings. This allows more precise handling of the cache behavior of the previous level when simulating the next level. On the other hand, the space overhead is exponential to the loop nesting depth. This model may be advantageous for secondary caches when primary caches are split. However, only a quantitative comparison between the two approaches can tell which approach is sufficiently suitable to yield tight bounds for execution times within reasonable overhead for the analysis. Overall, the extensions of timing analysis to unified caches and multi-level memory hierarchies are the first attempt to address these issues, to our knowledge. They provide another step towards bounding execution times for modern architectures and to use these bounds in schedulability analysis for real-time systems.

Acknowledgments

The anonymous referees raised several issues that contributed to the improvement of the quality of the paper and served as a motivation to address the issue of handling unified caches.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium*, September 1996.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[4] C. Ferdinand. Personal communications. January 1997.

[5] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[7] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[8] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, December 1996.

[10] S.-S. Lim, Y. H. Bea, G. T. Jang, B.-D. Rhee, S. L. Min, Y. C. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[11] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[12] F. Mueller. Generalizing timing predictions to set-associative caches. TR 96-66, Inst. f. Informatik, Humbolt University Berlin, July 1996.

[13] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.