To appear in Proc. of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), December 1999

A Method to Improve the Estimated Worst-Case Performance of Data Caching

Thomas Lundqvist Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{thomasl,pers}@ce.chalmers.se

Abstract

This paper presents a method for tight prediction of worst-case performance of data caches in high-performance real-time systems. Our approach is to distinguish between data structures that exhibit a predictable versus unpredictable cache behavior. Cache performance of accesses to predictable data structures can be automatically and accurately determined by our method whereas we let accesses to unpredictable data structures bypass the cache to simplify and improve the analysis. Through experimentation with a number of benchmark programs, we show that a vast majority of data accesses stems from predictable data structures. We analyze what kind of data structures that fall into this category. Remarkably, we find that all data structures in five out of the seven programs are predictable and will lead to a worst-case cache performance which is as high as the real performance. Moreover, for the remaining two benchmarks a majority of the accesses go to predictable data structures. Hence, empirically our data suggest that data caching is expected to improve worst-case performance considerably using our method.

1. Introduction

Cache memories have been a key innovation to shorten the average memory access time to the comparably slow memories. In many real-time systems, however, a high average performance is not the primary concern. Instead, an estimate of the worst-case performance is needed. A conservative approach to estimate the worst-case performance would be to assume that all memory accesses miss in the cache. Obviously, this is not only overly pessimistic; in fact, some of the memory accesses are predictable in the sense that whether they hit or miss in the cache could be determined before the program is run in absence of input data. The hope is that a vast majority of the memory accesses are predictable so that caches can help reducing the worst-case execution time significantly. Then, real-time systems can exploit the performance potential of caches used in contemporary high-performance microprocessors.

Several estimation methods have been proposed that predict the impact of caches on the worst-case execution time (WCET) [1–3,5–13]. One general observation regarding all methods is that the fetching of instructions is often highly predictable and can be accurately analyzed. Thus, instruction caching can often be used effectively in real-time systems. On the contrary, memory accesses going to the data cache are often unpredictable and difficult to analyze. It is presently unclear to what extent data caches can improve worst-case performance.

The goal of this paper is to develop a method to make it possible to predict the expected effectiveness of data caches in real-time systems. We do this in the context of a single uninterrupted (non-preemptive) program execution. Based on a published state of the art WCET estimation method [9], which is described in Section 2, we develop concepts and propose a method to analyze the impact of predictable and unpredictable memory accesses on the worst-case performance of data caches. Our approach is to distinguish between memory accesses to data structures that are predictable and unpredictable.

In Section 3, we present a method that extends the WCET method in [9] with the capability of automatically identifying predictable and unpredictable data structures. We provide examples of frequently used types of data structures that fall into these two categories and also propose a new method for making their cache behavior predictable. In comparison with previously published methods [5, 13] that charge a penalty as high as two misses to each unpredictable access, our approach is to identify and not cache unpredictable data structures. Accesses to such data structures can thus be easily analyzed. The question becomes how big fraction of all data accesses that go to predictable data structures whose cache behavior can thus be accurately predicted.

In order to evaluate the fraction of accesses that go to predictable data structures, we have analyzed a number of benchmarks using our method. This is done in Section 4. Interestingly, we find that five out of seven benchmarks contain only predictable data structures; hence, data caches can be fully exploited. For the other two applications, more than 62 % of the memory accesses go to predictable data structures. Our results thus suggest that data caching can be predictably used with preserved good worst-case performance. We finally relate our findings to work done by others in Section 5 before we conclude in Section 6.

2. WCET Estimation

Before we consider how to identify and handle unpredictable accesses, we will first briefly describe how the WCET of a program can be estimated using a state of the art method. For details, please consult [9].

The WCET of a program is defined as the worst-case execution time of the longest possible path through the program. To obtain tight estimations of the WCET, it is necessary to do both accurate path and timing analyses. In our view, path analysis is responsible for eliminating infeasible (non-executable) program paths and timing analysis is responsible for estimating the execution time of all the remaining (feasible) paths in the program.

Our previously presented WCET estimation method performs both automatic path and timing analyses using cycle-level symbolic execution. Cycle-level symbolic execution means that the timing of all feasible paths are analyzed using cycle-level timing models of the target system. Each path is executed symbolically by dealing with operands that can have an unknown value. Thus, if a branch is encountered that does not depend on unknown input data, the path that is infeasible is automatically excluded from the analysis. This is achieved by extending the execution model, in essence an architectural simulator, with the capability of handling unknown data. The semantics of all arithmetic operations associated with instructions is extended to operate on unknown input data. As a result, loop bounds, for example, that can be statically computed can be derived during the symbolic execution. Therefore, an accurate path and timing analysis can be achieved.

A potential complication of the method is its computational complexity associated with the exponential growth of paths to be analyzed. In a loop with several feasible paths in the loop body, for example, the number of paths to analyze grows exponentially. This problem is alleviated using a path-merge strategy: each time several simulated paths meet in the program they are merged into one. The merging procedure aims at discarding the path that may not contribute to the longest path through the program. The approach taken is to compare the worst-case impact on the future execution of the two paths. Based on this, the shortest path is discarded. In addition, a worst-case system state is assumed in which variable values that differ in the two paths will be assumed to be unknown. Thus, the method may cause additional variable values to become unknown in addition to those that depend on unknown input data.

The method has been empirically evaluated by applying it to WCET estimation of a number of programs run on a multiple-issue pipelined processor with instruction and data caches. In [9], it has been shown that the WCET can be accurately estimated for six out of the seven studied programs.

This method naturally analyzes the impact of data caches on the WCET because it can take into account operands whose values are either known or *unknown*. Therefore, cache behavior of accesses whose reference addresses are statically known is completely predictable. On the other hand, accesses whose reference addresses are *unknown*, either as a result of *unknown* input data or as a result of operand values that become *unknown* during the analysis, will result in unpredictable cache behavior. In the next section, we will use this property to develop a method that can distinguish between predictable and unpredictable data caching so as to automatically analyze the worst-case performance of data caches.

3. Approach to Improved Cache Analysis

In order to extend state of the art WCET methods to make accurate and tight predictions of data cache performance, we start in Section 3.1 to state concepts to reason about data cache predictability. We then present our method in Section 3.2 which is based on the notion of distinguishing between predictable and unpredictable data structures. Finally, based on this, we build intuition into what types of data structures are expected to be predictable. We will test our intuition experimentally in the subsequent sections of this paper.

3.1. Data cache predictability definitions

To understand how our method handles unpredictable accesses, we need to distinguish between the following: a data memory access instruction in the program code, the actual memory access done by the instruction at some point in time, and the data structure in the main memory that is the target of the access.

Definition 1 *An unpredictable memory access is a load or store access whose reference address is unknown during 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.*

The reasons why the reference address is unknown are twofold: First, it could be unknown because it depends on unknown input data. Second, even if the reference address does not depend on unknown input data, the WCET method could introduce uncertainties that actually make some of the operands in the program unknown. For example, in the method outlined in the previous section, the system state of two paths that meet is compared. If the same operand has different values in the two paths, it will be assumed to be unknown in order to make it possible to drop one of the paths and make sure that a worst-case value has been assumed. Therefore, whether an access is predictable or not depends also on the WCET estimation method used.

Definition 2 *An unpredictable memory access instruction is a load or store instruction that generates at least one unpredictable memory access. Conversely, a predictable memory access instruction is a load or store instruction that only generate predictable memory access.*

Because loop constructs are common in most programs, a particular load or a store instruction is executed many times. The purpose of this definition is to identify all load and store instructions that can generate at least one unpredictable memory access. One way of exploiting the fact that a memory access instruction is unpredictable would be to tag it as non-cacheable, thereby avoiding it to interfere with data that can be predictably cached.

Definition 3 *An unpredictable data structure is a data structure that is accessed by at least one unpredictable memory access. Conversely, A predictable data structure is a data structure that is accessed by only predictable memory accesses.*

This definition illustrates an approach to achieve a high predictable data cache performance. If a data structure is only accessed by loads and stores whose reference addresses are known (predictable) during the WCET estimation, the data structure can be predictably cached. Thus, if it is possible to automatically distinguish between predictable and unpredictable data structures, it should be possible to achieve a predictable data cache behavior. The method presented in Section 3.2 has this goal in mind.

The most important characteristic of unpredictable memory accesses is that we connect it to the WCET estimation method used. This is reasonable since it is hard to give a general, method-invariant definition of unpredictable accesses that is not very vague and thereby useless. An implication of the definitions is that whether a data structure is unpredictable or not also depends on the method used. In this paper, when unpredictable data structures are mentioned, they are connected to the state of the art WCET estimation method presented in the previous section.

When looking at a fixed path through a program, predictable memory access instructions are typically those that always generate the same reference address or the same sequence of addresses, regardless of program input data. For example, many estimation methods [2, 5, 7, 11, 13] are capable of handling accesses to global scalar variables. These variables are accessed using a fixed reference address that is independent of input data and is fairly easily deduced by an estimation method. Thus, these variables are predictable data structures when regarding all methods.

3.2. A predictable cache analysis method

We will now describe a method that makes it possible to identify and handle unpredictable memory accesses. We especially study how to use our previously presented WCET estimation method, but the principles presented here can be applied also to other WCET estimation methods presented in the literature as we will discuss in Section 5.

As a first step, we identify all unpredictable accesses by performing a WCET analysis. As explained in Section 2, the WCET method automatically identifies all loads and stores whose reference addresses are unknown. Therefore, all memory accesses which have an unknown reference address can be collected. To reduce the amount of information all unknown references are not collected but only the corresponding memory access instructions. Thus, the result is a list of unpredictable memory access instructions and we consider all memory accesses from these instructions as unpredictable. In the next step, each unpredictable memory access instruction is connected to the corresponding data structure. This requires information from the compiler or user about which data structure each memory access instruction can touch. Each data structure that is accessed by an unpredictable memory access instruction is marked as unpredictable. In the final step, linking is redone in order to map all data structures marked as unpredictable into a special memory area. This memory area will be marked as non-cacheable.

When all unpredictable data structures have been identified and properly remapped, a final estimation of the WCET can be done. Now, the time taken to access an unpredictable structure is equal to the miss penalty. Assuming a memory hierarchy with a single level, the miss penalty is simply the memory access time. If we would have allowed caching of the unpredictable access, the time charged could in the worst-case have been two cache miss penalties: one for the potential cache miss and one for the possibility of replacing a useful block. In addition, the time taken for a cache lookup must be added. In summary, by letting all unpredictable accesses bypass the cache we may gain more than a factor of two on the worst-case performance of data caches.

Storage type	Explanation
Global	Global or static structures.
Stack	Stack allocated structures.
Heap	Dynamically allocated structures on the heap.

Access type	Explanation
Scalar	Only one element.
Regular array	Array accessed by regular, stride accesses.
Irregular	Irregular accesses but still input data independent.
Input dependent	Reference addresses directly depends on input data.

Table 1. Data structure classification based on storage type (upper) and access type (lower).

The approach presented here requires support in the computer system for controlling the caching of different memory areas. Fortunately, hardware mechanisms that permit us to do this exist in many microprocessors that use caches. In some systems, we can even choose if we want to cache or not to cache each individual data access. As an example, PowerPC 403 GA [4] has a double-mapped memory address space. This means that one physical memory location can be reached from two different addresses and we can choose different cacheability for the two addresses, one cache-enabled address and one cache-disabled address. Other hardware mechanisms also exist; in many general-purpose processors, support often exists for virtual memory where cacheability can be controlled on a per page basis. This can be used to create a similar double-mapping, allowing individual accesses to be either cached or not cached.

Even if hardware permits cacheability control of each individual memory access, this can be hard to exploit since it would require quite complex compiler support. Another available approach is to control each individual memory access instruction. This would allow caching of accesses to a data structure when it is used predictably in some parts of the program and not caching accesses when it is used unpredictably in other parts. Still, compiler interaction is needed and care must be taken to keep the contents of memory and cache consistent by flushing the cache at proper points or by using a write-through policy.

In this paper, for simplicity we have only evaluated how to control cacheability on a data-structure level. This requires only support from the linker in order to control the placement of individual data structures. Also, there is no consistency problem as data is either cached or not cached.

3.3. Data structure classification

Based on the notion of predictable versus unpredictable data structures, we now develop intuition into what data structures are expected to fall into each of these two categories. It should be noted though that this generally depends on the WCET method used. We will therefore comment on to what extent the intuition depends on a certain WCET method.

In Table 1 we classify data structures with respect to storage type (upper table) and access type (lower table). The storage type determines what base address is used to access an element in the data structure. This base address is typically stored in a specific register (global, stack, or heap). Additionally, the access type determines how elements using the same base address are accessed and typically uses a fixed offset (scalar) or a varying offset that is calculated for each access (regular, irregular or input data dependent). Whether a data structure is predictable or not typically depends on both the storage type and the access type.

Let us first discuss predictability properties of data structures of different storage types. Global storage is the most simple one because the base address is a fixed known value produced by the linker. Stack allocated structures may look hard to analyze. Yet, they are often handled predictably by all estimation methods. The general approach adopted is to only estimate the WCET for complete programs since then the stack pointer is known. If only a single function or procedure in the program is analyzed the stack pointer would be unknown. The estimation method must keep track of the function call stack, and if a function is called from several places in the program each invocation of the function must be treated as a separate instance. Then, each function instance will have a fixed stack pointer value and thereby a known base address for stack allocated structures. This is supported by the WCET method used in this paper.

The most tricky storage type is heap storage, i.e., dynamically allocated memory on the heap. Due to its unpredictable behavior, this kind of allocation is not always permitted in real-time systems. However, the WCET method used in this paper allows some limited use of dynamic allocation. A necessary condition for predictable dynamic allocation is that the memory must be allocated in an order and in an amount that is independent of input data. This means that at each point in time, we always know which objects

reside on the heap and in which order they are allocated. Then, the base address of the allocated data structures are known. Using dynamic allocation in this controlled manner may seem a bit pointless but can still be useful. For example, it would allow the programmer to write programs that reuse memory in a straight-forward way.

A known base address of a data structure is not enough to make it predictable. The access type must also be taken into consideration. For scalar variables, the base address is the only thing used. They are therefore predictable whenever the base address is predictable. Regular array accesses, i.e., accesses with a constant stride, are considered to be such accesses that can be predicted using e.g. data dependency analysis. This kind of analysis is treated in for example [2] and typically handles the case where the reference address is a simple function of the loop iteration variables. Many methods can analyze this kind of accesses.

A data structure accessed by irregular accesses may in theory be predictable since the accesses are independent of input data. However, many estimation methods would classify it as unpredictable because of lacking analysis of complex data dependencies. Nevertheless, as we will see in Section 4, the estimation method used in this paper manages to handle a case of irregular accesses, showing that some irregular data structures can indeed be predictable.

Finally, a data structure accessed by input data dependent accesses will always be unpredictable as long as input data is considered to be unknown.

In summary, we can note that many types of data structures are expected to be predictable using state of the art WCET estimation methods. This makes us expect that data caching can be effective in achieving a high worst-case performance for data caches. We test this hypothesis in the next section.

4. Experimental Results

In order to evaluate how much the worst-case performance is improved when using data caching, we have classified data memory accesses based on WCET estimation of seven benchmark programs and determined the fraction of memory accesses that go to unpredictable data structures. We have also measured the data cache hit-rates and the corresponding WCET when counting all accesses to unpredictable data structures as misses.

4.1. Methodology

The state of the art WCET estimation method presented in [9] has been used to first classify all data structures according to the method presented in Section 3.2. Then, the WCET estimation tool has been extended to also collect all data memory accesses along the worst-case path during the analysis. The collected memory accesses have been classified according to which type of data structure they access. The categories of data structures considered are the ones presented in Section 3.3.

Name	Description
matmult	Multiplies two 50x50 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
DES	Encrypts 64-bit data
DES-opt	DES compiled with optimizations enabled
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

Table 2. Characteristics of the programs used.

The timing model assumed for the hit-rate and WCET estimations represents an ideal architecture containing only a data cache. All instructions execute in a single cycle except on a data cache miss when a miss penalty of 10 cycles is added. The data cache is a 2 Kbyte direct-mapped cache with 16-byte block size. When regarding the data cache, loads and stores were treated as equivalent to each other.

An overview of the seven programs can be seen in Table 2. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *jfdctint*, *DES* and *compress*. The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the programs. No optimization was enabled except for *DES-opt* which was compiled with the option `-O2`. The simulated run-time environment contained no operating system; consequently, we disabled all calls to system functions such as I/O in the programs.

4.2. Results

The results from the memory access classification can be seen in Figure 1. The first five benchmarks, *matmult*, *bsort*, *isort*, *fib*, and *jfdctint*, were found to contain only predictable data structures. In all these five benchmarks, the majority of the accesses aimed at scalar variables allocated on the stack. Another common effect is that most regular array accesses were either allocated globally (*matmult*, *bsort*, and *jfdctint*) or on the stack (*isort*). All the accesses to these data structures were found to be predictable when estimating the WCET and we thereby confirm the intuition from Section 3.3 that scalar and regular array accesses to global or stack allocated structures are predictable.

	matmult	**bsort**	**isort**
Scalar, global	1.4 %	0.1 %	2.6 %
Scalar, stack	49.9 %	77.0 %	66.6 %
Regular, global	48.7 %	22.9 %	
Regular, stack			30.8 %
Total accesses	1057597	130652	876
	fib	**jfdctint**	
Scalar, global	3.5 %	0.5 %	
Scalar, stack	96.5 %	80.9 %	
Regular, global		18.6 %	
Total accesses	397	2754	
	DES	**DES-opt**	**compress**
Scalar, global	4.1 %	10.3 %	24.4 %
Scalar, stack	84.5 %	59.2 %	36.4 %
Regular, global	5.4 %	14.5 %	1.7 %
Regular, stack	0.6 %	1.5 %	
Irregular, global	4.9 %	13.0 %	
Input dep, global	0.6 %	1.5 %	37.5 %
Total accesses	45876	17200	8852

Figure 1. Classified memory accesses from the worst-case program path.

In the last benchmarks, *DES*, *DES-opt*, and *compress*, unpredictable data structures were found. In *DES* two arrays are read using an index that depends on unknown input data. However, only 0.6 % of all accesses went to these arrays and the majority of accesses did again reference scalar variables allocated on the stack. The same is valid for *DES-opt*, although the amount of scalar stack accesses has been reduced. This reduction is expected and can be explained by the fact that scalar stack storage is often used as temporary storage by the compiler. Part of this temporary storage can often be eliminated by the compiler when enabling optimization. The reduction of scalar stack accesses makes the other accesses relatively more significant. Still, only 1.5 % of all accesses is unpredictable in *DES-opt*.

An interesting fact is that the WCET method managed to handle some cases of irregular arrays accesses in *DES* and *DES-opt*. Some arrays were accessed using an index obtained from another, regular array access. Thus, the array references are independent of input data and were found to be predictable by the WCET estimation method we used. To the best of our knowledge, this kind of array accessing would have been found unpredictable by all other WCET estimation methods.

In the final benchmark, *compress*, four data structures were found to be unpredictable. The dominant structures in this case were two hash tables (total size 1.5 Kbyte) indexed by unknown input data. In total, 37.5 % of all accesses went to these unpredictable data structures. Of these 37.5, unpredictable accesses contributed with 30. The rest, 7.5 was the contribution from predictable accesses during the initialization of the hash tables. This means that it would have been better to control the cacheability on the instruction level instead of a data structure level. However, the gain would have been small.

Table 3 summarizes the amount of memory accesses that were found to be predictable. Also shown is the corresponding data cache hit ratio. As explained before, almost all accesses were predictable in all benchmarks except in *compress*. This can also be seen in the hit ratio numbers which are close to 100 % in all benchmarks except *compress* where the unpredictable accesses cause a significant reduction of the hit ratio.

To understand the importance of data caching, we have also included the estimated WCET for two cases: one when all accesses to predictable data structures are cached and another when caching is disabled and all accesses are counted as misses. Ratio is the cache-disabled WCET divided by the cache-enabled WCET. It represents the improvement of the worst-case performance obtained when including a data cache. Clearly, for the majority of the programs studied,

Name	Fraction predictable	Data cache hit ratio	WCET	No data cache WCET	No data cache Ratio
matmult	100 %	92.1 %	7899863	17639883	2.2
bsort	100 %	97.8 %	320887	1598547	5.0
isort	100 %	98.1 %	2765	11355	4.1
fib	100 %	96.5 %	838	4668	5.6
jfdctint	100 %	98.7 %	6361	33551	5.3
DES	99.4 %	98.8 %	124276	577436	4.7
DES-opt	98.5 %	96.8 %	53905	220345	4.1
compress	62.5 %	62.0 %	82190	137050	1.7

Table 3. Fraction predictable accesses and corresponding hit-rate.

data caching is efficient and improves the worst-case performance significantly, in many cases by a factor of four or more. Even for *compress*, we get a considerable improvement of a factor 1.7 when using a data cache in spite of all the unpredictable accesses encountered.

5. Discussion and Related Work

The WCET estimation method used in this paper handles quite complex data dependencies and identifies many data structures as predictable, even the irregular ones in *DES* as seen from the results. An interesting question is if the use of another WCET estimation method would have changed the results drastically. According to our belief, other methods would also make a good analysis of many of the programs. For example, the methods presented in [2, 5, 7, 13] should be able to produce similar results as the method we used for the first five benchmarks. These benchmarks only contain data structures that are fairly simple to handle. The most complex one is regular array accesses which, according to our belief, could be handled by all mentioned methods. On the other hand, the other methods would probably perform worse on *DES* and *DES-opt*, due to the irregular array accesses present, which are accurately analyzed by the method we used.

The handling of *compress* by other methods is largely dependent on which strategy they use to handle unpredictable accesses. This strategy is not always made clear from the descriptions of the methods but in for example [5] they adopt the strategy of caching unpredictable accesses. Then, to make a safe estimate of the WCET, two miss penalties must be added for each unpredictable access: one for the possibility of a cache miss and another for the possibility of replacing another useful block. This strategy would reduce the worst-case performance quite drastically for *compress*. The resulting estimated WCET by our method when using this strategy is 108690 cycles which is only a factor of 1.26 better than not caching at all. This can be compared to a factor of 1.7 obtained when using the method in this paper.

The method of identifying unpredictable data structures, described in Section 3.2, is not limited to the WCET estimation method used in this paper. The same method could be used together with other estimation methods. The important criterion is that a list of all instructions generating accesses with an unknown reference address can be created during the estimation. This information can probably be generated early in the estimation procedure since deducing the reference addresses of data accesses is often the first step needed when analyzing data cache behavior.

In this paper, a state of the art WCET estimation method presented in [9] has been used to identify unpredictable data structures and estimate the WCET of seven benchmarks programs. The results obtained are probably very hard to improve upon since the unpredictable accesses in for example *compress* depends on unknown input data and are very random in nature. Thus, the results obtained for the benchmarks in this paper is as good as it can get. However, for other programs it is possible that accesses to data structures that we would classify as unpredictable could be turned into predictable when using other methods. As an example, consider regular accesses to an array allocated on a heap with an unknown base address. The WCET method we use would then classify the array as unpredictable. However, another method could predict some of the behavior based on upper or lower bounds on the number of misses or hits in the cache. The feasibility of this approach depends on how advanced data dependency analysis that can be made and we find it currently unclear if other methods could handle a case like this. For predictable data structures, many methods [2, 5, 7, 13] use the approach of calculating upper and lower bounds on the number of misses or hits.

A rather different approach of handling data cache analysis is described by Basumallick and Nilsen [1] where they use a register allocation algorithm to allocate data into different cache blocks. In this way, they arrange the data in a way that guarantees a certain number of cache hits. However, the same problem with unpredictable accesses will still be present.

The focus of this paper has been to obtain tight WCET estimations for a system with a data cache. If tight WCET estimation is not of primary concern it may be more fruitful to cache also unpredictable data since this probably reduces the average execution time. Then, to make a safe estimate, the approach taken by [5] must be adopted which means that two miss penalties must be added for each unpredictable access. Hence, it is possible to trade tightness of WCET estimation for increased average performance.

6. Conclusions

In this paper, we have presented a new method for improving the estimated worst-case performance of data caching. Based on a definition of unpredictable data structures, our method identifies unpredictable data structures and place them in a non-cacheable part of the memory. In this way, only predictable accesses pass through the cache which makes it possible to improve the WCET estimations. In the paper, different data structures used in programs are studied and common predictable data structures are identified.

The method, which is an extension of a state of the art WCET method, has been experimentally evaluated in order to estimate the WCET of seven benchmark programs. The results show that all data structures in five of the seven benchmarks are predictable. Thus, for these programs, the cache behavior is perfectly predicted and data caching is efficiently used. In the remaining programs, all unpredictable data structures were identified. Fortunately, the fraction of accesses to them were low. Thus, our empirical data suggest that data caching results in a very good worst-case performance.

7. Acknowledgments

This research is supported by a grant from the Swedish Research Council on Engineering Science (TFR) under contract number 221-96-214.

References

[1] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[2] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, June 1998.

[3] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[4] IBM. PowerPC 403 GA user's manual. http://www.chips.ibm.com.

[5] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996.

[6] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.

[7] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[8] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[9] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999.

[10] F. Mueller. Timing predictions for multi-level caches. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1997.

[11] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 47–55, June 1997.

[12] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, December 1998.

[13] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3nd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.