CHALMERS

Empirical Bounds on Data Caching in High-Performance Real-Time Systems

Thomas Lundqvist

Per Stenström

Technical Report 99-4

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 1999

Empirical Bounds on Data Caching in High-Performance Real-time Systems

Thomas Lundqvist and Per Stenström

Department of Computer Engineering, Chalmers University of Technology
SE-412 96 Gothenburg, Sweden

Email: thomasl@ce.chalmers.se, URL: http://www.ce.chalmers.se/~thomasl

Abstract

In this paper we study to what extent hard real-time programs can exploit the performance potential of data caches. Data structures that are accessed by memory instructions whose addresses are input data independent, can be safely cached. Based on a set of non-trivial programs from the SPEC95 benchmark suite, we find that more than 84% of the data accesses are predictable. With static analysis methods, it appears that a high predictable hit rate can be obtained which can result in tighter estimations of the worst-case execution time.

1. Introduction

Many time-critical real-time applications need high-performance microprocessors to meet their performance demands. When developing software for such systems, a main problem is to verify that it meets time constraints specified as deadlines. The industrial practice is to carefully estimate the worst-case execution time through exhaustive measurements which is both time-consuming and error-prone. This has motivated us and other research groups [1, 2, 3, 13] to automate the task of estimating the worst-case execution time. Our goal is to develop methods for integration in a combined compiler/timing analysis tool. This tool not only generates code but provides also estimations of the worst-case execution time of a program.

The task of estimating the worst-case execution time of a program can be formulated as a graph problem. Consider an acyclic[1] control-flow graph representation of the program, where each vertex is a basic block of instructions or an acyclic graph with its associated (worst-case) execution time. The worst-case execution time is then the longest path from the entry to the exit point. For simple microcontrollers with constant instruction execution times, fairly tight worst-case execution time estimates can be derived this way [10]. For high-performance embedded microprocessors that employ pipelining and caching, however, this methodology can provide very pessimistic estimations. This is because the worst-case instruction execution time can be an order of magnitude longer than the best-case execution time; in the worst-case an instruction can result in two cache misses which can easily stall the processor for tens of cycles. At the same time, cache memories often handle a vast majority of all memory accesses in a single cycle. This has motivated us to develop methods to predict data cache behavior so as to provide tighter bounds on the worst-case execution time.

The challenge is to determine the set of data accesses that are independent of input data. Such memory instructions are allowed to cache data. Recently a few studies have been published on methods to allow for predictable data caching. The basic method in [7] handles all unpredictable data accesses as if they result in two cache misses because in the worst case, a memory instruction will miss and replace data. If the predictable hit rate is lower than 50%, this method will produce more pessimistic results than if data caches were not used. Our approach is instead to let all memory accesses that are not predictable bypass the cache. Support for this exists in most embedded microprocessors. [6] is concerned with the problem of how to model conflicts between predictable data cache accesses and does not address the problem of how to handle unpredictable data accesses.

Ultimately, the effectiveness of data caches in hard real-time systems is dictated by the fraction of all memory accesses that can be predicted, i.e., are input-data independent. The purpose of this paper is to make an estimation of this fraction by (1) formulating what predictable data caching is and by (2) providing empirical data on what fraction of data accesses that are predictable. We do this by analyzing a set of non-trivial programs from the SPEC95 benchmark suite. While our study is preliminary, our empirical data so far look promising; more than 84% of the data accesses are indeed predictable. This suggests that data caching is effective in hard real-time systems although the challenge is to find tractable methods to come close to this bound. The fraction of such predictable data accesses that can be covered by such methods establishes a practical bound on the effectiveness. In Section 2, we provide our approach to predictable data caching. Sections 3 and 4 present the experimental results and, finally, Section 5 discusses our ongoing work.

1. Constraints on loop bounds, for example, must be available to make the control flow graph acyclic.

2. Predictable Data Caching

Predicting cache behavior can typically be divided into two steps. The first step is to find out the reference address/addresses of all instructions. The next step is to use this information to statically model cache behavior so as to predict its impact on the worst-case execution time. Such a method exists for instruction caches [4].

Recently, some methods to analyze data caches have been proposed [6, 7]. Their approach is to extend older methods that handle instruction caches to also handle data caches. This works because in both cases the goal is to statically determine the set of memory accesses for each execution path in the program. However, one big difference is that the addresses generated from load/store instructions to access the data cache are not always known or predictable. A typical case is when the reference address of a load instruction depends on unknown input data. If the address of all accesses to the data cache are known, it is possible to predict whether each memory access will hit or miss in the data cache. Unknown reference addresses of a memory instruction, however, lead to the pessimistic assumption that the memory instruction will miss in cache and results in poor estimations of the worst-case execution time.

Definition (*Predictable memory instruction*): With a predictable memory access instruction (load/store instruction) we mean an instruction that generates the same reference address or the same sequence of addresses, regardless of the unknown input data, and also regardless of the path taken through the program as long as the path includes the instruction.[2]

Predictable memory instructions will put an upper-bound on the prediction of the number of cache hits. Unfortunately, all predictable memory instructions cannot be determined in practice through static compiler analysis (e.g. dataflow analysis). We will call the set of predictable memory instructions that at the same time can be analyzed as *analyzable memory instructions*. The fraction of all executed memory instructions that at the same time are analyzable will put a practical upper bound on the effectiveness of data caching in hard real-time systems.

An interesting question is how good data cache prediction we can get. How many of the memory accesses have an unknown address? If we succeed to analyze all predictable accesses, what improvement in the form of increased hit rate will we see?

To answer the question of how many accesses are predictable, we decided to take a look at memory accesses from non-trivial programs and we chose the SPEC95 benchmark suite as a suitable object of study. These programs are not written specifically for real-time systems, but are interesting due to two facts. First, they are quite big and represent real programs doing some useful work. Second, they are well-known in the computer architecture community in performance evaluations and thereby suitable to use as a reference.

Given that we can determine the set of predictable memory instructions, the question is how this could be used to reach a high and predictable cache hit rate. To make this upper bound on predictability more useful, it is important that unpredictable memory instructions do not change the cache state. Fortunately, a hardware mechanism that permits us to do this exists in most embedded microprocessors that use caches. We can choose if we want to cache or not to cache each individual data accesses. As an example, PowerPC 403 GA [8] has a double mapped memory address space. This means that one memory location can be reached from two addresses and we can choose different cacheability for the two addresses, one cache-enabled address and one cache-disabled address. Since an unpredictable and predictable memory instruction may potentially access the same memory location, a write-through write policy must be chosen to avoid inconsistency.

Another solution is if the system supports virtual memory. Then cacheability is usually controlled at page level, and we can easily arrange a similar double mapping scheme in this case. There are two ways to exploit this. We can either put our unpredictable data structure in a special region in memory, marked as uncacheable, or we can control each individual load/store instruction and make it cacheable or non-cacheable. In the latter case, we must make sure the cache is consistent. To control the cacheability at a data structure level, we need to control the mappings done by the linker. For an instruction level control one might use compiler support. Data structure control seems a little bit simpler and we have chosen this level for our experimental classification of memory accesses. It is then useful to define the term *unpredictable data structure*. This is a data structure (part of the memory) that is accessed by at least one unpredictable memory access instruction. Such data structures are not cached.

2. This means that for a fixed path through a program with only predictable memory instructions, we will get a completely statically known sequence of reference addresses, regardless of variations in input data that still give us the same fixed path.

3. Experimental Results

3.1 Methodology

We have classified data memory accesses and determined the fraction of accesses that goes to unpredictable data structures. To this date we have studied two programs from the SPEC95 suite.

The SPEC95 suite consists of 8 C-programs using integer arithmetic and 12 fortran programs using floating point arithmetic. We have examined two programs, Compress, an integer program, and Swim, a floating-point program. Compress is an in-memory version of the common UNIX utility. Swim calculates shallow water equations. Swim was first translated from fortran to C by using f2c, a fortran to C translator, because the simulator we used didn't support fortran programs[3].

To analyze the data memory accesses we ran our programs on the simulator SimICS [9]. SimICS simulates the SPARC V8 instruction-set and emulates the SunOS 5.x operating system. When simulating a program it delivers a stream of memory accesses to our classification and data cache simulation program.

We classify each memory access as one of the categories in Table 1, depending on which type of data structure the access refers to. We first classify each data structure used in the program by manual inspection of the source code as being predictable or unpredictable. This information is fed to the classification program which registers the number of accesses to each data structure. Since we are only interested in memory accesses originating from the application, memory accesses from library code do not affect the statistics in our simulations.

Table 1: Categories of classification

Type	Explanation
Scalar, stack	A single reference address to something in the stack area
Scalar, data	A single reference address to a local or global symbol
Array, data	An access to an array, referenced with a predictable series of addresses
Unpredictable	The exact reference addresses to this data structure depends on unknown input data

3. The fortran libraries use unimplemented operating system calls.

All SPEC95 programs can be run with two different data sets, one reference data set and one test data set. The reference data set is quite time consuming to simulate and the test data set is usually too small for a serious analysis. We have chosen a medium-sized data set for each of the two programs that is not too simple and not too time consuming. In the end we plan to run the programs with the full reference data set.

3.2 Compress

For compress, the results of the classification can be seen in Figure 1. A total of 110 million memory accesses were classified and 84% were found to be predictable.

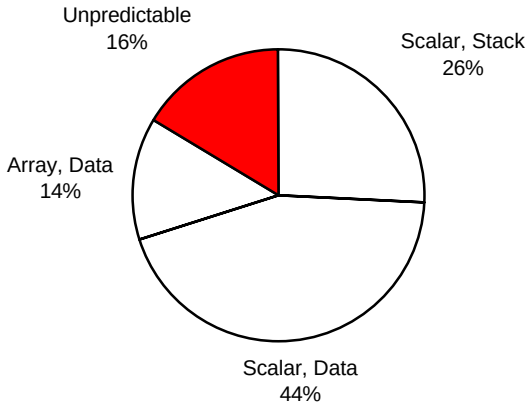

Figure 1: Memory accesses from compress

Compress works by first filling an array with random characters. Then, it compresses and decompresses this array 25 times. The compression method used is a modified Lempel-Ziv (LZW), which finds common substrings and replaces them with a variable size code. The core of the algorithm processes one character at a time and uses it as an index to a hash table. We regard the content of the array (the random characters) as unknown input data, all other parameters are fixed.

All stack accesses in compress go to scalar variables (scalar data structures have only one possible reference address) and are therefore predictable[4]. The same is valid for most global and local data. Among the array variables, there are 3 arrays that are unpredictable. These arrays are used as a hash table indexed by the unknown input data.

4. Even if references to the stack area are relative to some stack pointer or frame pointer, it is still possible to statically predict their absolute addresses.

Most of the predictable array accesses consist of accesses in the form of strides. A simple loop variable is used to index the array. The number of stack accesses is quite high and would be even higher if run on a different processor, due to the fact that the SPARC processor has register windows, which are used for function parameter passing instead of using the stack.

3.3 Swim

For Swim, the results of the classification can be seen in Figure 2. A total of 658 million memory accesses was classified and all accesses were found to be predictable.

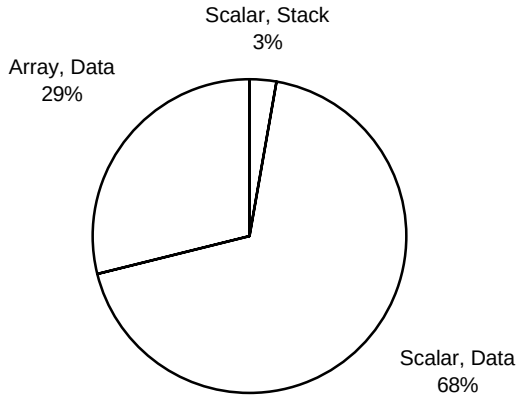

Figure 2: Memory accesses from Swim

Swim consists mainly of matrix calculations. Starting from some initial values, it does the same calculation repeatedly for a fixed number of iterations. The unknown input data in this case would be the initial values of the matrices. The calculations done are independent of the actual data inside the matrices. Therefore, all accesses are predictable.

All of the array accesses are in the form of strides. The addresses used is a linear combination of one or two loop index variables. The number of stack accesses is very small. Very few local, stack allocated variables are used, and there is enough registers to hold temporary values during computations.

3.4 Hit Rate Measurements

We have seen that a vast majority of the data memory accesses are predictable. If we cache these predictable accesses, what hit rate would we achieve?

More generally speaking, we would like to know how many of the accesses that are analyzable. This depends on the method used and we will not try to answer that question here, but we can still take a look on how it influences the hit rate.

We have simulated a data cache[5] and let it cache different number of the data structures. For example, we could guess that the scalar variables are analyzable. Then, we cache the references to the scalar variables and do not cache the others (i.e. all other references will count as a cache miss) and see what the resulting hit rate becomes. The result for Compress is seen in Figure 3 and for Swim in Figure 4.

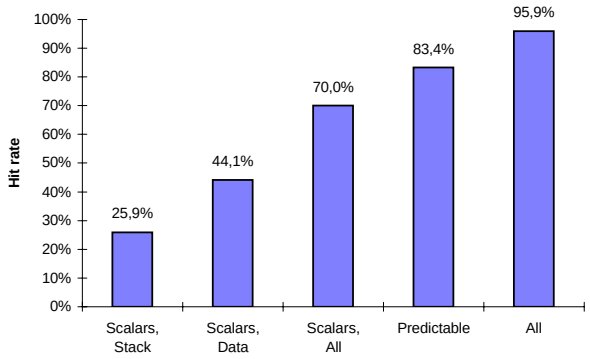

Figure 3: Hit rate when caching different parts of the data accesses for compress

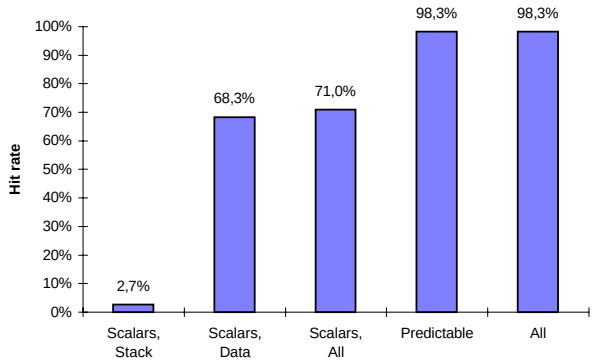

Figure 4: Hit rate when caching different parts of the data accesses for swim

These diagrams tell us that if we succeed in analyzing all predictable accesses, we will reach a fairly high hit rate. In Compress and Swim, the predictable hit rates are

5. The data cache was direct mapped, with a total size of 32768 bytes and a block size of 32 bytes.

83% and 98% (the same as the actual), respectively. The predictable hit rates are thus close to the actual hit rates.

We can also translate these hit rates into execution times. Let us assume a simple model for a microprocessor with a data cache. Each instruction takes 1 cycle, a cache hit adds no extra cycle and a cache miss will add 10 extra cycles. For our two programs the measured relative frequency of memory instructions is approximately 23%. This simple model gives us the following relation (Figure 5).

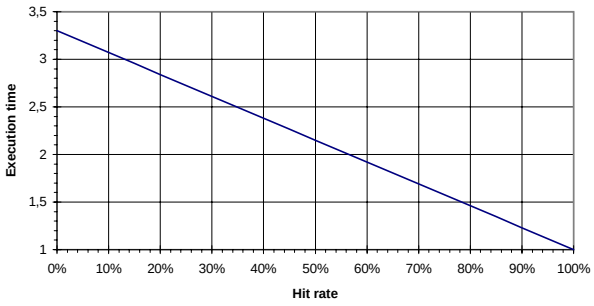

Figure 5: Execution time as a function of hit rate

Figure 5 tells us that if we have a hit rate of 0%, it will take a factor of 3.3 times longer time to execute compared to a hit rate of 100%. For Compress (83%) we get a factor of 1.4. We have thus cut the time by more than half compared to a disabled data cache.

4. Discussion

We have found that the potential of data caching based on the two programs we have studied is high. However, to be able to use the full potential we must find methods to analyze the predictable part of the accesses.

We have implicitly assumed that a program is run without preemption. However, it is possible to use the methodology of classifying accesses as predictable under preemptive scheduling policies as well given that different processes use different partitions of the cache. This can be achieved through software-based cache partitioning as discussed in [5]. This same technique can also be used to allow caching of unpredictable data structures given that they use another partition.

Another interesting question is if we can somehow do better than the predictable limit. The answer to that is yes, we can, but only if we have more information. With more information about the unknown input data (converting it into known input data maybe), we can in some cases make unpredictable accesses into predictable. Also, more information about the program itself (the source code or the algorithm), maybe in the form of annotations made by a programmer, will also surely prove useful. This and other issues remain open questions for future research.

5. Work in progress

Currently, we are working on a method to estimate the worst-case execution time of a program. A tool using this method is also under construction. Our main design principle is to estimate the worst-case execution time by using architectural simulation. This simulation must be feasible in spite of unknown input data. To accomplish this we have combined architectural simulation with symbolic execution. Thus, the simulator does not only work with real values but also with abstract values.

There are several advantages with doing a functional and temporal simulation of a program. Firstly, we dynamically resolve many dependencies present in the program without the need for extra annotations. This may automatically eliminate many false paths in the program graph. Secondly, we can use more complex annotations, which may be expressed with the help of variables in the program that are known during simulation. Annotations and false-path elimination are means to tighten the estimation on a high level. At the same time an architectural simulator have its strength at a low level since it can accurately mimic the hardware of a system.

Our method is similar to a recently proposed method [11]. However, we do not use a pure branch-and-bound algorithm. Instead we have added a path merge strategy to keep the complexity at a manageable level. The work done in [12] has inspired us, but they are only focusing on the annotation side of the problem.

In the near future we will try to analyze the Compress application and find out how many of the predictable data accesses are analyzable by our method. Our tool will primarily handle a simple architecture with the data cache as the only complex feature.

Acknowledgments

We are indebted to Peter Magnusson of SICS for providing us with the SimICS simulation system. This research is supported by the Swedish Research Council for Engineering Sciences (TFR).

References

[1] D. B. Healy, D. B. Whalley, and M. G. Harmon "Integrating the Timing Analysis of Pipelining and Instruction Caching" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 288-297.

[2] Y Hur et al. "Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 308-319

[3] Y. S. Li, S. Malik, and A. Wolfe "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software" in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995, pages 198-307

[4] F. Mueller and D. B. Whalley "Fast Instruction Cache Analysis via Static Cache Simulation" in the *Proceedings of the 28th Annual Simulation Symposium*, April 1995, pages 105-114.

[5] F. Mueller "Compiler Support for Software-Based Cache Partitioning" in *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, June 1995.

[6] Y.S. Li, S. Malik, and A. Wolfe ”Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches” in the *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996, pages 254-263.

[7] S.-K. Kim, S.L. Min, and R. Ha. ”Efficient Worst Case Timing Analysis of Data Caching” in the *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, June 1996.

[8] PowerPC 403 GA Users’s Manual by IBM. See http://www.ibm.com

[9] The SimICS simulator by SICS. See http://www.sics.se/simics

[10] Allan C Shaw. ”Reasoning About Time in Higher-Level Language Software” in the *IEEE Transactions on Software Engineering*, 15(7): 875-889 1989.

[11] Peter Altenbernd. ”On the False Path Problem in Hard Real-Time Programs” in the *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, June 1996, pages 102-107.

[12] Andreas Ermedahl and Jan Gustafsson. ”Deriving Annotations for Tight Calculation of Execution Time”, *EuroPar97*, August 1997 (*to appear*).

[13] Greger Ottosson and Mikael Sjödin. ”Worst-Case Execution Time Analysis for Modern Hardware Architectures”, *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*.