Time-predictable Cache Organization

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Abstract

Caches are a mandatory feature of current processors to deliver instructions and data to a fast processor pipeline. However, standard cache organizations are designed to increase the average case performance. They are hard to model for worst-case execution time (WCET) analysis. Unknown abstract cache states during the analysis result in conservative WCET bounds. Therefore, we propose to adapt the cache organization to simplify the analysis. The data cache is split into several independent caches for the stack, static data, constants, and heap allocated data.

1. Introduction

Standard computer architectures optimize the architecture for maximum throughput and average case performance. The resulting architectures are very problematic to model for worst-case execution time (WCET) analysis. Unknown state of the processor during the analysis results in conservative WCET bounds. We argue that future processors for real-time systems need to be designed to be time-predictable to overcome this analysis issue. The internal state of the processor has to be visible for the analysis.

Hiding the architectural details behind the instruction set architecture (ISA) is one of the key principles for success in general purpose computing. Programs can run faster on a new machine without recompilation. This abstraction turns WCET analysis into a nightmare. We need to know all the architecture details of the pipeline and the memory hierarchy to analyze the execution time of programs. Therefore, we argue that a time-predictable architecture has to expose all pipeline and memory hierarchy details to the compiler and the WCET analysis tool. In the embedded real-time domain, compatibility of the ISA is less important. On a processor change the application can be recompiled.

Between the middle of the 1980s and 2002, CPU performance increased by around 52% per year, but memory latency decreased only by 9% [11]. To bridge this growing gap between CPU and main memory performance, a memory hierarchy is used. Several layers with different tradeoffs between size, speed, and cost form that memory hierarchy. A typical hierarchy consists of the register file, first level instruction and data caches, one or two layers of shared caches, the main memory, and the hard disc for virtual memory.

Cache memories for the instructions and data are classic examples of the *make the common case fast* paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Plenty of research effort has been expended to integrate the instruction cache into the timing analysis of tasks [2, 9], the influence of the task preemption on the cache [4], and the integration of the cache analysis with the pipeline analysis [8].

A unified cache for data and instructions can easily destroy all the information on abstract cache states. Access to n unknown addresses in an n-way set-associative cache results in the state *not classified* for all cache lines. Modern processors usually have separate instruction and data caches for the first level cache. However, the second level cache is usually shared. Most chip-multiprocessor (CMP) systems also share the second level cache between the different cores. The possible interactions between concurrent threads running on different cores are practically impossible to model.

Caches in general, and particularly data caches, are hard to analyze statically. Therefore, we introduce caches that are organized to speed-up execution time and provide tight WCET bounds. We propose different caches for different data areas:

- An instruction cache for complete methods
- A stack cache
- A cache for static data
- A small, fully associative buffer for heap access
- A cache for constants

Furthermore, the integration of a program- or compiler-managed scratchpad memory can help to tighten the bounds for hard to analyze memory access patterns.

The concepts, presented in this paper, are language agnostic. Although, the discussion of virtual method dispatch tables is biased towards object-oriented languages, such as C++ and Java.

2. Related work

Edwards and Lee argue that a new research discipline is needed for time-predictable embedded systems: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [5]. A first simulation of their PRET architecture is presented in [14]. PRET implements the SPARC V8 ISA in a six-stage pipeline and performs chip level multithreading for six threads to eliminate data forwarding and branch prediction. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a TDMA scheme, called memory wheel, similar to the TDMA based arbiter used in the JOP CMP system [15]. The SPARC ISA is extended with a *deadline* instruction that stalls the current thread until the deadline is reached. This instruction is used to perform time based, instead of lock based, synchronization.

The most problematic processor features for WCET analysis are the replacement strategies for set-associative caches [10]. A pseudo-round-robin replacement strategy of the 4-way set-associative cache in the ColdFire MCF 5307 effectively renders the associativity useless for WCET analysis. The use of a single 2-bit counter for the whole cache destroys age information within the cache sets. Slightly more complex pipelines, with branch prediction and out-of-order execution, need an integrated pipeline and cache analysis to provide useful WCET bounds. Such an integrated analysis is complex and also demanding with respect to the computational effort. Consequently, Heckmann et al. suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches; (2) locally deterministic update strategies for caches; (3) static branch prediction; and (4) limited out-of-order execution.

Reineke et al. analyzed the predictability of different cache replacement policies [18]. It is shown that the least recently used (LRU) policy performs best with respect to predictability. Pseudo-LRU and FIFO perform similarly. Both perform considerably worse than LRU.

3. Split data caches

For the cache analysis the addresses of the memory accesses need to be predicted. The addresses for the instruction fetch are easy to determine[1] and the access to stack allocated data, e.g. function arguments and local variables, is also quite regular. The addresses can be predicted when the call tree is known.

The addresses for heap allocated data are very hard to predict statically – the addresses are only known at runtime.[2] A data cache that caches heap and stack content suffers from the same problem as a unified instruction and data cache: an unknown address for a heap access will evict one block from all sets in the abstract cache state and will increase the age of all cache blocks. Therefore, we propose to split the data cache into caches for different memory areas.

3.1. Stack data

Access patterns to stack allocated data are different from heap or static allocated data. Addresses in the stack are usually easy to predict statically as the allocation addresses of stack frames can be predicted by the analysis of the call tree. Furthermore, a new stack frame for a function call does not need to be cache consistent with the main memory. The involved cache blocks need no cache fill from the main memory.

The regular access pattern to the stack cache will not benefit from set associativity. Therefore, the stack cache is a simple direct mapped cache. The stack contains local variables and the write frequency is higher than for other memory areas. The high frequency mandates a write back organization.

A stack cache is similar to a windowed register file. When the cache overflows, which can happen only during a call, the oldest frame or frames have to be moved to the memory. A frame only needs to be loaded from the memory when a function returns. A write back occurs first when the program reaches a call depth resulting in a wrap around within the cache. A cache miss can only occur when the program goes up in the call tree and needs access to a cache block that was evicted by a call down in the call tree.

On a return, the previously used cache blocks can be marked empty, as function local data is not accessible after the return. As a result, cache lines will never need to be written back on a cache wrap around after return. The stack cache activity can be summarized:

- A cache miss can only occur after a return. The first miss is at least *one cache size* away from a leaf in the call tree
- Cache write back can only occur after a function call. The first write back is *one cache size* away from the root of the call tree

[1] Assuming avoidance of function pointers and computed gotos.

[2] We found no publication that described analysis of the data cache for heap allocated data.

We can make the misses and write backs more predictable by forcing them to occur at explicit points in the call tree. At these points, the cached stack frames are written back to the main memory and the whole stack cache is marked empty. If we place the flush points at function calls in the call tree that are within *one cache size* from the leaf functions, all cache accesses into that area are guaranteed hits. This algorithm can actually improve WCET because most of the execution time of a program is spent in inner loops further down the call tree.

Stack data is usually not shared between threads and no cache coherence and consistence protocol – the major bottleneck for CMP scaling – needs to be implemented for a CMP system.

In C it is possible to generate non-regular stack access patterns that violate the described access rules, e.g., propagate stack allocated data to callees or other threads. The compiler can detect these patterns by escape analysis and can generate safe code, e.g. allocating this data on a second stack on the heap. This detection is also needed for the register allocation of local variables.

3.2. Static data

For conservatively written programs with statically allocated data, the address of the data is known after program linking. Value analysis results in a good prediction of read and write addresses. The addresses are the input for the cache analysis. In [6], control tasks from a real-time benchmark were analyzed. For this benchmark 90% of the memory accesses were predicted precisely.

Therefore, we propose to implement an additional cache that covers the address area of the static data, e.g., class fields in Java. The address range of the cache needs to be configurable and is set after program loading. As static data is shared between threads, a CMP must implement a cache coherence protocol.

3.3. Heap allocated data

In a modern object oriented language, data is usually allocated on the heap. The addresses for the objects are only known at runtime. It is possible to analyze local cache effects with unknown addresses for an LRU set-associative cache. For an n-way associative cache the history for n different addresses can be tracked. As the addresses are unknown, a single access influences *all* sets in the cache. The analysis reduces the effective cache size to a single set.

Even when using such a language in a conservative style, where all data is allocated during an initialization phase, it is not easy to predict the resulting addresses. The order of the allocations determines the addresses of the objects. When the order becomes unknown at one point in the initialization phase, the addresses for all following allocations cannot be determined precisely.

We propose to implement the cache architecture exactly as it results from this analysis – a small, fully associative cache with an LRU replacement policy. The emphasis of the data cache is on associativity instead of capacity. To avoid false positives in the analysis, the cache line will be a single word.

The LRU policy is difficult to calculate in hardware and only possible for very small sets. Replacement of the oldest block gives an approximation of LRU. The resulting FIFO strategy can be used for larger caches. To offset the less predictable behavior of the FIFO replacement [18], the cache has to be much larger than an LRU based cache.

3.4. Constants

In procedural languages, such as C, the constant area primarily contains string constants and is small. For object oriented languages, such as C++ and Java, the virtual methods tables and class related constants consume a considerable amount of memory. The addresses of the constants are known after program linking and are simple to predict for the WCET analysis. On a uniprocessor system the constant area and the static data can share the same cache.

For CMP systems, splitting the static data cache and the constant cache is a valuable option. In contrast to static data, constants are per definition immutable. Therefore, cache coherence and consistence needs not to be enforced and the resulting cache is simpler and can be made larger.

Another option for constants is to embed them into the code. To support this option a PC relative addressing mode is needed. However, this option is only practical for a few constants. Large virtual method dispatch tables would thrash the instruction cache. Furthermore, if the address range for the PC relative addressing is restricted, some tables would need to be duplicated, increasing the code size.

3.5. Scratchpad memory

A common method for avoiding data caches is an on-chip memory called scratchpad memory, which is under program control. This program managed memory entails a more complicated programming model, although it can be automatically partitioned [1, 25].

A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [17] and the allocation of data in the scratchpad memory [26, 24] can be optimized for the WCET.

Exposing the scratchpad memory at the language level can further help to optimize the time-critical path of the application. The Real-Time Specification for Java [3] introduces scoped memory, which represents a memory re-

gion for a limited lifetime allocation without garbage collection. This scoped memory model can be used to represent scratchpad memory at the language level. In [27] thread-local scoped memory is introduced for a CMP system to represent per processor local scratchpad memory. This local scratchpad memory also avoids cache coherence protocols.

4. Instruction cache

A new form of organization for the instruction cache, the *method cache*, which has a novel replacement policy, is proposed in [19]. A whole function or method is loaded into the cache on a call or return.[3] This cache fill strategy lumps all the cache misses of a function together. All instructions except call and return are guaranteed cache hits. Only the call tree needs to be analyzed for the cache analysis. With the proposed cache organization, the cache analysis can be performed independently of the pipeline analysis.

For a full method load into the cache we need to know the length of the method. This information is available in the Java class file. For a compiled C program this information can be provided in the executable.

The WCET analysis is simpler with the method cache than with a direct-mapped cache. We only have to consider invoke and return instructions and not all instructions in a cache line for a cache analysis. As methods in the cache need to span contiguous cache blocks, a least-recently used replacement strategy is impractical. The method cache implements a FIFO replacement strategy.

The method cache needs the whole program call graph and the whole program control flow graph for the analysis. Whole program analysis with a FIFO replacement policy can become computationally impractical. Therefore, a practical approximation of the cache works as follows [13]: Within a loop, all possibly invoked methods are tested if they will fit together into the cache. If this is the case, all methods cause at most one miss in the loop.

In order to use the method cache in a RISC processor, the ISA is extended with a prefetch instruction to force the cache load. The prefetch instruction can be placed immediately before the call or return instruction. It can also be scheduled earlier to hide the cache load latency.

5. Discussion

The proposed splitting of caches is intended to lower the WCET bound – a design decision for time-predictable processors. Optimizing the WCET performance hurts the average case performance [22]. A data cache for all data areas is, in the average case, more efficient than splitting the same amount of on-chip memory for different data types. Therefore, the presented solution is intended for systems where the WCET performance is of primary importance.

[3] For functions that do not fit into the cache we provide a fall-back mechanism, either with compiler support or with hardware support.

5.1. Scheduling

Cache analysis usually ignores the influence of context switches. However, on a context switch the cache data becomes usually invalid. One option to consider this effect in the schedulability analysis is to include the cost for a complete refill of the cache in the context switch time. For an implementation of time-predictable caches we can enforce this assumption in hardware. The cache content can be saved and restored on a thread switch in the same way as registers are saved and restored. Under the assumption that saving and restoring the whole cache in one burst is better analyzable than individual cache loads, this architecture can reduce the WCET of a thread switch.

5.2. Chip multiprocessors

CMP systems share the memory bandwidth between the on-chip processors and the pressure to avoid memory accesses is increased. Therefore, these systems call for large and processor local caches. Furthermore, some data needs to be held consistent between the processor local caches. Cache coherence and consistence protocols are expensive to implement and limit the number of cores in a multiprocessor system. However, not all data needs to be held coherent. Thread local data, such as stack allocated data, constant data, and instructions can be cached processor local without such a protocol. This simplification is another argument for splitting the cache for different memory areas.

For a CMP system with a processor local scratchpad memory, thread migration between cores is problematic. A possible solution is to save the scratchpad memory on a context switch into main memory and restore it on a different processor on rescheduling. The scratchpad memory increases the thread context and the context switch becomes more expensive.

This organization of the proposed cache for heap allocated data fits very well as a buffer for hardware transactional memory [12]. We will explore transactional memory as an alternative to cache coherency protocols for massively parallel CMP systems.

5.3. Implementation

We have implemented some of the proposed caches in the Java processor JOP [21] and in the CMP version of JOP [16]. JOP is intended to be a time-predictable platform for real-time Java programs. As JOP is an easy target

for WCET analysis, two independent WCET analysis tools are available for JOP [23, 7].

The instructions are cached in the method cache. As the misses are only possible at invoke or return instructions, the pipeline can be simplified considerably. Invoke and return are implemented in microcode and wait explicitly till the method is transferred into the method cache from the main memory. As all other instructions are guaranteed hits, stalling of the pipeline, with the possible impact on the maximum clock frequency, does not have to be considered.

The stack cache in a Java processor contains – besides the stack frames for the return address and the local variables – also the operand stack. For an efficient implementation of the stack machine (the Java virtual machine is a stack machine), the two top elements are implemented in discrete registers [20]. Exchange of data between these two registers and the on-chip stack cache is automatically performed in the execution stage. The exchange between the stack cache and the main memory is performed on a thread switch. This exchange increases the context switch time, but avoids possible cache fill and write back operation at method invocations.

For a JOP CMP system we have implemented a processor local scratchpad memory [27]. This memory is mapped to a thread-local scoped memory and therefore under program control. Measurements of a simple benchmark showed an average case performance increase of a factor of up to two for a 8 processor system. Integration of the thread-local memory into the WCET analysis [23] is considered as future work.

A fully associative cache for heap allocated data is currently under development by Wolfgang Puffitsch. First experiments in a field-programmable gate array indicate that the practical limit for the associativity is around 8. Beyond that point, the cache hit detection becomes the critical path in the design. Pipelining the hit detection increases the possible associativity at the expense of an additional cycle latency.

6. Conclusion

For real-time systems we need a time-predictable processor to allow WCET analysis with tight bounds. In this paper we discussed the organization of caches – an architectural feature of modern processors that increases the average case performance. Caches, and especially data caches, are hard to analyze with respect to the WCET. A single access to an unknown address destroys the abstract cache state for a whole set. Therefore, we propose to split the data cache into independent caches for different memory areas. When caching stack allocated data, static data, and heap allocated data in different caches, the cache states are easier to predict. Stack and static data accesses are not disturbed by heap accessing instructions. Furthermore, splitting the physical caches and the cache analysis for different data areas reduces the state that need to be tracked by the analysis. The analysis will scale better for larger programs.

Acknowledgement

This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD). The author would like to acknowledge the discussions with Wolgang Puffitsch, who is currently implementing and evaluating a fully associative data cache for JOP. Furthermore, I thank Albrecht Kadlec for his detailed review of the paper.

References

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*, pages 318–326, New York, Oct. 30 Nov. 01 2003. ACM Press.

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

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. J. Gil, and A. J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[5] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[6] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[7] T. Harmon and R. Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.

[8] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.

[9] 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, 1995.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[12] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.

[13] B. Huber and M. Schoeberl. Comparison of ILP and model checking based WCET analysis. Technical Report 72/2008, Institute of Computer Engineering, Vienna University of Technology, December 2008.

[14] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In E. R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[15] C. Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[16] C. Pitter and M. Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[17] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[18] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[19] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[20] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[21] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[22] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems (to appear)*, 2009.

[23] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[24] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 223–232. IEEE Computer Society, 2005.

[25] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst*, 14(8):802–815, 2006.

[26] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE2005).*, pages 600–605 Vol. 1, March 2005.

[27] A. Wellings and M. Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.