Towards Time-predictable Data Caches for Chip-Multiprocessors

Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber

Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at, wpuffits@mail.tuwien.ac.at,
benedikt@vmars.tuwien.ac.at

Abstract. Future embedded systems are expected to use chip-multiprocessors to provide the execution power for increasingly demanding applications. Multiprocessors increase the pressure on the memory bandwidth and processor local caching is mandatory. However, data caches are known to be very hard to integrate into the worst-case execution time (WCET) analysis. We tackle this issue from the computer architecture side: provide a data cache organization that enables tight WCET analysis. Similar to the cache splitting between instruction and data, we argue to split the data cache for different data areas. In this paper we show cache simulation results for the split-cache organization, propose the modularization of the data cache analysis for the different data areas, and evaluate the implementation costs in a prototype chip-multiprocessor system.

1 Introduction

With respect to caching, memory is usually divided into instruction memory and data memory. This cache architecture was proposed in the first RISC architectures [1] to resolve the structural hazard of a pipelined machine where an instruction has to be fetched concurrently to a memory access. The independent caching of instructions and data has also enabled the integration of cache hit classification of instruction caches into the worst-case execution time analysis (WCET) [2]. While analysis of the instruction cache is a mature research topic, data cache analysis is still an open problem. After n accesses with unknown address to a n-way set associative cache, the abstract cache state is lost.

In previous work we have argued about cache splitting in general [3]. We have argued that caches for data with statically unknown addresses shall be fully associative. In this paper we evaluate time-predictable data cache solutions in the context of the Java virtual machine (JVM). We provide simulation results for different cache organizations and sketch the resulting modular analysis. Furthermore, an implementation in the context of a Java processor shows the resource consumptions and limitations of highly associative cache organizations.

Access type examples are taken from the JVM implemented on the Java processor JOP [4]. Implementation details of other JVMs may vary, but the general classification of the data areas remains valid. Part of the proposed solution can be adapted to other object-oriented languages, such as C++ and C#, as well.

2 Data Areas and Access Instructions

The memory areas used by the JVM can be classified into five categories:

Method area The instruction memory that contains the bytecodes for the execution. On compiled Java systems this is the native code area

Stack Thread local stack used for stack frames, arguments, and local variables

Class information A data structure representing the different types. Contains the type description, the method dispatch table, and the constant pool.

Heap Garbage collected heap of class instances. The object header, which contains auxiliary information, is stored on the heap or in a distinct handle area.

Class variables Shared memory area for static variables.

Caching of the method area and the stack area have been covered in [5] and [6]. In this paper we are interested in a data cache solution for the remaining data areas. On standard cache architectures these memory areas and the stack memory share the same data cache.

2.1 Data Access Types

Data memory accesses (except stack accesses) can be classified as follows:

CLINFO Type information, method dispatch table, and interface dispatch table. The method dispatch table is read on virtual and static method invocation and on the return from a method. The method dispatch table contains two words per method. Bytecodes: new, anewarray, multianewarray, newarray, checkcast, instanceof, invokestatic, invokevirtual, invokespecial, invokeinterface, *return.

CONST Constant pool access. Is part of the class information. Bytecodes: ldc, ldc_w, ldc2_w, invokeinterface, invokespecial, invokestatic, invokevirtual.

STATIC Access to static fields. Is the class variables area. Bytecodes: getstatic, putstatic.

HEADER Dynamic type, array length, and fields for garbage collection. The type information on JOP is a pointer to the method dispatch table within CLINFO. On JOP each reference is accessed via one indirection, called the handle, to simplify the compacting garbage collection. The header information is part of the handle area. Bytecodes: getfield, putfield, *aload, *astore, arraylength, *aload, *astore, invokevirtual, invokeinterface.

FIELD Object field access. Is part of the heap. Bytecodes: getfield, putfield.

ARRAY Array access. Is part of the heap. Bytecodes: *aload, *astore.

2.2 Cache Access Types

The different types of data cache accesses can be classified into four classes w.r.t. the cache analysis:

- The address is *always* known statically. This is the case for static variables (STATIC), which are resolved at link time, and for the constant pool (CONST), which only depends on the currently executed method.

- The address depends on the dynamic type of the operand, but not on its value. Therefore, the set of possible addresses is restricted by the receiver types determined for the call site. The class info table, the interface table and the method table are in this category (CLINFO).
- The address depends on the value of the reference. The exact address is unknown, as some value on the managed heap is accessed, but in addition to the symbolic address a relative offset is known. Instance fields and array fields, both showing some degree of spatial locality, belong to this category (FIELD, ARRAY).
- The last category contains handles, references to the method dispatch table, and array lengths (HEADER). They reside on the heap as well, but we only know the symbolic address.

2.3 Cache Coherence

For a chip-multiprocessor system the cache coherence protocol is the major limiting factor on the scalability. Splitting data caches also simplifies the cache coherence protocol. Most data areas are actually constant (CLINFO, CPOOL). Access into the handle area (HEADER) is pseudo-constant. The data written into the header area during object creation and can not be changed by a thread. However, the garbage collector can modify this area. To provide a coherent view of the handle area between the garbage collector and the mutators, a cache for the handle area has to be updated or invalidated appropriately.

Data on the heap (FIELD, ARRAY) and in the static area (STATIC) is shared by all threads. With a write-through cache the cache coherence can be enforced by invalidating the cache on monitorenter and before reads from volatile fields.

3 Cache Benchmarks

Before developing a new cache organization we run benchmarks to evaluate memory access patterns and possible cache solutions. Our assumption is that the hit rate on the average case correlates with the hit classification in the WCET analysis, when different access types are cached independently. Therefore, we can reason about useful cache sizes from the benchmark results.

For the benchmarks we use two real-world embedded applications [7]: Kfl is one node of a distributed control application and Lift is a lift controller deployed in industrial automation. The Kfl example is a very static application, written in conservative, procedural style. The application Lift was written in a more object-oriented style. Furthermore, two benchmarks from an embedded TCP/IP stack (Udplp and Ejip) are used to collect performance data. Figure 1 shows the access frequencies for the different memory areas for all benchmarks.

There are no write accesses to the constant data areas and also no write access to the pseudo-constant area (HEADER). As we measure applications without object allocation at runtime, the data in the HEADER area is not mutated. The general trend is that load instructions dominate the memory traffic (between 89% and 93%).

Table 1. Data memory traffic to different memory areas (in % of all data memory accesses)

	Kfl		Lift		UdpIp		Ejip	
	load	store	load	store	load	store	load	store
CLINFO	31.2	0.0	7.4	0.0	14.4	0.0	10.7	0.0
CONST	11.4	0.0	2.6	0.0	13.8	0.0	12.3	0.0
STATIC	28.3	7.6	2.6	0.6	8.8	1.1	12.3	3.4
HEADER	14.3	0.0	50.5	0.0	39.1	0.0	39.4	0.0
FIELD	0.0	0.0	24.9	0.8	6.3	1.8	6.4	1.0
ARRAY	3.9	3.2	4.7	5.7	10.7	4.0	10.6	4.0

For the Kfl application there are no field accesses (FIELD). Dominating accesses are to static fields (STATIC), static method invocation (CLINFO), and access to the constant pool (CONST). The rest of the accesses are related to array accesses (HEADER, ARRAY). The Lift application has a quite different access pattern: instance field accesses dominate all reads (FIELD and HEADER). There are less methods invoked than in the Kfl application and less static fields accessed. The array access frequency of both applications is similar (4%–5%), for the TCP/IP benchmark, due to many buffer manipulations, considerable higher (11% loads).

3.1 Cache Simulations

As first step we simulate different cache configurations with a software simulation of JOP (JopSim) and evaluate the average case hit count.

Handle Cache As all operations on objects and arrays need an indirection through the handle we first simulate a cache for the handle. The address of the handle is not known statically, therefore we assume a small fully-associative cache with LRU replacement policy. The results of the cache is shown in Table 2 for different sizes. The size is in single words. Quite interesting to note is that even a single entry cache provides a hit rate for the handle indirection access of up to 72%. Caching a single handle should be so simple, so a single cycle hit detection including a memory read start in the same cycle should be possible. In that case, even a uniprocessor JOP with a two cycle memory read will gain some speedup. A size of just 8 entries results in a reasonable hit rate between 84% and 95%.

Constants and the Method Table Mixing access to the method table and access to the constant pool in one direct mapped cache is an option when the receiver types can be determined precisely. However, if the set of possible receiver types is large, the analysis becomes less precise. Therefore, we evaluate individual caches for the constant pool access (CPOOL) and the access to the method table (CLINFO).

Table 3 shows that a small direct-mapped cache of 512 words (2 KB) gives a hit rate of 100%. Keeping the cache sizes small is important for our intended system. We

Table 2. Hit rate of a handle cache, fully associative, LRU replacement

	Hit rate (%)			
Size	Kfl	Lift	UdpIp	Ejip
1	72	15	43	69
2	82	20	80	78
4	84	94	87	82
8	88	95	91	84
16	92	95	94	84
32	95	95	96	86

Table 3. Hit rate of a constant pool cache, direct mapped

	Hit rate (%)			
Size	Kfl	Lift	UdpIp	Ejip
32	68	69	77	82
64	96	69	79	95
128	98	69	88	95
256	100	100	100	95
512	100	100	100	100

are targeting chip-multiprocessor systems with private caches, even for accesses to constants, to keep the individual tasks time-predictable. A shared cache would not allow to perform any cache analysis of individual tasks.

The hit rate of a direct-mapped cache for the method table (MTAB) shows a similar behavior as the constant pool caching, as shown in Table 4. A size of 256 words gives a hit rate between 95% and 100%. It has to be noted that the method table is accessed by static and virtual methods. While the MTAB entry is known statically for static methods, the MTAB entry for virtual methods depends on the receiver type. If data-flow analysis can determine most of the receiver types the combination of a single cache for the constant pool and the method table is an option further to explore.

Static Fields Table 5 shows the results for a direct mapped cache for static fields. For object-oriented programs (represented by Lift), this cache can be kept very small. Although the addresses are statically known as the addresses for the constants, a combination of these two caches is not useful. Static fields need to be kept cache coherent, constant pools entries are implicitly cache coherent. Cache coherence enforcement, with cache invalidation at synchronized blocks, limits the hit rate in Udplp and Ejip.

Object Fields Addresses of object fields are unknown for the analysis. Therefore, we can only attack the analysis problem via a high associativity. Table 6 shows hit rates of fully-associative caches with LRU replacement policy. For the Lift benchmark we

Table 4. Hit rate of a method table cache, direct mapped

	Hit rate (%)			
Size	Kfl	Lift	UdpIp	Ejip
32	64	83	62	49
64	85	83	77	74
128	91	100	85	93
256	100	100	97	95

Table 5. Hit rate of a static field cache, direct mapped

	Hit rate (%)			
Size	Kfl	Lift	UdpIp	Ejip
32	76	100	33	77
64	85	100	33	77
128	99	100	33	77
256	100	100	33	77

observe a moderate hit rate of 88% for a very small cache of just 8 entries. UdpIp and Ejip saturate at 8 entries due to cache invalidation during synchronized blocks of code.

3.2 Summary

From the experiments with simulation of different caches for different memory areas we see that quite small caches can provide a reasonable hit rate. However, as the memory access latency for a CMP system with time-sliced memory arbitration can be quite high,[1] even moderate cache hit rates are a reasonable improvement.

4 Cache Analysis

In the following section we sketch cache analysis as it will be performed in a future version of our WCET analysis tool [8]. We leverage the cache splitting of the data areas for a modular analysis, e.g., analysis of heap allocated objects is independent from analysis of the cache for constants or cache for static fields.

In multithreaded programs, it is necessary to invalidate the cache when entering a synchronized block or reading from volatile variables.[2] We require that accesses to shared data are properly synchronized, which is the correct way to access shared data in Java. In this case it is safe to assume that object references on the heap are not

[1] Our 8 core CMP prototype with a time slot of 6 cycles per core has a worst-case latency of 48 cycles.

[2] The semantics of volatile variables in the Java memory model is similar to synchronized blocks: the complete global state has to be locally visible before the read access. Simply bypassing the cache for volatile accesses is not sufficient.

Table 6. Hit rate of an instance field cache, fully associative, LRU replacement

	Hit rate (%)			
Size	Kfl	Lift	UdpIp	Ejip
1	84	17	47	9
2	84	75	59	13
4	84	86	65	18
8	84	88	67	20
16	84	88	67	20
32	84	88	67	20

changed by another thread at arbitrary points in the program, resulting in a significantly more precise analysis. The effect of synchronization, namely invalidating some of the caches, has to be taken into account though.

The running example is illustrated in Figure 1 and was taken from the *Lift* application. The figure comprises the source code of the method checkLevel and the corresponding control flow graph in static single assignment (SSA) form. Each basic block is annotated with the cache accesses it triggers.

4.1 Static and Type-Dependent Addresses

If we only deal with statically known addresses in a data cache, the standard cache hit/miss classification (CHMC) for instruction caches delivers precise results and is therefore a good choice [9]. In the example, there is only one static variable, LEVEL_POS. If we assume a direct-mapped cache for static variables, and a separate one for values on the heap, all but the first access to the field will be a cache hit every time checkLevel is executed.

When the address depends on the type of the operand, we have to deal with a set of possible addresses. The straight forward extension of CHMC to sets of memory addresses is to update the abstract cache state for each possible address, and then join the resulting states. This leads to a very pessimistic classification when dynamic dispatch is used, however, and therefore is only acceptable if the exact address is known for most references.

4.2 Persistence Analysis

If dynamic types are more common, a more promising approach is to classify program fragments, or partitions, where it is known that one or all memory addresses are locally persistent. If this is the case, they will be missed at most once during one execution of the program fragment.

For both direct-mapped and N-way set associative caches with LRU replacement, a dataflow analysis for persistence analysis has been introduced in [10]. For FIFO caches, the concept of persistence is useful as well, but it is not safe anymore to assume that a persistent address will be loaded at the first access.

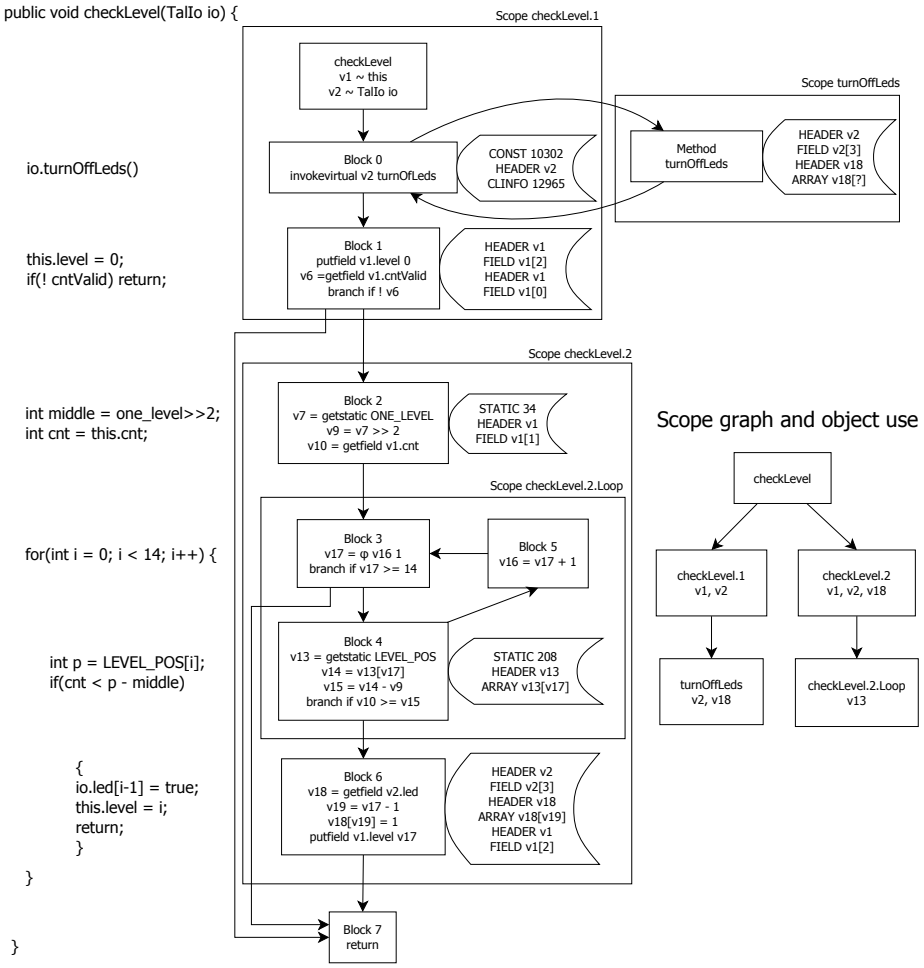

Fig. 1. Cache Analysis Example

Most work on persistence analysis focusses on dataflow equations and global persistence, leaving out some aspects which deserve more attention. Persistence for the whole program is rare and only of theoretical interest. We therefore identify a set of nested scopes [11], and identify for each scope which cache lines or cache sets are locally persistent. A scope is a subgraph of the control flow graph which represents a set of execution sequences. Methods, loops and loop bodies are typical examples of scopes, but partitions of less regular shape are possible as well. To reduce the amount of analysis work, persistence is checked in a bottom-up manner, starting at the leaves of the scope nesting graph. In the example we partitioned the flow graph of checkLevel into two scopes, the first of which contains the method turnOffLeds, while checkLevel.2.Loop is a subscope of the second one.

4.3 Object Header and Fields

As the address of the object header and the object fields depends on the instance and is not known at compile time, we use small, fully associative caches to track the cache state. There is usually a high number of handle accesses in object-oriented programs, but many of them do not change often. In our architecture, the object header is fully transparent at the bytecode level, and is managed by the runtime system. Caching is hence expected to gain a substantial benefit. On other platforms, which compile the bytecode and hold handles and array length values in registers, caching those values is probably less beneficial.

To calculate the symbolic addresses of object headers and fields used in some scope, the data dependencies of the control flow graphs in SSA form are analyzed. In SSA form, each variable is only defined once. If the definition is of the form $v = \phi(v_1, v_2)$, the definition is called a ϕ node, and the value of v is either that of v_1 or v_2.

For each object header used, those data depenencies which are defined in the same scope, and might be executed more than once within the scope, are identified. If all of those definitions are neither ϕ nodes nor depend on an indeterministic instruction, the variable representing the object corresponds to a unique symbolic address.

Finally, if all references used within a scope correspond to a unique symbolic address, we are able to perform a local persistence analysis. Additionally, using a variant of the global value numbering technique used in optimizing compilers [12], the quality of the analysis is further improved by identifying variables mapping to the same symbolic address.

In the running example, no handle has a data dependency on a ϕ node, and therefore persistence analysis is relatively simple. If a fully associative cache with four cache lines is used, all object headers of scope checkLevel are locally persistent. If the object header cache only has two entries, at least those headers used in scope turnOffLeds and checkLevel.2.loop are locally persistent.

5 Cache Implementation

We have implemented various forms of caches in the context of the Java processor JOP [4]: (1) a small fully associative cache with LRU replacement, (2) a fully associative cache with FIFO replacement, and (3) a direct mapped cache. We have combined the different caches to distinguish between different data areas.

5.1 LRU and FIFO Caches

The crucial component of an LRU cache is the tag memory. In our implementation it is organized as a shift register structure to implement the aging of the entries (see Figure 2). The tag memory that represents the youngest cache entry (cache line 0) is fed by a multiplexer from all other tag entries and the address from the memory load. This multiplexer is the critical path in the design and limits the maximum associativity.

Table 7 shows the resource consumption and maximum frequency of the LRU and FIFO cache. The resource consumption is given in logic cells (LC) and in memory

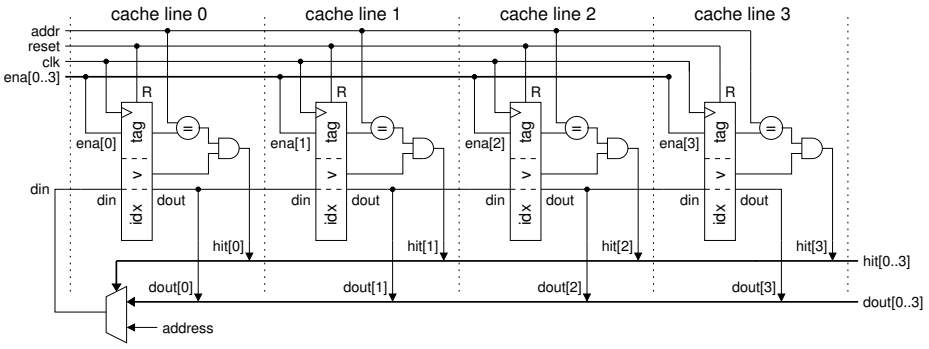

Fig. 2. LRU tag memory implementation

bits. As a reference, a single core of JOP consumes around 3500 LCs and the maximum frequency in the Cyclone-I device without data caches is 88 MHz. We can see the impact on the maximum frequency of the large multiplexer in the LRU cache on configurations with a high associativity.

Table 7. Implementation results for LRU and FIFO based data caches

	LRU Cache			FIFO Cache		
Associativity	LC	Memory	Fmax	LC	Memory	Fmax
16-way	783	0.5 KBit	102 MHz	633	0.5 KBit	119 MHz
32-way	1315	1 KBit	81 MHz	1044	1 KBit	107 MHz
64-way	2553	2 KBit	57 MHz	1872	2 KBit	94 MHz
128-way	4989	4 KBit	36 MHz	3538	4 KBit	89 MHz
256-way	10256	8 KBit	20 MHz	9762	8 KBit	84 MHz

The implementation of a FIFO replacement strategy avoids the change of all tag memories on each read. Therefore, the resource consumption is less than for an LRU cache and the maximum frequency is higher. However, hit detection still has to be applied on all tag memories in parallel and one needs to be selected.

5.2 Split Cache Implementation

We have combined a direct mapped cache and an LRU cache with one JOP core. The LRU cache stores the object header and the object fields; the direct mapped cache stores class info, constants, and static fields; array data is not cached.

Table 8 shows the resources and the maximum system frequency of different cache configurations. The first line gives the base numbers without any data cache. From the resource consumptions we can see that a direct mapped cache is cheap to implement. Furthermore, the maximum clock frequency is independent of the direct mapped cache

Table 8. Implementation results for a split cache design

Cache size		DM Cache		LRU Cache		System		
DM	LRU	LC	Memory	LC	Memory	LC	Memory	Fmax
0 KB	0	0	0 KBit	0	0 KBit	3530	61 KBit	88 MHz
1 KB	8	199	12 KBit	515	0.25 KBit	4731	73 KBit	85 MHz
2 KB	16	199	23.5 KBit	1045	0.5 KBit	5142	85 KBit	85 MHz
4 KB	32	172	46 KBit	1369	1 KBit	5344	108 KBit	81 MHz
8 KB	64	171	90 KBit	3235	2 KBit	7257	153 KBit	79 MHz

size. A highly associative LRU cache (i.e., 32-way and more) dominates the maximum clock frequency and consumes considerable logic resources.

6 Related Work

Early work on data cache access classification by White et al. focusses on computing addresses and analyzing array access patterns [13]. It is assumed, however, that the exact memory accesses can be resolved. Ferdinand et al. [10] discuss the use of dataflow analysis for data cache analysis. They suggest to use persistence analysis to deal with memory accesses which reference one out of a set of possible addresses.

To overcome the problems with unknown memory addresses, Lundquist et al. [14] suggest to distinguish unpredictable and predictable memory accesses to improve the analysis of data caches. If an address cannot be resolved at compile time, accesses to that address are considered as unpredictable. Those data structures which might be accessed by unpredictable memory accesses are marked for being moved into an uncached memory area. Vera et al. [15] lock the cache during accesses to unpredictable data . The locking proposed there affects all kinds of memory accesses though, and therefore is necessarily coarse grained.

7 Conclusion

Chip-multiprocessor systems increase the pressure on the memory bandwidth and caching of instructions and data is mandatory. In order to estimate tight WCET values, we propose to split data caches for different data areas. Benchmarking of embedded applications show possible tradeoffs between achievable hit rates and sizes of the different caches. Splitting the data cache for different access types (e.g., constant pool and heap) allows to modularize the cache analysis. Furthermore, unknown addresses of one data type access have no impact on data accesses of a different type. Caches for data where the address is not known statically (e.g., heap allocated data), can only be analyzed when the cache has a very high associativity. From our prototype implementation within an FPGA we conclude that LRU caches scale up to an associativity of 16 and FIFO caches up to an associativity of 64.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

References

1. Patterson, D.A.: Reduced instruction set computers. Commun. ACM **28**(1) (1985) 8–21
2. Arnold, R., Mueller, F., Whalley, D., Harmon, M.: Bounding worst-case instruction cache performance. In: Proceedings of the Real-Time Systems Symposium 1994. (December 1994) 172–181
3. Schoeberl, M.: Time-predictable cache organization. In: Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009), Tokyo, Japan, IEEE Computer Society (March 2009)
4. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture **54/1–2** (2008) 265–286
5. Schoeberl, M.: 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., Agia Napa, Cyprus, Springer (October 2004) 371–382
6. Schoeberl, M.: Design and implementation of an efficient stack machine. In: Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005), Denver, Colorado, USA, IEEE (April 2005)
7. Schoeberl, M.: Application experiences with a real-time Java processor. In: Proceedings of the 17th IFAC World Congress, Seoul, Korea (July 2008)
8. Huber, B.: Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria (2009)
9. Theiling, H., Ferdinand, C., Wilhelm, R.: Fast and precise WCET prediction by separated cache and path analyses. Real-Time Syst. **18**(2/3) (2000) 157–179
10. Ferdinand, C., Wilhelm, R.: On predicting data cache behavior for real-time systems. In: LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, London, UK, Springer-Verlag (1998) 16–30
11. Engblom, J., Ermedahl, A.: Modeling complex flows for worst-case execution time analysis. In: RTSS '00: Proceedings of the 21st IEEE Real-Time Systems Symposium, Los Alamitos, CA, USA, IEEE Computer Society (December 2000) 163–174
12. Click, C.: Global code motion/global value numbering. SIGPLAN Not. **30**(6) (1995) 246–257
13. White, R.T., Mueller, F., Healy, C., Whalley, D., Harmon, M.: Timing analysis for data and wrap-around fill caches. Real-Time Syst. **17**(2-3) (1999) 209–233
14. Lundqvist, T., Stenström, P.: A method to improve the estimated worst-case performance of data caching. In: RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, Washington, DC, USA, IEEE Computer Society (1999) 255–262
15. Vera, X., Lisper, B., Xue, J.: Data caches in multitasking hard real-time systems. In: RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (2003) 154–165