A Time-predictable Object Cache

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
Email: masca@imm.dtu.dk

***Abstract*—Static cache analysis for data allocated on the heap is practically impossible for standard data caches. We propose a distinct object cache for heap allocated data. The cache is highly associative to track symbolic object addresses in the static analysis. Cache lines are organized to hold single objects and individual fields are loaded on a miss. This cache organization is statically analyzable and improves the performance. In this paper we present the design and implementation of the object cache in a uniprocessor and chip-multiprocessor version of the Java processor JOP.**

***Keywords*-real-time systems; time-predictable computer architecture; worst-case execution time analysis**

I. Introduction

One of the big challenges in worst-case execution time (WCET) analysis are are hit and miss predictions of data caches. Access to program data, cached in data caches, is harder to predict than instruction cache hits. The addresses of the data, which are mandatory for classic cache analysis, are not always know statically. Especially for object-oriented programing languages most of of the data is allocated during runtime on the heap. The addresses of those objects are not known statically. Therefore, a single access to such an object destroys the abstract cache state of one way of the cache in classic cache analysis.

To solve this analysis issue for object-oriented languages we propose a dedicated cache for heap allocated objects. This cache has two advantages:

1) Access to heap allocated objects do not disturb the cache analysis of stack allocated or static data.
2) The object cache can be organized to allow WCET analysis by symbolic tracking of object field accesses.

The proposed object cache is organized to cache single objects in a cache line. The cache is highly associative to track object field accesses via the symbolic reference instead of the actual address.

Object-oriented programming is considered problematic for WCET analysis. The analysis of a C++ virtual function dispatch with the WCET tool aiT resulted in cache miss predictions (about 40 cycle on a PowerPC 755 for each miss) of two dependent memory loads.[1] The first memory reference goes into the heap to determine the object type and the second into the virtual function dispatch table. On a standard data cache the address of the first access is unknown and destroys the abstract cache state of one way. The second access is type dependent and to determine the address, the WCET tool needs exact knowledge how the compiler organizes virtual function dispatch tables.

Our approach is to enable tight WCET analysis of object oriented programs by designing a cache architecture that avoids the former described issues on standard cache organizations. The proposed object cache is one example of a computer architecture that is optimized for real-time systems.

The paper is organized as follows: in the following Section proposals of object caches are reviewed. Section III describes the design of the proposed object cache. In Section IV the benefits for the WCET analysis of the object cache is described. In Section V the object cache is evaluated in the context of the Java processor JOP. Section VI concludes the paper.

II. Related Work

One of the first proposals of an object cache [14] appeared within the Mushroom project [15]. The Mushroom project investigated hardware support for Smalltalk-like object oriented systems. The cache is indexed by a combination of the object identifier (the reference in the Java world) and the field offset. Different combinations, including xoring of the two fields, are explored to optimize the hit rate. The most effective generation of the hash function for the cache index was the xor of the upper offset bits (the lower bits are used to select the word in the cache line) with the lower object identifier bits. Considering only the hit rate, caches with a block size of 32 and 64 bytes perform best. However, under the assumption of realistic miss penalties caches with 16 and 32 bytes lines size result in lower average access times per field access. This result is a strong argument against just comparing hit rates.

With an indirection based access to an object two data structures needs to be cached: the actual object and the indirection to that object. In the paper a common cache for both data structures and a split cache is investigated. As the handle indirection cache is only accessed when the object cache results in a miss, a medium hit rate on the handle indirection cache is sufficient. Therefore, the best

[1] Private communication with Reinhard Wilhelm

configuration is a large object cache and a small handle indirection cache.

A dedicated cache for heap allocated data is proposed in [13]. Similar to our proposed object cache, the object layout is handle based. The object reference with the field index is used to address the cache – it is called virtual address object cache. Cache configurations are evaluated with a simulation in a Java interpreter and the assumption of 10 ns cycle time of the Java processor and a memory latency of 70 ns. For different cache configurations (up to 32 KB) average case field access times between 1.5 and 5 cycles are reported. For most benchmarks the optimal block size was found to be 64 bytes, which is quite high for the medium latency (7 cycles) of the memory system. The proposed object cache is also used to cache arrays, whereas our object cache is intended for *normal* objects only. Therefore, the array accesses favor a larger block size to benefit from spatial locality. Object access and array access are quite different from the WCET analysis point of view. The field index for an object access is statically known, whereas the array index usually depends on the loop iteration count.

Wright et al. propose a cache that can be used as object cache and as conventional data cache [17]. To support the object cache mode, the instruction set is extended with a few object oriented instructions such as load and store of object fields. The object layout is handle based and the cache line is addressed with a combination of the object reference (called object id) and part of the offset within the object. The main motivation of the object cache mode is in-cache garbage collection of the youngest generation [16].

All proposed object caches are optimized for average case performance. It is common to use a hash function by xoring part of the object identifier with the field offset in order to equally distribute object within the cache. However, this hash function defeats WCET analysis of the cache content. In contrast, our proposed object cache is designed to maximize the ability to track the cache state in the WCET analysis [3].

III. Object Cache Design

The proposed object cache is based on the idea that the data cache shall be split to cache different data areas within their own caches [8]. These caches can be optimized for the typical access patterns and for the WCET analysis.

A. Split Data Cache

In the first RISC processors the cache was split into an instruction cache and a data cache to decouple the instruction fetch in the first pipeline stage from the data load and store in a later pipeline stage. All current processors have a split cache at the first level in the cache hierarchy. This splitting of the cache enabled independent cache analysis for instructions and data. Instruction caches are relative easy to analyze as the fetch addresses are statically know. Access to static data and the caching of those data is also analyzable. However, the data cache serves for several quite different data areas: static data (scalars and arrays), stack allocated data (local variables and stack allocate objects), constants, method dispatch tables, and heap allocated data (objects and arrays). All those different data areas have different properties with respect to WCET analysis: e.g., loads of constants are simple to predict, whereas access to heap allocated data is hard to predict. Furthermore, different data areas benefit from different cache organizations: many data areas are sensible to accesses that conflict in the cache block placement, where access to stack allocated data is practically immune to those conflicts. The former data areas benefit from a higher associativity, whereas stack allocated data is served well from a direct mapped cache.

Considering these differences in data access patterns and also different WCET analysis properties we argue that the data cache shall also be split for different memory areas [8]. Each data area has its own cache. The distinction between the different accesses is easy on a Java processor as the JVM bytecodes contain more semantic information than loads and stores on a standard processor. On a standard processor the differentiation between different memory areas needs cooperation between the compiler and the memory management unit (MMU). Different accesses have to be mapped to different (virtual) addresses and the MMU needs to be configured with this address mapping to correctly redirect the accesses to the different caches.

With chip-multiprocessing (CMP) cache organization becomes further complicated. With a shared memory system core local caches need to be hold consistent with the other cores and the main memory. A cache coherence protocol is responsible for the coherent and consistent view of the global state in the main memory for all cores. This cache coherence protocol is also the major limiting factor for CMP scaling. However, not all data areas need a cache coherent view of the main memory: constants are read only and are implicit coherent; stack data is usually thread local[2] and needs no cache coherence protocol.

B. Object Cache Organization

In a modern object-oriented language, data (objects and arrays) is allocated on the heap. The addresses for these objects are only known at runtime. Therefore, static WCET analysis cannot classify accesses to those object fields as hits or misses. However, it is possible to statically analyze local cache effects with unknown addresses for a set-associative cache. For an n-way set associative cache the history for n different addresses can be tracked symbolically. As the addresses are unknown, a single access influences *all* sets

[2]The exception is leaking a reference to stack allocated data to a concurrent thread. This *violation* of stack local data can be detected by a compiler and the shared data can be allocated on a second stack that is placed in the cache coherent data area.

of one way in the cache. The analysis reduces the effective cache size to a single set.

The proposed object cache architecture is optimized for WCET analysis instead of average case performance. To track individual cache lines symbolically, the cache is fully associative. Without knowing the address of an object, all cache lines in one way map to a single line in the analysis. Therefore, the object cache contains just a single line per way. Instead of mapping blocks of the main memory to those lines, whole objects are mapped to cache lines. For this mapping the reference to the object, instead of the object's address, is used for the tag memory. This organization is similar to a virtual memory cache. For a handle based object layout, as it is the case on JOP, usage of the reference to access the cache avoids the handle indirection on a cache hit.

The index into the cache line is the field index. To compensate for the resulting small cache size with one cache line per way, we propose to use quite large cache lines. To reduce the resulting large miss penalty, only the missed word in the cache line is filled on a miss. To track which words of a line contain a valid entry, one valid bit per word is added to the tag memory.

As the object cache is organized to cache a whole object per cache line, each cache line can only contain a single object. Objects cannot cross cache lines. If the object is bigger than the cache line, the fields at higher indexes are not cached. While this might sound like a drastic restriction, it is the only way to keep the cache content WCET analyzable for data with statically unknown addresses. That this design is still efficient is shown in the evaluation section.

Furthermore, the cache organization is optimized for the object layout of JOP. The objects are accessed via an indirection called the handle. This indirection simplifies compaction during garbage collection. The tag memory contains the pointer to the handle (the Java reference) instead of the effective address of the object in the memory. Furthermore, the object cache reduces the overhead of using handles. If an access is a hit, the cost for the indirection is zero – the address translation has been already performed.

The effective address of an object can only be changed by the garbage collection. For a coherent view of the object graph between the mutator and the garbage collector, only a cached address of an object needs to be updated or invalidated after the move. The cached fields, however, are not affected by changing the object's address, and can stay in the cache.

To simplify static WCET analysis the cache is organized as write-through cache. Write back is harder to analyze statically, as on each possible miss an additional write back needs to be accounted for in the analysis. In a recent paper [1] it has been explained that write-back caches are handled as follows: on a cache miss it is assumed that the cache line that needs to be loaded is dirty and needs to be written back. Current WCET analysis tools do not track the dirty bit of a cache line. Therefore, write-back caches add considerable overhead – or conservatism – to the WCET analysis. In [1] it is recommended to use write-through caches. Furthermore, a write-through cache simplifies the cache coherence protocol for a chip multiprocessor (CMP) system [6].

As objects cannot cross cache lines, the number of words per cache line is one important design decision. For a fully associative cache the tag and comparator implementation consumes considerable resources, whereas the data memory can be implemented in on-chip memory, which is cheap. Therefore, cache lines for 8, 16, or even 32 words are a reasonable design. Furthermore, the layout of the fields in the object can be optimized by the compiler to place often accessed fields at lower indexes.

The object cache is only used for objects and not for arrays. The access behavior for array data is quite different as it explores spatial locality instead of temporal locality. Therefore, we believe that a cache organized as a small set of prefetch buffers is more adequate for array data. As arrays use a different set of bytecodes and are thus distinguishable from ordinary objects, access to arrays can be redirected to their own cache. The details of a time-predictable cache organization for array data is not considered in this paper.

C. Cache Coherence

Standard cache coherence and consistence protocols are expensive to implement and limit the number of cores in a multiprocessor system. The Java memory model (JMM) [2] allows for a simple form of cache coherence protocol [6]. With a write-through cache, the caches can be held consistent according to the rules of the JMM by invalidating the cache on start of a synchronized block or method (bytecode monitorenter) or a read from a volatile variable. In the implementation within JOP we have added a microcode instruction to invalidate the cache.

D. Cache Design

Figure 1 shows the design of the object cache. In this example figure the associativity is two and each cache line is four fields long. All tag memories are compared in parallel with the object reference. Therefore, the tag memory uses dedicated registers and cannot be built from on-chip memory. Parallel to the tag comparison, the valid bits for the individual fields are checked. The field index performs the selection of the valid bit multiplexer. The output of the tag comparisons and valid bit selection is fed into the encoder, which delivers the selected cache line. The line index and the field index are concatenated and build the address of the data cache. This cache is built from on-chip memory. As current FPGAs do not contain asynchronous memories, the input of the data memory contains a register. Therefore, the cache data is available one cycle later. The hit is detected in

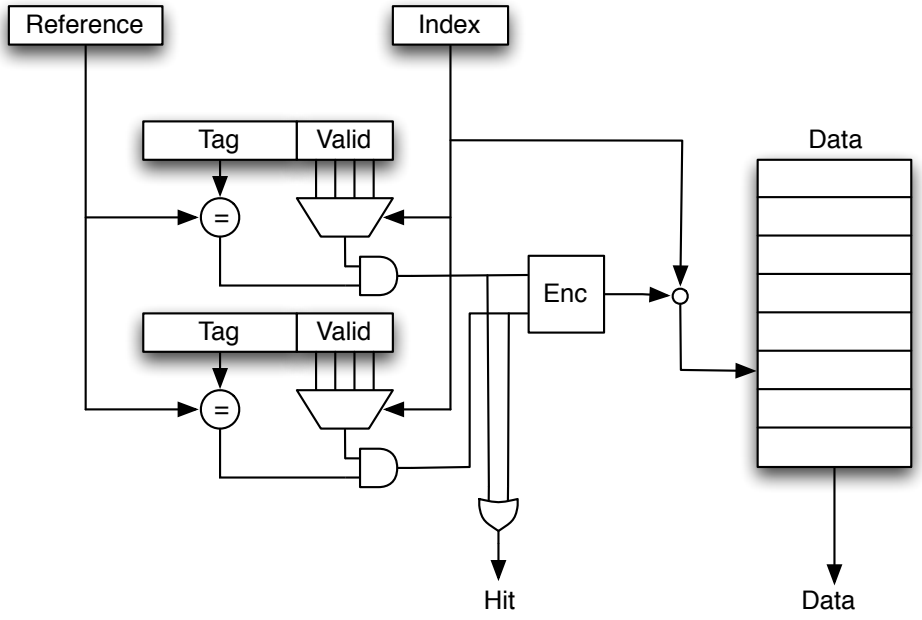

Figure 1. Object cache with associativity two and four fields per object

the same cycle as reference and index are available in the pipeline, the data is available one cycle later.

A cache line is allocated and the tag valid if at least one valid bit is set (i.e., at least one field is in the cache). When the tag is a hit, but the field is not (yet) in the cache it is loaded from main memory, put into the data memory of the cache, and the valid bit is set. This cache update is performed on a missed getfield and on a missed putfield. If the tag comparison is a miss, a new cache line is allocated on a getfield, but not on a putfield. The rational for this difference is as follows: The scare resource of cache ways (tags) is not spent on a single write, when the object is not yet in the cache. If the object has already an allocated tag, updating the cache on a putfield is for free.

The cache lines (tags) are allocated in first-in first-out (FIFO) order. FIFO allocation is simpler in hardware than least recently used (LRU) order.

IV. WCET Analysis

The WCET analysis of the object cache [3] is lexical scope based. If all references to the objects and their fields fit into the object cache a single miss for each access is assumed. In case that the lexical scope contains a loop with iteration count n, $n-1$ accesses are classified as hits. If not all accessed objects fit into the cache, all accesses are classified as misses. The analysis starts with the innermost scopes and enlarges the scopes until the maximum size with hit classification is reached.

For this type of analysis the cache replacement can be LRU or FIFO. It is important that no conflict misses can happen. Therefore, we can choose to implement the cheaper FIFO policy. This lexical scope based analysis is cheap and still effective [3]. It has to be noted that this type of analysis is only feasible for processors without cache related timing anomalies [4], where a cache hit can result in a higher overall WCET. As JOP is free from such types of timing anomalies, we can use this compositional approach to WCET analysis of caches.

In [3] we have integrated the object cache analysis in the WCET analysis tool of JOP [11]. By analyzing the influence of the object cache organization on the resulting predicted miss cycles we where able to show that the object cache, as it is proposed in this paper, is the most efficient solution. We also evaluated the influence on the main memory system (short latency with SRAM and longer latency, but high bandwidth with SDRAM) on the resulting miss cycles per field read. Even for a SDRAM memory loading single fields, or a small number of fields, on a miss is more beneficial than loading a full cache line. The main cache effect for object field accesses comes from temporal locality; spatial locality plays a marginal role.

The actual execution time of a getfield in the implementation of JOP depends on the main memory access time. For a memory access time of n cycles a getfield with a cache miss takes

$$t_{getfield\_miss} = 6 + 2n$$

cycles. Two memory access times are needed, as the handle indirection and the actual field value have to be read. If the

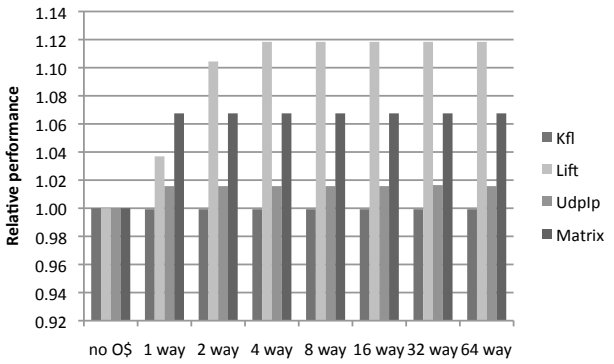

Figure 2. Performance impact of different associativities with a line size of 8 fields

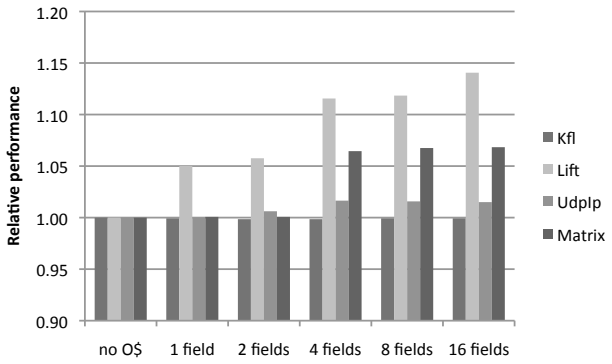

Figure 3. Performance impact of different line sizes with an associative of 4

access is a cache hit, the execution time of getfield is

$$t_{getfield\_hit} = 5$$

cycles. Besides not accessing the main memory at all, another cycle is saved that is needed between the handle indirection and field read to move the data from the memory read to the memory address. For a SRAM based main memory with 2 cycles access time, as it is used in the evaluation, a missed getfield consumes 10 cycles, double the time as in the hit case.

To verify the design decisions we have also evaluated the proposed object cache by executing large Java benchmarks (DaCapo) with cross-profiling [9]. With a simulation of the object cache we where able to compare the high-associative organization with a standard cache organization. The results are similar to the results from WCET based analysis: only temporal locality effects are important for the cache and single field loads are most efficient. The object cache is almost as efficient as a direct mapped cache in the average case, but is WCET analyzable.

V. Evaluation

We have implemented the proposed object cache in the Java processor JOP [7] and the CMP version of JOP [5]. In the original JOP architecture instructions are cached in the method cache and stack allocated data in the stack cache. Access to heap allocated data, the constant pool, and the method dispatch table go directly to the main memory. For the evaluation the object cache has been added to JOP. The other data areas are not cached. The results are based on the implementation in an FPGA from Altera, the Cyclone EP1C12.

The object cache is integrated in the memory management unit (MMU) that is responsible for bytecodes that access the main memory. The hit detection is performed in a single cycle and on a hit the processor pipeline can continue without a stall.

For the evaluation we use several benchmarks from the benchmark suit JemBench [10]. *Kfl* is a benchmark, derived from a real-world application that tilts up a railway contact wire. The application was written in a very conservative style using mostly static data and static methods. Therefore, we do not expect a performance improvement by caching objects. *Lift* is derived from a real-world application that controls an industrial lift. This application is programmed in a more object oriented style. Although all data is preallocated at the application start to avoid influence of the garbage collection, objects are used at runtime. *UdpIp* is is a benchmark derived from the implementation of a simple UDP/IP stack. The UDP/IP stack is in industrial use on a railway device. The *Matrix* benchmark measures the performance of matrix multiplication. It is an artificial benchmark to measure CMP speedup.

We have synthesized several different configurations of the object cache with a uniprocessor version and a CMP version of JOP. The object cache has two configurable parameters: 1.) the number of ways (cache lines) and 2.) the number of cached fields in one cache line. We have varied the number of ways between 1 and 64, and the number of fields per cache line between 1 and 16. With parameter values as power of two and uniprocessor vs. CMP configuration of JOP this resulted in more than 50 configurations that have been benchmarked. In the following, only the results of the most interesting configurations are presented.

Figure 2 shows the relative performance improvement compared to a non-caching version of JOP. The configuration is with a line size of 8 fields (32 bytes) and the number of lines (ways) is varied between 1 and 61. Figure 3 shows the performance impact of the variation of the line size for a 4 way cache configuration.

The *Kfl* benchmark was coded in a very conservative, static style – almost no objects are used and all functions are static. Therefore, the object cache has no impact for this

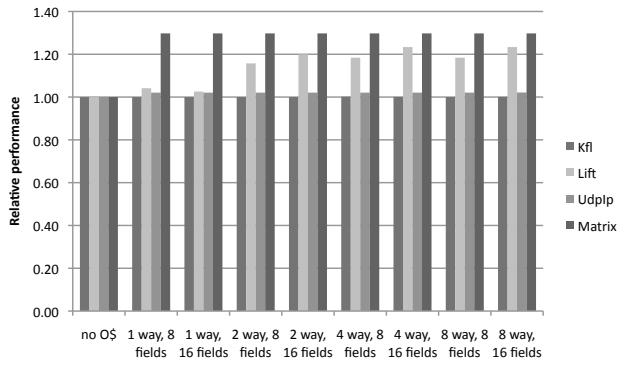

Figure 4. Performance impact on a CMP system

benchmark. The other two benchmarks show performance improvements. The *Lift* benchmark is faster by 12%. The *UdpIp* benchmark spends most of the time in array access for network package processing. Despite of this workload, there is still a speedup of 2%. The *Matrix* benchmark improves by 7%, even with a single cache line. For the benchmarks under test, caches with a higher associativity than 4 show no further performance improvement. In Figure 3 we can see that even a single field cache line provides performance improvement for *Lift*. The cache line sizes of 4 and 8 fields perform similar for all benchmarks. Only the *Lift* benchmark improves slightly (14%) with a line size of 16 fields.

The speedup of a few percent is remarkable for the minimal resources invested for the object cache. Especially when considering the quite fast main memory, which is based on an SRAM with a latency of 2 clock cycles.

In a CMP setting the demand on the memory bandwidth is high and with a time-division multiple access (TDMA) based memory arbitration the access latency increases linear with the number of cores. Therefore, even a small hit rate results in a considerable performance enhancement. The memory interface to the shared main memory is usually the bottleneck. In Figure 4 the speedup on a 3 core CMP system (with a TDMA based memory arbitration) with the object cache is shown. Due to resource limitations in the FPGA (12000 LCs) the maximum associativity of the 3 object caches for the CMP system is 8. Due to the high pressure on the memory bandwidth the performance improvement with an object cache is higher than on the uniprocessor version. The improvement is up to 23% for the *Lift* benchmark and up to 30% for the *Matrix* benchmark.

A. Resource Consumption

The object cache consumes considerable logic cells for the implementation of the tag memory and the parallel comparators. The data memory can be implemented in on-chip memory. In the FPGA that was used for the evaluation the size of an on-chip memory block is 4 KBit. Depending on the size of the data memory, the synthesize tool uses logic cells for small memories instead of an on-chip memory block. Table I shows the resource consumption in logic cells (LC) and memory bits for different cache configurations.

Table I
RESOURCE CONSUMPTION OF DIFFERENT CONFIGURATIONS OF THE OBJECT CACHE

Configuration		Resources	
Ways	Fields	Logic (LC)	Memory (Bit)
2	4	431	0
2	8	147	512
2	16	182	1024
4	4	181	512
4	8	218	1024
4	16	273	2048
8	4	331	1024
8	8	390	2048
8	16	492	4096
16	4	617	2048
16	8	745	4096
16	16	960	8192
32	4	1216	4096
32	8	1443	8192
32	16	1875	16384
64	4	2438	8192
64	8	2946	16384
64	16	3761	32768

B. Discussion

The evaluation of the object cache in hardware shows a considerable average case speedup even for small cache configurations. From the evaluation we draw the conclusion that a configuration of 4 ways and a line size of 16 fields is a good tradeoff between performance improvement and resource consumption.

The results are in line with the evaluation of the object cache based on static WCET analysis [3] and cache simulation based on cross-profiling [9]. The improvement scales to some extent with larger object caches. However, as we have seen with the cross-profiling based simulation, useful object caches sizes are up to a few KB. This result is verified by the presented benchmarks. For further speedup, especially on CMP systems, other data areas, such a constant pool or the method dispatch table, need their own cache. The address for constants and the class information is known statically. Therefore, for those areas a standard cache organization can be used.

The object cache is a good fit for Java processors. It would also be a good enhancement of the Java processors jamuth [12] and SHAP [18], which are similar to JOP. However, even standard processor architectures can be enhanced by a cache for heap allocated data. Either typed load and store instructions are added to the instruction set or the different memory areas can be distinguished by the address mapping for the virtual memory. With typed load/store

instructions one can even imagine to have n caches or scratchpad memories and let the compiler decide which data is cached by which cache unit. This approach is language agnostic and could also enhance time predictability of C programs.

VI. Conclusion

In this paper we presented a time-predictable solution to cache heap allocated objects. As the addresses of heap allocated data are not known for static WCET analysis, the object cache is highly associative. The associativity allows WCET analysis to track heap access via symbolic references. As the architectures of JOP is time compositional, the object cache can be analyzed with a static, scoped based analysis. If all heap accesses within a program scope fit into the object cache, the miss penalty for the WCET analysis is a single miss of each distinct field access. The implementation of the object cache provides a speedup of a several percent for object oriented applications at a minimum hardware cost.

The proposed object cache is one example for a time-predictable driven computer architecture. We believe that future real-time systems need a new design paradigm for the processor architecture to deliver a time-predictable platform.

Acknowledgement

I would like to thank Benedikt Huber for the implementation of the WCET analysis for the object cache that verified that the idea of a special cache for heap allocated data for time-predictable architectures is sound. Furthermore, I would like to thank Wolfgang Puffitsch for the implementation of various data caches for JOP to evaluate the split-cache idea.

References

[1] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.

[3] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object cache. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, pages 26–35, New York, NY, USA, 2010. ACM.

[4] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[6] Wolfgang Puffitsch. Data caching, garbage collection, and the Java memory model. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2009)*, pages 90–99, New York, NY, USA, 2009. ACM.

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

[8] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[9] Martin Schoeberl, Walter Binder, and Alex Villazon. Design space exploration of object caches with cross-profiling. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, 2011.

[10] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[11] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[12] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[13] N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.

[14] Ifor Williams and Mario Wolczko. An object-based memory architecture. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130, Martha's Vineyard, MA (USA), September 1990.

[15] Ifor W. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, 1989.

[16] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. Technical Report SML–TR–2005–143, Sun Microsystems Laboratories, February 2005.

[17] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. *Sci. Comput. Program*, 62(2):145–163, 2006.

[18] Martin Zabel, Thomas B. Preusser, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.