WCET driven design space exploration of an object cache

Huber, Benedikt; Puffitsch, Wolfgang; Schoeberl, Martin

Published in:
Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)

Link to article, DOI:
10.1145/1850771.1850775

Publication date:
2010

Document Version
Early version, also known as pre-print

WCET Driven Design Space Exploration of an Object Cache

Benedikt Huber
Institute of Computer Engineering
Vienna University of Technology
benedikt@vmars.tuwien.ac.at

Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology
wpuffits@mail.tuwien.ac.at

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

ABSTRACT

In order to guarantee that real-time systems meet their timing specification, static execution time bounds need to be calculated. Not considering execution time predictability led to architectures which perform well in the average case, but require very pessimistic assumptions when bounding the worst-case execution time (WCET).

Computer architecture design is driven by simulations of standard benchmarks estimating the expected average case performance. The design decisions derived from this design methodology do not necessarily result in a WCET analysis-friendly design. Aiming for a time-predictable computer architecture, we propose to employ WCET analysis techniques for the design space exploration of processor architectures. We exemplify this approach by a WCET driven design of a cache for heap allocated objects.

Depending on the main memory properties (latency and bandwidth), different cache organizations result in the lowest WCET. The evaluation reveals that for certain cache configurations, the analyzed hit rate is comparable to the average case hit rate obtained by measurements. We believe that an early architecture exploration by means of static timing analysis techniques helps to identify configurations suitable for hard real-time systems.

1. INTRODUCTION

The correctness of (hard) real-time systems depends on whether they meet their timing specification. In order to provide formal correctness guarantees, it is hence essential to obtain reliable upper bounds on the execution time of tasks. Computing the worst-case execution time (WCET) requires both a precise model of the architecture's timing and static program analysis to build a model of possible execution traces. As it is usually infeasible to analyze each execution trace on its own, abstractions of the timing relevant state of the hardware need to be computed.

Architectures with predictable timing support precise timing analysis, and consequently allow one to build precise abstractions of the timing relevant hardware state. Architectural features which do provide a speedup according to measurements, but which are intractable to be analyzed precisely, are unfavorable for hard real-time systems.[1] If there is a large gap between the observed and analyzed execution time, systems need to be over-dimensioned to provide formal guarantees. This is a waste of resources, not only in terms of silicon and power, but also in the sense that unnecessarily complex timing models are difficult to build and verify. Moreover, if static analysis is not able to provide tight bounds, the chances that relevant industries accept formal timing verification methods are decreased.

In order to build predictable architectures, static WCET analysis should be performed in an early stage of the architecture's design, in the same way benchmarks provide guidelines for building architectures which perform well in average-case measurements. In this paper, we discuss this methodology on the example of a data cache designed for predictability, the object cache. Common organizations of data caches are usually difficult to analyze in a static way. Standard data caches map the address of a datum to some location in the cache. But inferring the address of heap-allocated data is very difficult: The address assigned to an object at creation time depends on the allocator state and the allocation history. As data might be shared between threads, the address is not necessarily in control of the analyzed task. Furthermore, in the presence of a compacting real-time garbage collector, the address is modified during the object's lifetime. If the cache line depends on the address, it is therefore intractable to determine whether fields of distinct objects are mapped to different cache lines. Consequently, it is hard to find suitable abstractions for the state of the cache, and to obtain precise execution time bounds.

References in Java are often thought of as pointers, but are actually higher-level constructs. The JVM has the freedom to decide on the implementation and the meaning of a reference. Two different implementations of object references are common: either directly pointing to the object store or using an indirection to point to the object store. This indirection is commonly named *handle*. With an indirection, relocation of an object by the garbage collector is simple, because only a single pointer needs to be updated. With a direct reference to the store of the object field access is faster, but relocation of an object needs to update all references to the object. Early implementations of a JVM used the indirection, but current optimized JVMs prefer direct pointers. For a real-time JVM, compaction of the heap is mandatory and the constant time operation of updating a single location is preferable.

In this paper, we investigate the design of an object cache that uses handles as indices into the cache. As the cache is directly

[1] While any deterministic feature can be modeled accurately, without suitable abstractions the state space becomes too large to explore every possible behavior. Similar effects occur during testing, where a large state space makes exhaustive testing impossible.

JTRES'10 August 19–21, 2010 Prague, Czech Republic

indexed by handles, the indirection is eliminated on a cache hit. Handles can be reused when the object they have been pointing to is garbage. In this case, the corresponding entry in the object cache has to be invalidated, as otherwise fields of the dead object would be interpreted as cached fields of the new object. Caching the translation of an object identifier to the actual object address is similar to a translation loookahead buffer (TLB) for a virtual memory system. It can be implemented as part of the object cache or as a stand alone cache. When the object cache performs well, the pressure on an efficient translation cache is reduced; only misses in the object cache need a translation. Caching of this translation is not the topic of the paper though.

The proposed object cache is fully-associative, which implies that the cache analysis does not need to know the address of a handle. How does this organization help to increase predictability? Suppose it is known that some method operates on a set of $k \leq N$ objects, where N is the associativity of the object cache. The analysis is now able to infer that within this method, each handle will be a cache miss at most once. With a standard data cache, in contrast, this is impossible without knowing the memory address of each handle.

For the hardware designer, it is important to find a tradeoff between low resource usage and good performance. For average case performance, the standard methodology is to build a simulator collecting statistics, and run it on a set of benchmarks covering the expected use cases. As can be seen from the example of standard data caches, this will not necessarily lead to an analysis-friendly design. Therefore, we explore another design methodology in this paper, using WCET analysis techniques to investigate a good cache design. This complements the average case statistics collected earlier and presented in a technical report [11].[2]

The rest of the paper is organized as follows. In the following two sections background on WCET analysis, data cache splitting and related work on object caches are described. Section 4 discusses different organizations of the object cache. Section 5 gives an introduction data cache analysis for WCET calculation, presents the static analysis for the object cache and discusses pitfalls in WCET driven architecture evaluation. Section 6 introduces the evaluation methodology and presents the set of benchmarks used to evaluate the object cache. The results of the evaluation as well as the implications for our data cache design are presented. Section 7 concludes the paper.

2. BACKGROUND

Computing the WCET of a piece of code amounts to solving the problem of maximizing the execution time over all possible initial hardware states and execution traces. WCET analysis is concerned with two problems: Identifying the set of instruction sequences which might be executed at runtime (path analysis), and bounding the time needed to execute basic blocks (low-level analysis).

As it is usually intractable to enumerate all possible execution traces, an implicit representation that over-approximates the set of valid execution paths has to be used. The implicit path enumeration technique (IPET) [8], restricts the set of valid instruction sequences paths by imposing linear constraints on the execution frequency of control flow graph (CFG) edges.

In addition to the representation of all execution traces, the execution time of basic blocks is needed to compute the WCET. To obtain precise timings, the set of possible states of hardware components such as the data cache needs to be restricted. In particular, we need to know how often some access to the cache is a hit or miss.

The most widespread strategy used to calculate the WCET bound statically relies on dataflow analysis frameworks and integer linear programming (ILP). It proceeds in several stages: First, the call graph and control flow graph (CFG) of the task are reconstructed. Several dataflow analyses are run to gather information on the type of objects, values of variables, and loop bounds. This is followed by an analysis of hardware components, most prominently cache and pipeline analysis. Given the WCET of each basic block, and a set of linear constraints restricting the set of feasible execution paths, an ILP solver is used to find the maximum cost. The maximum cost is an upper bound for the WCET. The solution to the ILP problem assigns each basic block an execution frequency. Therefore, one only obtains a set of worst-case paths, not one valid execution trace. For the evaluation of the object cache we adapted the WCET analysis tool WCA [14].

With respect to caching, memory is usually divided into instruction memory and data memory. This cache architecture was proposed in the first RISC architectures [6] to resolve the structural hazard of a pipelined machine where an instruction has to be fetched concurrently to a memory access. This division enabled WCET analysis of instruction caches.

In former work we have argued that data caches should be split into different memory type areas to enable WCET analysis of data accesses [10, 13]. We have shown that a JVM accesses quite different data areas (e.g., the stack, the constant pool, method dispatch table, class information, and the heap), each with different properties for the WCET analysis. For some areas, the addresses are statically known; some areas have type dependent addresses (e.g., access to the method table); for heap allocated data the address is only known at runtime.

When accessing statically unknown addresses, it is impossible to predict which cache line from one way is affected by the access. From the analysis' point of view, a n-way set-associative cache is reduced to n cache lines when all addresses are unknown. Simpler cache organizations than a fully associative cache are therefore pointless for the analysis. However, such caches are expensive and must therefore be kept small. In contrast, accesses to datums with statically known addresses can be classified as hits or misses even for simple direct-mapped caches. For such a cache, accessing an unknown address would void information about all other accesses. Therefore, splitting the cache simplifies the static analysis and allows for a more precise hit/miss classification. For most data areas standard cache organizations are a reasonable fit. Only for heap allocated data we need a special organization – the object cache for objects and a solution that benefits mainly from spatial locality for arrays. The WCET analysis driven exploration of the object cache organization is the topic of this paper.

The object cache is intended for embedded Java processors such as JOP [9], jamuth [15], or SHAP [20]. Within a hardware implementation of the Java virtual machine (JVM), it is quite easy to distinguish between different memory access types. Access to object fields is performed via bytecodes `getfield` and `putfield`; array accesses have their own bytecode and also accesses to the other memory areas of a JVM. The instruction set of a *standard* processor contains only untyped load and store instructions. In that case a Java compiler could use the virtual memory mapping to distinguish between different access types.

[2] A paper for that average case statistics of the object cache, based partly on the technical report, is under submission.

3. RELATED WORK

One of the first proposals of an object cache [17] appeared within the Mushroom project [18]. The Mushroom project investigated hardware support for Smalltalk-like object oriented systems. The cache is indexed by a combination of the object identifier (the handle 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. When 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.

A dedicated cache for heap allocated data is proposed in [16]. 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 relatively low 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. 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 a loop iteration.

Wright et al. propose a cache that can be used as object cache and as conventional data cache [19]. 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.

The object caches proposed so far 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.

Design space exploration based on WCET analysis is presented in [5]. Given a reference application, the best processor and memory configuration can be selected based on the calculated WCET. The authors present a simplified version of the *aiT* WCET analysis tool to speedup the evaluation. In contrast to this approach, we do not want to select the best architecture for an application, but use WCET analysis techniques to provide a detailed analysis of the object cache performance, aiming to find a suitable design for embedded hard real-time systems. We use the WCET analysis tool WCA [14], which supports Java for a Java processor.

4. THE OBJECT CACHE

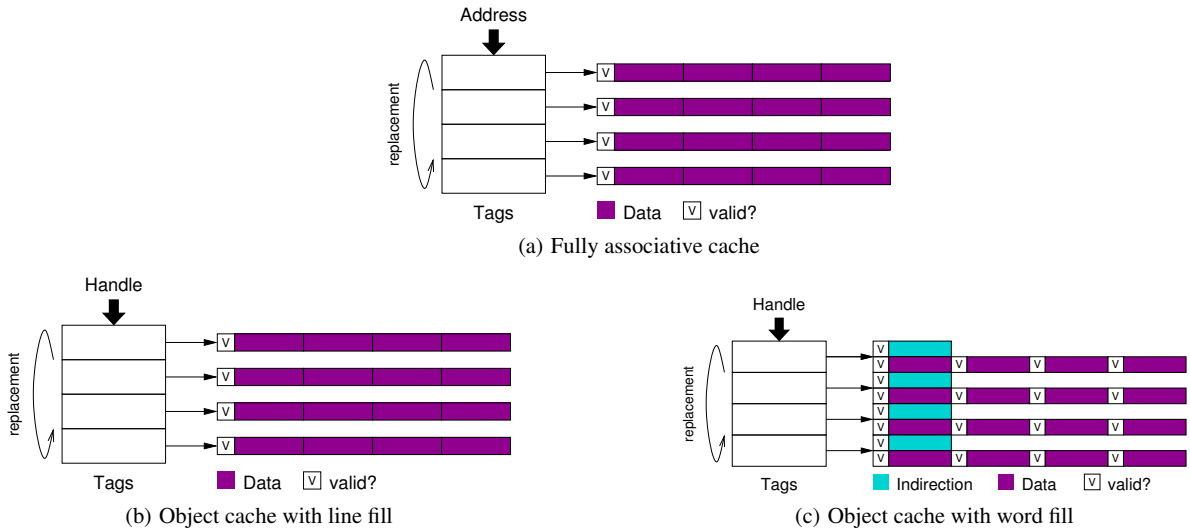
The object cache architecture is optimized for WCET analysis instead of average case performance. To track individual cache lines symbolically, the cache is fully associative. Without knowing the address of an object, all cache lines in one way map to a single line in the analysis. Therefore, the object 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. 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 also explore quite large cache lines. To reduce the resulting large miss penalty we also consider to fill only the missed word into the cache line. To track which words of a line contain a valid entry, one valid bit per word is added to the tag memory.

The object cache is a data cache with cache lines indexed by unique object identifiers (the Java reference). The object cache is thus similar to a virtually-addressed cache, except that the cache index is unique and therefore there are no aliasing issues (different object identifiers map to different objects). In our system, the tag memory contains the pointer to the handle (the Java reference) instead of the effective address of the object in memory. This simplifies changing the address of an object during the compacting garbage collection. For a coherent view of the object graph between the mutator and the garbage collector, the 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. 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 object cache is organized to cache one object per cache line. If an object needs more space than available in one cache line, fields with higher indices are not cached. The decision not to cache fields with higher field indices than words in the cache line simplifies the tag memory and the hit detection. If we would allow that different fields of one object can map to the same word in the cache line, we would need an additional tag entry per field. To compensate for possible misses with large objects, we explore quite long cache lines. The cost for the cache line is less then the cost of the tag memory, as all tags have to be compared in parallel, but the cache line needs only be read out. For an implementation in an FPGA this means that the tag memory has to implemented with discrete registers, but the cache lines can be implemented in standard on-chip memory blocks.

As objects thus cannot cross cache lines, the number of words per cache line is one important design decision to be taken. To avoid that less frequently accessed fields are cached, a compile time optimization may rearrange the order of object fields. Both benchmarking results and the results of the static analysis may be used to classify the access frequency of fields. We investigate two different behaviors on a cache miss: The first is to fill the whole cache line on a miss, loading all fields into the cache at once. This might be attractive if the memory has a longer latency for the first word accessed. The second option is to only load the missed field, which requires an additional tag bit for each word. It would also be possible to consider tradeoffs between these two extremes, e.g., loading four words on a cache miss, though this is not explored here.

Figure 1 outlines the differences between a fully associative data cache, and object caches with word and line fill. While the fully associative cache in Figure 1(a) uses the actual address as tag, the object caches use the handle. When filling the whole cache line, it is not necessary to keep the indirection pointer in the cache (Figure 1(b)). Once an object is cached, the indirection is resolved implicitly. For an object cache with word fill policy (Figure 1(c)), each word requires a valid flag, and the indirection must also be cached to allow for efficient loading of words that are not yet cached.



(a) Fully associative cache

(b) Object cache with line fill

(c) Object cache with word fill

Figure 1: Comparison of a fully associative cache and object cache variants

For comparison, we also investigate the performance of a *field cache*. In this case, cache lines are indexed by both the object identifier and the index of a field. This kind of a cache needs less memory space, but a higher associativity to achieve the same performance. As highly-associative caches are expensive, while larger cache lines are relatively cheap, we did not expect the field cache to be a viable alternative. However, the field cache, though expensive, marks the best result achievable with a given cache size, and is thus a good metric to compare the object cache configurations against.

To simplify static cache analysis we chose to organize the cache as write through cache. Write back is harder to analyze statically, as on each possible miss another write back needs to be accounted for. Furthermore, a write-through cache simplifies the cache coherence protocol for a chip multiprocessor (CMP) system [7]. In the evaluation, we assume that the cache line is not allocated on a write. Furthermore, synchronization, which usually requires a cache flush, has not been considered in the evaluation either.

The object cache is only used for objects and not for arrays. This is because arrays tend to be larger than objects, and their access behavior rather exposes spatial locality, in contrast to the temporal locality of accesses observed for objects. Therefore, we believe that a cache organized as a small set of prefetch buffers is more adequate for array data. As on the one hand arrays use a different set of bytecodes and are thus distinguishable from ordinary objects, but on the other hand have a structure similar to objects, this decision does not restrict our choices for further exploration.

5. DATA CACHE ANALYSIS

The challenge in many static program analyses is to find a way to merge information from different paths in a precise yet efficient way. For caches, this amounts to taking the union of data that may be in the cache, and the intersection of data that must be in the cache. If the address of a datum accessed is known, the analysis is able to infer that after accessing it, one known cache line will change, and all others will remain unchanged. For instruction caches, the cache block accessed at some instruction is always known statically. Intuitively, this is the reason that classifying whether some cache access is a cache hit or a cache miss works quite well for instruction caches.

If one out of a set of possible memory addresses is potentially accessed by an instruction, the analysis has to assume that more than one cache line is affected. More importantly, it is unknown to the analysis which cache line contains the data after the cache has been updated. Therefore, if it cannot be shown that all of the addresses accessed have been in the cache before, the analysis cannot classify an access to a datum with unknown address as cache hit.

For data cache accesses, the address accessed cannot always be determined precisely. Consequently, a hit or miss classification of cache accesses does not work well here. Instead, a persistence analysis [3] should be used for data caches. An access is *locally persistent* within some scope (program fragment, e.g. basic block, loop or method), if the access might be a cache miss the first time, and from there on is guarantueed to be a cache hit. For fully-associative caches using a first-in first-out (FIFO) replacement strategy, persistence is slightly weaker. For FIFO replacement, cache accesses can be classified as miss once, i.e. one cache access, though not necessarily the first one, will be a cache miss.

5.1 Object Cache Analysis

The details of the object cache analysis are not the focus of this paper, but some insights into its workings are necessary to interpret the evaluation results. As a consequence of the fact that the dataflow analysis abstracts the actual program, we do not always know the exact object a variable points to, but only have a set of objects the variable may point to at hand. Therefore, we perform a persistence analysis instead of a hit/miss classification, and try to identify scopes where accesses to an object are persistent. A particularly simple criteria for persistency in an object cache with associativity N is that at most N *distinct* objects are accessed within one scope. The main challenge for the analysis is thus to identify the number of distinct objects possibly accessed in a scope.

With this form of persistence analysis it is irrelevant if the replacement policy is LRU or FIFO. With *classic* cache analysis, FIFO replacement results in less hit classifications than LRU replacement [4].

In our analysis framework, we first perform a context sensitive receiver type analysis, which computes an approximation to the set of virtual methods possibly invoked at some instruction. Next, we

perform a value analysis, which tries to restrict the set of possible values of (integer) variables, and a loop bound analysis, which restricts the number of loop iterations.

For the object cache analysis, we perform a symbolic *Points-To* analysis for each program fragment of interest. The analysis assigns a set of object names to each use site of a variable with reference type. The name of an object is represented by an access path [2], consisting of a root name, and a sequence of field names. The root names used in the analysis of a virtual method scope include the implicit `this` parameter, the names of the reference type parameters passed to the method, and the names of all static fields with reference type that may be accessed when executing the method. The effect of a `getfield` instruction is to append the accessed field to the access paths assigned to the receiver object. For an `aaload` instruction, which loads a reference from an array, the access paths are modified depending on the possible values of the index in the analyzed scope. Examples of access paths thus include `this`, `this.f1.f2` and `staticField.f1[0]`. The special name $\top$, denoting the union of all names, is assigned to an object if either the analysis is not able to infer a meaningful set of object names, or the number of names exceeds a fixed threshold.

Objects allocated within the analyzed scope are assigned a unique name depending on the allocation site, if the corresponding `new` instruction is only executed once in the the analyzed scope. Otherwise, $\top$ is assigned as object name. One complicating factor are `aastore` instructions and `putfield` instructions with reference type, as they might change the object corresponding to one or more access paths. To this end, the analysis maintains *alias sets*, which are modified by `aastore` / `putfield` instructions, and merged into the set of possible access paths associated with an object. The set of objects affected by these instruction is currently computed based on the object's type. Finally note that the analysis is *local*, i.e. the analysis results depend on the scope the analysis acts on.

Given the results of the object reference analysis, the next task is to find out how many distinct objects are accessed within one scope. This is another optimization problem solvable using IPET: Maximize the number of field accesses, with the constraint that each object is accessed at most once. If this number can be shown to be less than the associativity of the cache, the scope is a persistence region: Each handle access will be a cache miss at the first access only (LRU cache) or at most one miss (FIFO replacement). This knowledge is included in the timing model by adding pseudo CFG nodes modeling the cache misses, and constraints restricting the execution frequency of this cache miss nodes.

Though the object reference analysis is more localized and easier than an address analysis, it has its weaknesses, as does every static analysis. In this case, the problems concern the positive effects of aliasing on the object cache performance. If two different access path are known to always point at the same object, they will map to the same cache line. A typical example is a tree-like data structure with both children and parent references. We believe that taking the results of a must-alias analysis into account should help to improve the analyzed hit rate for this examples. In general, however, it is intractable to always determine precise must alias information.

5.2 Possible Pitfalls in WCET Analysis Based Architecture Evaluation

There is an important difference between exploring the timing of an architecture using average case benchmarking and worst-case timing analysis. For a given machine code representation of a benchmark, timings obtained for different architectures all relate to the same instruction sequence. Even when using different compilers for different architectures, the frequency of different instructions and cache access patterns stays roughly the same. WCET analysis, however, computes the maximum time the benchmark needs on *any* path. As a consequence, changing one timing parameter of the architecture does not only change the WCET, but may also lead to a completely different worst-case path. This may cause the designer to draw the wrong conclusions when investigating one component of the architecture, e.g. the data cache.

As an example, consider investigating a set-associative instruction cache by means of WCET analysis. For one particular benchmark and a given associativity, assume there are two execution traces with roughly the same WCET. On one path, cache miss costs due to cache conflicts make up a significant fraction of the WCET, while in the second one, cache costs do not influence the WCET at all.

Increasing the associativity of the cache, to a lower the number of cache conflicts, the first path will become significantly cheaper to execute. But the WCET will stay roughly the same, as the second path is now the dominating worst-case path. The designer might conclude that increasing the associativity has little effect on the execution time, if she is not aware that the worst-case path has changed. In general, while absolute comparisons (*X is better than Y*) are valid, relative comparisons (*X is twice as good as Y*) have to be analyzed carefully to avoid drawing wrong conclusions. This problem is well known amongst computer architecture designers aiming to minimize the worst-case timing delay in chip designs, but does not occur in simulation-based architecture evaluation.

We investigate the cache on its own, isolating it as far as possible from other characteristics of the target. Therefore, we do not suffer from the problem that a different worst-case path changes the relative contribution of e.g. arithmetic to load/store instructions. Still, it should be kept in mind that switching the worst-case path might distort relative comparisons.

As the analysis assumes the worst possible cache state when entering a scope, embedding the scope in a larger context cannot increase the costs for object accesses. Consider the case where an object is accessed only once within a scope. The analysis then has to assume a cache miss within this scope. However, when embedding the scope in a larger context, it may be possibly to classify this access as cache hit, because the object may have been accessed in the larger context.

6. DESIGN SPACE EXPLORATION

The WCET analysis results will guide the design of the object cache. In the following section the evaluation methodology is explained and different cache organizations are analyzed with five different embedded Java benchmarks.

6.1 Evaluation Methodology

For the evaluation of the object cache we consider several different system configurations: (1) the main memory is varied between a fast SRAM memory and a SDRAM memory with a higher latency; (2) uniprocessor and a 8 core chip-multiprocessor are considered. Finally we explore the difference between single word and full cache line loads on a cache miss. To compare the WCET analysis results with average case measurements the same configurations as in [11] are used.

The best cache configuration is dependent on the properties of the next level in the memory hierarchy. Longer latencies favor longer cache lines to spread the latency over possible hits due to spatial locality. Therefore, we evaluate two different memory configuration that are common in embedded systems: static memory (SRAM) and synchronous DRAM (SDRAM). For the SRAM configuration we assume a latency of two cycles for a 32 bit word read

Table 1: Miss penalty for a memory read operation in clock cycles

		8 core CMP	
	1 CPU	min.	max.
SRAM 1w	2	2	17
SRAM 2w	4	4	35
SRAM 4w	8	8	71
SDRAM 1w	12	12	107
SDRAM 2w	14	14	125
SDRAM 4w	18	18	162

access. As an example of the SDRAM we select the IS42S16160B, the memory chip that is used on the Altera DE2-70 FPGA board. The latency for a read, including the latency in the memory controller, is assumed to be 10 cycles. The maximum burst length is 8 locations. As the memory interface is 16 bit, four 32 bit words can be read in 8 clock cycles. The resulting miss penalty for a single word read is 12 clock cycles, for a burst of 4 words 18 clock cycles. For longer cache lines the SDRAM can be used in page burst mode. With page burst mode, up to a whole page can be transferred in one burst. For shorter bursts the transfer has to be explicitly stopped by the memory controller. We assume the same latency of 10 clock cycles in the page burst mode.

Furthermore, a single processor configuration and a CMP configuration of 8 processor cores are compared. The CMP configuration is according to an implementation of a JOP CMP system on the Altera DE2-70 board. As shared caches severly complicate cache analysis (or even make it impossible), each core is assumed to have its own object cache. The memory access is arbitrated in TDMA mode with a minimum slot length s to fulfill a read request according to the cache line length. For n CPUs the TDMA round is $n \times s$ cycles. The effective access time depends on the phasing between the access request and the TDMA schedule. In the best case, the access is requested at the begin of the slot for the CPU and is $t_{min} = s$ cycles. In the worst case, the request is issued just in the second cycle of the slot and the CPU has to wait a full TDMA round till the start of the next slot:

$$t_{max} = n \times s - 1 + s = (n+1) \times s - 1$$

In contrast to [11] we use the worst-case access time for the miss penalty. Table 1 shows the memory access times (miss penalty) for the different configurations.

6.2 The Benchmarks

For the evaluation of different object cache configurations we used five different benchmarks. *Lift* is a tiny, but real world, embedded application. It controls a lift in an automation factory. The next two benchmarks have two different implementations of JOP's TCP/IP stack at their core. Both *UdpIp* and *Ejip* are artificial client/server applications exchanging data via the TCP/IP stack. Although the client and server code are artificial, the TCP/IP stack, which contains most of the code, is also used in industrial applications. The benchmarks are part of the embedded Java benchmark JemBench [12].

The fourth application we use, the trading benchmark, is based on the demo application presented in [1].[3] It emulates a financial transaction system, which must react to market changes within a bounded amount of time. One thread receives the market updates, while a second thread continuously checks whether any sell or buy orders should be placed. For the evaluation, we chose the method `OrderManager.checkForTrade()`, which performs the core functionality of the trader thread.

The final application (Cruise Control) chosen for evaluation is a cruise control that controls the throttle and brake of a simple car model. The application reads speed sensor messages from each wheel and target speed messages from a higher-level control system. One thread per wheel filters the speed messages. A speed manager thread fuses the filtered wheel speeds and provides the current speed and the target speed to the actual control algorithm. The thread to dispatch messages to the individual threads for further processing is the most interesting thread for our analysis, as it accesses a number of objects while parsing the raw message and translating it to an internal representation.

[3]We thank Eric Bruno and Greg Bollella for open-sourcing this demo application. It is available at `http://www.ericbruno.com`.

6.3 Evaluation

Figures 2–6 illustrate the miss penalty per field access as derived from the WCET analysis. The horizontal axis is labeled with the associativity of the cache configuration. N_0 is the cache miss penalty if no cache is used at all, N_k groups samples with associativity k. Different colors of the bars correspond to different cache line configurations. Cache configurations with a line size of j words are denoted by L_j. The suffix $-fill$ indicates that full line fill was assumed.

Table 2 displays the detailed analysis results for the UDP/IP benchmark. To save space, only the most interesting results of one table are included here. The complete tables are available in an accompanying technical report [11]. The results are shown for a cache configuration with single word fill on a miss, complete line fill on a miss, and as a reference a cache configuration for single words, as we have presented it in [13].

From the results we can see that the maximum analyzable hit rate without line fill is between 46 % and 93 % [11]. This hit rate can be achieved with a moderate associativity between 2 and 16 way, depending on the program.

When the cache is configured with full line fill the hit rate naturally is increased as some fields that are later used will be loaded on the line fill. Longer lines result in higher hit rates. However, the miss penalty also increases and so the miss cycles per field access. The best configuration depends on the relation between latency and bandwidth of the main memory. For a main memory with a short latency, as represented by the SRAM configuration, individual field loads on a miss give a better miss cycles per access rate than filling the whole cache line.

For the SDRAM memory the optimal line size and whether individual fields should be filled is not so clear. There is at least one configuration for every benchmark, where a line fill configuration with 8 to 32 bytes per line (depending on the benchmark) performs better than the individual field fill. However, there is no single line size, which gives better results on line fill than on word fill for all benchmarks. Moreover, the performance gained using the optimal line fill configuration is relatively small. Choosing the line size optimal for one benchmark, results in a higher miss penalty for other benchmarks than using a word fill configuration. This result is a little bit different from average-case measurements with DaCapo application benchmarks [11]. With the DaCapo benchmarks single field fill was always more efficient than loading whole cache lines, which indicates only small spatial locality in heap allocated objects. For these reasons, we lean slightly towards filling only individual fields even with a SDRAM main memory.

7. CONCLUSION

In this paper, we discussed the use of WCET analysis techniques for computer architecture exploration. Using static timing analysis in an early design stage helps to identify whether it will be possible to derive precise execution time bounds. For real-time systems, it is therefore a good complement to simulation based evaluation.

We have applied this approach to the design of the object cache, a data cache for heap allocated objects. The results have been comparable to the results obtained using simulations before, indicating that the cache design indeed enables precise WCET analysis.

We have evaluated different object cache configurations with five non-trivial application benchmarks and it is possible to analyze guaranteed hits between 46 % and 93 %. The results with a high latency memory show that object field accesses have a low spatial locality. The hits came mainly from temporal locality. Only some dedicated cache line sizes benefit from a full line fill, but not a single configuration works well for all benchmarks. Therefore, we argue to better fill only individual words on a miss and provide long cache lines to fill most of the object fields.

We believe that applying WCET analysis techniques to evaluate computer architecture features is worth the effort. We will use the insights obtained so far for the implementation of JOP's split cache architecture. We will further investigate the potential and shortcomings of this technique, and plan to apply it to obtain quantitative comparisons of other components.

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) and 214373 (Artist Design).

8. REFERENCES

[1] Eric J. Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[2] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 2–13, Apr 1992.

[3] Christian Ferdinand and Reinhard Wilhelm. 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*, pages 16–30, London, UK, 1998. Springer-Verlag.

[4] Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, July 2010.

[5] Stefana Nenova and Daniel Kästner. Worst-case timing estimation and architecture exploration in early design phases. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[6] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[7] 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.

[8] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

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

[10] 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.

[11] Martin Schoeberl, Benedikt Huber, Walter Binder, Wolfgang Puffitsch, and Alex Villazon. Object cache evaluation. Technical report, Technical University of Denmark, 2010.

[12] 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)*, Prague, Czech Republic, August 2010. ACM Press.

[13] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number LNCS 5860, pages 180–191. Springer, November 2009.

[14] 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.

[15] 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.

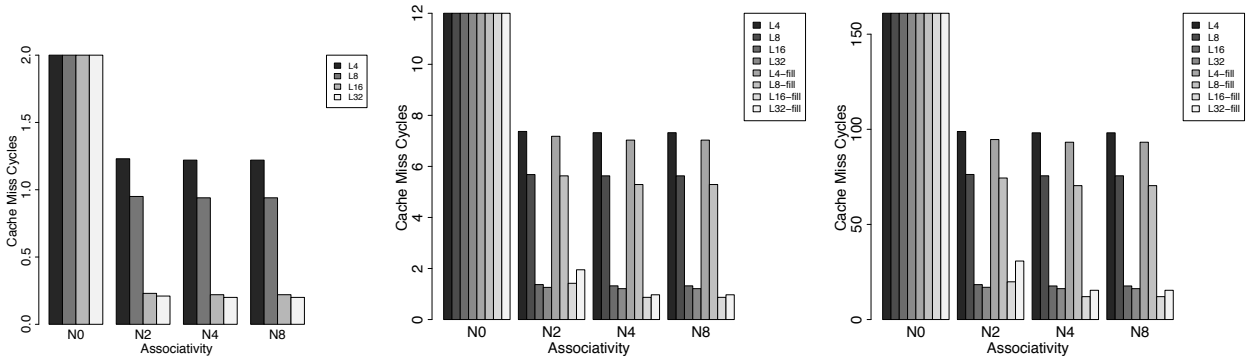
[16] N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.

[17] 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.

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

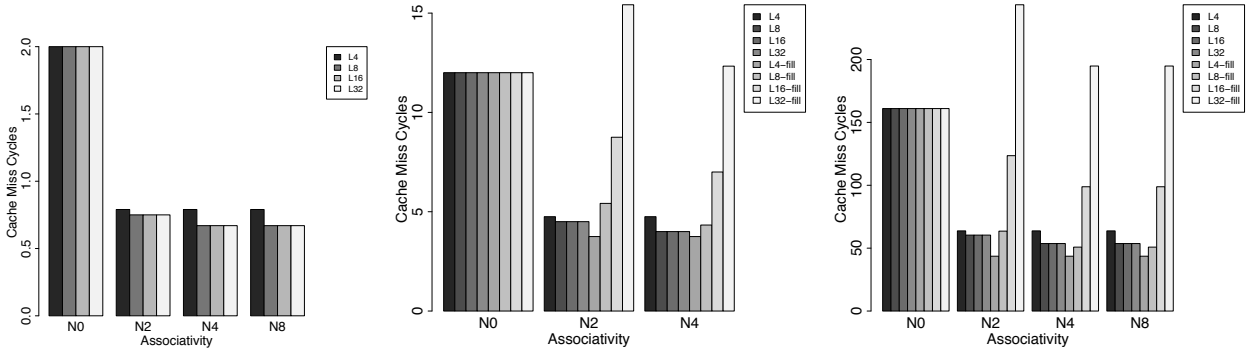
[19] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. *Sci. Comput. Program*, 62(2):145–163, 2006.

[20] 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.



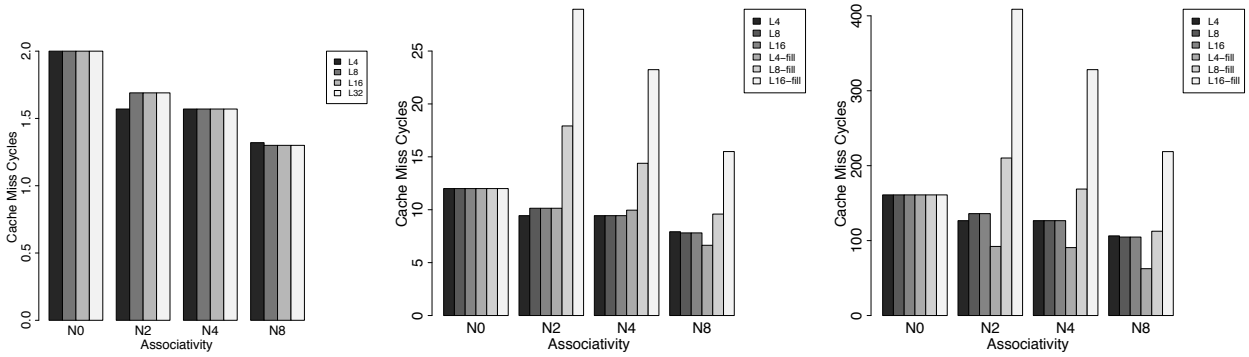
(a) SRAM, word fill (b) SDRAM, word and line fill (c) SDRAM / 8-core CMP, word and line fill

Figure 2: Lift benchmark



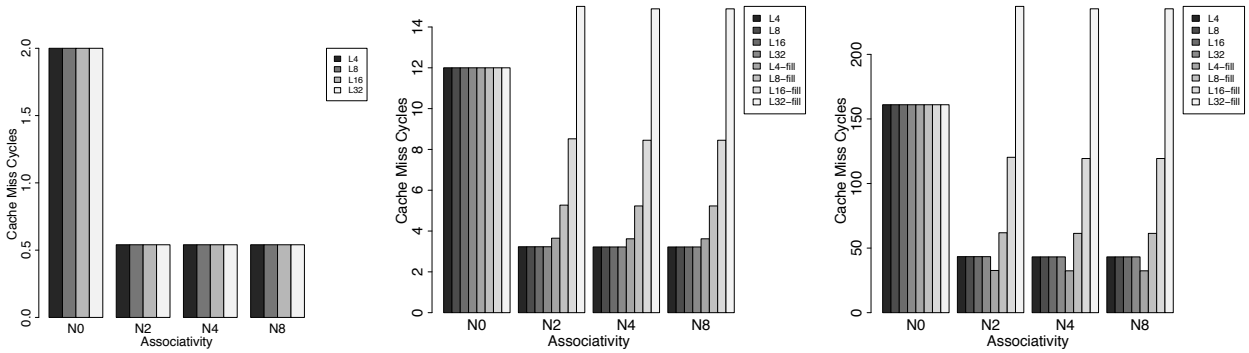
(a) SRAM, word fill (b) SDRAM, word and line fill (c) SDRAM / 8-core CMP, word and line fill

Figure 3: UDP/IP benchmark



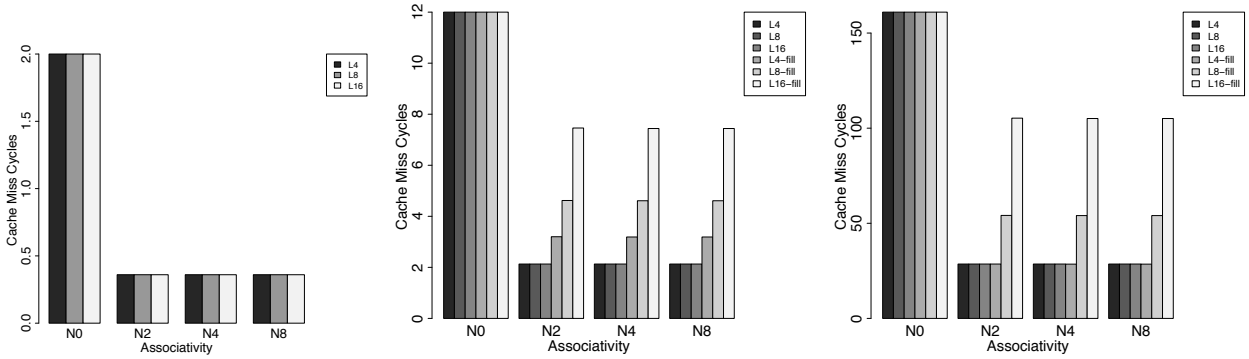
(a) SRAM, word fill (b) SDRAM, word and line fill (c) SDRAM / CMP, word and line fill

Figure 4: EJIP benchmark



(a) SRAM, word fill (b) SDRAM, word and line fill (c) SDRAM / CMP, word and line fill

Figure 5: Order Manager benchmark



(a) SRAM, word fill (b) SDRAM, word and line fill (c) SDRAM / CMP, word and line fill

Figure 6: Cruise Control Benchmark

Table 2: Object cache hit rate and miss penalty per field access for the `UDP/IP` benchmark.

Cache					Field access cost per access			
					Uniprocessor		8 core CMP	
Type	Size	Line	Assoc.	Hit rate	SRAM	SDRAM	SRAM	SDRAM
word fill	0 B	128 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	8 B	1 way	25.00 %	1.50	9.00	12.75	120.75
	16 B	16 B	1 way	58.33 %	0.83	5.00	7.08	67.08
	32 B	32 B	1 way	54.17 %	0.92	5.50	7.79	73.79
	64 B	64 B	1 way	54.17 %	0.92	5.50	7.79	73.79
	8 B	4 B	2 way	2.08 %	1.96	11.75	16.65	157.65
	16 B	8 B	2 way	27.08 %	1.46	8.75	12.40	117.40
	32 B	16 B	2 way	60.42 %	0.79	4.75	6.73	63.73
	64 B	32 B	2 way	62.50 %	0.75	4.50	6.38	60.38
	128 B	64 B	2 way	62.50 %	0.75	4.50	6.38	60.38
	16 B	4 B	4 way	2.08 %	1.96	11.75	16.65	157.65
	32 B	8 B	4 way	27.08 %	1.46	8.75	12.40	117.40
	64 B	16 B	4 way	60.42 %	0.79	4.75	6.73	63.73
	128 B	32 B	4 way	66.67 %	0.67	4.00	5.67	53.67
	256 B	64 B	4 way	66.67 %	0.67	4.00	5.67	53.67
line fill	0 B	128 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	8 B	1 way	29.17 %	1.75	8.83	14.71	114.04
	16 B	16 B	1 way	70.83 %	1.58	4.50	12.96	46.96
	32 B	32 B	1 way	66.67 %	5.33	8.67	43.00	101.67
	64 B	64 B	1 way	66.67 %	10.67	14.00	85.67	197.67
	8 B	4 B	2 way	2.08 %	1.96	11.75	16.65	157.65
	16 B	8 B	2 way	31.25 %	1.54	8.42	13.02	110.69
	32 B	16 B	2 way	72.92 %	1.04	3.75	8.60	43.60
	64 B	32 B	2 way	79.17 %	3.33	5.42	26.88	63.54
	256 B	128 B	2 way	79.17 %	13.33	15.42	106.88	243.54
	16 B	4 B	4 way	2.08 %	1.96	11.75	16.65	157.65
	32 B	8 B	4 way	31.25 %	1.54	8.42	13.02	110.69
	64 B	16 B	4 way	72.92 %	1.04	3.75	8.60	43.60
	256 B	64 B	4 way	83.33 %	5.33	7.00	42.83	98.83
	512 B	128 B	4 way	83.33 %	10.67	12.33	85.50	194.83
single field	0 B	4 B	0 way	0.00 %	2.00	12.00	17.00	161.00
	4 B	4 B	1 way	0.00 %	2.00	12.00	17.00	161.00
	8 B	4 B	2 way	4.17 %	1.92	11.50	16.29	154.29
	16 B	4 B	4 way	62.50 %	0.75	4.50	6.38	60.38
	32 B	4 B	8 way	66.67 %	0.67	4.00	5.67	53.67
	64 B	4 B	16 way	66.67 %	0.67	4.00	5.67	53.67