CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1763

Worst-case execution time analysis-driven object cache design

Benedikt Huber[1,*,†], Wolfgang Puffitsch[1] and Martin Schoeberl[2]

[1]*Institute of Computer Engineering*, *Vienna University of Technology*, *Vienna*, *Austria*
[2]*Department of Informatics and Mathematical Modeling*, *Technical University of Denmark*, *Copenhagen*, *Austria*

SUMMARY

Hard real-time systems need a time-predictable computing platform to enable static worst-case execution time (WCET) analysis. All performance-enhancing features need to be WCET analyzable. However, standard data caches containing heap-allocated data are very hard to analyze statically. In this paper we explore a new object cache design, which is driven by the capabilities of static WCET analysis. Simulations of standard benchmarks estimating the expected average case performance usually drive computer architecture design. The design decisions derived from this methodology do not necessarily result in a WCET analysis-friendly design. Aiming for a time-predictable design, we therefore propose to employ WCET analysis techniques for the design space exploration of processor architectures. We evaluated different object cache configurations using static analysis techniques. The number of field accesses that can be statically classified as hits is considerable. The analyzed number of cache miss cycles is 3–46% of the access cycles needed without a cache, which agrees with trends obtained using simulations. Standard data caches perform comparably well in the average case, but accesses to heap data result in overly pessimistic WCET estimations. We therefore believe that an early architecture exploration by means of static timing analysis techniques helps to identify configurations suitable for hard real-time systems. Copyright © 2011 John Wiley & Sons, Ltd.

Received 2 December 2010; Revised 9 March 2011; Accepted 17 March 2011

KEY WORDS: real-time systems; time-predictable computer architecture; worst-case execution time analysis; data cache

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 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, the timing relevant state of the hardware needs to be abstracted.

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[‡]. 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

*Correspondence to: Benedikt Huber, Institute of Computer Engineering, Vienna University of Technology, Treitlstraße 3/182/1, 1040 Vienna, Austria.
†E-mail: benedikt@vmars.tuwien.ac.at
‡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.

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 that 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 difficult to analyze statically for heap-allocated data. 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 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. For standard caches it is therefore intractable to determine to which cache line an object address maps and whether an object is in the cache when accessed. Consequently, it is hard to find suitable abstractions for the state of the cache, and to obtain precise execution time bounds.

For the reasons stated above, object-oriented programming, where memory allocation at runtime is common, is considered problematic for WCET analysis. The analysis of a C++ virtual function dispatch with the WCET tool aiT resulted in cache miss predictions of two dependent memory loads (about 40 cycles on a PowerPC 755 for each miss)[§]. 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 of 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 issues mentioned before. The capabilities of static WCET analysis guide the cache design, instead of the usual way where the WCET analysis has to follow the given hardware design. In this paper, we investigate the design of an object cache with the help of WCET analysis [1]. The object cache caches individual objects; it is addressed with the object reference, which is not necessarily the memory address of the object. The individual words within a cache line are addressed by the field index.

The proposed object cache is fully associative, which implies that the cache analysis does not need to know the address of the object. 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 access to an object field will be a cache miss at most once. With a standard data cache, in contrast, this is difficult without knowing the address of each object and object field accessed.

For the hardware designer, it is important to find a tradeoff between resource usage and 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.

The paper is an extended version of [2]. The additional contributions are:

- A more flexible object cache model, which caches the object address and the type reference, and permits a more fine-grained tradeoff between memory latency and expected spatial locality of field accesses.
- An improved object cache analysis, including a detailed formal description of the dataflow analysis we developed.
- Evidence that in the average case, the split cache architecture including the object cache performs reasonably well compared to a standard data cache.

[§]Private communications with Reinhard Wilhelm.

In the following section the background and related work on WCET analysis, cache analysis, data cache splitting, object cache organizations, and time-predictable architectures are described. Section 3 introduces the object cache design, and discusses different organizations and the respective benefits. Section 4 presents the static analysis of the object cache's timing relevant behavior. Section 5 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 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

The proposal of a time-predictable object cache and the respective WCET analysis algorithm combines research on data cache analysis, architectures for object-oriented architectures, and the emerging research on time-predictable computer architectures. The following section provides the background on those different, but interrelated research domains.

2.1. *Worst-case execution time analysis*

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 that might be executed at runtime (path analysis), and bounding the time needed to execute basic blocks (low-level analysis) [3].

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) [4] restricts the set of valid paths by imposing linear constraints on the execution frequency of control flow graph 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 a safe WCET bound relies on dataflow analyses and integer linear programming (ILP). It proceeds in several stages: First, the call graph and control flow graph of the task are reconstructed. Several dataflow analyses are run to gather information about the types of objects, values of variables, and loop bounds. This is followed by an analysis of hardware components, most prominently cache and pipeline analyses. 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, and not a single execution trace. For the evaluation of the object cache we adapted the WCET analysis tool WCA [1].

2.2. *Cache analysis*

With respect to caching, memory is usually divided into instruction memory and data memory. This cache architecture was proposed in the first RISC architectures [5] 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, which is outlined below.

The most common cache architecture for both instruction and data caches are set-associative caches (Figure 1). To determine the cache block that an address refers to, the address is split into the set index, the tag used to identify blocks within one set, and the offset in the cache block. Conceptually, the cache lookup mechanism first selects a cache set depending on the index part of the address, and then updates the cache blocks in that set. The update depends on the tags of the cache blocks in the selected set, and the replacement strategy used. Common replacement strategies are LRU (Least Recently Used), FIFO (First-In First-Out) and Pseudo-LRU. If the block

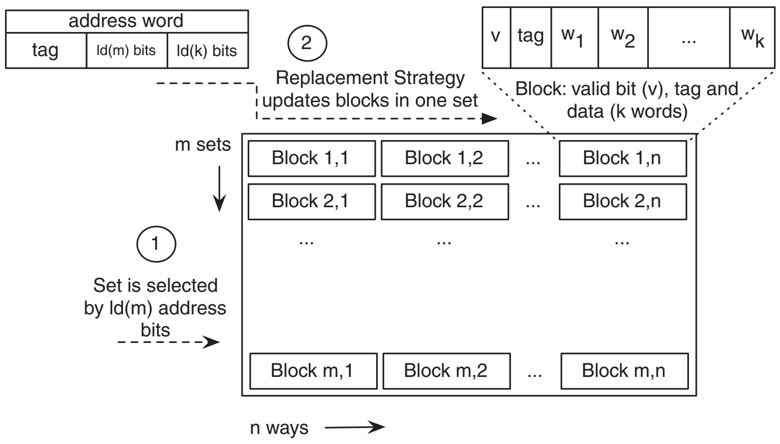

Figure 1. Set-associative cache with n ways.

is not in the cache, there is a cache miss, and the corresponding data is loaded from the next level memory. For the purposes of a cache analysis, a direct-mapped cache can be understood as a set-associative cache with only one way.

A commonly used framework for the low-level timing analysis of set-associative instruction caches is cache hit/miss classification (CHMC), which determines whether a memory access at some point of execution is a cache hit, a cache miss, or cannot be classified [6]. The information necessary for this classification is provided by a preceding dataflow analysis. The abstract cache state used in the dataflow analysis for LRU caches assigns each cache block a minimum and maximum *age*. When a block is accessed, it has age 0, and after accessing k distinct other blocks its age is k. The information from different predecessors has to be merged at join points. For caches, this amounts to taking the union of cache blocks that may be in each set, and the intersection of blocks that must be in one set, i.e. the minimum and maximum of all ages, respectively. If at some program point, the block corresponding to an accessed address has an age less than the associativity of the cache, it is safe to assume that the access will always be a cache hit.

The CHMC technique works well for LRU instruction caches, as it is always known statically which cache block an instruction corresponds to. Detecting cache hits due to the spatial locality of instructions mapping to the same cache block is easy: When an address in the same cache block has been accessed before, the age of the cache block in the abstract cache state is zero. In order to classify all but the first access to an instruction in a loop as cache hit, the first iteration of the loop body is treated separately (virtual loop unrolling).

However, if the exact address accessed by an instruction is unknown, cache analysis, and especially hit/miss classification, becomes more difficult and less precise.

1. If the exact memory address is unknown, we do not know which cache block is actually accessed. Therefore, one cannot predict whether one particular cache block is loaded into the cache or not due to the access.
2. If the address is unknown, it is not possible to determine which cache set was actually affected. In any of the affected cache sets, the oldest block might be evicted from the cache.

For the first issue, consider a sequence of accesses where it is only known that each address is within a certain range. An example would be a series of array accesses where the index is unknown to the static analysis. In this case, the cache hit/miss classification cannot be applied; as it is unknown which of the accesses will be a cache hit. However, given a sufficiently large cache, it is possible to infer that every cache block belonging to the array will be loaded at most once. If a cache block is loaded at most once during execution, it is said to be persistent. Therefore, so-called persistence analysis is better suited to data caches than CHMC. It is only applicable however, if a precise hit/miss classification is not absolutely necessary for pipeline analysis. Therefore, the architecture used has to be *timing compositional* [7]. That means that the architecture has no timing anomalies or unbounded timing effects [8].

As cache blocks are rarely persistent throughout the execution of the whole program, one is usually interested in local persistence with respect to a program fragment, or scope. To this end, the scope graph [9] of a program is traversed, performing persistence analyses for each scope of interest. One possible choice for scopes are methods—in this case the scope graph is equivalent to the call graph of the program.

The second issue is difficult to solve from the analysis side. If there are n accesses to data where the address is not known statically, any cache block in an n-way set-associative cache might have been evicted, and we cannot tell which one. From the analysis' point of view, only one cache set of an n-way set-associative cache is useful when all addresses are unknown.

State-of-the-art WCET analysis tools, such as AbsInt's aiT, consider accesses to heap-allocated data as unknown addresses. For a four-way associative cache all information about the cache content is lost after accesses to four unknown addresses. Unknown addresses remain unknown; no symbolic tracking is implemented in aiT[¶]. Therefore, nothing comparable to our proposed object cache analysis is implemented in aiT. We are not aware of any implementation of symbolic address tracking in the other available WCET analysis tools.

To improve data cache analysis in the presence of unknown memory addresses, Lundquist *et al.* [10] suggest to distinguish between unpredictable and predictable memory accesses. If an address cannot be resolved at compile time, accesses to that address are considered as unpredictable. Those data structures that might be accessed by unpredictable memory accesses are marked for being moved into an uncached memory area. This idea complements our split cache design, as the knowledge within the compiler can be used to redirect accesses to different caches instead of just redirecting data structures to an uncached memory area. Vera *et al.* [11] 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.

2.3. Split cache design

In the former work we have argued that data caches should be split into different memory type areas to enable WCET analysis of data accesses [12, 13]. We have shown that a Java virtual machine (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. In Java, the type of the memory area for load and store instructions can be inferred from the bytecode (e.g. putfield versus putstatic), and the implementer of a JVM can choose how these bytecodes are translated into machine code.

For some areas, the addresses are known statically; some areas have type dependent addresses (e.g. access to the method table); for heap-allocated data the addresses are only known at runtime. As argued before, using parts of the address to lookup the cache set is pointless if a significant fraction of accessed addresses are unknown to the static analysis. In contrast, accesses to areas with statically known addresses can be classified as hits or misses even for simple direct-mapped caches.

Splitting the cache for different data areas simplifies the static analysis and allows for more precise analysis results. 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 exploits the spatial locality of arrays. The mapping of accesses to the individual caches of the split cache is known statically, which simplifies the design of the hardware and static WCET analysis. From the processor core's point of view, the caches can be treated as if they were unrelated; only a relatively simple multiplexing/demultiplexing unit is needed to merge the memory accesses from the caches. The focus of this paper is on the WCET analysis-driven exploration of the object cache organization and the tradeoffs between size and analyzable hits.

[¶]Private communications with Christian Ferdinand.

2.4. Object caches

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 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 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 block 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 surprisingly high for the relatively low latency (seven cycles) of the memory system.

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.

The object caches proposed so far are optimized for average case performance. It is common to use a hash function (xoring part of the object identifier with the field offset) in order to equally distribute objects 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.

2.5. Time-predictable computer architecture

The object cache, and the WCET analysis supporting it, is part of the research on time-predictable computer architectures [18]. This research field gets momentum as WCET analysis of modern processor architecture is getting almost impractical. Edwards and Lee argue: ‘It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function’ [19].

Heckmann *et al.* [20] suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches; (2) locally deterministic update strategies for caches; (3) static branch prediction; and (4) limited out-of-order execution. The authors argue for restriction of processor features of processors for embedded systems. Extending on this proposal we want to actively add time-predictable features, such as the object cache, to the restricted processor.

At Berkeley the concept of a precision timed (PRET) architecture is developed [21, 22]. The PRET is a multi-threaded RISC pipeline with scratchpad memories, time-predictable access to DRAM by assigning each thread a dedicated bank in the memory chips, and a deadline instruction to enforce cycle accurate timings of program sections and I/O accesses.

Whitham and Audsley work on time-predictable out-of-order processors [23] and on a scratchpad memory management unit (SMMU) [24]. Virtual traces allow static WCET analysis and constrain the out-of order scheduler of the CPU to execute deterministically. The SMMU manages pointer redirection (similar to Java handles) to simplify the usage of scratchpads instead of caches.

3. THE OBJECT CACHE

The object cache architecture is optimized for WCET analysis instead of average case performance. Therefore, the design has to take into account the capabilities of WCET analysis in general and cache analysis in particular. Typically, objects are dynamically allocated and have unknown addresses. As detailed in Section 2, unknown addresses are mapped to a single set in the cache analysis; having more than one set would not provide any benefits. Consequently, the cache contains only a single set, or in other words, it is a fully associative cache.

As high associativity is expensive in terms of hardware, the number of objects that can be cached is limited. However, caching a larger number of words per object is relatively cheap.

References in Java are often thought of as pointers, but are actually higher level constructs. The concrete implementation of the Java virtual machine 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. For a real-time JVM, where compaction of the heap is mandatory, such an organization is preferable. The indirection increases the number of memory accesses, and must be handled efficiently to provide good performance.

In our system, the tag memory contains references, i.e. pointers to the object handle, and not the effective address of object fields, which is a key difference to conventional caches. This also makes it possible to resolve the handle indirection efficiently in case of cache hits. In case of a cache hit, the cost for the indirection is zero—the address translation has already been performed. To some degree, the object cache is similar to virtually addressed caches, except that the cache index for the object cache is unique and therefore there is no need to handle aliasing.

With regard to the amount of cache memory that is associated with each object, several design decisions have to be made. First, the amount of data that can be cached per object is limited, and a tradeoff has to be found between implementation cost and cache performance. Second, as the size of the memory per object is limited, one has to decide how the fields for objects that are too large to fit completely into the cache are treated. Third, the policy for filling the cache has to be chosen.

Regarding the filling of cache lines, there are several possible solutions. One solution would be to load individual words into the cache upon cache misses. This requires valid flags for each word and minimizes the amount of data that is loaded into the cache. On the other end of implementation choices, one could fill the whole per-object memory, treating it as a single cache line. While this increases memory traffic, it is potentially beneficial for memories with high bandwidth but a long latency for accessing the first word. In between these extremes, it is also possible to fill the cache in bursts of several words, similar to conventional caches. The optimal fill size depends on the properties of the main memory and the spatial locality of object field accesses. As shown in the evaluation, field accesses benefit primarily from temporal locality and only little from spatial locality.

The object cache is organized to cache one object per cache way. If an object needs more space than available in one way, fields with higher indices are either not cached at all, or cached in a wrap-around manner, replacing other fields of the same object. While the second solution could enable the exploitation of locality at different field indices, it also complicates the cache design. For a direct-mapped organization of individual cache lines of an object an additional tag field for the individual fields (or blocks of fields) would be necessary. This design would result in another cycle for the access of a cached field. We decide to optimize the design for the common case of small objects and leave fields at higher indices uncached. In order to minimize bypassing of uncached fields, the object layout may be optimized such that heavily accessed fields are located at lower indices.

Figure 2 shows the conceptual organization of the object cache. It has an associativity of n and can consequently cache up to n objects. Unlike conventional caches, where the effective address

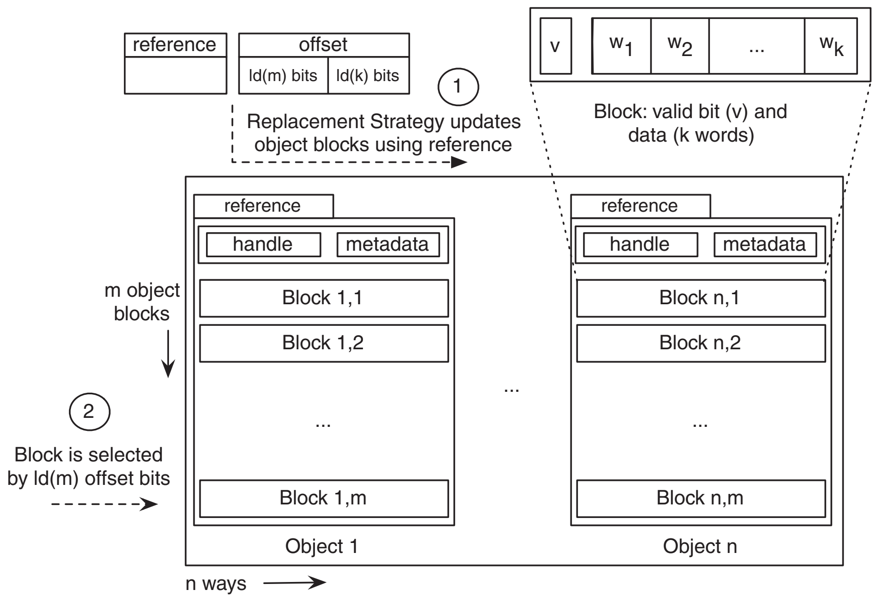

Figure 2. Object cache caching the first $m \times k$ words of an object.

determines the placement of data in the cache, the object cache uses the reference to determine the way, and the field offset to determine the cache block and word index within the block. For each object, the effective object address and some metadata are cached; in our case, the metadata consists only of the type information pointer. The cache contains m blocks consisting of k words for each object. The choices for m and k depend on the overall cache size and the cache fill policy. For example, when caching 16 words for each object and filling the cache in bursts of four words, both m and k equal 4.

As for any cache, it also has to be decided whether writes are handled with a write-back or a write-through policy. In a recent paper [7] it has been explained that write-back caches are handled in WCET analysis 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. The current WCET analysis tools do not track the dirty bit of a cache line. Therefore, write-back caches add considerable conservatism to the WCET analysis results. Heeding this insight, we chose to organize the cache as write-through cache. Furthermore, a write-through cache simplifies the cache coherence protocol for a chip multiprocessor (CMP) system [25]. In the evaluation, we assume that the cache line is not allocated on a write.

The object cache is used only 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 arrays usually have a layout similar to objects, it would however be possible to use the object cache for small arrays as well.

The object cache is intended for embedded Java processors such as JOP [26], jamuth [27], or SHAP [28]. 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.

As a result of the WCET-driven evaluation of the object cache, it has been implemented within JOP [29]. The cache size (associativity) and the number of fields per cache line are configurable via constants in a VHDL file. Average case measurements in the hardware show the same trend as the WCET based analysis: a medium associativity and relatively short cache lines are enough to cache most object data. Full associativity for the object cache lines is expensive in hardware (e.g. the logic needed for 64 ways is almost as much as needed for the processor core). A configuration of four ways and a line size of 16 fields is found to be a good tradeoff between performance improvement and resource consumption. For an object-oriented benchmark, such as the *Lift* application that is used in the evaluation section, the average case performance increases by 14% for a single processor and by 23% for an eight core CMP system. Non-object-oriented benchmarks do not benefit from the object cache. The 4-way, 16 fields per line cache configuration consumes less than 10% of the full processor design.

4. OBJECT CACHE ANALYSIS

The object cache analysis is the basis for the design choices in the implementation of a time-predictable object cache. It relies on a dataflow analysis, which computes the set of possible referenced objects for each field access. As dataflow analyses abstract the actual program, we do not always know the exact object a variable points to, but only have a set of objects that the variable *may point to* at hand. Therefore, we perform a persistence analysis instead of a hit/miss classification.

A particularly simple criterion 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. The persistence analysis is run for each scope of interest. Persistence in a larger scope subsumes persistence in subsets of that scope. Therefore, we traverse the call graph of the program to be analyzed, and try to prove that all objects in the traversed scopes are persistent. If we succeed, we skip the investigation of the sub-scopes.

Our persistence analysis aims to show that, within one specific program scope, one particular cache block *never* leaves the cache once it is loaded. This is in contrast to persistence analyses based on dataflow frameworks [30], where they aim to compute precise abstraction of the cache state *for each program point*. According to our observations, the latter problem is much harder for data caches. Furthermore, with *classic*, dataflow-oriented cache analyses, the FIFO replacement policy is problematic, and results in less classified hits than the LRU replacement policy [31]. With the form of persistence analysis we use, it is irrelevant whether the replacement policy is LRU or FIFO.

The object cache analysis performs three steps for each scope: First, we assign each variable of reference type a set of symbolic object names it may point to. Each name corresponds to one object on the heap. Second, we try to prove that at most n distinct objects are referenced during one execution of that scope. If this is the case, we add, as final third step, constraints to the IPET problem, restricting the cache miss cost for the object cache.

We illustrate the object cache analysis using the small example in Listing 1. In this example, there are two root objects possibly accessed in this method: `datap`, which is passed as a parameter, and the instance variable `this.syncp`. At the end of the if-then-else statement, `packet` either points to the object named `datap` or to `syncp`. Next, the buffer `packet.buf`, i.e. either `datap.buf` or `syncp.buf`, is modified. Finally, the method dispatch table of `linkLayer` is accessed, passing either `datap` or `syncp` to the method `enqueue()`. For the sake of simplicity, we assume that `enqueue()` does not access any objects other than `linkLayer` and the one passed as a parameter.

In summary, the following objects are accessed once during a single execution of `header()`: `datap`, `this`, `this.linkLayer` and either `datap.buf` or `syncp.buf`. Now assume that `header()` is invoked n times in a row with the same object `datap` as parameter, but with different values of `datap.prot`. The analysis will infer that at most those five objects are accessed. In an eight-way object cache with FIFO or LRU replacement, it follows that none of the objects will be evicted from the cache once loaded, i.e. they are persistent. The way the cache is designed, we also know that each cache block accessed needs to be loaded at most once.

```
public void header(Packet datap)
{
   Packet packet;

   if (datap.prot == SYNC) { packet = this.syncp; }
   else                    { packet = datap; }

   packet.buf[0] = 0x45000000 + packet.len;

   linkLayer.enqueue(packet);
}
```

Listing 1: Object analysis example.

The rest of this section explains the details of the dataflow analysis, the technique used to count the maximum number of distinct objects accessed in a scope, and how to extend the IPET formulation to restrict the cache miss costs.

4.1. Dataflow analysis for the object cache

In our analysis framework, we first perform a 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. All analyses are call context sensitive, so invocations from different call sites can be distinguished. Furthermore, for WCET analyzable programs, the maximum number of times an instruction is executed within a scope is always known—otherwise it would be impossible to infer any WCET bound at all. If the dataflow analysis fails to infer loop bounds, they have to be provided by means of manual annotations.

The dataflow analysis for the object cache is a *symbolic points-to analysis*, run for each scope of interest. The analysis assigns a set of object names to each variable with reference type. The name of an object is represented by an access path [32], consisting of a root name, and a sequence of field names or indices. We write $n.f$ to denote the symbolic name obtained by appending the field f to the object name n. Examples of access paths are anObject, this.f1.f2 and staticField.f1.0.

The root names used in the analysis of a virtual method 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 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.

The domain of the dataflow analysis is a pair $\langle \mathscr{P}, \mathscr{A} \rangle$. The first component, $\mathscr{P}$, is a mapping assigning access paths to variables. It models which variables may point to which objects, but does not take dependencies due to the mutation of references into account. This is achieved by introducing a second component, $\mathscr{A}$, which keeps tracks of fields or array elements that are modified during the execution of the analyzed scope. The join function $\sqcup$ for this dataflow domain

$$\langle \mathscr{P}', \mathscr{A}' \rangle = \langle \mathscr{P}_1, \mathscr{A}_1 \rangle \sqcup \langle \mathscr{P}_2, \mathscr{A}_2 \rangle$$

is defined as the pointwise union of $\mathscr{P}_1$ and $\mathscr{P}_2$, and $\mathscr{A}_1$ and $\mathscr{A}_2$, respectively.

$$\mathscr{P}'(x) = \mathscr{P}_1(x) \cup \mathscr{P}_2(x)$$

$$\mathscr{A}'(x) = \mathscr{A}_1(x) \cup \mathscr{A}_2(x)$$

Definition of the transfer functions affecting $\mathscr{P}$. If a reference pointing to a constant, for example a string literal (ldc), is assigned to a variable, $\mathscr{P}$ maps the variable to the singleton set containing the name for the constant (Equation (1)). If a null pointer, i.e. a value which cannot be dereferenced, is assigned to a variable, the set of object names the variable may point to is empty (Equation (2)).

Equation (3) defines the transfer function for accessing the field of an object (getfield). The field name is appended to all access paths associated with the dereferenced variable. The union of

```
public static void f(Packet p1, Packet p2)
{
   Packet x = p1;
   x.link.reset();

   p2.link = p2.prev.link;

   Packet y = p1;
   y.link.reset();
}
```

Listing 2: Aliasing example.

this set and of all objects the field may point to due to aliasing is assigned to the left-hand side variable.

For an aaload instruction, which loads a reference from an array, the result depends on the possible values of the index in the analyzed scope. If the index i is known statically, or if the value analysis computes a reasonably small interval for the possible values of the index at the analyzed instruction, all possible indices are appended to all possible source objects (Equation (4)). The resulting object set is merged with the objects the array may contain due to aliasing. If the interval for the index is too large, or unknown within the analyzed scope, $\top$ is assigned to the variable.

For instructions allocating new objects (new), we need to assign an otherwise unused object name to the variable. To this end, we use our WCET analyzer's information about the maximum number of times the new instruction is executed within the analyzed scope. If a new instruction is executed at most n times ($f(loc) \leq N$), then n object names, obtained by appending an index to the unique *location* of the new instruction, approximate the set of objects the variable may point to (Equation (5)).

$$\mathsf{y} = \mathsf{ldc(id)} \quad \mathscr{P}(y) \leftarrow \{id\} \tag{1}$$

$$\mathsf{y} = \mathsf{null} \quad \mathscr{P}(y) \leftarrow \{\} \tag{2}$$

$$\mathsf{y} = \mathsf{x.f} \quad \mathscr{P}(y) \leftarrow \{o.f \mid o \in \mathscr{P}(x)\} \cup \mathscr{A}(f) \tag{3}$$

$$\mathsf{y} = \mathsf{x[i]} \text{ with } L \leq i \leq U \quad \mathscr{P}(y) \leftarrow \{o.K \mid o \in O, K \in [L, U]\} \cup \mathscr{A}(\mathsf{typeof(x)}) \tag{4}$$

$$\mathsf{loc:}\ \mathsf{y} = \mathsf{new\ T} \text{ with } f(\mathsf{loc}) \leq N \quad \mathscr{P}(y) \leftarrow \{\mathsf{loc}.K \mid 1 \leq K \leq N\} \tag{5}$$

Mutation of heap-allocated objects. Instructions that modify the content of an array (aastore) or the fields of an object (putfield), need to be treated in a special way. In order to illustrate why we use an additional mapping to deal with mutation of reference type fields, consider the following program fragment shown in Listing 2. Both x and y point to the object passed as the first parameter p1. Now assume that p1 and p2 reference the same object. In this case modifying p2.link also modifies p1.link, and consequently x.link and y.link reference different objects, although the same access path (p1.link) is associated with them. Taking aliasing into account, we thus need to assume that y.link may point to either p1.link or p2.prev.link.

If an instruction modifies a reference type field f of an object o, then the object referenced by this field might have been changed for each variable which *may alias* with o. Equation (6) reflects our implementation and states a simpler approximation: any field named f may alias the objects that are assigned to $y.f$. Similarly, if one element of an array a is changed, accesses to the same element of another array variable, which may alias with a, might reference the assigned object. For arrays, we use the type of the array to distinguish different alias sets (Equation (7)).

$$\mathsf{y.f} = \mathsf{x} \quad \mathscr{A}(f) \leftarrow \mathscr{A}(f) \cup \mathscr{P}(x) \tag{6}$$

$$\mathsf{y[i]} = \mathsf{x} \quad \mathscr{A}(\mathsf{typeof}(y)) \leftarrow \mathscr{A}(\mathsf{typeof}(y)) \cup \mathscr{P}(x) \tag{7}$$

4.2. *Determining persistency and IPET integration*

Given the results of the symbolic points-to analysis, the next task is to find out how many distinct objects are accessed within one scope. This is another optimization problem solved using IPET: Maximize the number of field accesses, with additional constraints enforcing that each object is only counted once. If the number of accessed objects can be shown to be less than or equal the associativity of the cache, the scope is a persistency region. Each handle access may be a cache miss at the first access only (LRU cache) or at some access (FIFO replacement).

The results of the object cache analysis are included in the timing model by adding *miss nodes* to the program model, or equivalently, cache miss variables to the ILP. Whenever an instruction might access an object named o, a cache miss node $n \in cm(o)$ is added. To reflect that a cache miss implies a cache access, the execution frequency of this node has to be less than the execution frequency of the instruction accessing the object. Now assume that all objects in scope S are persistent. We need to assert that all persistent objects are loaded at most once. For each object name o, we add a linear constraint stating that the execution frequency of all cache miss nodes ($cm(o)$) has to be less than or equal to the execution frequency of scope S:

$$\sum_{n \in cm(o)} f(n) \leq f(S)$$

In our object cache design, there is no conflict between fields of the same object. Therefore, the same strategy is used for cache blocks in the object cache. For each potential access to cache block b, we introduce a cache miss node $n \in cm(b)$. In a persistency scope, the total execution frequency of these nodes is again less than or equal to the execution frequency of the scope.

$$\sum_{n \in cm(b)} f(n) \leq f(S)$$

Note that if there is a lot of aliasing, counting objects becomes imprecise, as some of the objects will inevitably be counted more than once. 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 these examples. In general, however, it is intractable to always determine precise must-alias information. As a consequence, aliasing should be kept to the necessary minimum in hard real-time applications, which need to be analyzed statically.

5. 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 four different, object-oriented embedded Java benchmarks.

5.1. *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 by executing the benchmarks 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 data 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 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 a change of the worst-case path might distort relative comparisons.

5.2. Evaluation methodology

The best cache configuration depends on the properties of the next level in the memory hierarchy. Longer latencies favor longer cache blocks to amortize the miss penalty by possible hits due to spatial locality. Therefore, we evaluate two different memory configurations 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 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 four 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 page burst mode.

To evaluate the performance of the object cache, we analyzed the average number of miss cycles per access, i.e. cycles spent for accessing the main memory. All accesses to heap-allocated data are mapped to the object cache. They include accesses to the handle to resolve the indirection, accesses to the immutable object type reference, and accesses to data fields of the object.

If a field of an object is read, two (independent) accesses to the main memory are needed without caching: one to resolve the handle indirection and one to read the field. With the object cache two distinct cases have to be distinguished: on a field hit, the result is delivered without a memory access. If the field is a cache miss, but the handle is in the cache, the cache resolves the handle indirection and the individual field (or block of fields) has to be fetched from memory.

In the object cache, a two-word block comprising the handle and the object type is loaded if the object handle is not in the cache. If the field is bypassed, i.e. never cached because its index is too large, one word needs to be loaded from main memory on every access. For accessing the object's type (the reference to the class information), one access to the main memory is needed without a cache. When caching the class reference, on a miss in the object cache, a two-word block (comprising the handle and the class reference) is loaded into the cache.

While the main evaluation of the object cache is based on static WCET analysis, we also performed some average case measurements. For those measurements a cache simulation of standard caches and the split cache have been developed and integrated in the software simulator for JOP (JopSim).

5.3. The benchmarks

For the evaluation of different object cache configurations we used four different benchmarks. *Lift* is a tiny, but real-world, embedded application. It controls a lift in an automated factory. The next two benchmarks have two different implementations of JOP's TCP/IP stack at their

Table I. List of benchmarks and properties.

Benchmark	Methods	Instructions
Lift	13	1200
UdpIp	32	1904
Ejip	27	1423
Cruiser	38	1667

Table II. Average case comparison of a standard and a split cache.

		Data cache 2 kB		Split cache 2 kB	
Benchmark	Accesses	Hit rate (%)	Miss cycles	Hit rate (%)	Miss cycles
Lift	310	90.00	558	90.97	492
UdpIp	219	77.17	900	79.68	771
Ejip	422	78.35	1643	81.33	1264
Cruiser	784	87.02	1832	79.97	2766

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 [33].

The final application (*Cruiser*), chosen for the 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.

Table I summarizes the benchmarks used. The column *Methods* gives that static method count of analyzed and executed methods, the column *Instructions* the static instruction count (bytecodes) of the benchmark.

5.4. Split caches

To evaluate whether the split cache for data accesses is a reasonable design, we perform an average case comparison between a standard data cache and the split cache. The comparison includes all data memory accesses except accesses to the stack and array accesses. The stack is cached in its own cache and we do not (yet) have a time-predictable solution for array accesses in the split cache. The numbers are collected with a cache simulation executed in the software simulator (JopSim) of JOP.

The standard data cache is a four-way set-associative cache with a block size of 4 words and 32 blocks per set (2 kB). The split cache consists of two direct-mapped caches for constants and static variables, respectively, and an object cache. The direct-mapped cache for constants has 64 cache blocks with four words each (1 kB), the cache for static variables has 32 cache blocks with four words each (512 B). The object cache has eight ways, four blocks per object, and four words per block (512 B for field data). In total, both the standard data cache and the split cache have a size of 2 kB.

In both configurations, array accesses go directly to main memory. In our benchmark set, the number of array accesses is, on average, in the same order as the number of field accesses. Therefore, their impact on the embedded benchmarks does not dominate as one may suspect. Nevertheless, a full split cache implementation will also provide a solution for arrays.

Table II shows the comparison results between the two cache designs. The column *Accesses* gives the dynamic count of memory accesses (excluding instruction, stack, and array accesses). The miss cycles are calculated for the SDRAM configuration.

The hit rate and number of miss cycles is in the same range for both configurations. Three of the four benchmarks perform slightly better with the split cache than with the standard cache. We contribute this performance enhancement to the elimination of cache conflicts between constants and variables, and to the higher associativity of the object data cache.

5.5. The object cache

The object cache is evaluated with two different main memory configurations and different block sizes for the SDRAM configuration. The vertical axis in all figures shows the statically analyzed miss cycles per memory access for object field, object handle, and type information. The baseline without a cache uses single word accesses, the cache versions access memory in bursts corresponding to the block length. Therefore, non-cached accesses take 12 cycles for the SDRAM configuration and 2 cycles for the SRAM configuration. The horizontal axis is labeled with the abbreviations of different object cache configurations. The leftmost data points (NO) give the number of memory access cycles without a cache. The label n/k corresponds to a configuration with *n* ways, with *k* words of object data. The cache size is therefore $4 \times k \times n$ bytes.

The cache miss cycles for the SDRAM configuration, using a block size of four words, are illustrated in Figure 3. Each line represents the results for one benchmark. The miss cycles per access vary greatly depending on the benchmark. For most benchmarks an associativity of 4–8 is enough to reduce the access time considerably. The analyzed methods of the Lift benchmark access few objects, but do so frequently, and thus the results for this benchmark are the best. The other benchmarks have a lower degree of temporal locality in their field accesses, but using the object cache still gives a considerable improvement over not caching heap-allocated data at all.

The tradeoff between different block sizes for the SDRAM configuration is shown in Figure 4. The optimal configuration varies between the benchmarks, depending on the size of the objects, and the spatial locality of field accesses. For small caches the configuration with single word loads performs best as not enough spatial locality can offset the higher miss penalty due to a block load. If the cache is sufficiently large, a block size of four words turns out to be a good choice for the Ejip and Lift benchmark. The Cruiser benchmark has apparently less spatial locality and even for the SDRAM configuration single word load gives the best performance.

The results for the SRAM configuration are shown in Figure 5. We used one-word block, as this fits the characteristics of the fast SRAM memory best. The general trend is similar to the

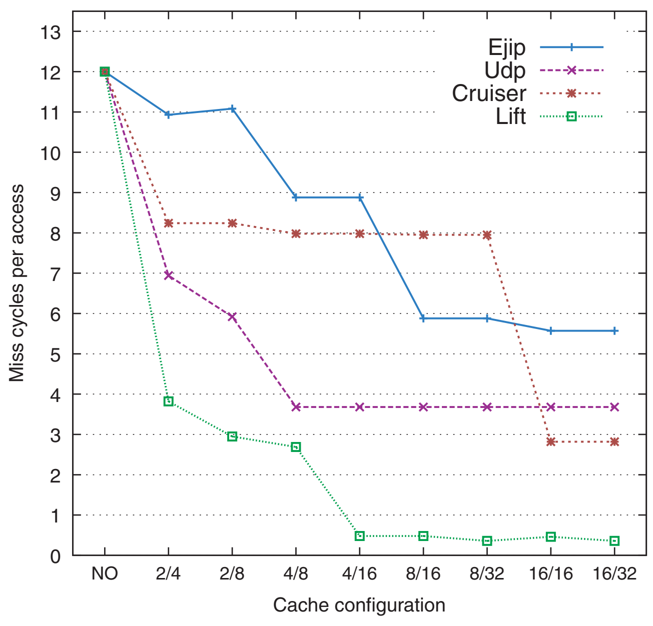

Figure 3. Analyzed cache miss cycles per memory access (SDRAM, four word blocks).

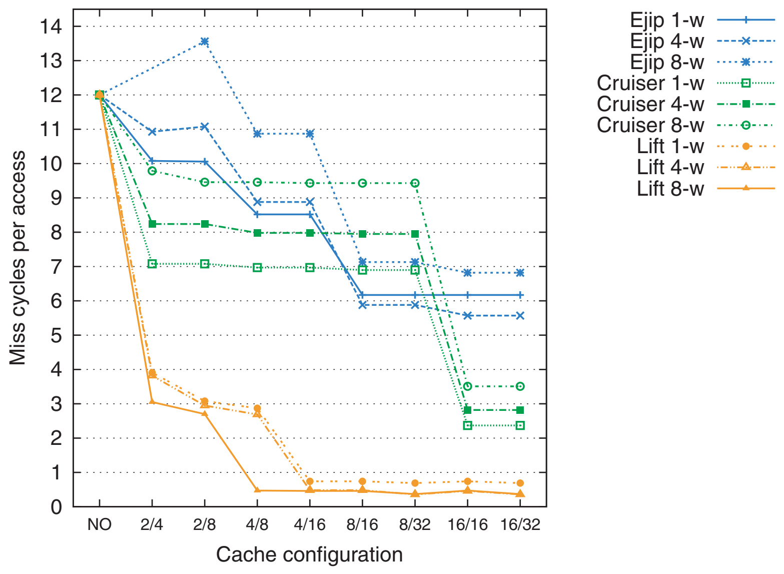

Figure 4. Analyzed cache miss cycles per memory access. Evaluation of different block sizes for selected benchmarks (SDRAM).

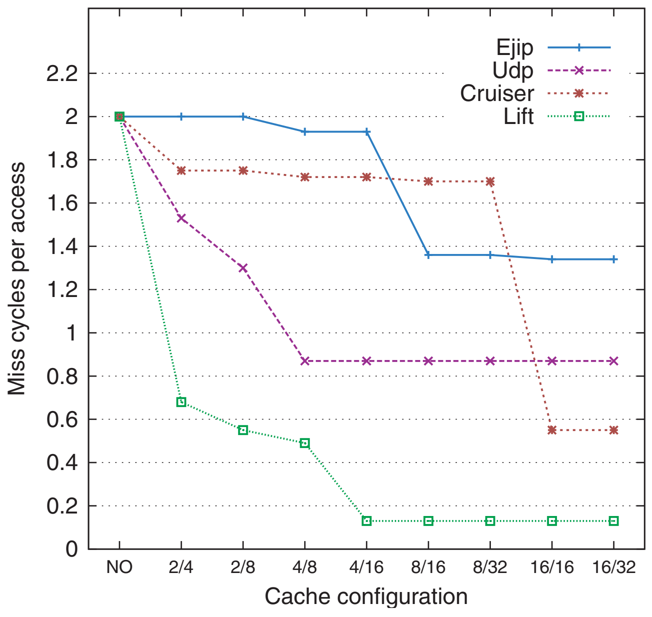

Figure 5. Analyzed cache miss cycles per memory access (SRAM, one word blocks).

SDRAM configuration. However, the benefit of using the object cache design for this memory is, as expected, not as big as for the SDRAM configuration. Moreover, accessing two words on each object handle miss incurs a higher relative penalty than for SDRAM, which results in a higher number of cache miss cycles for benchmarks with low hit rates, such as Ejip. Therefore, with this configuration the handle and type information should be loaded individually.

In Table III, the total cache miss cycles for the same configuration are shown for both average case simulations and the static analysis. Unfortunately, the results are not directly comparable, as the number of memory access cycles differs, even if no cache is used. This is because the measurements do not execute the worst-case path maximizing the number of cache miss cycles. Nevertheless,

Table III. Measured and analyzed total cache miss cycles (SDRAM, four word blocks).

	No cache	2/4	2/8	4/8	4/16	8/16	8/32	16/16	16/32
Ejip measured	1734	646	670	574	574	491	491	402	402
Ejip analyzed	2748	2502	2538	2034	2034	1346	1346	1276	1276
Udp measured	792	125	113	106	106	106	106	106	106
Udp analyzed	1272	736	628	390	390	390	390	426	426
Cruiser measured	3108	430	430	334	334	320	320	320	320
Cruiser analyzed	4476	3074	3074	2978	2978	2964	2964	1052	1052
Lift measured	3024	360	354	354	186	186	168	186	168
Lift analyzed	5544	1764	1362	1242	222	222	168	222	168

there are several interesting points to observe. The benchmarks show similar characteristics in both simulations and static analysis: For the Lift benchmark, the cache is most effective, while it is least effective for Ejip. In the simulations, a rather low associativity already gives good benefits. In contrast, the analysis needs at least twice as many ways before cache miss cycles drop significantly. We believe that this is partly due to the fact that in a worst-case scenario more objects are accessed. The other reason is probably the static and rather coarse grained nature of persistency scopes. Over all, the analyzed number of cache miss cycles (the miss penalty) is 3%–46% of the access cycles needed without a cache.

6. 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 evaluation based on simulation.

We have applied this approach to the design of the object cache, a data cache for heap-allocated objects. The evaluation results for the static analysis we developed demonstrate that for this cache design, it is possible to derive useful static bounds on cache miss costs. To the best of our knowledge, in the presence of heap-allocated objects, no comparable results have been achieved for other data cache designs.

We have evaluated different object cache configurations with four non-trivial application benchmarks. Using a safe static analysis, it is possible to show that the analyzed number of cache miss cycles is 3–46% of the access cycles needed without a cache. The results also show that object field accesses have a relatively low spatial locality. For SDRAM with 10 cycles latency and 2 cycles read delay, using four word cache blocks turned out to be a good compromise.

We believe that applying WCET analysis techniques to evaluate computer architecture features is worth the effort. We 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.

REFERENCES

1. Schoeberl M, Puffitsch W, Pedersen RU, Huber B. Worst-case execution time analysis for a Java processor. *Software*: *Practice and Experience* 2010; **40**(6):507–542.
2. Huber B, Puffitsch W, Schoeberl M. WCET driven design space exploration of an object caches. *Proceedings of the Eighth International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2010*). ACM: New York, NY, U.S.A., 2010; 26–35.
3. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P. The worst-case execution time problem—Overview of methods and survey of tools. *Transactions on Embedded Computing Systems* 2008; **7**(3):1–53.

4. Puschner P, Schedl A. Computing maximum task execution times—A graph-based approach. *Journal of Real-Time Systems* 1997; **13**(1):67–91.
5. Patterson DA. Reduced instruction set computers. *Communications of the ACM* 1985; **28**(1):8–21.
6. Ferdinand C, Wilhelm R. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 1999; **17**(2–3):131–181.
7. Cullmann C, Ferdinand C, Gebhard G, Grund D, Maiza C, Reineke J, Triquet B, Wilhelm R. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems*, Toulouse, France, 2010.
8. Lundqvist T, Stenström P. Timing anomalies in dynamically scheduled microprocessors. *Proceedings of the 20th IEEE Real-time Systems Symposium* (*RTSS 1999*). IEEE Computer Society: Washington, DC, U.S.A., 1999; 12–21.
9. Engblom J, Ermedahl A, Sjödin M, Gustafsson J, Hansson H. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer* (*STTT*) 2003; **4**(4): 437–455.
10. Lundqvist T, Stenström P. A method to improve the estimated worst-case performance of data caching. *RTCSA '99*: *Proceedings of the Sixth International Conference on Real-time Computing Systems and Applications*. IEEE Computer Society: Washington, DC, U.S.A., 1999; 255–262.
11. Vera X, Lisper B, Xue J. Data caches in multitasking hard real-time systems. *RTSS '03*: *Proceedings of the 24th IEEE International Real-time Systems Symposium*. IEEE Computer Society: Washington, DC, U.S.A., 2003; 154–165.
12. Schoeberl M. Time-predictable cache organization. *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems* (*STFSSD 2009*). IEEE Computer Society: Tokyo, Japan, 2009; 11–16.
13. Schoeberl M, Puffitsch W, Huber B. Towards time-predictable data caches for chip-multiprocessors. *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems* (*SEUS 2009*) (*Lecture Notes in Computer Science*, vol. 5860). Springer: Berlin, 2009; 180–191.
14. Williams I, Wolczko M. An object-based memory architecture. *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, U.S.A., 1990; 114–130.
15. Williams IW. Object-based memory architecture. *PhD Thesis*, Department of Computer Science, University of Manchester, 1989.
16. Vijaykrishnan N, Ranganathan N. Supporting object accesses in a Java processor. *IEE Proceedings—Computers and Digital Techniques* 2000; **147**(6):435–443.
17. Wright G, Seidl ML, Wolczko M. An object-aware memory architecture. *Science of Computer Programming* 2006; **62**(2):145–163.
18. Schoeberl M. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems* 2009; **2009**:2:1–2:17.
19. Edwards SA, Lee EA. The case for the precision timed (PRET) machine. *DAC '07*: *Proceedings of the 44th Annual Conference on Design Automation*. ACM: New York, NY, U.S.A., 2007; 264–265.
20. Heckmann R, Langenbach M, Thesing S, Wilhelm R. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE* 2003; **91**(7):1038–1054.
21. Lickly B, Liu I, Kim S, Patel HD, Edwards SA, Lee EA. Predictable programming on a precision timed architecture. *Proceedings of the International Conference on Compilers*, *Architecture*, *and Synthesis for Embedded Systems* (*CASES 2008*), Altman ER (ed.). ACM: Atlanta, GA, U.S.A., 2008; 137–146.
22. Liu I, Reineke J, Lee EA. A PRET architecture supporting concurrent programs with composable timing properties. *2010 Conference Record of the 44th Asilomar Conference on Signals*, *Systems and Computers*, Asilomar, CA, U.S.A., 2010.
23. Whitham J, Audsley N. Time-predictable out-of-order execution for hard real-time systems. *IEEE Transactions on Computers* 2010; **59**(9):1210–1223.
24. Whitham J, Audsley N. Implementing time-predictable load and store operations. *Proceedings of the International Conference on Embedded Software* (*EMSOFT 2009*), Grenoble, France, 2009.
25. Puffitsch W. Data caching, garbage collection, and the Java memory model. *Proceedings of the Seventh International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2009*). ACM: New York, NY, U.S.A., 2009; 90–99.
26. Schoeberl M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* 2008; **54**(1–2):265–286.
27. Uhrig S, Wiese J. jamuth: an IP processor core for embedded Java real-time systems. *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2007*). ACM Press: New York, NY, U.S.A., 2007; 230–237.
28. Zabel M, Preusser TB, Reichel P, Spallek RG. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures*, *Methods and Tools* (*DSD 2007*), Lübeck, Germany, 2007; 59–62.
29. Schoeberl M. A time-predictable object cache. *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing* (*ISORC 2011*), Newport Beach, CA, U.S.A., 2011.

30. Ferdinand C, Wilhelm R. On predicting data cache behavior for real-time systems. *LCTES '98*: *Proceedings of the ACM SIGPLAN Workshop on Languages*, *Compilers*, *and Tools for Embedded Systems*. Springer: London, U.K., 1998; 16–30.
31. Grund D, Reineke J. Precise and efficient FIFO-replacement analysis based on static phase detection. *Proceedings of the 22nd Euromicro Conference on Real-time Systems* (*ECRTS 2010*), Brussels, Belgium, 2010.
32. Deutsch A. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. *Proceedings of the 1992 International Conference on Computer Languages*, Oakland, CA, U.S.A., 1992; 2–13.
33. Schoeberl M, Preusser TB, Uhrig S. The embedded Java benchmark suite JemBench. *Proceedings of the Eighth International Workshop on Java Technologies for Real-time and Embedded Systems* (*JTRES 2010*). ACM: New York, NY, U.S.A., 2010; 120–127.