Data cache organization for accurate timing analysis

Schoeberl, Martin; Huber, Benedikt; Puffitsch, Wolfgang

Published in:
Real-Time Systems

Link to article, DOI:
10.1007/s11241-012-9159-8

Publication date:
2013

Real-Time Systems manuscript No.
(will be inserted by the editor)

Data Cache Organization for Accurate Timing Analysis

Martin Schoeberl, Benedikt Huber,
Wolfgang Puffitsch

Received: 1 Mai 2010 / Accepted: 12 June 2012

Abstract Caches are essential to bridge the gap between the high latency main memory and the fast processor pipeline. Standard processor architectures implement two first-level caches to avoid a structural hazard in the pipeline: an instruction cache and a data cache. For tight worst-case execution times it is important to classify memory accesses as either cache hit or cache miss. The addresses of instruction fetches are known statically and static cache hit/miss classification is possible for the instruction cache. The access to data that is cached in the data cache is harder to predict statically. Several different data areas, such as stack, global data, and heap allocated data, share the same cache. Some addresses are known statically, other addresses are only known at runtime. With a standard cache organization all those different data areas must be considered by worst-case execution time analysis. In this paper we propose to split the data cache for the different data areas. Data cache analysis can be performed individually for the different areas. Access to an unknown address in the heap does not destroy the abstract cache state for other data areas. Furthermore, we propose to use a small, highly associative cache for the heap area. We designed and implemented a static analysis for this cache, and integrated it into a worst-case execution time analysis tool.

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
E-mail: masca@imm.dtu.dk
Benedikt Huber, Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology, Austria
E-mail: benedikt@vmars.tuwien.ac.at, E-mail: wpuffits@mail.tuwien.ac.at

1 Introduction

With respect to caching, memory is usually divided into instruction memory and data memory. This cache architecture was proposed in the first RISC architectures [28] to resolve the structural hazard of a pipelined machine where an instruction has to be fetched concurrently to a memory access. This partitioning into instruction and data memory is also known as *Harvard architecture*, in contrast to the *von Neumann architecture* with unified memory, relating the cache splitting to an early computer architecture with separate storages for instructions and data.

The independent caching of instructions and data has enabled the integration of cache hit classification of instruction caches into the worst-case execution time (WCET) analysis [2] long before data caches. While analysis of the instruction cache is a mature research topic, data cache analysis for heap allocated data is still an open problem. After N accesses with unknown addresses to a N-way set associative cache, the complete abstract cache state is lost.

In previous work we have argued about cache splitting in general [36] and provided average case simulation results of hit rates for split caches [42]. Simulation showed that small, individual caches for constants, object handles, method lookup tables, and static fields lead to good hit rates for embedded benchmarks. For the caching of object fields in a fully associative cache, the hit rate saturated at an associativity of 8.

The proposed splitting of caches is intended to lower the WCET bound – a design decision for time-predictable processors. Optimizing the WCET performance hurts the average case performance [37]. A data cache for all data areas is, in the average case, as shown by current main-stream processor architectures, more efficient than splitting the same amount of on-chip memory for different data types. Therefore, the presented solution is intended for systems where the WCET performance is of primary importance.

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

The organization of the proposed highly associative cache for heap allocated data fits very well as a buffer for hardware transactional memory [16]. Transactional memory as an alternative to cache coherency protocols for CMP systems under real-time constraint is explored in [40].

The proposed split cache is an approach to increase the time predictability of a system and to decrease the WCET. For hard real-time systems the possible increase of the average case execution time is not relevant, as the overall system has to be dimensioned for the WCET of tasks. However, an average case comparison shows that a split cache configuration performs similar to a unified cache for data accesses [19].

In this paper we focus on L1 caches for data accesses. However, the same principle of cache splitting can be applied to L2 caches. The extension of the WCET analysis to consider timing models for a cache hierarchy is straight forward. The timing model of the main memory is configurable with access latency and bandwidth. Therefore, it can be used for SRAM as well DRAM.

In this paper we evaluate time-predictable data cache solutions in the context of the Java virtual machine (JVM). Implementation results for different cache organizations show the resource consumptions and limitations of highly associative cache organizations.

Access type examples are taken from the JVM implemented on the Java processor JOP [35]. Implementation details of other JVMs may vary, but the general classification of the data areas remains valid. The concepts presented in this paper are language agnostic. However, the discussion of virtual method dispatch tables is biased towards object-oriented languages, such as C++ and Java.

This paper is an extension of [36] and [42]. The contribution of the paper is the proposal of individual caches for different data areas. For global, stack allocated, and constant data standard WCET analysis methods can be used. For heap allocated objects we present the results of a scope based persistence analysis [19], which tracks symbolic object references instead of memory addresses.

The paper is organized as follows: In the following section the mechanics of caches are summarized. Section 3 presents related work on data cache analysis and scratchpad memories. The different types of data areas for the common languages C/C++ and Java are presented in Section 4. Section 5 describes our cache analysis for heap-allocated data. Implementation details and the evaluation of the proposed caching solution are covered in Section 6. Section 7 concludes our findings.

2 Caches

Between the middle of the 1980s and 2002, CPU performance increased by around 52% per year, but memory latency decreased only by 9% [15]. To bridge this growing gap between CPU and main memory performance, a memory hierarchy is used. Several layers with different tradeoffs between size, speed, and cost form that memory hierarchy. A typical hierarchy consists of the register file, first level instruction and data caches, one or two layers of shared caches, the main memory, and the hard disc for virtual memory.

The only memory layer that is under direct control of the compiler is the register file. Other levels of the memory hierarchy are usually not visible – they are not part of the instruction set architecture abstraction. The placement of data in the different layers is performed automatically. While caches are managed by the hardware, virtual memory is managed by the operating system. The access time for a word that is in a memory block paged out by the OS is several orders of magnitude higher than a first level cache hit. Even the difference between a first level cache access and a main memory access is in the order of two magnitudes.

2.1 Why Caches Work

Caches exploit the principle of locality. An accessed data word is likely to be accessed again in the near future (temporal locality), and it is likely that words that are located close to that word are accessed (spatial locality). Typical caches in general-purpose processors exploit both types of locality. Temporal locality is exploited by keeping data in the fast memory of the cache. Spatial locality is exploited by fetching several words from main memory into a cache line. For dynamic RAM, accessing consecutive addresses is cheap compared to accessing random addresses. Fetching a whole cache line requires only little more latency than accessing a single word, but turns out beneficial if accesses exhibit spatial locality.

Programs that mainly iterate through arrays show a high degree of spatial locality. Other programs however will not gain as much from large cache lines. Caches in general-purpose processors must find a compromise such that the cache configuration fits the temporal and spatial locality of “average” programs. It has been proposed to split the data cache to exploit different types of locality [11,26] and use the chip area budget more efficiently. Despite promising initial performance results, the concept of split caches has not prevailed in mainstream architectures.

2.2 WCET Analysis

Cache memories for instructions and data are classic examples of the *make the common case fast* paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Plenty of research effort has been expended to integrate the instruction cache into the timing analysis of tasks [2,13], the influence of the task preemption on the cache [4], and the integration of the cache analysis with the pipeline analysis [12].

A unified cache for data and instructions can easily destroy all the information on abstract cache states. Access to N unknown addresses in an N-way set-associative cache results in the state *not classified* for all cache lines. Modern processors usually have separate instruction and data caches for the first level cache. However, the second level cache is usually shared. Most chip-multiprocessor (CMP) systems also share the second level cache between the different cores. The possible interactions between concurrent threads running on different cores are practically impossible to model. A second level cache can be made predictable by partitioning the L2 cache for the different cores. The concept of split caches can also be applied to L2 caches.

Caches in general, and particularly data caches, are hard to analyze statically. Therefore, we introduce caches that are organized to simplify the analysis. Such caches do not only improve the average case performance (compared to uncached data accesses), but also enable the computation of tight WCET bounds.

3 Related Work

Most published papers on cache analysis consider the effects of the instruction cache on the WCET. Data cache analysis papers are rare. This can be explained by two factors: (1) The influence of the instruction cache on the WCET is considerably higher than the influence of the data cache. Each instruction includes one fetch and assuming misses slows down each instruction. Data is accessed only every few instructions. (2) Instruction cache analysis is easier than data cache analysis. Instruction addresses are known statically, whereas the addresses of a data accesses depends on several factors. Some addresses, e.g., access to heap allocated data, cannot be predicted statically.

Kim et al. extend Shaw's timing schema [44] to support pipelined processors and data caches [22]. Load and store operations are categorized into static and dynamic accesses. Static accesses are load/stores where the address does not change (e.g., access to global data via the `gp` register and stack allocated data via the `sp` register). Dynamic accesses are penalized by two cache misses: one for the unknown access and a second for the evicted cache line that might cache a static access. In the paper the classification of static load and stores, which are not using the global or stack pointer register, but are in fact static, is tightened. One example is an access to a local variable not addressable through the stack pointer with an offset due to the restriction of the offset field in the RISC processor. Furthermore, the conservative assumption of two miss penalties for a dynamic access is reduced to one miss for cases where no cache information is propagated in the path for the extended timing schema. Or in other words, where the following accesses will be classified as misses anyway.

White et al. present an analysis of data caches when the addresses can be statically predicted [52]. The paper focuses on static, global data, and on stack allocated data. Furthermore, for iterations through an array, spatial locality (data that is prefetched due to loading of complete cache lines) is included in the analysis. Heap allocated data is not considered in the analysis and the example programs use static data only. The presented results (analyzed hit ratio between 76% and 97%) are promising for caching static and stack allocated data. With our proposed split cache architecture these analysis results for static data will not be influenced by accesses to heap allocated data.

Ferdinand and Wilhelm propose abstract interpretation for instruction cache analysis in the presence of set associative caches [10]. The main idea is to provide three classifications: *always hit*, *always miss*, and *not classified* for individual access points. Based on a least recently used (LRU) replacement policy the update and merge functions for different control flow paths are given. To benefit from hits in loop iterations the concept of splitting a loop into the first iteration and following iterations is proposed.

Reineke et al. analyzed the predictability of different cache replacement policies [33]. It is shown that the LRU policy performs best with respect to predictability. Pseudo-LRU and FIFO perform similarly, but considerably worse than LRU.

Lundqvist and Stenström propose to analyze only predictable data structures and bypass the cache for unpredictable data structures [24]. Memory loads and stores are unpredictable due to following reasons: the address depends on unknown input data or the WCET analysis itself introduces uncertainties to the memory address, e.g., when execution paths are merged. The mapping of unpredictable loads and stores to the unpredictable data structures needs compiler or user support. Those data structures are then located by the linker into an uncached memory segment.

A common solution to avoid data caches is an on-chip memory, named scratchpad memory (SPM), that is under program control. This program managed memory implies a more complicated programming model. However, scratchpad memory can be automatically partitioned [3,1,49]. A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [30] and the allocation of data in the scratchpad memory [50, 45,6] can be optimized for the WCET. A comparison between locked cache blocks and a scratchpad memory with respect to the WCET can be found in [31]. While former approaches rely on the compiler to allocated the data or instructions in the scratchpad memory an algorithm for runtime allocation is proposed in [25]. We consider SPM as a complimentary technology to data caching to enable local memory usage for data structures with hard to predict addresses. Within our evaluation platform (the Java processor JOP) we have implemented a SPM. To allow programs to allocate objects in the SPM, the SPM is mapped to RTSJ style scoped memory region [51].

Vera et al. lock the cache during accesses to unpredictable data [46]. The locking proposed there affects all kinds of memory accesses though, and therefore is necessarily coarse grained. Cache locking can be combined with cache partitioning for multiple tasks to achieve a time-predictable system in the case of task preemption [47,48]. The idea of restricting caching of data to only predictable data structures is along the same line as our proposed cache splitting. In contrast, we propose to adapt the cache structure to cache all data accesses, but split the cache for accesses to different data areas.

Whitham and Audsley propose a special form of SPM, which includes scratchpad memory management unit (SMMU) [53,55]. The SMMU is in charge to redirect pointers into the SPM when data is moved into the SPM. Therefore, the pointer aliasing issue is solved in hardware. The SPM with the SMMU is somehow similar to our approach for caching of heap allocated data. Individual objects and arrays are allocated at runtime in the SPM and hardware supports the redirection of accesses into the SPM. Both proposals can handle dynamic, heap allocated data structures. The difference between the the SPM/SMMU and our heap cache is that data is moved into the SPM under software control, whereas our heap cache performs data movement (and caching decisions) in hardware. Similar to the heap cache, the SMMU needs a fully associative tag memory for pointer lookup, which limits the maximum number of objects that can be allocated in the SPM. However, similar to our findings in [19], they show that a moderate associativity (simultaneous objects in the SPM) of 8 is enough to cover most dynamically allocated data structures [55]. With an enhancement of the SMMU to support read-only objects and tiling of larger data structures it is shown that the WCET with a SMMU/SPM is close to the average case execution time with a conventional data cache [54].

Herter et al. [18,17] tackle the analysis of dynamically allocated memory from the allocation side. One approach [18] is to guide `malloc()` to allocate data such that it maps to a certain cache line. A second approach [17] is to automatically convert dynamic allocations to static allocations. The conversion takes into account the cache effects of memory mappings to find an optimal mapping with regard to the WCET. Although these techniques look promising, it remains to be evaluated whether they can be applied to real-world programs effectively.

The most problematic processor features for WCET analysis are the replacement strategies for set-associative caches [14]. A pseudo-round-robin replacement strategy of the 4-way set-associative cache in the ColdFire MCF 5307 effectively renders the associativity useless for WCET analysis. The use of a single 2-bit counter for the whole cache destroys age information within the cache sets. Slightly more complex pipelines, with branch prediction and out-of-order execution, need an integrated pipeline and cache analysis to provide useful WCET bounds. Such an integrated analysis is complex and also demanding with respect to the computational effort. Consequently, Heckmann et al. [14] 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. Further suggestions for time-predictable architectures and memory hierarchies are given in [57]. Another project on time-predictable architectures is the PRET project [7] where scratchpad memories are suggested instead of caches.

An overview on WCET analysis in general and various WCET tools from industry and academia is given in [56].

Area	Address predictability	Spacial locality	Coherence
Constants	+	−	No
Stack	◦	+	No
Global and static	+	−	Yes
Type dependent	◦	−	No
Heap (headers)	−	◦	Partial
Heap (objects)	−	−	Yes
Heap (arrays)	−	+	Yes

Table 1 Properties of Data Areas

4 Data Areas

In the following, we describe various data areas. Although we use Java terminology, we also provide links to the respective memory areas in C/C++. In these languages, memory areas are usually divided into the `.text`, `.data` and `.bss` linker segments, heap data (allocated through `malloc()`) and the stack.

Table 1 summarizes key features of various memory areas. The details and rationale for this classification is provided in the following sections.

4.1 Constants

In procedural languages, such as C, the constant area primarily contains string constants and is small. For object oriented languages, such as C++ and Java, the virtual methods tables and class related constants consume a considerable amount of memory. The addresses of the constants are known after program linking and are simple to handle in the WCET analysis.

On a uniprocessor system the constant area and the static data can share the same cache. For CMP systems, splitting the static data cache and the constant cache is a valuable option. In contrast to static data, constants are per definition immutable. Therefore, cache coherence and consistence do not need to be enforced and the resulting cache is simpler and can be made larger.

Another option for constants is to embed them into the code. Many instruction sets allow constructing large constants directly. For cases where this is not possible, support for a PC relative addressing mode is needed. However, this option is only practical for a few constants. Furthermore, if the address range for the PC relative addressing is restricted, some tables would need to be duplicated, increasing the code size.

In C/C++, constants are traditionally placed in the `.data` linker segment. Some compilers (e.g., GCC) place constant data in the `.rodata` segment, indicating that the data is read-only. Constants that are embedded into the code are placed in the `.text` segment.

4.2 Stack

Local variables are allocated on a stack frame. As the access frequency of local variables is very high, this data area benefits from caching. The addresses of the local variables are easy to predict when the call tree is known [24]. A cache that serves only stack allocated data can be optimized compared to a standard data cache. In the following such a special stack cache is described.

A new stack frame for a function call does not need to be cache consistent with the main memory. The cache blocks for the new stack frame can be allocated without a cache fill from the main memory. On a return, the previously used cache blocks can be marked invalid, as function local data is not accessible after the return. As a result, cache lines will never need to be written back after returning from a function. The stack cache activity can be summarized:

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

The regular access pattern to the stack cache will not benefit from set associativity. Therefore, the stack cache is a simple direct mapped cache. However, the stack cache exhibits spatial locality. Even without large cache lines, this can be taken advantage of when performing cache fills at function returns. Stack data is thread local and needs no cache coherence protocol in a chip-multiprocessor system.

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

4.3 Global and Static Data

For conservatively written programs with statically allocated data, the address of the data is known after program linking. The addresses are the input for the cache analysis. In [9], control tasks from a real-time benchmark were analyzed. For this benchmark 90% of the memory accesses were predicted precisely.

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

Global and static data is placed in the `.data` and `.bss` segments in C/C++. Addresses for global data in C/C++ are easy to predict. In Java global data is allocated on the heap, and large amounts of static fields are uncommon. Therefore, the pressure to handle heap allocated data is higher in Java than in C/C++ code.

4.4 Type Dependent Data

Virtual method and interface dispatch tables are constant, but their address depends on the actual type of the object. Receiver type analysis, as implemented in our program analysis tool, can narrow the possible types. Type dependent data can be cached together with the constant area if the receiver type analysis delivers reasonably precise type sets. Otherwise a distinct cache for this area with moderate associativity decouples constant access analysis from the type dependent access analysis.

4.5 Heap Allocated Objects

In object-oriented languages the objects are usually allocated in a data area called the heap. The main issue for WCET analysis are statically unknown object addresses. If the heap shares the data cache with other data areas, a single access to a heap allocated data structure destroys the abstract cache state for one way.

To avoid this coupling between heap allocated data structures and other data areas in the WCET analysis, we propose to use a distinct cache for heap data. This dedicated cache can be optimized for the heap data area.

Different objects can be tracked symbolically by the analysis, but the address of an object is not known before runtime. A cache of l cache lines with an associativity of n has $a = \frac{l}{n}$ lines per way. Which of the a lines is used depends on part of the address. As the address is not known statically, all a lines have to be merged by the analysis. Therefore, a cache with an associativity of n is needed to track n different objects. Nothing is gained by providing more than n cache lines.

We propose to implement the cache architecture according to the possibilities of the WCET analysis – a small, fully associative cache similar to a victim cache [20].

A cache for the heap allocated objects can be implemented in (at least) two organizations: (1) caching individual fields or (2) caching larger parts of an object in a cache line. Which organization performs better, with respect to the WCET, depends on the spatial locality of field accesses and the latency and bandwidth of the main memory. In the analysis either individual fields or whole objects need to be tracked. We have implemented the analysis for both types of object caches and compare the hit rates in the evaluation section. It has to be noted that we are not suggesting to adapt the cache structure for an individual application, but find out which organization works best for the domain of embedded real-time applications.

Depending on the concrete analysis, the replacement policy shall be either LRU or FIFO. When LRU replacement is too expensive for high associativities, replacement of the oldest block gives an approximation of LRU. The resulting FIFO strategy can be used for larger caches. To offset the less predictable behavior of the FIFO replacement [33], the cache has to be larger than an LRU based cache.

4.5.1 Header Data and Handles

Additionally to the actual data, objects have meta-data associated with them. This header data contains auxiliary information for the run-time system like the type of an object and information for garbage collection. This header data is usually only modified by the run-time system. For example, the type of an object remains immutable during the lifetime of an object. As changes to the header data occur in a controlled fashion, this data can be considered pseudo-constant and does not require full cache coherence support.

On JOP, the header data is allocated in a *handle area*. Objects are accessed through pointers (the *handles*), which simplifies garbage collection considerably. As all field accesses must follow this indirection, the handles are obvious candidates for caching.

This object layout also lends itself to a cache organization, where the handle instead of the object address can be used to index into the cache. In that case the indirection for the access is implicitly resolved on a cache hit.

Depending in the length of one cache line (either a single field, or part of the object), the field index is either appended to the handle for the cache tag memory comparison, or used to index into the cache line.

4.6 Heap Allocated Arrays

As access to arrays benefits mainly from spatial locality we propose prefetch and write buffers for array accesses. For operations on two arrays (e.g., vector operations) two prefetch buffers are needed. Each array shall continue to use its own prefetch buffer. Which prefetch buffer will be used depends on the base address of the array. In Java, array access bytecodes consist of a base address and an index into the array. Therefore, the base address can be used for the tag memory of the prefetch buffer. For C based languages on a standard processor target, support from the compiler and the instruction set needs to be added to distinguish between the two arrays and also between array access and other memory operations.

For array operations, such as string copy, one prefetch buffer and one write buffer is needed. However, a write buffer introduces timing dependencies between unrelated array write instructions. The write timing can be localized with a write buffer flush instruction, inserted by the compiler at the end of the write loop.

4.7 Memory Access Types

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

– The address is always known statically. This is the case for static variables, where the address is resolved at link time, as well as for constants.
– The address depends on the dynamic type of the operand, but not on its value. Therefore, the set of possible addresses is restricted by the receiver types determined for the call site. The class meta-data and the virtual method dispatch tables belong to this category.
– The address depends on the value of the object reference. The data resides on the heap and the actual address is unknown. Object meta-data, instance fields and arrays allocated at runtime belong to this category. For fields and arrays, in addition to the symbolic address a relative offset is known.

4.8 Access Type Distinction

In order to benefit from different caches for different data areas the load and store units need a mechanism to distinguish between the different memory areas. One possibility is to use typed load/store instructions. For a RISC based architecture the instruction set needs to be extended and the compiler has to emit the typed load/store instructions. In Java bytecode, typed memory access is already part of the instruction set. For example, object fields are accessed with bytecode getfield and putfield. There are no *general* load/store instructions availabel in Java bytecode. Therefore, we use typed memory accesses in the Java processor JOP.

Another option is to use different memory segments for different data areas. In that case, standard load/store instructions can be used and no changes in the compiler are needed. The linker and program loader are responsible for the correct placement. The memory management unit can be extended to communicate the information about the typed memory pages to the cache subsystem.

5 Heap Cache Analysis

The cache analysis of heap allocated data has to handle data with unknown addresses. We therefore require a fully associative heap cache, whose replacement behavior can be analyzed independently of the actual physical address of the object. The basic idea of our analysis is to represent objects using access paths [5], and find the set of objects each instruction might use.

A points-to analysis [8] associates each variable with a set of memory locations it might reference. As the set of objects used in a program usually is of unbounded size, objects are grouped into equivalence classes. One class consists, e.g., of all objects with the same type, or allocated at the same site.

We perform a local points-to analysis with respect to a program fragment, or scope, maintaining a set of objects that variables defined outside the scope may point to. The heap cache analysis, however, has to determine the exact set of distinct objects accessed during the execution of the scope. Therefore, each object in the analysis model must correspond to at most one concrete object.

Objects are represented by one access path pointing to them at the beginning of the scope. An access path consists of a root object defined outside the currently analyzed scope, and a sequence of fields and array indices accessed. Examples include global1.field1, arg1.field0.field1 or arg2[2].x[1].

Object allocations are only handled if they are executed at most once in the program fragment (e.g., allocate at the beginning). In this case, a fresh variable name is used for the object. For array accesses, the same strategy is used if the instruction is executed only once. Otherwise, we use the results of a preceding value analysis, assuming one access for each possible value of the index.

Finally, the analysis maintains sets of objects a type may alias to, due to heap modifications within the scope. The result of the analysis associates each heap access instruction with the set of objects it may access.

5.1 Dataflow Analysis

The dataflow analysis is performed separately for each program fragment of interest. The underlying lattice is a pair $\langle P, A\rangle$. P associates all static fields and all variables defined in outer scopes with a set of access paths. A is used to take heap modifications into account, mapping types (classes) to the set of access paths that variables of this type may alias with. Both P and A are finite maps, whose range type are sets with a maximal size k. All sets with size greater than k are in one equivalence class ($\top$).

Initialization At the entry edges of the scope analyzed, P maps all variables v defined outside the scope to their name $\overline{v}$ and for all types τ, $A(\tau) = \emptyset$.

Join The join $\langle P, A\rangle$ of two values $\langle P_1, A_1\rangle$ and $\langle P_2, A_2\rangle$ is defined as a element-wise set union.

$$P(v) = P_1(v) \cup P_2(v)$$
$$A(\tau) = A_1(\tau) \cup A_2(\tau)$$

Transfer There are two different kind of transfer equations:

$$[\![e_1 := e_2]\!]_P(v) = X$$
$$[\![e_1 := e_2]\!]_A(\tau) = Y$$

In the first kind of equation, the statement $e_1 := e_2$ changes the points-to set $P(v)$ to X. In the second one, the alias set $A(\tau)$ is changed to Y. The equations for those entries of P or A that do not change are not further considered during the analysis. The expression $\tau(v)$ denotes set of possible runtime types of variable v, determined by a preceding type analysis.

Variables and Static Fields: If v_1, v_2 are local variables or static fields (global variables) with reference type, we have

$$[\![v_1 := v_2]\!]_P(v_1) = P(v_2)$$

Instance Fields Access: Consider the assignment $v_1 := v_2.F$, with F being an instance field (member variable) with reference type. The objects v_1 may point to are obtained by appending F to all access paths v_2 may point to, and adding all possible aliases for $\tau(v_1)$.

$$\begin{aligned}[\![v_1 := v_2.F]\!]_P(v_1) &= P_f \cup A(\tau(v_1)) \\ \textit{where } P_f &= \{\ n.F \mid n \in P(v_2)\ \}\end{aligned}$$

Array Access and Object Creation: Both when fetching an object from an array and when creating a new object, we distinguish whether the instruction is executed at most once in the analyzed scope ($:=^1$) or not ($:=^*$). $\pi(i)$ denotes the interval assigned to the index expression e by the preceding value analysis.

$$\begin{aligned}[\![v :=^1 newT]\!]_P(v) &= \{\ n_{new}\ \} \\ [\![v :=^* newT]\!]_P(v) &= \top \\ [\![v_1 :=^1 v_2[e]]\!]_P(v_1) &= \{\ n_{new}\ \} \\ [\![v_1 :=^* v_2[e]]\!]_P(v_1) &= P_a \cup A(\tau(v_1)) \\ \textit{where } P_a = \{\ n[i]\ &\mid\ n \in P(v_2),\ i \in \pi(i)\ \}\end{aligned}$$

Heap Modifications: When writing to an instance field or an array of objects, the alias information is updated.

$$\begin{aligned}[\![v_1.F := v_2]\!]_A(T \in \tau(v_2)) &= A(T) \cup P(v_2) \\ [\![v_1[e] := v_2]\!]_A(T \in \tau(v_2)) &= A(T) \cup P(v_2)\end{aligned}$$

The dataflow analysis is run several times from the WCET tool, which uses the results to add ILP constraints restricting the number of cache misses.

5.2 Heap Cache WCET Analysis

Most of the published techniques for instruction cache analysis try to classify cache accesses as hit (the referenced value is always cached), or miss (the referenced value is never cached). This also works for certain data cache areas, when the accessed memory address is statically known. If a reference may point to one out of several addresses, simply introducing a non-deterministic access and performing a hit/miss classification will not lead to satisfying results. An alternative analysis approach is to perform a persistence analysis. A persistent reference is one which is missed the first time accessed, and then stays in the cache during the execution of a program fragment.

Our cache analysis framework, which is also used for the method cache analysis [43], tries to identify scopes where all object references are persistent. For those scopes, ILP constraints stating that each object is missed at most once per execution of the scope are added. Lexical scopes (methods, loops, blocks) suggest themselves for such an analysis, though less regular shaped subgraphs of the program's control flow graph work as well.

In addition to its simplicity, this technique has the advantage that it works for both LRU and FIFO caches. In order for this analysis to be sound, the timing of the cache must not influence the timing of other components.

A scope graph is a graph whose vertices are scopes, and which has an edge from scope S_1 to scope S_2 if the control flow graph nodes contained in S_2 are are subset of those in S_1. We require that each basic block is in at least one scope, and that $S_1 \cap S_2 = \emptyset$ if neither $S_1 \subseteq S_2$ nor $S_2 \subseteq S_1$.

Let N be the associativity of the heap cache. The objects in a scope are persistent, if at most N distinct objects are accessed during one execution of that scope. This check is implemented using IPET, with a similar ILP model as the one used in the actual WCET analysis. We add one binary variable for each accessed address. With the objective function set to the sum of those binary variables, we obtain a bound on the maximum number of distinct objects accessed within the scope.

To include the heap cache in the IPET-based WCET analysis, the analysis first identifies the set of relevant scopes R. A scope is relevant if all addresses are persistent, but this is not the case in at least one parent scope. Then parts of the control flow model are duplicated to ensure that no method is both called from within and outside a relevant scope.

In a second step, ILP constraints restricting the number of cache misses in each *relevant* scope $S \in R$ are added. For each relevant scope S, we add two integer variables $a_{S,hit}$ and $a_{S,miss}$ for each address a possibly accessed in S.

Let $i \in \mathcal{I}_a$ be the set of instruction accessing a in S. $f(i)$ denotes the linear expression for the execution frequency of i. Now the following facts are modeled in the ILP:

– An access to an object field is either a hit or miss:

$$f(a_{S,miss}) + f(a_{S,hit}) = \sum_{i \in \mathcal{I}_a} f(i)$$

– If the scope is executed $f(S)$ times, we have at most $f(S)$ cache misses:

$$f(a_{S,miss}) \leq f(S)$$

– Each time an object field is accessed, time to access the cache has to be added to the WCET. With $c(a_{hit})$ and $c(a_{miss})$ being the time needed for a cache hit respectively miss, the following linear expression is added to the objective function:

$$c(a_{hit})f(a_{S,hit}) + c(a_{miss})f(a_{S,miss})$$

Not that $c(a_{miss})$ is the worst-case miss time, which includes a possible L2 cache miss and memory bus arbitration times.

Together with the other parts of the IPET formulation, the solution of the ILP gives us the worst-case execution time, including the cost for the object cache. Analyzing the optimal solution, we find that

$$\sum_{S \in \mathcal{S}} a_{S,miss}$$

corresponds to the number of times the address a has to be loaded into the cache on the worst case path.

6 Evaluation

We have implemented various split cache configurations and an object cache in the context of the Java processor JOP and the data cache analysis for the heap cache.

For the benchmarks we use two benchmark collections: JemBench [41], which is a collection of embedded Java programs, and GrinderBench, which contains programs for mobile Java. Furthermore, jPapaBench is a Java port of the real-time benchmark PapaBench [27]. JemBench includes two real-world embedded applications [34]: Kfl is one node of a distributed control application and Lift is a lift controller deployed in industrial automation. The Kfl example is a very static application, written in conservative, procedural style. The application Lift was written in a more object-oriented style.

6.1 Access Type Frequency

Before developing a new cache organization we run benchmarks to evaluate memory access patterns. Figure 1 shows the access frequencies for the different memory areas for all benchmarks. The different categories are:

ARRAY:	heap allocated arrays
STATIC:	static fields
FIELD:	fields of heap allocated objects
INTERN:	JVM internal memory access (e.g., garbage collection)
ALEN:	array length
HANDLE:	object header
CONST:	constant data
CLINFO:	class information

The benchmarks show a considerable different pattern on access type distribution. Common to all benchmarks is a very low write frequency. Therefore, write-through caches are a good design choice for a Java processor. Most benchmarks access the class information (e.g., method dispatch table) relative frequently. Application that access array elements also access the array length

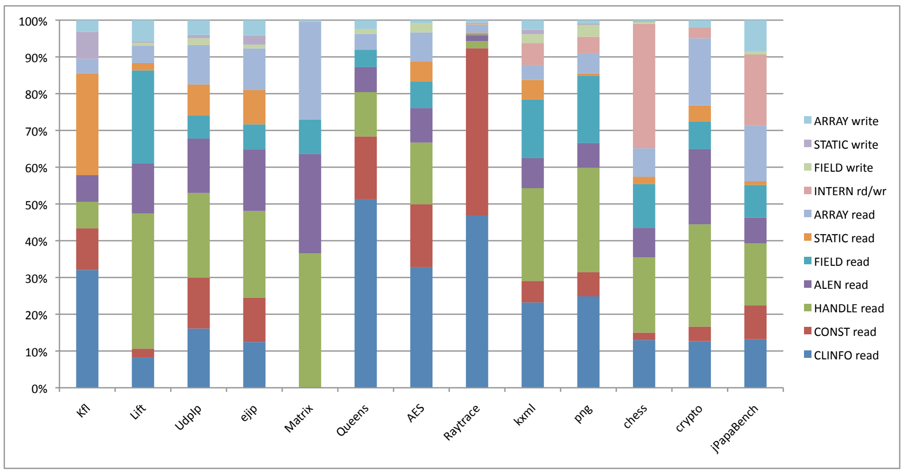

Fig. 1 Data memory traffic to different memory areas (in % of all data memory accesses)

field, as bounds checks are mandatory in Java. Two benchmarks are considerable different to the rest: Kfl and Raytrace. Kfl has a high access frequency to static fields, but no object field accesses. Kfl was written in a very procedural style and uses no object oriented features of Java. Raytrace is a number crunching benchmark, which is dominated by floating point operations.

The benchmarks Matrix, crypto, and jPapaBench are dominated by a relative high array access frequency. Therefore, those applications will not benefit from the object cache architecture.

In general, the different benchmarks show quite different distributions of access types. Finding a good balance of resource distribution for the individual cache areas is an interesting topic of future work.

6.2 Cache Implementation in JOP

The Java processor JOP [35] traditionally includes two caches: a method cache and a stack cache. Our implementation adds two more caches: a direct-mapped cache for data with predictable addresses, and a fully associative cache for heap-allocated data. The fully associative cache is configurable for LRU and FIFO replacement.

Accesses to constant and static data access the direct-mapped cache. In multi-processors these accesses should be split further, because cache coherence mechanisms can be omitted for constant data. The fully associative cache handles accesses to handles and object fields. Accesses to array elements are not cached. Full associativity is expensive in terms of hardware, and therefore the size of this cache is limited to 16 or 32 cache lines.

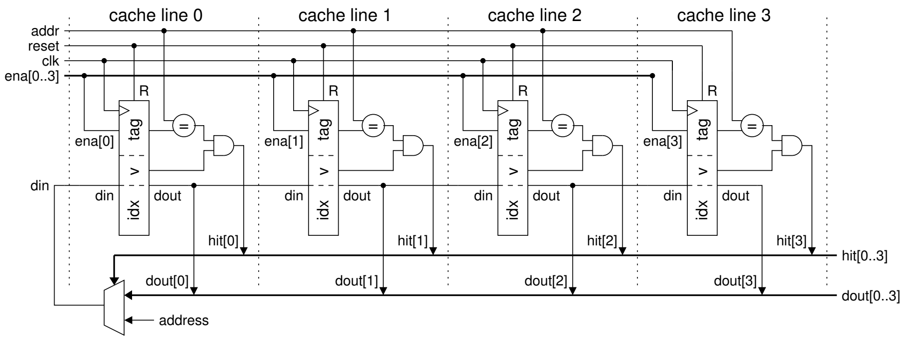

Fig. 2 LRU tag memory implementation

Table 2 Implementation results for LRU and FIFO based data caches

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

6.3 LRU and FIFO Caches

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

Table 2 shows the resource consumption and maximum frequency of the LRU and FIFO cache. The resource consumption is given in logic cells (LC) and in memory bits. As a reference, a single core of JOP consumes around 3500 LCs and the maximum frequency in the Cyclone-I device without data caches is 88 MHz. We can see the impact on the maximum frequency of the large multiplexer in the LRU cache on configurations with a high associativity.

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

Although it is known that FIFO based caches are hard to analyze with common techniques, the simpler implementation (less hardware resources, higher clock frequency) is an argument for a different analysis approach, such as the analysis presented in Section 5. If, e.g., in the context of a loop, the maximum read set on the heap fits into the FIFO cache, all accesses can be classified as *miss once.*

6.4 Object Cache

Besides the fully associative single word cache we have implemented an object cache, further optimized for the object layout of JOP [38]. The object cache is organized to cache whole objects in a cache line. Each cache line can only contain a single object. Objects cannot cross cache lines. If the object is bigger than the cache line, the fields at higher indexes are not cached. Furthermore, the implementation in JOP is optimized for the object layout of JOP. The objects are accessed via an indirection called the handle. This indirection simplifies compaction during garbage collection.

The tag memory contains the pointer to the handle (the Java reference) instead of the effective address of the object in the memory. If the access is a hit, additional to the field access the cost for the indirection is zero – the address translation has already been performed. The effective address of an object can only be changed by the garbage collection. For a coherent view of the object graph between the mutator and the garbage collector, the handle cache needs to be updated or invalidated after the move. The object fields can stay in the cache.

To enable static cache analysis the cache is organized as write through cache. Write back is hard 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 CMP system [32]. The cache line is not allocated on a write.

6.4.1 Implementation

Figure 3 shows the design of the object cache. In this example figure the associativity is two and each cache line is four fields long. All tag memories are compared in parallel with the object reference. Therefore, the tag memory uses dedicated registers and cannot be built from on-chip memory. Parallel to the tag comparison, the valid bits for the individual fields are checked. The field index performs the selection of the valid bit multiplexer. The output of the tag comparisons and valid bit selection is fed into the encoder, which delivers the selected cache line. The line index and the field index are concatenated and build the address of the data cache. This cache is built from on-chip memory. As current FPGAs do not contain asynchronous memories, the input of the data memory contains a register. Therefore, the cache data is available one cycle later. The hit is detected in the same cycle as reference and index are available in the pipeline, the data is available one cycle later.

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

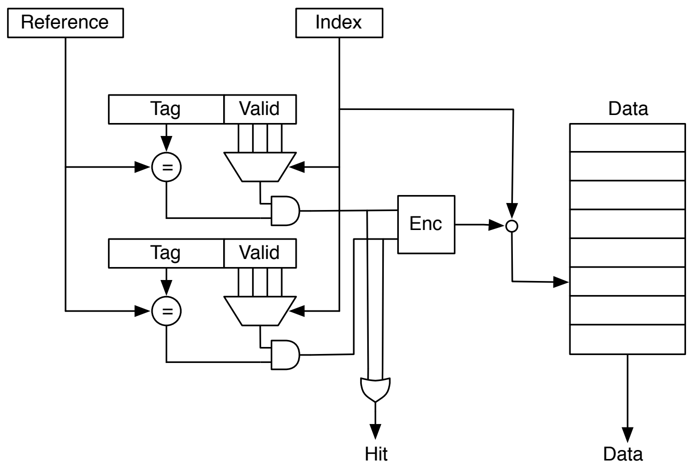

Fig. 3 Object cache with associativity two and four fields per object

a getfield with a cache miss takes

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

cycles. Two memory access times are needed, as the handle indirection and the actual field value have to be read. If the access is a cache hit, the execution time of getfield is

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

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

6.4.2 Resource Consumption

The object cache consumes considerable logic cells (LC) for the implementation of the tag memory and the parallel comparators. The data memory can be implemented in on-chip memory. In the FPGA that was used for the evaluation the size of an on-chip memory block is 4 KBit. Depending on the size of the data memory, the synthesize tool uses logic cells for small memories instead of an on-chip memory block. Table 3 shows the resource consumption in logic cells (LC) and memory bits for different cache configurations. The main overhead, compared to a standard cache or a scratchpad memory, comes from the tag memory, which consumes logic cells.

Table 3 Resource consumption of different configurations of the object cache

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

Table 4 Implementation results for a split cache design

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

6.4.3 Adaption for Arrays

The object cache is only used for objects and not for arrays. The access behavior for array data is quite different as it explores more the spatial instead of the temporal locality. A variation of the cache for arrays is currently being implemented. The array cache loads full cache lines and has additional to the array base address also the array index of the first element in the cache line in the tag memory. Furthermore, the cache includes also the array length and the mandatory array bounds check is performed in the array cache.

6.5 Split Cache Implementation

Table 4 shows the resources and the maximum system frequency of different cache configurations. The first line gives the base numbers without any data cache. From the resource consumptions we can see that a direct mapped cache is cheap to implement. Furthermore, the maximum clock frequency is independent of the direct mapped cache size. A highly associative LRU cache (i.e., 32-way and more) dominates the maximum clock frequency and consumes considerable logic resources.

We have implemented the proposed analysis of the heap cache in the dataflow analysis and WCET framework for JOP [43]. The object reference analysis is performed for each scope separately, and relies on a preceding type analysis to deal with dynamic dispatch. The results are then used to add ILP constraints to the WCET formulation. As we are interested in the performance of the cache analysis, we present the analyzable hit rate of the cache instead of the execution time of the application.

To evaluate the effectiveness of the heap cache analysis, we performed a simplified WCET computation only considering the cost of object field misses. We assume a fully associative cache with N ways. The two different heap cache organizations are compared. The single word cache, caches individual fields and on a miss only that field is loaded into the cache. The object cache, caches complete objects (with up to 32 fields). On a miss the whole object is loaded into the cache. Therefore, the second configuration will also benefit from spatial locality. At the same associativity the object cache will result in a higher hit rate. However, the miss penalty is higher as more words are transferred into the cache.

Table 5 lists the result for our benchmark applications. The column N gives the associativity, the value of 0 means no cache and serves as reference on the maximum possible misses. The hit ratio is relative to this value.

In the Lift benchmark, a controller method is analyzed. With the single word cache the hit ratio is at 85 % for 16 lines, indicating that between 9 and 16 different fields are accessed. The object cache indicates with a hit rate of 97 % at an associativity of 2 that only two different objects are read in the controller method.

Both the UdpIp and Ejip benchmarks are network stack implementations. UdpIp has a good analyzed hit rate with an object cache of four lines. The single word cache results in about 2/3 of the hit rate. Therefore, this benchmark benefits from spatial access locality in the object cache.

For the Ejip benchmark, the analysis is relatively imprecise, because some objects which always alias at runtime are considered to be distinct in the analysis. A global must-alias or shape information would significantly improve the result.

Table 6 shows the heap cache measurement with the simulation of the caches in JopSim. As we cannot trigger the worst-case path in the simulation the number of heap accesses is lower than in the analysis table. However, the trend in the measured values is similar to the analysis results. This indicates that the scope based analysis is a reasonable approach to WCET analysis of caches for heap allocated data.

Simulations with larger workloads (DaCapo benchmarks) show a hit rate of 70% to 90% for small heap caches [39]. In this paper it is also shown that most hits stem from temporal locality and not spatial locality. Due to the low spatial locality it is more beneficial, at least for the average case, to update single words in a cache line instead of filling the whole line on a miss. For the

Table 5 Evaluation of the heap cache analysis

		Single word cache		Object cache	
Benchmark	N	Misses	Hit ratio	Misses	Hit ratio
Lift	0	228	0.00 %	228	0.00 %
	1	228	0.00 %	221	3.07 %
	2	228	0.00 %	6	97.37 %
	4	195	14.47 %	3	98.68 %
	8	35	84.65 %	3	98.68 %
	16	24	89.47 %	3	98.68 %
	32+	23	89.91 %	3	98.68 %
UdpIp	0	48	0.00 %	48	0.00 %
	1	48	0.00 %	16	66.67 %
	2	46	4.17 %	10	79.17 %
	4	18	62.50 %	4	91.67 %
	8	16	66.67 %	4	91.67 %
	16	12	75.00 %	4	91.67 %
Ejip	0	103	0.00 %	103	0.00 %
	1	103	0.00 %	71	31.07 %
	2	97	5.83 %	72	31.07 %
	4	87	15.53 %	57	44.66 %
	8	81	21.36 %	38	63.11 %
	16	68	33.98 %	38	63.11 %
	32	68	33.98 %	38	63.11 %
	64+	68	33.98 %	34	66.99 %

fully associative organization this is even true for a main memory based on SDRAM devices.

A small fully associative cache for the heap allocated data results in a moderate hit rate. As we are interested in time-predictable CMP systems [29], where the requirements on the memory bandwidth are quite hight, even the moderate hit rates will give a considerable WCET performance increase.

6.7 WCET Based Evaluation

We have included the implemented object cache in the WCET analysis tool. Besides the benchmarks from JemBench we have also included tasks from the Java port of the PapaBench WCET benchmark [27,21].

Table 7 shows the WCET analysis results of tasks. The WCET is given in clock cycles for different cache configurations. All object cache configurations use a line size of 16 words. The analysis is configured to assume single word loads on a miss, as it is implemented in the hardware. The table shows results for different number of lines. The column with 0 lines is the reference where all object accesses are served by the main memory. As we have seen from the hit rate analysis and simulation, the efficiency of the object cache saturates at 4 to 16 lines. Therefore, we limit the analysis to up to 16 lines. That means scopes with up to 16 different objects are tracked by the WCET analysis.

Table 6 Simulation of average case heap cache hits

		Single word cache		Object cache	
Benchmark	N	Misses	Hit ratio	Misses	Hit ratio
Lift	0	122	0.00 %	122	0.00 %
	1	101	17.21 %	95	22.13 %
	2	17	86.07 %	3	97.54 %
	4	15	87.70 %	3	97.54 %
	8	15	87.70 %	3	97.54 %
	16+	15	87.70 %	3	97.54 %
UdpIp	0	41	0.00 %	41	0.00 %
	1	19	53.66 %	3	92.68 %
	2	13	68.29 %	2	95.12 %
	4	11	73.17 %	2	95.12 %
	8	10	75.61 %	2	95.12 %
	16+	10	75.61 %	2	95.12 %
Ejip	0	73	0.00 %	73	0.00 %
	1	57	21.91 %	27	63.01 %
	2	52	28.77 %	22	69.68 %
	4	47	35.62 %	17	75.34 %
	8	45	38.36 %	15	79.45 %
	16	41	43.84 %	12	83.56 %
	32	34	53.42 %	12	83.56 %
	64	34	53.42 %	12	83.56 %

Table 7 WCET analysis results in clock cycles, including the object cache

	Object cache configuration (number of lines)			
Benchmark	0	4	8	16
Lift	7619	6609	6609	6609
UdpIp	127318	127158	127168	127168
Ejip	38083	38023	37973	37973
CheckMega128Values	9144	9029	9144	9104
Navigation.courseComputation	242512	242512	242472	242472
Navigation.update	32465419	32470574	32465254	32465169
RadioControl	65203	65138	65253	65213
AltitudeControl	29060	29070	29030	29030
ClimbControl	138170	138235	138105	138105
Stabilization	165874	165769	165884	165879
SendDataToAutopilot	11504	11434	11859	11849
TestPPM	3470	3440	3380	3380
CheckFailsafe	72602227	72561317	72561317	72561227

Most benchmarks show only a very small improvement in the WCET. The best result is shown with the Lift benchmark, where the WCET is reduced by 15% with a 4 line object cache. This result is in line with the analyzed hit rate as shown in Table 5. Although the analyzed hit rate of UdpIp and Ejip are reasonable, the effective reduction of the WCET is minimal. Looking at the WCET path, as it is reported by the WCET tool, reveals that the WCET is dominated by the checksum calculation of IP packets. The number of object field accesses is quite low: as shown in Table 5, the worst-case object accesses is 48 for UdpIp and 103 for Ejip. In the best case, when almost all accesses could be classified as hit, the WCET could be reduced by about 250 and 500 clock cycles with our current memory subsystem.

From the jPapaBench tasks only a few have a noticeable reduction of their WCET when an object cache is used. The WCET path is often dominated by floating point operations, which are in the current version of JOP implemented in software.

Some benchmarks actually increase the WCET with larger object caches. This fact can be explained by the phenomena that larger scopes, which are possible with more cache lines, can also introduce a higher uncertainty of the receiver objects. However, the WCET of a smaller object cache is a safe approximation of the larger object cache. The same effect can even happen when comparing the non-cached WCET with the 4 line object cache, as seen with benchmark ClimbControl.

The WCET reduction is less than what we expected from the first evaluation with analyzable miss rates. Non object-oriented code still dominates the execution time. The first three benchmarks are written in a relative conservative programming style and the original code of PapaBench is in C. Therefore, they are not written in *typical* Java style. We hope that analyzable caching for object-oriented programming will enable development of real-time applications in a more object-oriented style. We also expect that the upcoming standard for safety-critical Java [23] will increase usage of Java in real-time systems. In that case caching of objects will become more important.

7 Conclusion

Caching accesses to different data areas complicate WCET analysis of the data cache. Accesses to unknown addresses, e.g., for heap allocated data structures, destroy information about the cache for other, simpler to predict, data areas. We propose to change the data cache architecture to enable tighter data cache analysis. The data cache is split for different data areas. Besides enabling a more modular data cache analysis, those different data caches can be optimized for their data area. A cache for the stack and for constants can be a simple direct mapped cache, whereas the cache for heap allocated data has a high associativity to track objects with statically unknown addresses.

For heap allocated objects we presented a scope-based local persistence analysis. In the analysis we considered two different heap cache organizations: a single word cache and an object cache. Both configurations result in a moderate to good hit classification. The effect on the WCET of tasks has been evaluated with real-time benchmarks. As those benchmarks are still written in a more procedural style, the WCET is dominated by non object-oriented operations and the effect of a object cache is minimal. However, a solution to predict heap access cache hits can reduce the entry cost of using object-oriented languages in real-time applications.

Acknowledgement

We would like to thank the anonymous reviewers for their detailed comments, which helped to improve the paper. This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Source Access

The analysis tool, which includes the heap cache analysis, is part of the JOP source distribution and available from `http://www.jopdesign.com/`. At the time of this writing the cache analysis is not yet included in the master branch, but in the remote branch splitcache_analysis.

References

1. Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*, pages 318–326, New York, October 30 November 01 2003. ACM Press.
2. Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.
3. Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.
4. José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.
5. 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.
6. Jean-Francois Deverge and Isabelle Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.
7. Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.
8. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. pages 242–256, 1994.
9. Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.
10. Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

11. Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1995. ACM.
12. Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.
13. Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.
14. Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.
15. John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.
16. M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.
17. Jörg Herter and Jan Reineke. Making dynamic memory allocation static to support WCET analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.
18. Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.
19. Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, doi: 10.1002/cpe.1763, 2011.
20. Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.
21. Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive testing of safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, pages 164–174, New York, NY, USA, 2010. ACM.
22. S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 230–240, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.
23. Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. Safety-critical Java technology specification, public draft, 2011.
24. Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*, pages 255–262. IEEE Computer Society, 1999.
25. Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM.
26. V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay. A new cache architecture concept: the split temporal/spatial cache. In *Electrotechnical Conference, 1996. MELECON '96., 8th Mediterranean*, volume 2, pages 1108–1111 vol.2, May 1996.
27. Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.
28. David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.
29. Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

30. Isabelle Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.
31. Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2007)*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.
32. 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.
33. Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.
34. Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.
35. Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
36. 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.
37. Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.
38. Martin Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.
39. Martin Schoeberl, Walter Binder, and Alex Villazon. Design space exploration of object caches with cross-profiling. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 213–221, Newport Beach, CA, USA, March 2011. IEEE Computer Society.
40. Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 326–333, Sierre, Switzerland, March 2010. ACM Press.
41. Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.
42. 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 5860 in LNCS, pages 180–191. Springer, November 2009.
43. 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.
44. Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.
45. Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 223–232. IEEE Computer Society, 2005.
46. Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-03)*, volume 31, 1 of *Performance Evaluation Review*, pages 272–282, New York, June 11–14 2003. ACM Press.
47. Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 154–165. IEEE Computer Society, 2003.

48. Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7:4:1–4:38, December 2007.
49. Manish Verma and Peter Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst*, 14(8):802–815, 2006.
50. Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE2005).*, pages 600–605 Vol. 1, March 2005.
51. Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, pages 275–282, Tokyo, Japan, March 2009. IEEE Computer Society.
52. Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, 1999.
53. Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2009)*, 2009.
54. Jack Whitham and Neil Audsley. Investigating average versus worst-case timing behavior of data caches and data scratchpads. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 165–174, Washington, DC, USA, 2010. IEEE Computer Society.
55. Jack Whitham and Neil Audsley. Studying the applicability of the scratchpad memory management unit. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 205–214, Washington, DC, USA, 2010. IEEE Computer Society.
56. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.
57. Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

Martin Schoeberl has a PhD from the Vienna University of Technology. Since 2005 he has been Assistant Professor at the Institute of Computer Engineering. 2010 he joined the Technical University of Denmark as Associate Professor. His research interest is in time-predictable computer architecture and real-time Java.

Benedikt Huber is a research and teaching assistant at the Institute of Computer Engineering at the Vienna University of Technology. He received his Computer Science Master's degree from the Vienna University of Technology in 2009. His current research focus is on time predictable systems, and on the WCET analysis of object-oriented languages.

Wolfgang Puffitsch is a research and teaching assistant at the Institute of Computer Engineering at the Vienna University of Technology. He has studied Computer Engineering at the Vienna University of Technology and received his Dipl.-Ing. (Master's) degree in 2008. Currently, he is working on time-predictable computer architectures and on his PhD on real-time garbage collection for multi-processor systems.