

A Time Predictable Instruction Cache for a Java Processor

Martin Schoeberl

JOP.design, Vienna, Austria
martin@jopdesign.com

Abstract. Cache memories are mandatory to bridge the growing gap between CPU speed and main memory access time. Standard cache organizations improve the average execution time but are difficult to predict for worst case execution time (WCET) analysis. This paper proposes a different cache architecture, intended to ease WCET analysis. The cache stores complete methods and cache misses occur only on method invocation and return. Cache block replacement depends on the call tree, instead of instruction addresses.

1 Introduction

Worst case execution time (WCET) analysis [1] of real-time programs is essential for any schedulability analysis. To provide a low WCET value, a good processor model is necessary. However, the architectural advancement in modern processor designs is dominated by the rule: *'Make the common case fast'*. This is the opposite of *'Reduce the worst case'* and complicates WCET analysis.

Cache memory for the instructions and data is a classic example of this paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET value. Plenty of effort has gone into research into integrating the instruction cache in the timing analysis of tasks [2, 3] and the cache's influence on task preemption [4, 5]. The influence of different cache architectures on WCET analysis is described in [6].

We will tackle this problem from the architectural side — an instruction cache organization in which simpler and more accurate WCET analysis is more important than average case performance. In this paper, we will propose a method cache with a novel replacement policy. The instruction set of the Java virtual machine contains only relative branches, and a method is therefore only left when a return instruction has been executed. It has been observed that methods are typically short [7] in Java applications. These properties are utilized by a cache architecture that stores complete methods. A complete method is loaded into the cache on both invocation and return. This cache fill strategy lumps all cache misses together and is very simple to analyze.

2 Cache Performance

In real-time systems we prefer time predictable architectures over those with a high average performance. However, performance is still important. In this section we will

give a short overview of the formulas from [8] that are used to calculate the cache's influence on execution time. We will extend the single measurement *miss rate* to a two value set, memory read and transaction rate, that is architecture independent and better reflect the two properties (bandwidth and latency) of the main memory. To evaluate cache performance, MEM_{clk} memory stall cycles are added to the CPU execution time (t_{exe}) equation:

$$t_{exe} = (CPU_{clk} + MEM_{clk}) \times t_{clk}$$

$$MEM_{clk} = Misses \times MP_{clk}$$

The miss penalty MP_{clk} is the cost per miss, measured in clock cycles. When the instruction count IC is given as the number of instructions executed, CPI the average clock cycles per instruction and the number of misses per instruction, we obtain the following result:

$$CPU_{clk} = IC \times CPI_{exe}$$

$$MEM_{clk} = IC \times \frac{Misses}{Instruction} \times MP_{clk}$$

$$t_{exe} = IC \times (CPI_{exe} + \frac{Misses}{Instruction} \times MP_{clk}) \times t_{clk}$$

As this paper is only concerned with the instruction cache, we will split the memory stall cycles into misses caused by the instruction fetch and misses caused by data access.

$$CPI = CPI_{exe} + CPI_{IM} + CPI_{DM}$$

CPI_{exe} is the average number of clock cycles per instruction, given an ideal memory system without any stalls. CPI_{IM} are the additional clock cycles caused by instruction cache misses and CPI_{DM} the data miss portion of the CPI. This split between instruction and data portions of the CPI better reflects the split of the cache between instruction and data cache found in actual processors.

The misses per instruction are often reported as misses per 1000 instructions. However, there are several drawbacks to using a single number:

Architecture dependent: The average number of memory accesses per instruction differs greatly between a RISC processor and the Java Virtual Machine (JVM). A typical RISC processor needs one memory word (4 bytes) per instruction word, and about 40% of the instructions [8] are *load* or *store* instructions. Using the example of a 32-bit RISC processor, this results in 5.6 bytes memory access per instruction. The average length of a JVM bytecode instruction is 1.7 bytes and about 18% of the instructions access the memory for data load and store.

Block size dependent: Misses per instruction depends subtly on the block size. On a single cache miss, a whole block of the cache is filled. Therefore, the probability that a future instruction request is a hit is higher with a larger block size. However, a larger block size results in a higher miss penalty as more memory is transferred.

Main memory is usually composed of DRAMs. Access time to this memory is measured in terms of latency (the time taken to access the first word of a larger block) and

bandwidth (the number of bytes read or written in a single request per time unit). These two values, along with the block size of a cache, are used to calculate the miss penalty:

$$MP_{clk} = Latency + \frac{Block\ size}{Bandwidth}$$

To better evaluate different cache organizations and different instruction sets (RISC versus JVM), we will introduce two performance measurements — memory bytes read per instruction byte and memory transactions per instruction byte:

$$MBIB = \frac{Memory\ bytes\ read}{Instruction\ bytes}$$

$$MTIB = \frac{Memory\ transactions}{Instruction\ bytes}$$

These two measures are closely related to memory bandwidth and latency. With these two values and the properties of the main memory, we can calculate the average memory cycles per instruction byte $MCIB$ and CPI_{IM} , i.e. the values we are concerned in this paper.

$$MCIB = \left(\frac{MBIB}{Bandwidth} + MTIB \times Latency \right)$$

$$CPI_{IM} = MCIB \times Instruction\ length$$

The misses per instruction can be converted to MBIB and MTIB when the following parameters are known: the average instruction length of the architecture, the block size of the cache and the miss penalty in latency and bandwidth. We will examine this further in the following example:

We will use as our example a RISC architecture with a 4 bytes instruction length, an 8 KB instruction cache with 64-byte blocks and a miss rate of 8.16 per 1000 instructions [8]. The miss penalty is 100 clock cycles. The memory system is assumed to deliver one word (4 bytes) per cycle. Firstly, we need to calculate the latency of the memory system.

$$Latency = MP_{clk} - \frac{Blocksize}{Bandwidth} = 100 - \frac{64}{4} = 84\ clock\ cycles$$

With $Miss\ rate = \frac{Cache\ miss}{Cache\ access}$, we obtain MBIB.

$$MBIB = \frac{Memory\ bytes\ read}{Instruction\ bytes} = \frac{Cache\ miss \times Block\ size}{Cache\ access \times Instruction\ length}$$

$$= Miss\ rate \times \frac{Block\ size}{Instruction\ length} = 8.16 \times 10^{-3} \times \frac{64}{4} = 0.131$$

MTIB is calculated in a similar way:

$$MTIB = \frac{Memory\ transactions}{Instruction\ bytes} = \frac{Cache\ miss}{Cache\ access \times Instruction\ length}$$

$$= \frac{Miss\ rate}{Instruction\ length} = \frac{8.16 \times 10^{-3}}{4} = 2.04 \times 10^{-3}$$

For a quick check, we can calculate CPI_{IM} :

$$MCIB = \frac{MBIB}{Bandwith} + MTIB \times Latency = \frac{0.131}{4} + 2.04 \times 10^{-3} \times 84 = 0.204$$
$$CPI_{IM} = MCIB \times Instruction\ length = 0.204 \times 4 = 0.816$$

This is the same value as that which we get from using the miss rate with the miss penalty. However, MBIB and MTIB are architecture independent and better reflect the latency and bandwidth of the main memory.

$$CPI_{IM} = Miss\ rate \times Miss\ penalty = 8.16 \times 10^{-3} \times 100 = 0.816$$

3 Proposed Cache Solution

In this section, we will develop a solution with a predictable cache. Typical Java programs consist of short methods. There are no branches out of the method and all branches inside are relative. In the proposed architecture, the full code of a method is loaded into the cache before execution. The cache is filled on calls and returns. This means that all cache fills are lumped together with a known execution time. The full loaded method and relative addressing inside a method also result in a simpler cache. Tag memory and address translation are not necessary.

3.1 Single Method Cache

A single method cache, although less efficient, can be incorporated very easily into the WCET analysis. The time needed for the memory transfer need to be added to the invoke and return instruction. The main disadvantage of this single method cache is the high overhead when a complete method is loaded into the cache and only a small fraction of the code is executed. This issue is similar to that encountered with unused data in a cache line. However, in extreme cases, this overhead can be very high. The second problem can be seen in following example:

```
foo() {  
    a();  
    b();  
}
```

The main drawback of the single method cache is the multiple cache fill of `foo()` on return from methods `a()` and `b()`. In a conventional cache design, if these three methods can be fitted in the cache memory at the same time and there is no placement conflict, each method is only loaded once. This issue can be overcome by caching more than one method. The simplest solution is a two block cache.

3.2 Two Block Cache

The two block cache can hold up to two methods in the cache. This results in having to decide which block is replaced on a cache miss. With only two blocks, Least-Recently

Used (LRU) is trivial to implement. The code sequence now results in the cache loads and hits as shown in Table 1. With the two block cache, we have to double the cache memory or use both blocks for a single large method. The WCET analysis is slightly more complex than with a single block. A short history of the invocation sequence has to be used to find the cache fills and hits. A memory (similar to the tag memory) with one word per block is used to store a reference to the cached method. However, this memory can be slower than the tag memory as it is only accessed on invocation or return, rather than on every cache access.

Table 1. Cache load and hit example with the two block cache

| Instruction | Block 1 | Block 2 | Cache |
|-------------|---------|---------|-------|
| foo() | foo | – | load |
| a() | foo | a | load |
| return | foo | a | hit |
| b() | foo | b | load |
| return | foo | b | hit |

We can improve the hit rate by adding more blocks to the cache. If only one block per method is used, the cache size increases with the number of blocks. With more than two blocks, LRU replacement policy means that another word is needed for every block containing a use counter that is updated on every invoke and return. During replacement, this list is searched for the LRU block. Hit detection involves a search through the list of the method references of the blocks. If this search is done in microcode, it imposes a limit on the maximum number of blocks.

3.3 Variable Block Cache

Several cache blocks, all of the size as the largest method, are a waste of cache memory. Using smaller block sizes and allowing a method to spawn over several blocks, the blocks become very similar to cache lines. The main difference from a conventional cache is that the blocks for a method are all loaded at once and need to be consecutive.

Choosing the block size is now a major design tradeoff. Smaller block sizes allow better memory usage, but the search time for a hit also increases.

With varying block numbers per method, an LRU replacement becomes impractical. When the method found to be LRU is smaller than the loaded method, this new method invalidates two cached methods. For the replacement, we will use a pointer *next* that indicates the start of the blocks to be replaced on a cache miss. Two practical replace policies are:

Next block: At the very first beginning, *next* points to the first block. When a method of length *l* is loaded in the block *n*, *next* is updated to $(n + l) \% \text{block count}$.

Stack oriented: *next* is updated in the same way as before on a method load. It is also updated on a method return — independent of a resulting hit or miss — to point to the first block of the leaving method.

We will show these different replacement policies in an example with three methods: a(), b() and c() of block sizes 2, 2 and 1. The cache consists of 4 blocks and is therefore too small to hold all the methods during the execution of the following code fragment:

```

a() {
  for ( ; i ) {
    b();
    c();
  }
}

```

Tables 2 and 3 show the cache content during program execution for both replacement policies. The content of the cache blocks is shown after the execution of the instruction. An uppercase letter indicates that this block is newly loaded. A right arrow depicts the block to be replaced on a cache miss (the *next* pointer). The last row shows the number of blocks that are filled during the execution of the program.

Table 2. Next block replacement policy

| Instruction | a() | b() | ret | c() | ret | b() | ret | c() | ret | b() | ret | c() | ret |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Block 1 | A | →a | →a | C | A | a | a | a | a | B | b | →- | →- |
| Block 2 | A | a | a | →- | A | a | a | a | a | →- | A | a | a |
| Block 3 | →- | B | b | b | →b | →b | →b | C | c | c | A | a | a |
| Block 4 | - | B | b | b | b | b | b | →- | →- | B | →b | C | c |
| Fill count | 2 | 4 | 5 | 7 | 8 | 10 | 12 | 13 | | | | | |

Table 3. Stack oriented replacement policy

| Instruction | a() | b() | ret | c() | ret | b() | ret | c() | ret | b() | ret | c() | ret |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Block 1 | A | →a | a | a | a | →a | a | a | a | →a | a | a | a |
| Block 2 | A | a | a | a | a | a | a | a | a | a | a | a | a |
| Block 3 | →- | B | →b | C | →c | B | →b | C | →c | B | →b | C | →c |
| Block 4 | - | B | b | →- | - | B | b | →- | - | B | b | →- | - |
| Fill count | 2 | 4 | 5 | 7 | 8 | 10 | 11 | | | | | | |

In this example, the stack oriented approach needs slightly fewer fills, as only methods b() and c() are exchanged and method a() stays in the cache. However, if, for example, method b() is the size of one block, all methods can be held in the cache using the *next block* policy, but b() and c() would be still exchanged using the *stack* policy. Therefore, the first approach is used in the proposed cache.

4 WCET Analysis

The proposed instruction cache is designed to simplify WCET analysis. Due to the fact that all cache misses are included in two instructions (*invoke* and *return*) only, the instruction cache can be ignored on all other instructions. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the length of the method. On an *invoke*, the length of the invoked method is used, and on a *return*, the method length of the caller.

With a single method cache this calculation can be further simplified. For every *invoke* there is a corresponding *return*. That means that the time needed for the cache load on *return* can be included in the time for the *invoke* instruction. This is simpler because both methods, the caller and the callee, are known at the occurrence of the *invoke* instruction. The information about which method was the caller need not be stored for the *return* instruction to be analyzed.

With more than one method in the cache, a cache hit detection has to be performed as part of the WCET analysis. If there are only two blocks, this is trivial, as (i) a hit on *invoke* is only possible if the method is the same as the last invoked (e.g. a single method in a loop) and (ii) a hit on *return* is only possible when the method is a leaf in the call tree. In the latter case, it is always a hit.

When the cache contains more blocks (i.e. more than two methods can be cached), a part of the call tree has to be taken into account for hit detection. The variable block cache further complicates the analysis, as the method length also determines the cache content. However, this analysis is still simpler than a cache modeling of a direct mapped instruction cache, as cache block replacement depends on the call tree instead of instruction addresses.

In traditional caches, data access and instruction cache fill requests can compete for the main memory bus. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instructions. With a data cache, this situation can be even worse. The worst case scenario for the memory stall time for an instruction fetch or a data load is two miss penalties when both cache reads are a miss. This unpredictable behavior leads to very pessimistic WCET bounds.

A *method cache*, with cache fills only on *invoke* and *return*, does not interfere with data access to the main memory. Data in the main memory is accessed with *getfield* and *putfield*, instructions that never overlap with *invoke* and *return*. This property removes another uncertainty found in traditional cache designs.

5 Caches Compared

In this section, we will compare the different cache architectures in a quantitative way. Although our primary concern is predictability, performance remains important. We will therefore first present the results from a conventional direct-mapped instruction cache. These measurements will then provide a baseline for the evaluation of the proposed architecture.

Cache performance varies with different application domains. As the proposed system is intended for real-time applications, the benchmark for these tests should reflect

this fact. However, there are no standard benchmarks available for embedded real-time systems. A real-time application was therefore adapted to create this benchmark. The application is from one node of a distributed motor control system [9]. A simulation of the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark for simulating the real-world workload.

The data for all measurements was captured using a simulation of a Java processor [10] and running the application for 500,000 clock cycles. During this time, the major loop of the application was executed several hundred times, effectively rendering any misses during the initialization code irrelevant to the measurements.

5.1 Direct-Mapped Cache

Table 4 gives the memory bytes and memory transactions per instruction byte for a standard direct-mapped cache. As we can see from the values for a cache size of 4 KB, the kernel of the application is small enough to fit completely into the 4 KB cache. The cache performs better (i.e. fewer bytes are transferred) with smaller block sizes. With smaller block sizes, the chance of unused data being read is reduced and the larger number of blocks reduces conflict misses. However, reducing the block size also increases memory transactions (MTIB), which directly relates to memory latency.

Table 4. Direct-mapped cache

| Cache size | Block size | MBIB | MTIB |
|------------|------------|------|-------|
| 1 KB | 8 | 0.28 | 0.035 |
| 1 KB | 16 | 0.38 | 0.024 |
| 1 KB | 32 | 0.58 | 0.018 |
| 2 KB | 8 | 0.17 | 0.022 |
| 2 KB | 16 | 0.25 | 0.015 |
| 2 KB | 32 | 0.41 | 0.013 |
| 4 KB | 8 | 0.00 | 0.001 |
| 4 KB | 16 | 0.01 | 0.000 |
| 4 KB | 32 | 0.01 | 0.000 |

Which configuration performs best depends on the relationship between memory bandwidth and memory latency. Examples of average memory access times in cycles per instruction byte for different memory technologies are provided in Table 5. The third column shows the cache performance for a Static RAM (SRAM), which is very common in embedded systems. A latency of 1 clock cycle and an access time of 2 clock cycles per 32-bit word are assumed. For the synchronous DRAM (SDRAM) in the fourth column, a latency of 5 cycles (3 cycle for the row address and 2 cycle CAS latency) is assumed. The memory delivers one word (4 bytes) per cycle. The Double Data Rate (DDR) SDRAM in the last column has an enhanced latency of 4.5 cycles and transfers data on both the rising and falling edge of the clock signal.

The data in bold give the best block size for different memory technologies. As expected, memories with a higher latency and bandwidth perform better with larger block

Table 5. Direct-mapped cache, average memory access time

| Cache size | Block size | SRAM | SDRAM | DDR |
|------------|------------|-------------|-------------|-------------|
| 1 KB | 8 | 0.18 | 0.25 | 0.19 |
| 1 KB | 16 | 0.22 | 0.22 | 0.16 |
| 1 KB | 32 | 0.31 | 0.24 | 0.15 |
| 2 KB | 8 | 0.11 | 0.15 | 0.12 |
| 2 KB | 16 | 0.14 | 0.14 | 0.10 |
| 2 KB | 32 | 0.22 | 0.17 | 0.11 |

sizes. For small block sizes, the latency clearly dominates the access time. Although the SRAM has half the bandwidth of the SDRAM and a quarter of the DDR, with a block size of 8 bytes it is faster than the DRAM memories. In most cases a block size of 16 bytes is the fastest solution and we will therefore use this configuration for comparison with the following cache solutions.

5.2 Fixed Block Cache

Cache performance for single method per block architectures is shown in Table 6. A single block that has to be filled on every invoke and return requires considerable overheads. More than twice the amount of data is read from the main memory than is consumed by the processor.

The solution with two blocks for two methods performs almost twice as well as the simple one method cache. This is due to the fact that, for all leaves in the call tree, the caller method can be found on return. If the block count is doubled again, the number of misses is reduced by a further 25%, but the cache size also doubles. For this measurement, an LRU replacement policy applies for the two and four block caches.

Table 6. Fixed block cache

| Type | Cache size | MBIB | MTIB |
|---------------|------------|------|-------|
| Single method | 1 KB | 2.32 | 0.021 |
| Two blocks | 2 KB | 1.21 | 0.013 |
| Four blocks | 4 KB | 0.90 | 0.010 |

The same memory parameters as in the previous section are also used in Table 7. As MBIB and MTBI show the same trend as a function of the number of blocks, this is reflected in the access time in all three memory examples.

5.3 Variable Block Cache

Table 8 shows the cache performance of the proposed solution, i.e. of a method cache with several blocks per method, for different cache sizes and number of blocks. For this measurement, a *next block* replacement policy applies.

Table 7. Fixed block cache, average memory access time

| Type | Cache size | SRAM | SDRAM | DDR |
|---------------|------------|------|-------|------|
| Single Method | 1 KB | 1.18 | 0.69 | 0.39 |
| Two blocks | 2 KB | 0.62 | 0.37 | 0.21 |
| Four blocks | 4 KB | 0.46 | 0.27 | 0.16 |

Table 8. Variable block cache

| Cache size | Block count | MBIB | MTIB |
|------------|-------------|------|-------|
| 1 KB | 8 | 0.80 | 0.009 |
| 1 KB | 16 | 0.71 | 0.008 |
| 1 KB | 32 | 0.70 | 0.008 |
| 1 KB | 64 | 0.70 | 0.008 |
| 2 KB | 8 | 0.73 | 0.008 |
| 2 KB | 16 | 0.37 | 0.004 |
| 2 KB | 32 | 0.24 | 0.003 |
| 2 KB | 64 | 0.12 | 0.001 |
| 4 KB | 8 | 0.73 | 0.008 |
| 4 KB | 16 | 0.25 | 0.003 |
| 4 KB | 32 | 0.01 | 0.000 |
| 4 KB | 64 | 0.00 | 0.000 |

In this scenario, as the MBIB is very high at a cache size of 1 KB and almost independent of the block count, the cache capacity is seen to be clearly dominant. The most interesting cache size with this benchmark is 2 KB. Here, we can see the influence of the number of blocks on both performance parameters. Both values benefit from more blocks. However, a higher block count requires more time or more hardware for hit detection. With a cache size of 4 KB and enough blocks, the kernel of the application completely fits into the variable block cache, as we have seen with a 4 KB traditional cache. From the gap between 16 and 32 blocks (within the 4 KB cache), we can say that the application consists of fewer than 32 different methods.

It can be seen that even the smallest configuration with a cache size of 1 KB and only 8 blocks outperforms fixed block caches with 2 or 4 KB in both parameters (MBIB and MTIB). In most configurations, MBIB is higher than for the direct-mapped cache. It is very interesting to note that, in all configurations (even the small 1 KB cache), MTIB is lower than in all 1 KB and 2 KB configurations of the direct-mapped cache. This is a result of the complete method transfers when a miss occurs and is clearly an advantage for main memory systems with high latency. As in the previous examples, Table 9 shows the average memory access time per instruction byte for three different main memories.

The variable block cache directly benefits from the low MTBI with the DRAM memories. When comparing the values between SDRAM and DDR, we can see that the bandwidth affects the memory access time in a way that is approximately linear. The high latency of these memories is completely hidden. The configuration with 16

Table 9. Variable block cache, average memory access time

| Cache size | Block count | SRAM | SDRAM | DDR |
|------------|-------------|------|-------|------|
| 1 KB | 8 | 0.41 | 0.24 | 0.14 |
| 1 KB | 16 | 0.36 | 0.22 | 0.12 |
| 1 KB | 32 | 0.36 | 0.21 | 0.12 |
| 1 KB | 64 | 0.36 | 0.21 | 0.12 |
| 2 KB | 8 | 0.37 | 0.22 | 0.13 |
| 2 KB | 16 | 0.19 | 0.11 | 0.06 |
| 2 KB | 32 | 0.12 | 0.08 | 0.04 |
| 2 KB | 64 | 0.06 | 0.04 | 0.02 |

or more blocks and dynamic RAMs outperforms the direct-mapped cache of the same size. As expected, a memory with low latency (the SRAM in this example) depends on the MBIB values. The variable block cache is slower than the direct-mapped cache in the 1 KB configuration because of the higher MBIB (0.7 compared to 0.3-0.6), and performs very similarly at a cache size of 2 KB. In Table 10, the different cache solutions with a size of 2 KB are summarized. All full method caches with two or more blocks have a lower MTIB than a conventional cache solution. This becomes more important with increasing latency in main memories. The MBIB value is only quite high for one or two methods in the cache. However, the most surprising result is that the variable block cache with 32 blocks outperforms a direct-mapped cache of the same size at both values.

Table 10. Caches compared

| Cache type | MBIB | MTIB |
|---------------------|------|-------|
| Single method | 2.32 | 0.021 |
| Two blocks | 1.21 | 0.013 |
| Variable block (16) | 0.37 | 0.004 |
| Variable block (32) | 0.24 | 0.003 |
| Direct mapped | 0.25 | 0.015 |

We can see that predictability is indirectly related to performance — a trend we had expected. The most predictable solution with a single method cache performs very poorly compared to a conventional direct-mapped cache. If we accept a slightly more complex WCET analysis (taking a small part of the call tree into account), we can use the two block cache that is about two times better. With the variable block cache, it could be argued that the WCET analysis becomes too complex, but it is nevertheless simpler than that with the direct-mapped cache. However, every hit in the two block cache will also be a hit in a variable block cache (of the same size). A tradeoff might be to analyze the program by assuming a two block cache but using a version of the variable block cache.

6 Conclusion

In this paper, we have extended the single cache performance measurement *miss rate* to a two value set, memory read and transaction rate, in order to perform a more detailed evaluation of different cache architectures. From the properties of the Java language — usually small methods and relative branches — we derived the novel idea of a *method cache*, i.e. a cache organization in which whole methods are loaded into the cache on method invocation and the return from a method. This cache organization is time predictable, as all cache misses are lumped together in these two instructions. Using only one block for a single method introduces considerable overheads in comparison with a conventional cache, but is very simple to analyze. We extended this cache to hold more methods, with one block per method and several smaller blocks per method.

Comparing these organizations quantitatively with a benchmark derived from a real-time application, we have seen that the variable block cache performs similarly to (and in one configuration even better than) a direct-mapped cache, in respect of the bytes that have to be filled on a cache miss. In all configurations and sizes of the variable block cache, the number of memory transactions, which relates to memory latency, is lower than in a traditional cache.

Filling the cache only on method invocation and return simplifies WCET analysis and removes another source of uncertainty, as there is no competition for main memory between instruction cache and data cache.

References

1. Puschner, P., Koza, C.: Calculating the maximum execution time of real-time programs. *Real-Time Syst.* **1** (1989) 159–176
2. Arnold, R., Mueller, F., Whalley, D., Harmon, M.: Bounding worst-case instruction cache performance. In: *IEEE Real-Time Systems Symposium*. (1994) 172–181
3. Healy, C., Whalley, D., Harmon, M.: Integrating the timing analysis of pipelining and instruction caching. In: *IEEE Real-Time Systems Symposium*. (1995) 288–297
4. Lee, C.G., Hahn, J., Seo, Y.M., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M., Kim, C.S.: Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.* **47** (1998) 700–713
5. Busquets-Mataix, J.V., Wellings, A., Serrano, J.J., Ors, R., Gil, P.: Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In: *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, Washington - Brussels - Tokyo, IEEE Computer Society Press (1996) 204–213
6. 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* **91** (2003)
7. Power, J., Waldron, J.: A method-level analysis of object-oriented techniques in java. Technical report, Department of Computer Science, NUI Maynooth, Ireland (2002)
8. Hennessy, J., Patterson, D.: *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303 (2002)
9. Schoeberl, M.: Using a Java optimized processor in a real world application. In: *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, Austria, Vienna (2003) 165–176
10. Schoeberl, M.: JOP: A Java optimized processor. In: *Workshop on Java Technologies for Real-Time and Embedded Systems*. Volume LNCS 2889., Catania, Italy (2003) 346–359