Approaches to Handle TLBs in Real-Time Systems Running on State of the Art Processors

Alexander von Bülow Jürgen Stohr Georg Färber
Institute for Real–Time Computer Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{**Alexander.Buelow,Juergen.Stohr,Georg.Faerber**}**@rcs.ei.tum.de**

Abstract

State of the art processors like Intel Pentium or AMD Athlon implement large cache memories. These caches bridge the gap between the high speed processors and the comparatively slow main memories. However, for the use in real–time systems, caches are a source of predictability problems. A lot of progress has been achieved to cope with caches in real–time systems to determine safe and precise bounds of the execution times in the presence of cache memories. In this paper, we deal with the Translation-Lookaside Buffers (TLBs). These little caches are used for the translation of a virtual address into a physical address. They are accessed each time when code or data is referenced. We show how memory layout affects usage of the TLBs and we discuss two approaches to use them in real-time systems.

1. Introduction

Present-day processors like the AMD Athlon or the Intel Pentium 4 are not designed to act in hard real-time systems. They are optimized to deliver a good performance in the average case. Nevertheless, there are some properties which make them interesting for use in real-time systems: They are very fast in comparison to other processor architectures, they are cheap in price, the technological progress goes on rapidly and they are in most cases downwardly compatible.

In a real-time system the correctness of a result depends on the date at which it is produced. Therefore, it is essential to know theses dates exactly. In order to proof the real-time capabilities of a system, one has to choose an adequate scheduling policy and to perform a real-time analysis. One determining parameter is the worst-case execution time (WCET) of each task of the system. The significant part of the formation of WCETs on modern processors is wether the corresponding code and data is in cache when a task is executing.

But not only the code and data caches account for the WCET, the TLBs also have a non-neglectable influence on the WCET. They are used to hold the information needed for address translation. If an access to the TLBs fails, the result is at least one memory access in addition to the one needed to access a code or data object, even if this object is in cache. To compute precise bounds for the WCET on systems using TLBs, one has to investigate their behavior during run-time of the real-time system.

This paper is organized as follows: Section 2 gives an overview of related work. The next section describes the scenario we deal with and section 4 gives a short introduction about TLBs. In section 5 we discuss different approaches to cope with TLBs in the context of real-time systems and give a short example. Section 6 concludes the paper.

2. Related Work

To the best of our knowledge, there are not any papers that deal with the influence of the TLBs on the WCET explicitely. A framework for data cache analysis with respect to real-time capabilities is given in [8]. They investigate techniques like cache partitioning, dynamic cache locking and static cache analysis for predictability of multitasking hard real-time systems. They also include data TLBs in their research.

Bennet and Audsley discuss virtual memory management for real-time systems in [1]. Sebek deals in [6] with the architecture of cache memories in general and

its influence on real-time systems.

An analytic approach to cache and pipeline analysis to compute save bounds for the WCET on modern processor architectures is pushed by Wilhelm *et al.* in many papers. Especially, they model cache behavior for state of the art processors and investigate the impact of different cache architectures on the WCET [3] [5] [7].

In this paper, we want to analyse the impact of main memory management on the usage of the TLBs. Among other things, this is very important for algorithms that rearrange items in memory for an optimal usage of the processors' caches (not TLBs) like in [2], [4] and [9].

3. Scenario

The system consists of a set of N procedures

$$P = \{p_i(s_i, D_i) : i = 1...N\}$$

Each procedure p_i consists of one code object with size s_i in bytes and a set D_i of K_i data objects with

$$D_i = \{d_{i,k}(s_{i,k}) : k_i = 1...K_i\}$$

where the object $d_{i,k}$ has the size $s_{i,k}$ in bytes. The whole object code of a procedure is referred to as a *code object*. *Data objects* $d_{i,k}$ are the variables used by the procedure p_i. The stack region that each procedure p_i can have is a data object, too. In general, each task of the system consists of one or more procedures. The same applies to the mechanisms of the RTOS (Real-Time Operating System) like the scheduler. The sizes of code and data objects are *aligned*, this means, they are rounded up to the next aligned size, if needed. For example, if the alignment is four, an object with size six will be handled like an object with size eight.

Each of these objects can have an arbitrary location in main memory. This location in main memory predetermines its location in cache. The cache is presumed to be a physically tagged *n-set associative cache*. Its cache line size is m and it has l sets, so its size s denotes to $s = n \cdot m \cdot l$. Parts of the physical address correspond with the set number where a cache line will be stored in cache. The cache is supposed to have a *harvard architecture*. This means, it is split into a cache for instructions and data, respectively.

The memory is divided into *pages* with size s_p in bytes. To simplify the discussion, we deal with pages of one size only. Nevertheless, the approach can be easily extended to a combination of pages with different sizes. This is supported to a certain extend by all modern processors.

4. Translation Lookaside Buffers

The *Translation Lookaside Buffers* are little caches that contain information needed for the translation of a virtual into a physical address. In general, this information consists of entries of tables in memory the processor needs to compute the physical address. Figure 1 shows the address translation for an AMD Athlon processor and a page size $s_p = 4096$ Byte.

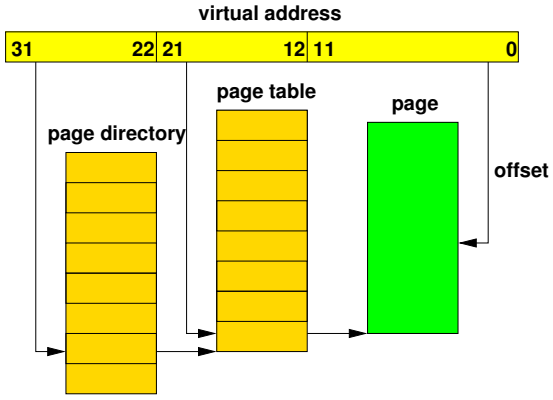

Figure 1: Translation of virtual adresses

In this example, the TLBs must cache the entry of the page directory and the entry of the page table. Both are needed for the address translation. This information is cached in one entry in the TLBs and is sufficient for the address translation within one page.

The structure of the TLBs is very similar to the structure of the caches. There are TLBs for accesses to instructions and for accesses to data, respectively. If there exists more than one level of cache, there exists a corresponding level of TLBs, too. Normally, the level one TLBs are fully-associative caches and the level two TLBs are set-associative caches. Both dismiss their entries following a LRU strategy.

5. Caches, TLBs, and Memory Layout

One approach to optimize cache usage for real-time software is to arrange code and data objects in memory in such a manner that the number of cache misses for an application is minimized or made more predictable. For example, such approaches can be found in [2], [4] or [9]. To put one object to a certain set in cache, it is necessary to put it to a physical address that corresponds with this

set. Figure 2 shows this scenario for a 2-way set associative cache. As one can see, a location in memory that

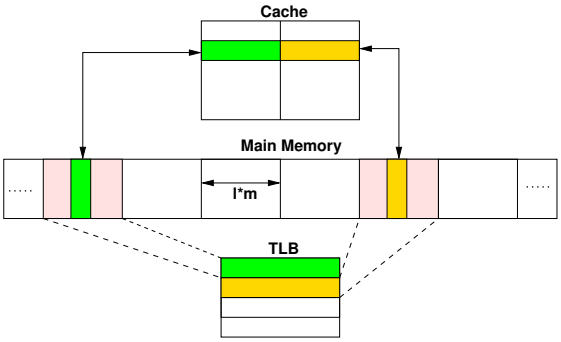

Figure 2: Cache, memory layout and TLBs

corresponds with one set in cache repeats in intervals of $l \cdot m$. Let s_m be the size of main memory in bytes. Then, the number of possible locations in memory that correspond with one set denotes to $x = \lfloor \frac{s_m}{l \cdot m} \rfloor$.

Everytime software accesses one of these locations, another entry in the TLBs must be used (assumed that $l \cdot m > s_p$). That means, if code and data objects are distributed over a large range in memory, the number of TLB entries needed by the software will increase. Every TLB miss means at least one additional access to memory. For the system shown in figure 1, there are two additional misses. These misses influence the execution time of software in the same way as a conventional cache miss does.

The relationship between locations in memory and sets in cache (see figure 2) shows that a good placement for objects in cache might be a bad placement regarding the TLBs. From the point of view of the cache, it does not matter at which of the possible locations an object actually is. But from the point of view of the TLBs, it is very important.

Local and Global Optimization We suggest two approaches to deal with TLBs in real-time systems:

- **Local optimization:** This means to minimize the number of TLB entries needed by a single procedure. In general, every procedure p_i needs at least one entry for code and one for data.
- **Global optimization:** Here, the goal is to minimize the number of TLB entries that are needed for the *whole* system. It is possible that the number of TLB entries for a particular procedure increases.

Both approaches can be realized by placing code and data objects in memory. This should be part of an effort to optimize cache usage for software.

The local optimization minimizes the number of TLB entries needed by a single procedure. In the worst case, the number of possible memory accesses due to a TLB miss when executing a procedure is at its minimum. This is useful for procedures that are running very frequently. Another expedient application are interrupt service routines that must suffer short delays only.

The global optimization is suitable for systems where all procedures are running evenly. There is no reason to optimize the TLB usage of one procedure for the cost to increase total TLB usage. It is advisable to use global optimization for systems which get by with TLB entries using global optimization but not local optimization. Getting along with the resources is always the best solution.

Thus, one needs to compute the number of TLB entries needed for global optimization. To do this, one has to know that the compiler distributes objects according to their properties in *sections*. Basically, a section is a continuous area in memory. There is one section for code (.text section), one for data (.data section) with known storage size at compile time, and one for uninitialized data (.bss section). There can be more sections, even user defined ones.

To compute the number of TLBs needed for a fixed distribution in memory, one needs to know how many sections are needed and how large each section is. The size of a section is dependend on the size of the objects and the distance between the objects within this section.

Let $M = \{m_j : j = 1...J\}$ be the J sections that are needed for the real-time system and s_{m_j} the size in bytes that is needed by section m_j. The number of TLB entries needed for global optimization denotes to

$$n_{TLB,global} = \left\lceil \frac{1}{s_p} \sum_{j=1}^{J} s_{m_j} \right\rceil$$

If this is lower or equal to the available number of TLB entries, global optimization is the best choice. If it is not, local optimization might be the better solution. It enables the user to optimize critical parts of the system at the cost of reducing performance for less critical parts. The number of TLB entries for one procedure in case of a local optimization can be computed like $n_{TLB,global}$ if one interprets M as the set of sections needed by one procedure and s_{m_j} as the size of m_j per procedure.

A Small Thought Experiment To get an idea of the possible differences between local and global optimization, have a look at this small procedure: p_1 has a code size of 5 kB and uses four data objects: $s_{1,1} = 4$ byte, $s_{1,2} = 128$ byte, $s_{1,3} = 32$ byte and $s_{1,4} = 1024$ byte. The page size is 4 kB. This procedure needs two entries in the code TLBs. But for the data TLBs, there are different possible solutions.

Here, it is important to know which section will be allocated by the compiler for each data object. If, for example, $d_{1,1}$ and $d_{1,2}$ are in the .data section and $d_{1,3}$ and $d_{1,4}$ are in the .bss section, it is possible that two entries in the data TLBs are needed, although the sum of the sizes of all four data objects is lower than 4 kB. This is true if the size of each section is greater than 4 kB which can be regarded as the common case for a real-time system running on state of the art processors. Keep in mind that p_1 is only a small part of the system.

But it can be worse. If the distance of data objects in one section is greater than 4 kB in memory, four entries of the data TLB will be required. This can happen because of a memory layout that optimizes cache usage but does not take care of the TLB usage.

If using a local optimization, procedure p_1 needed four entries in the TLBs: Two for code and two for data. This optimization does not provide distributing the data objects so that more TLB entries are needed. A global optimization would include all procedures of a system (not declared in this small example). In this case, six entries in the TLBs could be a good solution if the data objects of the other procedures fit completely into a memory range of $4 \cdot 4 = 16$ kB. Thus, for procedure p_1, four data TLB entries are required instead of two, but from the point of view of the whole system, this could be a good solution.

6. Conclusions and Future Work

In this paper, we have shown that the memory layout of real-time software not only affects the cache usage but also the usage of the TLBs. These little caches are not neglectable regarding WCET analysis. The consequences of a TLB miss are rather the same as the consequences of a regular cache miss. A memory layout that optimizes cache usage can result in many TLB misses, thus, it has to include an optimization of TLB usage.

Future work will be a deeper investigation of the impacts of TLB misses on the WCET of software. We want to extend our approach to use different page sizes simultaneously.

References

[1] M. D. Bennet and N. C. Audsley. Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems. In *In Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, Delft, The Netherlands, June 2001.

[2] B. Calder, K. Chandra, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[3] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT 2001)*, pages 469–485, Tahoe City, CA, USA, October 2001.

[4] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 1997.

[5] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. In *Proceedings of the IEEE*, volume 91, July 2003.

[6] Filip Sebek. Cache memories and real–time systems. Technical report, Department of Computer Engineering Mälardalen University Västerås, Sweden, 2001.

[7] L. Thiele and R. Wilhelm. Design for Timing Predictabilty. *Real-Time Systems*, 28:157–177, 2004.

[8] X. Vera, B. Lisper, and J. Xue. Data Caches in Multitasking Hard Real-Time Systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[9] Alexander von Bülow, Jürgen Stohr, and Georg Färber. Towards an Efficient Use of Caches in State of the Art Processors for Real–Time Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.