Towards an Efficient Use of Caches in State of the Art Processors for Real–Time Systems

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 present an extension to algorithms that aim to place code and data efficiently into cache memory. Our approach takes the architectural features of the implementation of the cache memory of state of the art processors into account. Furthermore this approach allows it to lock regions of memory in cache although these processors do not provide this feature in hardware. The benefits and the problems of this approach are discussed.

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 these 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 these WCETs on modern processors is weather the corresponding code and data is in cache when a task is executing.

Determining the WCET on processors with caches is a challenging task and extensive studies have been done on this. Our approach focuses not only on one level of cache or only on instruction caches, it treats the cache subsystem as an integrated whole. It can be seen as an extension to the algorithms presented in [6] and [7]. As a result of our extensions it is possible to predict tighter estimations of the WCET of real–time software. Furthermore, it is possible to lock certain code and data blocks in cache even though this feature is not supported in hardware. Throughout this paper we focus on the architecture of the AMD Athlon processor but the results can easily be adapted to similar architectures like the Intel Pentium 4 or Power PC processors.

This paper is organized as follows: Section 2 gives an overview of current approaches in conjunction with cache management and methods that deal with the consideration of cache effects in real–time analysis. Section 3 shows how the cache memory system is organized in present-day processors and what difficulties arise for estimating execution times. In section 4 our approach is presented and section 5 concludes the paper.

2. Current Approaches

Current approaches which deal with caches in real–time systems can be divided into two categories: The first category is the cache analysis where caches are used without any restrictions. These analysis techniques predict the worst-case impact of the caches on the schedulability of the system [4]. The second category are the cache partition techniques where portions of the cache are assigned to certain tasks to make their timing behavior predictable [5] [8] [9] [17]. Another approach is the use of cache locking techniques to lock several tasks in cache for the whole lifetime of the system (static cache locking) or to change the mapping dynamically (dynamic cache locking) [3] [13].
Petrank and Rawitz showed in [10] that the problem to find an optimal placement of contents in a cache memory in the sense that it minimizes the number of cache misses is NP–hard. The consequence is to find algorithms that optimize another target than the minimization of cache misses.

Hashemi et al. present in [7] an approach to optimize instruction cache usage. The optimization takes place at compile time. The method is based on a weighted call graph which represents the call structure and call frequency of the software. Additionally, it takes the procedure size, cache size and cache line size into account. This approach works for direct mapped and set associative caches. It can be extended to deal with basic blocks instead of procedures.

Calder et al. present in [6] an approach based on the one in [7] to place data in cache with the aim to minimize the number of cache misses. This approach considers data on stack, global data and dynamically allocated data.

Sebek deals in [14] with the architecture of cache memories in general and its influence on real–time systems. A method to measure the cache related preemption delay (CRPD) on multiprocessor systems is presented in [15]. Because of the complexity of state of the art processors it is impossible to get precise execution times without measuring them on the target system [11].

Our approach extends the algorithms presented in [6] and [7] to consider the features of the cache implementation of the x86 processors. It enables the feature of cache locking though not supported in hardware by these processors.

3. Cache Memory System

The AMD Athlon [1] processor is equipped with a large cache memory system, which is divided into two levels (L1 and L2 Cache) where the L1-Cache is split into one cache used for instructions only and one used for data only (Harvard architecture). The L2-Cache is unified which means that it can be used for instructions and data simultaneously. The Athlon features an exclusive cache design which means the whole size of the L2-Cache can be used for both, code and data. Both caches are organized as n set associative caches, so n cachelines share one set. A cacheline is a continuous block of memory which is loaded at once into cache when needed.

The assignment of the position in memory to a place in cache corresponds with the address in main memory. The lower l bits of the physical address in memory form the offset within one cacheline (cache line size = 2^l). The following s denote the set in cache so the total number of sets in the cache is 2^s. The place for a cache line within the set depends on the replacement strategy of the cache and is transparent to software. The AMD Athlon uses a pseudo least recently used strategy. The remaining bits of the physical address are used as a tag to identify a block of memory in cache. Therefore software can assign a block of code to a certain set by putting it to the corresponding address in memory.

If there is no space left in L1-Cache one cache line is displaced into L2-Cache and another from L2-Cache to main memory. This means that one transfer from L1 to L2-Cache is needed and, much worse, one from L2-Cache to main memory. A transfer from main memory to L1-Cache and a transfer from L1-Cache to L2-Cache are concurrent. If the contents of a cache line changed not while it was in cache it only needs to be invalidated and no transfer to a lower memory level is needed.

![Cache Architecture of the AMD Athlon](image)

**Figure 1: Cache architecture of the AMD Athlon**

Between the L1- and L2-Cache the AMD Athlon processor implements a victim buffer which is a small cache (8 entries) to compensate the differences in speed between the cache levels. Figure 1 shows the cache architecture of the AMD Athlon processor together with the execution times for certain transfers as stated by AMD [2] and verified by us. The time needed for a transfer from or to main memory depends on the underlying connection to the memory controller and the RAM architecture itself.

Accesses to main memory last not only much longer than accesses to cache memory, they are also much more unpredictable. They are dependend on the physical type of main memory being used and on the concurrent data traffic from or to main memory caused by other devices.

4. Main Memory Arrangement

**Scenario.** The real–time system consists of a set of tasks $T = \{ \tau_i : i = 1 \ldots N \}$. Each task $\tau_i = \{ B_k(A_k, S_k), D(k) : k = 1 \ldots K \}$ consists of basic blocks $B_k$ with the attributes $A_k = \{ ro, rw, locked \}$ and the size $S_k$ in bytes, where $D = \{ d_k(A_k, S_k) : l = 1 \ldots L \}$ denotes the data used by $\tau_i$ together with their attributes $A_k$ and size $S_k$. The number of bytes $b$ needed by each item denotes to $b_{e} = S_{e} + (S_{e} \mod a)$ when $a$ is the alignment (usually four bytes). The total number of cache lines needed by this system is

$$C_{total} = \sum_{n=1}^{N} \frac{1}{cl \text{ size}} \left( \sum_{k=1}^{K} b_{k} + \sum_{l=1}^{L} d_{l} \right)$$

where $cl \text{ size}$ denotes the size of a cache line in bytes. In the following we refer to an item as an element of $\tau_i$. 
Our Approach. We extend the approaches presented in [6] and [7] to consider the attributes of the different items. These attributes reflect additional properties of the cache architecture. We arrange our items in the way that only items which
- are read-only (attribute ro) or
- are writable (attribute rw) or
- should be locked (attribute locked)
share one set in cache. Items which are writable can cause a memory access when dismissed but they do not need to do so. Therefore we can never underestimate the WCET if we state that all accesses to these sets cause a main memory access.

The assembler code is used to analyze the real–time system. During this step we automatically locate all items, their sizes and attributes. We treat code in units of procedures (no code splitting). Data is treated in units of variables or structures or as a single stack unit. Dynamic data allocation during task execution is not useful for real–time tasks and not considered here. All these optimizations take place at link time or before the real–time system is started in an initialization phase.

Code is always read-only (self-modifying code is undesirable for predictable execution times), stack data is always read-write and the remaining global data objects can be divided in read-only and read-write according to their section. In the next step we arrange all items according to our scheme mentioned above. We have to take care of the correct alignment and not to spread code and data items of one procedure (spatial locality). This is very important for the use of the Translation Lookaside Buffers (TLB). These TLBs are small caches used for the calculation of the physical address from the virtual address. One entry can address 4 kbytes of memory (other page sizes are not used here) so it is important not to increase the TLB usage due to an unfavorable memory arrangement.

Discussion. This approach has two major advantages: Displacements of read-only items never cause an access to main memory which simplifies the analysis of cache related preemption delays (CRPD). This makes the real–time system more predictable and reduces the WCETs when executing code (remember that code is always read-only).

The second advantage is that not only the places in code where a displacement can take place are exactly known but also the type of displacement, with or without write-back to main memory. This is true for both code and data items. This knowledge paired together with the ascertainable times for a cache miss or a cache hit can be used by known methods for calculating the WCET.

This approach allows it to lock certain items (code or data) in cache. This feature is not supported in hardware by the AMD Athlon or Intel Pentium processors but it can be very useful for hard real–time systems (see [13]). A drawback of this kind of cache locking is that all locations in main memory that would be placed in a set containing locked cache lines cannot be used any more. So if you reserve 10% of the cache for locked items, you loose 10% of main memory minus the space needed for the locked items. Therefore cache locking in this way should be used sparingly and only for small items like interrupt service routines.

Multiprocessor Systems. On a multiprocessor system this approach provides new options for the software architecture of a real–time system. With multiple processors the cache of each processor can be used for real–time jobs individually. This requires that each processor has its own part of main memory. In SMP (Symmetric Multi Processing) systems this can be achieved by a real–time operating system (RTOS) that splits the main memory in parts used by only one processor, respectively. In computer systems that feature the NUMA (Non Uniform Memory Access) architecture each processor has its own physical main memory which makes it much more easier to arrange certain jobs in the cache of certain processors. The usage of separate memory regions for each processor also minimizes the waste of main memory when locking sets into the cache.

Furthermore, the effect of cache snooping can be avoided when using separate main memory regions. Cache snooping is a software transparent technique to maintain data coherence between the processors. That means, if one processor changes the contents of one cache line and another processor has the same location of main memory in his cache, this technique automatically updates the contents in cache.

5. Application Potential

To study the run time behavior we used a dual AMD Athlon computer (Athlon MP 1800+, 1533 MHz, 512 kbytes DDR RAM). We implemented the method of arranging the items of a real–time system as described in section 4. The computer system we use implements the RECOMS Software Architecture as described in [16].

The execution times of the real–time software are determined by measurement. The measurement attempt is based on the work described in [12]. We extended this approach for multiprocessor systems and optimized it for measuring with as low influence on the real–time software as possible. Describing all details of this approach is not the scope of this paper.

One important result is that the difference in speed between the L1-Cache and the L2-Cache is very small. Therefore it is justified to regard the cache as one memory and not to distinguish between different levels of cache (presumed the cache levels are on–die).

Furthermore, we observed execution times of about 250 cycles for accesses to main memory (without any blocking times caused by parallel bus transfers to main memory). This means a factor of about 25 for L1-Cache and a factor of about 12 for L2-Cache in comparison to cache hits. This shows the need to
avoid unnecessary main memory accesses caused by an unsuitable memory arrangement. This explicitly includes memory accesses to refill the TLBs.

Conclusions. The approach presented in this paper extends the algorithms presented in [6] and [7] to consider the implementation of the cache architecture of the AMD Athlon and Intel Pentium processors. It assigns attributes to different code and data items and places only items with identical attributes into one set in cache. Furthermore, this method makes the locking of items in the cache possible. All these optimizations take place at link time or before the real–time system is actually started (stack allocation). This approach makes the execution times of the real–time software more predictable and becomes especially interesting if the whole real–time system fits into the cache which is not unlikely regarding the cache sizes of state of the art processors.

Future Work. The next steps will be a deeper study of the influence of different cache architectures (in particular trace cache) and main memory types on the WCET to enhance the presented memory placement strategies for different hardware and software architectures. The correlation of cache contents and techniques like branch prediction or hyper threading should be examined.

References


