An Approach to Improve Predictability of Worst Case Execution Times of Real-Time Software Running on State of the Art Multiprocessor 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

Multiprocessor systems based on the Intel SMP or the AMD NUMA (Non-Uniform Memory Access) architecture are not designed to act as real-time systems. These systems deliver a high computing power in the average case, but rarely appearing worst case execution times (WCETs) are not considered at all. One main source of unpredictability of execution times in these systems is the memory system. It is composed of the caches and the main memory. The caches bridge the gap between the very fast processors and the comparatively slow main memory. In SMP systems, the processors share one main memory and compete for each other when accessing it. In NUMA systems, each processor has its own physical memory area, but if a task wants to access data in another than the own memory area, access times increase. As a result of these architectures, execution times of software can vary in a broad range. In this paper, we present a tool chain to utilize the cache memory system for real-time software. Further on, we propose an algorithm to place items in memory to improve predictability of WCETs. We present case studies to prove the benefits of our concepts.

1 Introduction

Modern multiprocessor systems based on commercial off-the-shelf (COTS) components are not designed to act as real-time systems. They are optimized to deliver a good performance in the average case, worst case scenarios are not taken into account. Nevertheless, there are many properties which make them interesting for use in real-time systems: they are very fast compared to other processor architectures, they are cheap, the technological progress goes on rapidly, and they are in most cases downwardly compatible.

One source of varying execution times of software is the memory system. It consists of the cache subsystem and the connection of the processors and peripheral devices to the memory. There are big differences in execution times depending on whether code and data are in cache or if they have to be fetched from memory. An access to memory is not only much slower than an access to the cache, it can also be enlarged by a concurring access to memory from another processor or a peripheral device. These delays can be long and their behaviour is hard to predict. The results are varying execution times of software. They depend on whether the needed data is in cache and whether a write back to memory is needed before some data can be loaded into the cache. The methods presented in this paper aim to reduce these effects and to cope with them to improve predictability for real-time software.

Cache memory is divided into several levels (usually two) which differ in size and access time. The fastest cache is the level one (L1) cache, which is split into a cache for code (instruction cache) and one for data (data cache), both are equal in size. Usually, the second level cache (L2) is a unified cache, that means, the whole size can be used for code and data simultaneously. The cache is organized as a *N set associative* cache, that means, each set in cache contains *N* cache lines. The decision, what cache line has to be displaced within a set is transparent for software. Normally, a *pseudo least recently used (PLRU)* algorithm is used. The arrangement of code and data in main memory corresponds directly with the arrangement in cache. Parts of the physical address in memory correspond with the set number in cache. For more detailed information regarding the build up of caches in state of the art processors see [13].

In this paper, we examine the influence of main memory arrangement of code and data on the execution time

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on “Realtime with Commercial Off-the-Shelf Multiprocessor Systems” under Grant Fa 109/15-1.

of software. On the one hand, we consider the timing behaviour of software that runs completely in the caches, on the other hand, we investigate the effects of other processors and peripheral devices that access the memory in parallel. We present methods to place code and data of real-time software in main memory in such a way that displacements in cache occur in a predictable manner. That means, we know which cache lines will be displaced during run-time and what kind of displacement will happen.

There are displacements that need a write-back of data to main memory and others that need an invalidation of a cache line only. The process of displacement works transparently for software. The only way to cope with this is to analyse the software to be able to foresee which kind of displacement will happen.

Our methods reduce the influence of competing memory accesses from several processors working in parallel significantly. Furthermore, our scheme of memory arrangement enables to lock certain code or data items in cache though this feature is not supported in hardware by the processors we are dealing with. We consider the *Translation Lookaside Buffers (TLB)* that are needed for the translation of a virtual address into a physical address. These little caches also have an appreciable influence on the execution time of software.

The paper is organized as follows: section 2 gives an overview of current work. Section 3 deals with a memory division for multiprocessor systems which is part of our approach. In section 4 we present our approach to arrange real-time software in main memory. Section 5 presents a tool chain which implements our concepts and section 6 gives some results to show the benefits of our approach. Section 7 concludes the paper.

2 Current Approaches

Liedtke and Härtig describe in [7] a method to partition the cache among different tasks. The operating system controls the allocation between tasks and cache portions. They use the memory management unit mechanisms to implement their system. They do not consider different data items or multiprocessor systems.

Schönberg investigates in [11] the influence of PCI-Bus transfers on the execution time of software. He introduces a *slowdown factor* to describe these effects.

Petrank and Rawitz showed in [8] 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 but the minimization of cache misses. They propose to investigate different application scenarios and to develop heuristics to achieve an optimal cache behaviour for these scenarios.

Hashemi *et al.* present in [6] 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 [4] an approach based on the one in [6] 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.

Petters states in [9] that due to the complexity of modern processors it is impossible to get precise execution times without measuring them on the target system. He proposes a measurement approach for processors of the AMD Athlon and Intel Pentium family.

Cache locking techniques and their benefits are discussed in [3] [5] [10]. One approach is to lock the contents in cache for the whole lifetime of the system (*static cache locking*) or to change the mapping dynamically (*dynamic cache locking*).

Our approach for memory optimization presented in subsection 4.1 extends the approaches presented in [4] and [6] 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. Additionally, it deals with the TLBs and multiple real-time processors working in parallel.

3 Memory Organisation

In order to perform hard real-time tasks in parallel to a general purpose operating system (GPOS), we developed the RECOMS (Realtime with Commercial off-the- Shelf Multiprocessor Systems) Software Architecture [12]. Figure 1 shows the memory management of the RECOMS Software Architecture. This is part of our approach to arrange code and data of real-time software in memory.

The CPUs of a multiprocessor system are divided into two groups: one CPU is called the *General Purpose Unit (GPU)*, the other CPUs are called the *Real-Time Units (RTUs)*. The GPOS is executed on the GPU exclusively. Thus, all tasks belonging to the GPOS are executed on the GPU. Furthermore, all interrupts that are needed by the GPOS are routed to the GPU. All real-time tasks are executed on the RTUs. Interrupts of peripheral devices served by real-time handlers are routed to the RTU that executes the corresponding real-time handler.

Real-time tasks and real-time interrupt handlers are bound to a specific RTU statically. The methods used for the assignment of a real-time task to a particular RTU depend

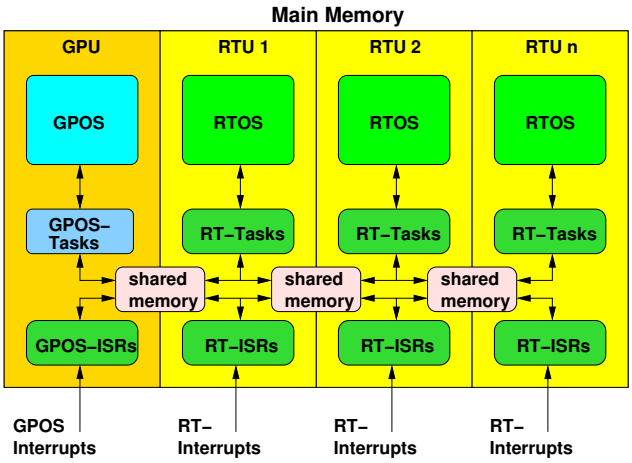

Figure 1. Memory division

on the resource requirements of that task. These approaches are not the scope of this paper.

Each CPU gets its own portion of *physical* memory. In NUMA systems, the portions of memory that every processor gets assigned to are equal to its physical memory. Code and data of tasks that are running on a particular CPU are assigned to the memory region of that processor. They run all in the same address space and privilege level of the processor. There are shared memory regions for data exchange between tasks. They can be used for tasks on the same processor or for tasks on different processors. Details of the handling of shared memory regions are given in section 4.

This memory architecture separates the GPOS from the RTOS (Real Time Operating System) and binds them to specific processors. Since the GPOS is not executed on the RTUs, there are no displacements in the caches and the TLBs that are caused by non real-time software. Additionally, the partitioning of memory avoids cache displacements caused by *cache snooping*. This software transparent technique keeps the caches of all processors in a system in a consistent state. That means, if a processor writes to a memory location which is in the cache of another processor, the corresponding cache line of this processor has to be reread to be consistent with the new state in memory.

4 Arrangement of Code and Data in Memory

Modern processors feature a huge cache memory which speeds up the execution of software. However, for real-time systems, these caches are a source of unpredictability. If the needed code or data is not in cache, the processor must fetch it from main memory. An access to main memory not only lasts much longer than an access to cache, it also competes against peripheral devices that want to transfer data from or to main memory.

Our goal is to keep as much as possible real-time code and data in cache to avoid accesses to main memory. If the real-time software is larger than the cache, the consequential cache displacements should be predictable. We have to know where in cache they occur and what kind of displacement it is. First of all, we define the scenario to investigate:

Scenario. The real-time system consists of a set of tasks $\mathcal{T} = \{\tau_i : i = 1 \dots N\}$. Each task $\tau_i = \{P_k(A_k, S_k), D(d_l) : k = 1 \dots K\}$ consists of procedures P_k with the attributes $A_k = \{ro, rw, locked\}$ and the size S_k in bytes, where $D = \{d_l(A_l, S_l) : l = 1 \dots L\}$ denotes the data used by τ_i together with their attributes A_l and size S_l. The number of bytes b needed by each item denotes to $b_x = S_x + (S_x \ mod \ a)$ when a is the alignment (usually four bytes). The total number of cache lines needed by this system is

$$C_{total} = \sum_{n=1}^{N} \frac{1}{clsize} \left(\sum_{k=1}^{K} b_k + \sum_{l=1}^{L} b_l \right)$$

where $clsize$ denotes the size of a cache line in bytes. In the following, we refer to an $item$ as an element of τ_i.

The cache we consider is a *N-set associative cache* with m bytes per cache line and s sets. Thus, the size S of the cache denotes to $s * N * m$ bytes.

Attributes. The first step is to assign an attribute to every item in memory. Attributes can be either *read-only (ro)*, *read-write (rw)* or *locked*. Code is always *read-only*, that means, during lifetime, it is never modified. Therefore, if an displacement occurs, we know there will be no write-back operation to memory. Data objects can be *read-write*, that means, they will be modified during lifetime and must be written back to memory if displaced. The third attribute *locked* means that this item must never be displaced from cache.

When arranging all items in cache, only items with the same attribute may share one set. Thus, we are able to predict if the displacement of a certain item has a write-back operation as consequence or not. Additionally, we are able to lock some items in cache though this is not supported in hardware by processors of the AMD Athlon or AMD Opteron family.

Multiprocessor Systems and TLBs. In multiprocessor systems, there is the need to keep the caches of the different processors in a coherent state. This need arises when these processors share the same memory regions. They implement a *cache snooping* protocol in hardware which automatically updates the caches of the processors. To avoid displacements from cache caused by *cache snooping*, it is important that every processor works on its own physical

address space. This is realized by our scheme of memory division (see section 3). In a NUMA architecture, each processor has its own physical memory and its own physical connection to it. Thus, to avoid *cache snooping* effects, software must be placed into the local memory of that processor where it is executed on.

When arranging the items in main memory, one has to take care of the TLBs. These little buffers work as caches for the information needed for the translation from a virtual to a physical address. They are build up similar to the caches. It is important not to spread out the items in memory that belong together like for example the code of a task and its data. This would generate additional TLB misses, that are accesses to main memory. A more detailed description of how this works is given in section 4.1.

Shared Memory Areas. Another important thing to mention is memory space that it used for exchanging data between tasks that run on different processors. These memory areas can cause *cache snooping* effects as described above. For example, one task on processor A wants to transfer data items to another task running on processor B. The assumption is, that the read and write accesses of the two tasks are synchronized, but this is a suitable account for a scenario like this. Now, there are two possible solutions to this problem:

- The first is to make the memory area used for this transmission *uncacheable*, that means, this location will not be cached at all. The advantage is that no *cache snooping* effects occur. It does not matter where in memory this area is located. The disadvantage is, that the processor has to access memory for each item and there is no prefetch effect like loading a whole cache line at once. This method makes sense if small data items have to be transferred.

- The second solution is to make the shared memory area *write-through*. That means, all read and write accesses are cached, but the write accesses are not only to the cache but also to main memory simultaneously. This is useful for processor A because it makes no sense to store data in cache exclusively that should be read by a task running on another processor. For processor B that runs the reading task, there is no disadvantage: the data to read is always new from the point of view of the cache. Thus, cache misses on this processor are unavoidable. But these misses are more predictable, because they occur exactly at the point of time the data is written or read and they last only the duration of one memory access. Otherwise, these misses would be produced by the *cache snooping* protocol and would last two memory access, one for the write-back of processor A and one to read the data. This would implicate much more unpredictability. The performance optimizing feature of *write-combining* (that is, to merge several write accesses to one access) is also supported when using *write-through*. This method makes sense if a larger quantity of data items has to be transferred. The shared memory area should be located in the memory area of one of the processors.

It depends on the scenario which solution is the best. It is also possible to combine these methods for different communication relations in one real-time system.

4.1 Memory Management

In the previous subsection, we defined the scenario to consider and the problems we have to deal with. In this subsection, we describe our approach to arrange code and data of real-time software in memory. All these steps take place before the real-time system starts to work. There is no dynamic memory allocation pretended. The approach can be subdivided into the following steps:

- First of all, one has to analyse the real-time software to determine all items and their size. This includes the real-time operating system.

- Then, an attribute has to be assigned to each of the items using the following scheme:

  – Code items are always *read-only*, that means, they are never modified during lifetime. This assumption is reasonable for real-time software because the execution time of self-modifying code is very hard to predict and it lowers the performance of the processor.

  – Data items can be either *read-only* or *read-write*. If they are modified during lifetime, they are *read-write*, otherwise they are *read-only*. The decision, which attribute is suitable for a certain data item is made by the programmer.

  – A special data item is the stack. Each task of the real-time system has got its own stack which is used to store temporary data or to deliver data for function calls. This data item is always *read-write*.

  – A special attribute is *locked*: it can be assigned to each item and indicates that this item must never be displaced from cache. This is useful for small items of very time critical tasks like for example interrupt service routines.

- Now, the items have to be arranged in main memory in that way, that only items with equal attributes share one set in cache.

- It is important to always take a subset of items that belong together (τ_i). These items should be positioned near to each other in main memory. The reason for this are the TLBs, that can address a certain area (*page*) of memory with one entry. It is advantageous to place the items of each τ_i in one page. Thus, additional memory accesses due to TLB misses can be avoided.
- The final step is to place the data areas that should be used for data exchange between different tasks. These areas are allocated once during the initialization of the real-time system and can be placed in areas with the attribute *read-write*.

The realization of this approach for multiprocessor systems based on commercial off-the shelf components is the topic of the next section. Before that, we want to give an example of how this algorithm works.

An Example. Consider the code shown in figure 2: this task implements the bubblesort-algorithm. The data that

```
unsigned long a[LENGTH];

void task1(void) {
  unsigned int n,i,interchanged,help;

  /* initialize data array */
  for(i=0; i<LENGTH; i++)
    a[i] = LENGTH-i;

  /* bubblesort */
  n = LENGTH-1;
  do {
      interchanged = 0;
      i = 0;
      do {
          if(a[i] > a[i+1]) {
              help   = a[i+1];
              a[i+1] = a[i];
              a[i]   = help;
              interchanged = 1;
            }
          i++;
        }
      while(i < n);
    }
  while(!interchanged);
}
```

Figure 2. Bubblesort algorithm

should be sorted is located in the array `a` which is of size `LENGTH` $\cdot\ 4\ bytes$ (4 bytes is the size of `unsigned long` when using gcc 3.3.2 on an IA32 architecture).

There is a single task $\tau_1 = \{P_k(A_k, S_k), D(d_l) : k = 1, l = 1 \ldots 5\}$ with $P_1 = (ro, 154\ bytes)$ and the data items $D(d_1(rw, \texttt{LENGTH} \cdot 4\ bytes), d_2 = d_3 = d_4 = d_5(rw, 4\ bytes))$. The *154* byte code size can be obtained from tools like *objdump*. The data items $d_2 - d_5$ are `n, i, interchanged` and `help`. The attributes of the items are assigned according to the scheme given in the paragraph before: code is always *read-only* (ro), the data item d_1 contains the unsorted data at the beginning of the algorithm and the sorted data at the end, so it must be *read-write* (rw). The other data items ($d_2 - d_5$) are local variables of the task and thus on the stack, which is always *read-write*.

To arrange these items in memory, d_1 is placed to a free region in memory considering the alignment ($start\_address\ mod\ m = 0$). Then, the stack of this task (the place where $d_2 \ldots d_5$ exist) is positioned so that no displacements with d_1 can occur. If $sum = \sum_{i=2}^{5} d_i$ fits into one cache line (in this example $m >= 16\ bytes$) it is placed into one set just behind the last set of d_1. Thus, it is ensured that a miss in d_1 will not generate a miss in $d_2 \ldots d_5$.

5 Tool Chain

As mentioned in the previous section, code and data of real-time software have to be put at specific physical addresses. Therewith, it is possible to map these items to dedicated sets in cache.

In order to arrange the items, we developed the tools *rcmc (RECOMS Colored Module Creator)* and *mmo (Memory Management Optimizer)*. Both tools use the ELF (Executable and Linkable Format) object code as input, there is no need for source code annotations or to recompile existing real-time software.

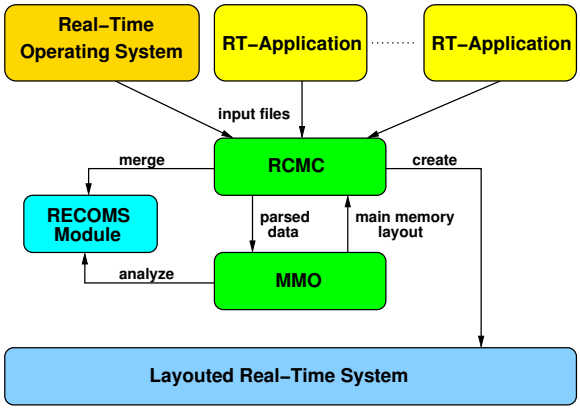

Figure 3. The RECOMS tool chain

The tool chain we use to create a rearranged binary is shown in figure 3. First of all, the object code has to be generated by a compiler. After that, the generated object files (RT-Applications and Real-Time Operating System) are parsed for the code (in form of functions) and data (in the form of single variables, data structures and stack usage). Then, the object files are merged to one all-embracing object file by *rcmc* (RECOMS Module in figure 3). The

relevant information like for example the dependencies of the items and their sizes, are forwarded to *mmo*, which generates the layout that is used for the memory arrangement. Now, *rcmc* creates a new object file using the memory layout given by *mmo*. In the next step, the binary code and data items are loaded into the memory areas of the RTUs. A new memory management driver developed by us ensures that the binary items are loaded to the correct addresses. After that, the real-time application can be started.

5.1 Rearranging an Object File

To rearrange functions and data within an ELF object file, the following information is needed from this file:

Sections: Binary code and data of real-time applications are grouped into sections. Without any restrictions, binary data can only be moved within a section.

Symbols: Symbols specify the offsets of items within an object file.

Relocation Entries: Relocation is the process of connecting symbolic references with symbolic definitions. This allows executable and shared object files to hold the right information for the executable image of a process. Relocation entries are the references that are needed for this.

Relative Addressing: These are the positions of jumps and function calls relative to an offset.

Functions: A function is a piece of binary code which can be seen as an individual object. In particular, a function has a dedicated entry and terminates with a *return* or *jump* instruction.

Variables: This is the information about the position and size of each variable that is used by functions that are defined in the object file.

If a function or variable should be moved to a new position, the following steps have to be performed: first of all, the area where the binary data should be stored has to be cleared. This is done by moving the overlapping functions or variables to the end of the respective section. If the section has got initialized data, the binary data is copied to the new position. After that, the values of the symbol of the function or data being moved have to be adapted. At last, the relative addresses of code which uses the moved binary data as target have to be adjusted.

This concept of moving function and variables within an object file leads to the following advantages:

- The semantics of the code is not modified. No instructions or variables are added or removed. Thus, if a piece of code works correctly, the rearranged version will also do.
- The information needed for moving functions and variables can be gathered from the object file. There is no need for additional source code annotations.
- Existing tool chains for developing software can be used further on, as the rearrangement only depends on the object code.
- In contrast to reprogramming the Memory Management Unit, this method is more fine-grained.

When the arrangement of functions and variables has finished, the resulting binary is *linked* to a specific virtual address. This address is obtained from our memory management and ensures a correct mapping of the binary code to physical addresses. Remaining dependencies are solved by using a *linker script*. After the linking process, the binary data is extracted from the object file and copied to the obtained virtual addresses. Then, the memory needed for sections containing uninitialized data is allocated. If everything is set up correctly, the real-time applications are started.

Memory Driver. The range of physical main memory where the application will be executed is managed by a special memory driver. This driver takes control of a certain amount of memory space that is used for real-time purposes exclusively. This driver manages the memory allocations for all items and also manages memory areas used for interprocess communication mechanisms like shared memory. The same applies to stack regions, that means, regions in memory that are used as stack for particular tasks.

6 Case Studies

In this section, we want to present results we achieved by examining the run-time behaviour of software that runs on multiprocessor systems based on commercial off-the shelf components. On the one hand, these results show the need for a suitable memory management for real-time systems. They motivate our approach to arrange items in memory and they show the benefits of this approach. On the other hand, it is necessary to have a tool chain like the one we presented in section 5 to get accurate results on the computer systems we used for our research.

We did the studies on two different computer systems (see figure 5): the first is a dual AMD Athlon SMP computer where each processor runs with 1533 MHz. It is equipped with 512 MB DDR-RAM. The second is a quad AMD Opteron system that features the NUMA architecture. Here, each processor runs with 1800 MHz and each of the four nodes is equipped with 2 GB DDR-RAM.

The four processors of the AMD Opteron system are connected via hypertransport links. Accesses to peripheral devices that are connected to the PCI-X bus must all tunnel through the GPU or RTU 2, respectively.

The processors of the AMD Athlon system are connected via the host bus and the north bridge (NB) to main memory. The PCI-bus is connected via the north bridge, too. That means, if a processor and a PCI-device want to access main memory simultaneously, they interfere at the north bridge. The same applies to two concurrent accesses of the processors: they also interfere at the north bridge.

The caches of both processor architectures are very similar. Both have 128 kB of L1 cache, that are split into 64 kB instruction cache and 64 kB data cache (harvard architecture). The L2 cache is a unified cache on both architectures. They only differ in size: the AMD Athlon is equipped with 256 kB of L2 cache, the AMD Opteron with 1024 kB. The cache line size is 64 Byte for both caches on both architectures. The L1 cache is 2-way set associative, the L2 cache 8-way set associative on both architectures.

Measuring Execution Times. Our approach to determine execution times on modern processors is like the methodology presented by Petters in [9]. We use the *time stamp counter* (TSC) of these processors. This is a register, that is increased by one with each clock cycle of the processor. The execution time of a piece of code is the difference between two time stamps taken at the beginning and the end of the code sequence. We make sure that all instructions are executed before taking the time stamps to be more accurate. We have the possibility to invalidate the caches and the TLBs before a new measurement. It is possible to fill the caches with data that has to be written back when displaced later on, or with data that needs no write-back operation. Additionally, it is possible to invalidate the instruction cache only, the data cache only, or both.

The objective is to investigate the effects of the caches and the TLBs on execution times of software. Thus, it is possible to bound the worst case execution time of a piece of code with respect to the architecture of the processor. This is not necessarily the overall WCET, because the WCET is not only dependend on the processor, but also on the activity of peripheral devices.

We are able to measure on several processors in parallel which offers the opportunity to investigate the effects of concurring memory accesses of different RTUs. Additionally, we use the *performance monitoring counters* (PMCs) to study the processes within the processor and to control our results. For example, the PMCs allow to count the number of L1 cache misses, L2 cache misses or TLB misses. Four events can be counted simultaneously. For more details on the TSC and the PMCs and the overall processor architecture refer to [1] and [2].

In the following paragraphs, we give results concerning the impact of memory management on the execution time of software. We give no depiction of how to determine the WCET of software. These studies account for our approach to arrange items in memory.

Different Levels of Cache. The caches are composed of two levels, the L1 cache and the L2 cache (see section 1). From the point of view of WCET analysis, it is important to know, how long it takes to access L1 cache compared to L2 cache. Figure 4 shows the memory subsystem.

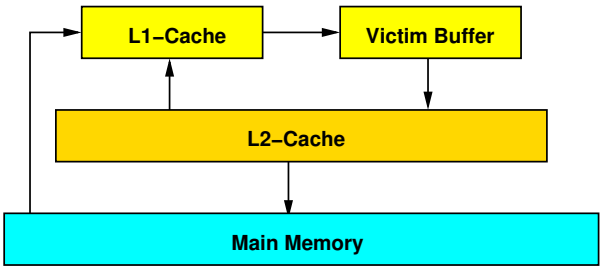

Figure 4. The memory subsystem

Between the L1 cache and the L2 cache, there is a *victim buffer* that holds some cache lines that were evicted from L1 cache. This buffer is written back to L2 cache if there is some idle time. Otherwise, it is written back before a cache line from L1 cache can be evicted. Our measurements show, that for the AMD Athlon and the Opteron system, there is nearly no difference in execution time when code and data are in L1 cache or L2 cache. The influence of the victim buffer is negligible on both systems.

That means, it does not matter in which level of cache an item is, the access time is nearly the same. The difference is in a magnitude of only 10-20 processor cycles. This fact simplifies the memory arrangement and thus WCET analysis, because you can treat the caches as if it was *one* cache. Concerning the feature of cache locking, this property is irrelevant, because items are always locked in L1 cache.

Displacements from Cache We did some research about the characteristics of displacements from cache to main memory. There are two possible scenarios: cache lines can be replaced without any *write* access to memory. Or, if some contents was modified, one write access for the modified cache line is necessary. We studied these scenarios on both computer systems. The measurements were taken *without* any disturbances from other devices.

On the AMD Athlon system, the difference in run-time between displacements with write-back and those without write-back ranges between 9% and 18%. On the AMD Opteron, this difference ranges from 50% for small

amounts of data (16 kB in this case) and about 5% for big amounts of data (bigger than the whole cache).

The read access for a new cache line and the write access for the old cache line proceed in parallel. The possibility to get a worst case situation, that is, the read access is delayed because of a write access, is smaller for large amounts of data than for small amounts of data. This does *not* mean that there are different execution times in the worst case depending on the amount of data. Thus, the longest delays we can determine have to be taken into account for WCET analysis.

These results emphasize the need to know, if a write-back is needed for a certain cache line or not.

Concurrent Accesses of Different Processors. One of the key characteristics of our approach to arrange items in memory is to separate code and data that is executed from different RTUs. Thus, unwanted und unpredictable displacements caused by *cache snooping* are avoided.

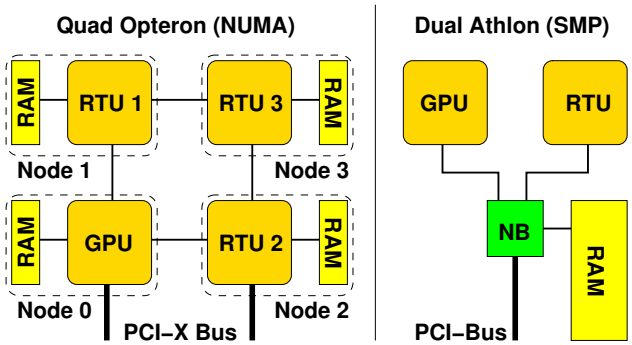

Figure 5. NUMA and SMP architecture

In SMP systems, the processors compete for each other when accessing physical memory in parallel (see figure 5). It is difficult to determine the impact of this competition on the execution time of software. Mostly, it depends on the synchronism of the concurring accesses. We did many measurements to get an order of magnitude of how this conflict affects the execution time of software. In the average case, we got an extension of about 10% of the execution times. But in worst case scenarios, we measured a run-time extension up to 100%. These numbers show, that the increase of execution times of software can vary within a large range.

These times are nearly impossible to predict for a WCET analysis, thus, always the worst case has to be taken into account. To improve this situation, it is possible to put all tasks that frequently need memory accesses on one RTU. Other tasks that fit into the cache can be put to another RTU to avoid collisions. Another solution is to lock small tasks into the cache.

Further on, we measured the execution times for the concurrent access of an identical area in memory and we had an increase of execution time up to 370%. This number stresses the need to avoid this scenario where possible. It is only necessary for memory areas that are needed for the exchange of data between tasks (see section 4). This experiment shows the influence of *cache snooping* on the execution times.

Accessing Different Memory Regions. In NUMA systems, the situation is different. Each processor has got its own physical memory. Thus, if software is running in parallel on different processors, there are no conflicts concerning memory accesses. An interesting question is, how long last memory accesses from one processor to the local memory of another processor. This scenario occurs when a transfer of data between tasks running on different processors is needed.

Table 1 shows the results we got when measuring the time that is needed to transfer 64 kB of data to different nodes. The setup of the system is shown in figure 5. These measurements were taken without any activity of the PCI-X buses.

target / source	RTU 1	RTU 2	RTU 3
RTU 1	83 μs	110 μs	97 μs
RTU 2	110 μs	83 μs	97 μs
RTU 3	97 μs	97 μs	83 μs

Table 1. Access times of different nodes

The results in table 1 show, that there are noticeable differences in access times depending on the source and the target node. An access from node one to node two involves an increase in execution time of 32.53%, compared to the access time to local memory. From node one to node three, there is an increase of 16.87%. The increase when changing the direction of the access remains constant. The increase for accesses from node three to node two and to node one is the same. This is also true for greater amounts of data.

We did the same measurements with concurring memory accesses from other processors. For example, accesses from node one to node three and vice versa in parallel. There was no noticeable difference in run-time. This is a consequence of the hypertransport protocol which defines two independent data streams per link.

Concerning our approach of memory arrangement, it can be useful to take a memory area on node three to share data between tasks running on node one and node two. Assumed that there is no need to use this area on node three, this is a suitable solution. When exchanging data between adjacent

nodes, one has to choose the memory of one of the two nodes.

6.1 Experimental Results

On the one hand, the time that is needed for a memory access depends on the underlying hardware, that is bus systems, the memory controller and the kind of memory itself (e.g. SRAM, DDR RAM). On the other hand, it depends on the connection of other processors or peripheral devices to the memory and the activity of these devices.

Here, we want to present some results to compare the run-time of software when it is arranged in cache with the run-time when it has to access memory. To give an order of magnitude of the impact of concurrent memory accesses on the run-time of software, we performed random hard disk and ethernet activity in parallel to the execution of a task. These results should point out the complexity and the dimension of uncontrolled memory accesses. These measurements were taken on the AMD Athlon system. Experiments on the AMD Opteron system show similar results.

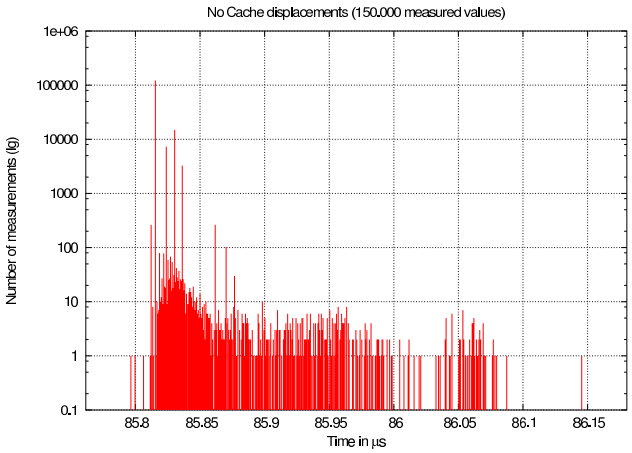

Figure 6. Run-time without displacements

Figure 6 shows the execution time of the first loop of the bubble-sort algorithm shown in figure 2 with `LENGTH`=8192. That means, an array of 32 kB is loaded into the data cache. In this case, the task was executed in cache without any displacements. This is the best case for real-time software. Remarkable is the very small jitter.

In contrast to this, figure 7 shows the same scenario, but this time, the code and data of the first loop has to be loaded from main memory first. Additionally, there are random memory accesses of peripheral devices in parallel. At random means, the highest value we measured is not necessarily the WCET of this task. Bounding the WCET in cases like that is not the scope of this paper.

Table 2 summarizes the results. One can see, if there are

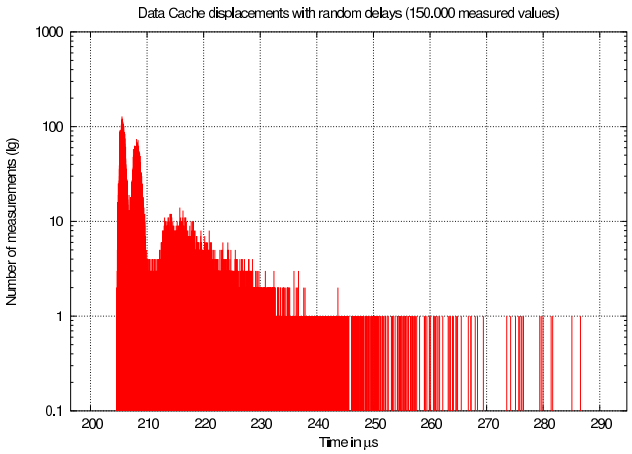

Figure 7. Run-time with concurrent accesses

Scenario	min. time	max. time	run-time variation
Best Case	85.80 μs	86.15 μs	0.4%
Displacements with delays	204.68 μs	286.59 μs	40%

Table 2. Summary of results

memory accesses in parallel, execution times of software vary in a broad range. Here, there is an increase in execution time of factor 3.33 compared to the best case scenario. Additionally to this, the run-time variation, which is a quantity for the predictability of execution times, increases by factor one hundred.

These results further stress the need to keep code and data in cache where possible. This does not only accelerate software execution, it makes run-times predictable. This is of great importance for real-time systems.

6.2 Conclusions

The case studies and experiments we presented in this section render the great impact of memory accesses on the execution time of software. The best case for real-time software is to fit into the caches. Thus, execution times are predictable and programm execution is as fast as possible. Our results show, that the two levels of cache on modern processors can nearly behave as one cache which simplifies the memory arrangement.

These results point up the benefits of cache locking. Tasks that run in cache suffer no extension of their run-time by memory accesses and are thus predictable and fast. This feature is very useful for static WCET analysis.

The analysis of the impact of displacements from the cache on execution times of software show that there are big differences depending on the situation. It is necessary to

distinguish between displacements with and without write-back operation. This is what our approach enables the system designer.

In addition to that, it is important to know what is happening around the processor during a memory access. Concurring accesses from further processors or even peripheral devices can extend access times for real-time software tremendously. Furthermore, the actual effects are nearly impossible to predict and thus one has to account for the worst case. Our research showed, that this worst case situation rarely appears, so this is a source of overestimation of computation time needed for a real-time task. In addition, this worst case situation is hard to adjust, so it is very difficult to determine a safe WCET for this. Our approach of memory arrangement enables the system designer to avoid situations like this where possible.

7 Summary and Future Work

In this paper, we presented an approach to arrange code and data of real-time software in memory. This approach aims to improve the speed and predictability of execution times of real-time software. The background of this work is the intention to be able to use the computation power of modern commercial off-the shelf computers for real-time systems. We presented a tool chain which implements our approach of memory arrangement. These tools are based on standard open source software and only need the object code as input.

Our case studies in section 6 showed the importance of a suitable memory management for real-time systems. They show the benefits of our approach, but they also show its limits. It is important to know the kind of displacement from cache (with or without write-back operation). Furthermore, it is important to know what is happening in parallel to memory accesses. The memory division for multiprocessor systems presented in section 3 enables the programmer of real-time software to control this up to a certain limit.

Another feature of our approach is the ability to lock certain code and data items in cache. Our case studies showed the benefits of this feature. Therewith, execution times of software are nearly constant and thus predictable.

In the future, we want to investigate different kinds of real-time applications to find suitable heuristics on how to arrange them best in memory. Furthermore, we want to delve into the characteristics of concurrent memory accesses in SMP and NUMA systems to find new methods of how to allocate processors for different tasks.

In the near future, new computer systems that combine SMP and NUMA systems will appear on the market. So it is important to do further research work on this field.

References

[1] AMD, Sunnyvale, CA, USA. *AMD Athlon Processor x86 Code Optimization Guide*, Sept. 2000.

[2] AMD, Sunnyvale, CA, USA. *AMD64 Architecure Programmer's Manual Volume 2: System Programming*, Sept. 2003.

[3] A. Arnaud and I. Puaut. Towards a Predictable and High Performance Use of Instruction Caches in Hard Real–Time Systems. In *Proceedings of the work-in-progress session of the 15th Euromicro Conference on Real-Time Systems*, pages 61–64, Porto, Portugal, July 2003.

[4] 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.

[5] A. Campoy, A. Perles, F. Rodríguez, and J. V. Busquets-Mataix. Static Use of Locking Caches vs. Dynamic Use of Locking Caches for Real–Time Systems. In *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2003)*, Montreal, Canada, May 2003.

[6] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 1997.

[7] J. Liedtke, H. Härtig, and M. Hohmuth. OS–Controlled Cache Predictability for Real–Time Systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 9–11 1997.

[8] E. Petrank and D. Rawitz. The Hardness of Cache Conscious Data Placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 101–112, Portland, Oregon, 2002.

[9] S. M. Petters. Bounding the Execution Time of Real–Time Tasks on Modern Processors. In *Proc. of the 7th Int. Conf. on Real–Time Computing Systems and Applications*, Cheju Island, South Korea, Dec. 12–14 2000.

[10] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real–time systems. In *Proceedings of the 2nd International Workshop on worst–case execution time analysis in conjunction with the 14th Euromicro Conference on Real–Time Systems*, Vienna, Austria, June 2002.

[11] S. Schönberg. Impact of PCI-Bus Load on Applications in a PC Architecture. In *Proceedings of 24th IEEE International Real–Time Systems Symposium*, Cancun, Mexico, Dec. 2003.

[12] J. Stohr, A. von Bülow, and G. Färber. Using State of the Art Multiprocessor Systems as Real-Time Systems – The RECOMS Software Architecture. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.

[13] A. von Bülow, J. Stohr, and G. 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.