ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, La Jolla, California, June 1995.

# Compiler Support for Software-Based Cache Partitioning

Frank Mueller
Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin (Germany)

*e-mail: mueller@informatik.hu-berlin.de* *phone: (+49) (30) 20181-276*

## Abstract

Cache memories have become an essential part of modern processors to bridge the increasing gap between fast processors and slower main memory. Until recently, cache memories were thought to impose unpredictable execution time behavior for hard real-time systems. But recent results show that the speedup of caches can be exploited without a significant sacrifice of predictability. These results were obtained under the assumption that real-time tasks be scheduled **non-preemptively**.

This paper introduces a method to maintain predictability of execution time within **preemptive**, cached real-time systems and discusses the impact on compilation support for such a system. Preemptive systems with caches are made predictable via software-based cache partitioning. With this approach, the cache is divided into distinct portions associated with a real-time task, such that a task may only use its portion. The compiler has to support instruction and data partitioning for each task. Instruction partitioning involves non-linear control-flow transformations, while data partitioning involves code transformations of data references. The impact on execution time of these transformations is also discussed.

## 1 Introduction

Cache memories have become a major factor to bridge the bottleneck between the relatively slow access time to main memory and the faster clock rate of today's processors. Yet, in the area of real-time systems, cache memories were thought to introduce *unpredictable* behavior in terms of worst-case execution time (WCET). In real-time systems, the results of schedulability analysis can only be applied for *predictable* WCET. As a result, real-time designers either disabled cache memories, or allowed only certain portions of the system to be cached, or even used processors without caches. These approaches become less feasible with the increasing importance of cache memories.

Results in schedulability theory provide a firm base for rate-monotone scheduling, earliest deadline first scheduling, and other preemptive scheduling paradigms [LL73]. This is also reflected in the increasing number of preemptive real-time operating systems [GL91, Hil92]. These systems are available for a number of cached processors.

Recently, it has been shown that tight predictions of the WCET of programs can be obtained even for cached systems [AMWH94, Mue94]. These results were obtained under the assumption that tasks be scheduled non-preemptively. This paper discusses how these results can be generalized for preemptive systems via cache partitioning. For an existing architecture, the cache space can be divided into partitions, each of which are only used by certain real-time tasks [Wol93]. To ensure that a task only uses its partition, the instructions and data can only be within a certain portion of the addressing space. These portions are scattered over the entire addressing space, thereby providing a non-linear addressing space for this real-time task.

This paper focuses on the compiler support for cache partitioning. The instructions of a conventional task comprise a linear (contiguous) addressing space. The compiler can transform the instruction layout to match a non-linear addressing space by splitting it into partitions. At the same time, the control flow has to be adjusted to preserve the functionality of the task. For example, unconditional jumps may have to be inserted as the last instruction of each partition. Similarly, data partitioning may require large data structures to be distributed into several non-linear memory regions. The access to these data structures has to be adjusted by the compiler.

Software-based cache partitioning is a trade-off between predictability and performance. The WCET of each task becomes predictable when distinct cache partitions are allocated to each real-time task. The code transformations by the compiler provide the means for cache partitioning but they also introduce additional code. This paper discusses the impact of partitioning and the additional code on the performance of a task.

The paper is structured as follows. In section 2, the software-based partitioning scheme is introduced. In section 3, the necessary compiler transformations to support partitioning are detailed. Sections 4 and 5 discuss the impact on object libraries and the operating system, respectively. Section 6 outlines how to generalize partitioning to other cache architectures than just direct-mapped caches. In section 7, estimates are given for the impact of partitioning on performance. Section 8 outlines future work. Section 9 reviews related work. Finally, conclusions are presented in section 10.

# 2 Software-Based Cache Partitioning

This section introduces a software-based cache partitioning scheme for existing architectures [Wol93]. For the following discussion, a processor with a split cache is assumed, where both data and instruction caches are direct-mapped caches. Extensions to other cache architectures are discussed in later sections.

A direct-mapped cache is divided into $l$ cache lines, each of which are of size $s$. For example, a 1kB cache with $s = 16B$ has $l = 64$ lines (see Figure 1). A cache tag is associated with each line. When a reference to an address is made, the address is split into tag $t$ (most significant bits), index $i$ , and offset $o$ (least significant bits). In our example, the offset comprises 4 bits, the index has 6 bits, and the remaining bits are used as a tag. The address reference results in a comparison of tag $t$ and the tag associated with cache line $i$. If the tags match, a cache hit occurs, *i.e.* the cache line is valid and can be used to resolve the reference by addressing the content at offset $o$ within the line. If the tags do not match, a cache miss occurs and the reference has to be resolved by loading the entire line from main memory into cache and updating the tag before the content at offset $o$ can be provided.

The cache can be partitioned into $n+1$ different sections for a set of $n$ real-time tasks $\{\tau_1, ..., \tau_n\}$. Considering the 1kB cache discussed before, a partitioning for $n = 5$ may be chosen to be $\{20, 10, 8, 8, 6, 12\}$ cache lines for the corresponding tasks, where the last number represents the number of lines for a shared partition $\tau_s$ (see Figure 2). The number of lines should be chosen with respect to the priority of the task and the code/data size of the task. The memory mapping ensures that each real-time task only accesses its cache lines with the exception of synchronization, when the shared partition is accessed. Non-real-time tasks only access the shared

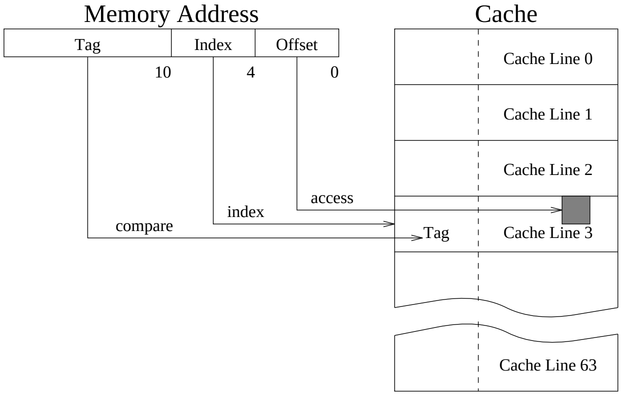


Figure 1: Indexing into a Direct-Mapped Cache

partition. Data and instruction caches can be partitioned differently, according to the demands of the task set. This partitioning scheme provides exclusive access of one task for a certain cache partition. The exclusive partition access provides the means to statically analyze each task separately and predict it's caching behavior.

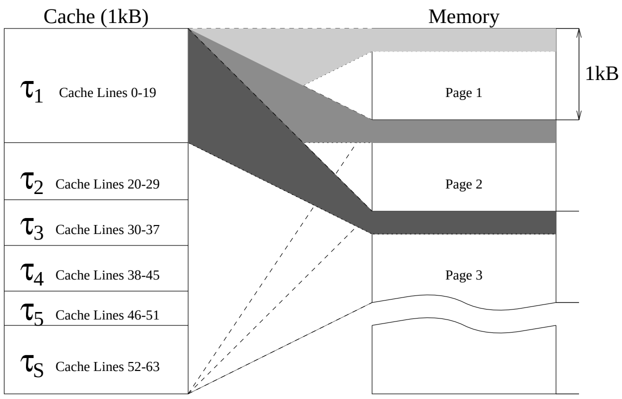


Figure 2: Cache and Memory Partitioning

The code and the data of a task have to be restricted to only those memory portions that map into the cache lines assigned to the task. If the code/data size of a task exceeds its cache partition size, the code/data has to be scattered over the address space. In the above example, consider $\tau_1$ with 10k instructions. The instruction space will be divided into 32 portions of 320B each, since $\tau_1$ was given the first 20 lines (320B) in the cache. Thus, the first 320B within each 1kB page in main memory contain instructions of $\tau_1$, up to page 32. (In the context of this paper, the memory page size is given by the cache size, which does not have to match the system page size.)

So far, the partitioning was performed under the assumption that each task needs a private partition of the cache. But this is not necessarily the case. Recent work

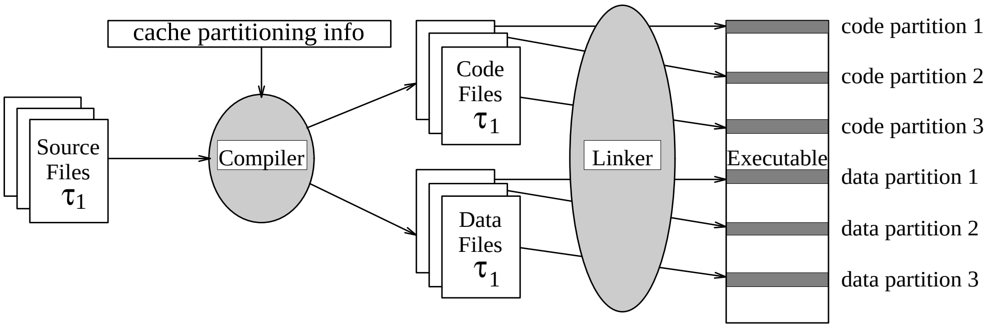


Figure 3: Compiling and Linking

by Audsley and Tindell [AT95] shows that tasks at the same priority level are scheduled *non-preemptively* relative to *each other* in an otherwise preemptive environment that supports FIFO scheduling at the same priority level. This observation can be used in the context of cache partitioning to let tasks at the same priority level *share* the same cache partition. A task can only be preempted between two context-switch points by a higher priority task. Thus, tasks at the same priority level cannot interfere with each other with respect to their cache accesses, even if they use the same cache partition. The predictability of cache behavior remains unchanges. The only difference is that the start-up time of tasks may increase since the caches lines of a periodic task are most likely replaced by another task (of the same priority) between two runs. The worst-case execution time analysis takes this restart overhead into account in any case. Furthermore, the partition at priority levels addresses the potential issue of unreasonable cache fragmentation. Rather than having to divide the cache space into $n$ partitions for $n$ tasks, it suffices to create $p$ partitions, one for each used priority level. This should provide a much better cache utilization.

# 3 Compiler Transformations

The distribution of a task's code and data over non-linear partitions in the addressing space can be automated via compiler and linker support, thereby making the partitioning transparent to the user. Figure 3 illustrates the compilation and linkage process. The compiler is supplied with the cache size and the partition size of a task as an additional input when the task is compiled. It produces separate object files for each code partition and each data partition. The object files of all tasks are combined into an executable by the linker. The figure only shows the positioning of object partitions for task $\tau_1$. The object partitions of other tasks are positioned in between, according to the order of cache partitioning. If the compiler did not produce separate object files, the linker would have to be modified to perform the partitioning of object code. But it seems that the partitioning can be done more easily by the compiler, requiring certain code and data transformations discussed in detail in the following.

## 3.1 Code Partitioning

The code generated by the compiler is split into portions of equal size according to the size of the instruction cache partition. Each portion, called a memory partition, is terminated by an unconditional jump to the next memory partition unless the last instruction in the partition already performed an unconditional transfer of control.[1] Each partition is stored in a separate object file. The file may be padded with no-ops at the end to extend it to the exact size given by the cache partition.

Besides adding an unconditional jump at the end of each partition, the transformations on the generated code can be restricted to instructions that perform a transfer of control. Transfers of control within a memory partition (*local* transfers) conform to the rules of a linear addressing space traditionally handled by compilers. The following discussion can thus be limited to transfers across memory partitions (*remote* transfers).

- A remote *conditional branch* to label $L$ may increase the distance between branch source and target when compared to a local branch. If the distance exceeds the number of bits in the encoding of the branch instruction, then the control transfer has to be performed as a local branch to $L1$, followed by a remote unconditional jump to label $L$ at the branch destination $L1$.
- A remote *unconditional jump* or a remote *call* should generally not be affected since most modern architectures allow the encoding of any destination within the addressing space. However, should a certain architecture not support the entire address

[1] For the sake of simplification, the discussion abstracts from branch delay slots. One would simply have to ensure that the delay slot is in the same partition as the unconditional jump.

```
int i, sum, a[1000];

...
for (i = 0; i < 1000; i++)
   sum += a[i];
```

(a) Original Code

```
...
for (i = 0; i < 1000; i++)
   sum += a[f(i)];
...
int f(i)
  int i;
{
  return (i/PS)*CS + i%PS;
}
```

(b) Indexing Function

```
int max_i, max = 1000;
max_i = (max/PS)*CS + max%PS;
...
for (i = 0; i < max_i; i++) {
  sum += a[i];
  if (i % PS == 0)
    i += CS - PS;
}
```

(c) Counter Manipulation

Figure 4: Transformation for Large Data Structures

space, then the jump/call can be transformed into an indirect jump/call.

- A remote *indirect jump* or a remote *indirect call* is typically not affected since the entire address space is supported as a destination. However, if an architecture supports indirect jumps only through offsets and if the offset size is exceeded, jump tables may have to be recoded as absolute addresses that are loaded into a register before the transfer of control can be performed.
- A *return* from a function to a remote caller is not affected since it is equivalent to an indirect jump through a register containing the return address. However, the destination of the return (*i.e.* the instruction following the call) has to be positioned in the same partition as the call.
- A *trap* into the operating system will always result in a remote transfer of control into the operating system task, handled as a separate task with its own cache partition (discussed in detail below). Thus, traps are not affected by the partitioning.

Notice that the additional code required by the transformations increases the code size within a partition. This has to be taken into account by the compiler when deciding where to cut the code between partitions.

## 3.2 Data Partitioning

The data of a task can be split into portions similar to the handling of instruction partitioning. In the following, the compiler transformations are discussed for global data, local (stack) data, and dynamically allocated data on the heap.

### 3.2.1 Global Data

Global data can be split into memory partitions of the data cache partition size. The compiler has to ensure that no data structure spans multiple partitions. In fact, global data structures can be rearranged in their positional order to fit into the partitions as long as their size does not exceed the cache partition size.

If the size of a data structure, *e.g.* a large array, exceeds the cache partition size, it is split over multiple memory partitions and the compiler needs to transform the access to the data structure. Consider the C code fragment in Figure 4a. An array, originally laid out linearly in memory, is indexed in a linear fashion. Once the layout is changed to accommodate portions of the array in several memory partitions, the array cannot be indexed linearly anymore. The array indexing can be handled in two different ways. Either the index is calculated by a function mapping the original (linear) index into the non-linear memory partitions (see Figure 4b). Or the loop counter is modified to skip to the next partition, thereby performing the remapping of the indexing function (see Figure 4c). The example assumes an integer size of 16 bits (one word), a data cache size of $CS = 64$ words ($1kB$), and a partition size of $PS = 20$ words (*i.e.* 320B or 20 cache lines) for $\tau_1$.

Notice that approach (c) changes the counter semantics, which may have undesired side-effects if the counter is used for other purposes inside or after the loop. Thus, approach (c) can only be used in the absence of sources for side-effects, as determined by the data-flow analysis of the compiler. The transformations, shown at the source-code level for better understanding, should be implemented in the back-end of a compiler after performing optimizations since more information about code and data is available at that time.

The generalization of both approaches to multi-dimensional arrays would potentially involve complicated indexing functions. The compiler may instead ensure that any row of an array resides within a partition, possibly wasting some data space at the end of a partition for the sake of efficiency. The space/time trade-off of such decisions can only be made case-by-case. Long records and arrays of records can be handled similarly.

### 3.2.2 Local Data

Local data on the stack can only be split into partitions by manipulating the stack pointer. Under the assumption that the stack pointer is only decremented on entering a function, a partitioning scheme can be implemented as follows. The stack allocation for the current function is transformed into a sequence of instructions to test if the stack allocation still fits into the current partition. If it fits, the allocation proceeds by decrementing the stack as always. If it does not fit, the stack pointer is forwarded to the bottom of the previous memory partition before the stack is decremented. Figure 5 shows the corresponding pseudo code, where *stackp* is the stack pointer, $CS$ the cache size, $PO$ the partition offset within the stack, and $PS$ the partition size, respectively.

```
if ((stackp / CS) * CS + PO > stackp - offset)
  stackp = ((stackp / CS) * CS - 1) * CS +
           PO + PS;
stackp -= offset;
```

Figure 5: Stack Decrement

On occasion, the stack requirements of a function may exceed the cache partition size. In this case, local data structures have to be split into multiple memory partitions. This results in the remote access of local data and is supported by referencing data relative to the stack pointer plus data offset and partition offset. If the combined offsets (data offset + partition offset) exceed the maximum offset supported by the load instruction, then the compiler generates code to move the offset into a register and then loads a value from the location of this register plus stack pointer. Local data structures exceeding the size of the cache partition are split across memory partitions similarly to global data structures, and the access is modified as already discussed.[2]

### 3.2.3 Dynamic Allocation

Dynamic storage allocation on the heap can be supported as long as the memory request does not exceed the cache partition size. The heap allocation algorithm can be adapted to skip from one memory partition to another when a request does not fit into the current partition. If a request exceeds the cache partition size, it can be scattered over multiple partitions just like global data. When a pointer is dereferenced, the offset calculation has to be performed with an indexing function. The indexing function itself has to be associated with the data type and is passed as a hidden parameter together with the base pointer for subroutine calls. Notice that any pointer parameter requires such a treatment with a dynamically bound indexing function (global, local or heap data). This association is also required at type castings to retain the proper indexing.

An alternative solution would be to resolve large heap requests by allocating memory in a special uncachable portion of the address space. The translation look-aside buffers (TLBs) of modern processors include a caching bit for each page entry. By rendering certain pages uncachable, heap allocation of large chunks can be supported. The price of this facility is the reduced performance due to the absence of cache usage. This seems a feasible compromise, based on the observation that hard real-time tasks will hardly ever use dynamic allocation due to its unpredictable behavior. For non-real-time tasks, the additional overhead is less of a concern due to the relaxed timing constraints. But the former approach, using the index function as an association, seems more orthogonal with the overall model.

## 4 Linker and Object Libraries

The linker gathers the object files corresponding to the data and instruction memory partitions by ordering them according to the cache partitioning to produce an executable. Linked-in library code has to be handled in a different manner. In general, linked-in library code cannot be partitioned in the same way source code is handled since library code is only available as object code and cannot be transformed by the compiler. There are two alternative solutions to the problem.

It is possible to precompile library code for a certain cache partition size. In this case, the task partition size should be chosen as a multiple of the library partition size to better utilize memory. The adjustment of the stack at function entry can be controlled by a task-specific variable indicating the task partition size. Each task has to be separately linked with the appropriate libraries. To take this approach one step further, all partitions could be of the same size (for each task as well as library code). Larger partitions have to integer multiples of this small partition size. Any task can then be composed of building blocks of the same size, providing a flexible, modular approach to the partitioning problem.

In an alternative approach, library code can be provided as source for the compilation. As seen before, library routines cannot be shared between tasks. For example, if two tasks use the heap allocation routine, then two different instances of the routine have to be gener-

[2] Programming languages supporting lexical scoping with nested procedures (*e.g.* Pascal) have to take the transformations for a remote accesses to non-local data structures into account (*e.g.* via displays). This can be supported by annotating the symbol table entries of remote data structures with some access information.

ated, one for each task, to reflect the different cache partition sizes and ensure that a task's code/data does not affect the cache partition of another task.

None of the approaches provides code sharing between tasks. But the first approach still supports object libraries. The second approach requires the availability of library source code, which is often not the case.

So far, the discussion has only focus on statically linked libraries. Dynamically linked (shared) libraries pose an even greater challenge since code is traditionally shared between tasks (locked into memory). This would cause an unacceptable interference between the cache partitions of tasks. Thus, it seems imperative to prohibit the use of shared libraries. Shared libraries are generally also provided as statically linkable libraries, which can be handled in the manner discussed before.

# 5 Operating System

The operating system can be handled as a separate task with its own cache partition. Calls to the operating system have to be handled as synchronization points, potentially involving a context switch to a different task. This view is coherent with the notion of most systems, where kernel calls always establish a new context, the kernel context.

Certain problems remain. For example, the private code and data of the kernel is often mapped into protected pages, indicated by a supervisor flag in the page entries of the TLB. If the object code of user tasks was mixed with kernel code within one page as suggested by the linking scheme, supervisor protection could not be provided for this page. Thus, kernel code and data have to be loaded into memory pages that do not contain any user code. In other words, the portion of a page besides the kernel code/data would remain unused. This restriction is feasible for architectures with sufficient memory and modern real-time micro-kernels. Other issues, such as the placement of trap tables in memory, should simply be handled by preventing the trap table page from being cached.

The operating system also has to provide the facility to control the mapping between virtual and physical memory. A real-time application has to establish the physical memory mapping illustrated in Figure 3. However, if the system page size is an integer multiple of the cache size, then no memory mapping support is needed. The positioning of code and data partitions within a memory page is sufficient in this case to provide the proper mapping into cache partitions.

# 6 Generalization to other Cache Architectures

So far, only direct-mapped split caches have been discussed. Software-based cache partitioning can be extended to other cache architectures as follows.

## 6.1 Set-Associative Caches

The level of associativity of current architectures has been declining over the years, due to the observation that a low level of associativity (if not even a direct-mapped cache) provides high hit ratios as cache sizes increase [Hil88]. Today's processors typically implement at most 4 levels of associativity.

Within an $n$-way set-associative cache, $n$ memory blocks with the same index can be cached at any given time. Since the replacement policy implemented in hardware determines which line is replaced when all $n$ sets are occupied (*e.g.* least-recently used replacement), the lines cached in a set cannot be predicted across tasks in a preemptive environment. Thus, all $n$ sets corresponding to a certain index $i$ have to be associated with the cache partition of one task. Notice that this task can store up to $n$ different blocks with this index, corresponding to $n$ different memory partitions. Thus, the cache capacity of a task is $n * linesize * numlines$ for an $n$-way set-associative cache, where $numlines$ denotes the number of lines of the task's partition. The compiler transformations can be applied as discussed above.

## 6.2 Unified Caches

When data and code are sharing a cache, software-based partitioning can still be applied. The compiler ensures that data and code are mapped into different cache partitions, thereby in effect forcing a split cache at the software level. No hardware modifications are necessary.

# 7 Performance Impact

There are three sources of performance degradation with the suggested partitioning scheme. First, with cache partitioning, a task can only use a small portion of the cache. Second, code transformations may introduce additional control-flow instructions. Third, data transformations may introduce additional instructions to access data structures of remote partitions. On the other hand, the response time after context switches is improved since tasks do not affect the caching of other tasks. This can result in some performance improvements under frequent context switching, in particular for lower-priority tasks.

### 7.1 Impact of Cache Partitioning

When a cache memory is partitioned such that a task only accesses its portion of the cache, capacity misses will increase while the hit ratio decreases (relative to an unpartitioned cache). For example, if the code of a frequently executed loop exceeds the cache partition size, then misses will be encountered on each loop iteration. Thus, a higher miss rate can be expected for any cache partitioning design, whether implemented in hardware or in software.

On the other hand, partitioning is the means to make preemptive systems predictable. In the past, caches have been disabled for preemptive real-time systems. Thus, systems with partitioned caches should be compared with uncached systems. A system with a partitioned cache will exhibit much better performance than an uncached system by exploiting spacial and temporal locality (within the limitations of the partition size).

### 7.2 Impact of Code Transformations

The additional instructions due to changes in the control flow will probably increase a task's execution time only slightly. In fact, the performance impact of code transformations should not result in a significant performance penalty compared with the much more severe impact of cache partitioning.

### 7.3 Impact of Data Transformations

Compiler transformations to access data of a remote partition may be expected to affect the overall performance. For example, matrix operations are commonly performed on large arrays within tight loops. In this case, an efficient implementation of counter manipulations can be used instead of explicitly using an indexing function. Thus, the new counter increment would induce only a shift-right-and-test, a branch, and an increment instruction. This is assuming that the modulo operation can be replaced with a shift operation, *i.e.* the partition size is a power of two.

When the indexing function has to be used, there will be the overhead of an integer division/modulo operation, compare, branch, and increment. The division operation, often quite expensive, can be replaced by cheaper bit manipulations when the data layout is arranged accordingly, which may involve padding the data to adjust record sizes to power-of-2 storage sizes. An additional function call and return may be inflicted if the compiler does not support inlining of the indexing function. Overall, cache partitioning is likely to have a higher impact than data transformations.

The additional overhead for stack allocation involves a shift right, shift left, add, compare, and a branch instruction before the stack is decremented, again assuming that the partition size is a power of 2. Since this overhead is relatively small compared to the impact of cache partitioning, it should not have a significant performance impact.

The access of large heap structures will inflict the same overhead as global data with the indexing function, provided that the indexing function is dynamically associated with the data structure.

### 7.4 Impact of Context Switch Frequency

The discussion about performance impact so far did not take the effect of context switches into account. In a preemptive system, context switches may occur at any given point. When regular caches (without partitioning) are used, the execution of a task triggered by a context switch often invalidates large portions of the cached data and instructions of the previous task.

Cache partitioning ensures that the cached data and instructions of a task are not invalidated by the execution of any other task. For high context switch frequencies, the benefit due to non-interference between tasks under cache partitioning can compensate for a portion of the performance loss due to partitioning. Furthermore, the response time after context switches will improve since cached code/data of a task will remain in memory across context switches. This is an important asset for real-time applications.[3]

In addition, the predictability gained by cache partitioning allows the use of static cache simulation[Mue94] to determine the worst-case execution time[AMWH94] and perform schedulability analysis in conventional cached systems that are preemptively scheduled. Notice that the timing analysis has to take into account that processor pipelines are flushed on context switches, potentially inflicting wait cycles up to the execution time of the slowest instruction (typically some floating point instruction).

## 8 Future Work

The impact of cache partitioning and compiler transformations should be evaluated via a quantitative analysis. This will require longer-term efforts to first implement the compiler transformations in the back end of an optimizing compiler and then perform the evaluation via cache simulation. The performance impact, a function of the cache partition size, context switch frequency, and overhead of compiler transformations, could also be

[3] If multiple tasks are scheduled at the same priority and mapped into the same partition, the above savings are most likely diminished. One task's data and instructions can then be replaced within the partition by another task at the same priority. Also see last paragraph of section 2.

compared experimentally with the average performance of an unpartitioned cached system. A comparison with a hardware-based partitioning scheme may also provide interesting insight, though it seems unlikely that future architectures will readily support hardware-based partitioning for common processors.

Another direction of future research could be the utilization of virtual memory mapping for the sake of cache partitioning. Consider a physically-mapped primary cache whose size is an integer multiple of the system page size. The MMU mapping from virtual to physical addresses can then be used to provide cache partitioning (at the physical level) and retain the view of a contiguous address space for the user (at the virtual level). The MMU is only used for the virtual-to-physical mapping; it is not used to implement a virtual memory management. This approach would not require any compiler transformation but simply operating system support to reprogram the MMU mapping. However, primary cache sizes have to be about 32 times larger than the system page size to support 32 priority levels before this approach becomes feasible. This estimation excludes the associativity level. Consider a 1kB system page size. A direct-mapped 32kB cache would suffice to support partitioning for 32 priority levels. A 4-way set-associative cache of the same size would only support $32/4 = 8$ priority levels since only 8 pages can be arbitrated. But 8 priority levels are often insufficient. It remains to be seen if this approach becomes feasible, depending on how primary cache sizes will develop over time.

# 9 Related Work

Caches can be partitioned by means of software or hardware. A hardware-based partitioning scheme has the advantage that the partitioning is transparent to the software. No special compiler support is required. Kernel calls can be used to identify real-time tasks, so that the operating system initializes the hardware contexts of each real-time task. Thereafter, the cache partitioning is entirely performed in hardware.

A hardware-based cache partitioning scheme, Strategic Memory Allocation for Real Time (SMART), has been proposed and implemented by Kirk [Kir89]. The cache memory is partitioned into equal-sized portions for each task and a larger partition, called the shared pool. The shared pool is used by non-real-time tasks and for synchronization between real-time tasks. The memory management unit is modified to use a task id to index the proper partition and a shared-pool hardware flag to arbitrate between task partitions and the shared pool. The task id is swapped during context switches. Thus, a task cannot invalidate cached portions of another task, thereby gaining predictability under a cached system.

However, hardware-based cache partitioning has some disadvantages. First, partition sizes are fixed. Software-based partitioning supports arbitrary application-specific partitioning. Second, costly custom-made hardware support is needed. Software-based partitioning can be applied to any on-the-shelf architecture.

Software-based cache partitioning was first proposed by Wolfe [Wol93], including the address-space partitioning described in section 2. He also proposed two schemes for resolving memory references by altering the traditional address decomposition into tag, index, and offset. One hypothetical scheme swaps the positioning of index and tag during address decomposition, another hybrid scheme uses some bits above the tag and some bits below the tag to determine the index. Both schemes would provide linear address spaces that do not require any special compiler or linker support. Unfortunately, hardware modifications are needed that cannot be performed for on-chip caches. Wolfe also reported results showing predictable execution times for low-priority tasks in a preemptive system with varying interrupt frequencies. These results were obtained by software-based partitioning of the instructions (object code) since the experimental system only had an instruction cache. Yet, he did not discuss the opportunities for compiler transformations of code or data.

An alternative to cache partitioning for preemptive systems is provided by the non-preemptive scheduling paradigm of the Spring system [NNS91]. Under the Spring system, the execution of a task between two scheduling/synchronization points cannot be interrupted. Thus, the caching behavior between these points can be predicted.[4] Caches are assumed to be flushed during context switches, which provides predictability but does not improve the response time. If a certain response time is required by the overall system, additional scheduling points may have to be inserted by hand into long-running code segments. Furthermore, the scheduling paradigm of the Spring system is aimed at minimizing the number of missed deadlines but cannot provide *a priori* guarantees for timely task completion, as provided by traditional schedulability analysis [LL73].

# 10 Conclusion

This paper describes how software-based partitioning can be used to preserve the predictability of task execution times in a preemptively-scheduled real-time system. Software-based cache partitioning has the advantage over hardware-based partitioning schemes that it

[4] Actually, the Spring system would be better called a pseudo-preemptive system since preemption cannot be provided at any arbitrary point in time.

can be readily applied to existing architectures. The paper focuses on the necessary compiler support to automatically support cache partitioning for real-time tasks. On one hand, transformations on the control flow are needed to support instruction cache partitioning. On the other hand, data reference have to be modified to support data cache partitioning. The partitioning scheme is detailed for different cache architectures and the performance impact is discussed.

The cache partitioning scheme can be readily used in conjunction with existing static cache simulation and worst-case execution time tools. Thus, schedulability analysis can finally be applied to preemptive real-time systems with caches, due to the predictability in execution time gained by cache partitioning.

## References


[AMWH94] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, pages 172–181, December 1994.

[AT95] N. C. Audsley and K. W. Tindell. On priorities in fixed priority scheduling. TR 95-???, Dept. of CS, Uppsala Univ. Sweden, May 1995.

[GL91] Bill O. Gallmeister and Chris Lanier. Early experience with POSIX 1003.4 and POSIX 1003.4a. In *IEEE Symposium on Real-Time Systems*, pages 190–198, December 1991.

[Hil88] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.

[Hil92] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, April 1992.

[Kir89] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Symposium on Real-Time Systems*, pages 229–237, December 1989.

[LL73] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[Mue94] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[NNS91] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable real-time caching in the spring system. In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87, 1991.

[Wol93] A. Wolfe. Software-based cache partitioning for real-time applications. In *Workshop on Responsive Computer Systems*, 1993.