Explicit Reservation of Cache Memory in a Predictable, Preemptive Multitasking Real-time System

JACK WHITHAM, NEIL C. AUDSLEY and ROBERT I. DAVIS, University of York

We describe and evaluate explicit reservation of cache memory to reduce the cache-related preemption delay (CRPD) observed when tasks share a cache in a preemptive multitasking hard real-time system. We demonstrate the approach using measurements obtained from a hardware prototype, and present schedulability analyses for systems that share a cache by explicit reservation. These analyses form the basis for a series of experiments to further evaluate the approach. We find that explicit reservation is most useful for larger task sets with high utilization. Some task sets cannot be scheduled with a conventional cache, but are schedulable with explicit reservation.

1. INTRODUCTION

In this paper, we apply earlier work on *explicit reservation of local memory* [Whitham and Audsley 2012] to the problem of reducing worst-case *cache-related preemption delay* (CRPD) [Busquets-Mataix et al. 1996]. CRPD is observed in preemptive multitasking systems. When a task τ_1 preempts another task τ_2, some of the cache blocks in use by τ_2 may be replaced by task τ_1. In this case, τ_2 will be forced to reload those blocks. The time required to do this is known as CRPD. CRPD is highly relevant in hard real-time systems with shared caches because it increases the running time of preempted tasks, and is a potential cause of deadline misses.

Note that cache misses due to CRPD are distinct from cache misses that occur during normal, non-preempted task execution. These are taken into account by conventional *worst-case execution time* (WCET) analysis. CRPD accounts for *additional* cache misses that may occur in worst-case scenarios when a task is preempted by other tasks.

The state of cache memory can be saved before each preemption, and restored after the preempting task completes. We call this *explicit reservation of cache memory*. Saving and restoring the cache state incurs a time cost just like the usual *implicit* form of CRPD. However, the cache state is restored in a single operation before the preempted task resumes, instead of multiple cache misses while the preempted task is running. This is more efficient, and means that the preempted task experiences less CRPD.

This paper contributes (1) a description of a hardware mechanism for explicit reservation of cache memory, which is prototyped in FPGA hardware; (2) measurements from the prototype that clearly demonstrate the benefits for simple systems; and (3) exact and sufficient schedulability analysis for large task sets using explicit reservation. The results of various experiments are used to compare and contrast the explicit reservation approach with the conventional non-reserved cache approach.

The paper is structured as follows. Related work and details of the problems being solved are given in section 2. Section 3 introduces explicit reservation of cache memory as a solution. Section 4 describes how explicit reservation is implemented in hardware. Section 5 provides an experimental demonstration of the explicit reservation approach as executed on our hardware prototype. Section 6 gives exact and sufficient schedulability analyses for the approach and section 7 uses schedulability analysis to evaluate the approach with large task sets. Section 8 is a discussion of the assumptions we made, with a qualitative evaluation, and section 9 concludes.

This work was supported by EPSRC project TEMPO, number EP/G055548/1 and FP7 project T-CREST, number 288008. Authors' address: Department of Computer Science, University of York. Email: jack@jwhitham.org / neil.audsley@york.ac.uk / rob.davis@york.ac.uk

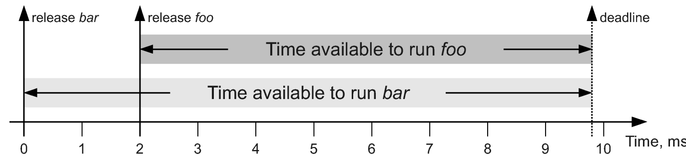

Fig. 1. The real-time system used to illustrate section 2. There are two tasks, *foo* and *bar*. *bar* has 9.8ms to complete its work. *foo* has 7.8ms, as it can only begin 2ms after the beginning of *bar*.

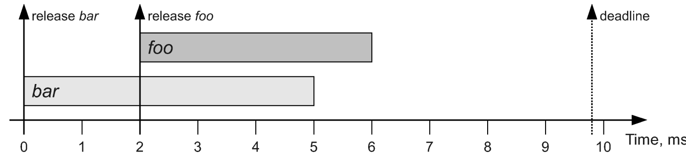

Fig. 2. Example tasks executed on dedicated CPU resources.

2. BACKGROUND

A *hard real-time system* is composed of a hardware platform and one or more hard real-time tasks. Each invocation (*job*) of each task must complete its work before a *deadline* [Burns and Wellings 2009]. The classic real-time systems problem is to prove that a particular task set on a particular system will always meet its deadlines. This property is called *schedulability*. A system is *schedulable* if the deadlines are met.

In order to achieve schedulability, we assume that we can compute the *worst-case execution time* (WCET) of each task, running in isolation without any resource sharing with any other task. The WCET may be computed using various analysis methods [Wilhelm et al. 2008] or by simple measurement in certain special cases, e.g. single path programs [Puschner 2005]. The WCET of a task depends on its functionality, on its implementation (e.g. machine code) and on the platform on which it runs.

2.1. Running Example

We illustrate the remainder of this section by introducing an example task set containing two tasks, named *foo* and *bar*. In the example, the tasks have a shared deadline of 9.8ms, but *foo* has an offset of 2ms after *bar*'s release, because it depends on some input from the environment. These facts can be represented within a timing diagram (Figure 1) which shows the time available to run each task.

2.2. Real-time Systems with Dedicated Resources

Ideally, each task would be given dedicated resources: CPU, memory, and so on. If the tasks must communicate, they can do so through dedicated, non-blocking channels. No task is ever blocked by any other task, so it is straightforward to determine whether tasks meet their deadlines, because only the WCET and deadline are required for the proof. If *foo* and *bar* are executed with dedicated resources, the result is as shown in Figure 2. The two run in parallel and both complete long before the deadline, because the WCET of *foo* is 4ms and the WCET of *bar* is 5ms.

This arrangement is not unknown in real implementations, such as cell phone chipsets, but real-time systems sold in smaller markets often avoid dedicated resources to reduce engineering costs. Usually, an off-the-shelf platform is used, and a single CPU is shared between multiple tasks.

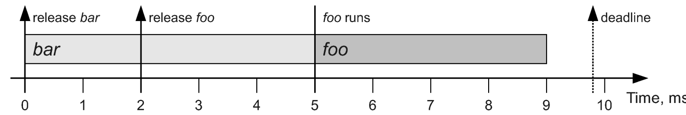

Fig. 3. Example tasks executed by a cyclic executive.

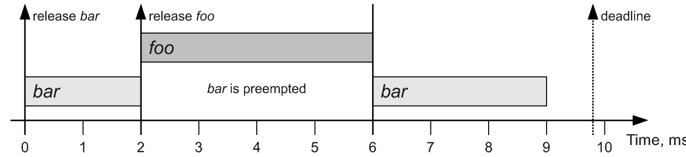

Fig. 4. Example tasks sharing a CPU with preemptive multitasking.

2.3. Real-time Systems Sharing the CPU

Schedulability analysis determines if a CPU can be shared by multiple hard real-time tasks [Burns and Wellings 2009]. The analysis computes the *worst-case response time* (WCRT) of each task. The WCRT of a task is the longest possible interval between the time when that task is released (becomes runnable) and the point where it completes. In a hard real-time system, the task WCRT must be no greater than the task deadline.

The simplest schedulability analysis is possible when jobs run in a predefined order with a *cyclic executive* (Figure 3). However, this is restrictive, because it cannot easily accommodate task sets with widely differing WCETs, offsets and deadlines. It only works for our example because the example is so simple. *bar* has a WCRT of 5ms, completing 5ms after release, and *foo* has a WCRT of 7ms, completing 7ms after release.

The restriction is relaxed by using *preemptive multitasking* as the sharing policy (Figure 4). This involves more complex schedulability analysis, because higher-priority jobs suspend (*preempt*) the execution of lower-priority jobs in response to externally-generated events [Audsley et al. 1993; Tindell et al. 1994]. For our example, *foo* is assigned a higher priority, and when it becomes runnable at 2ms, it preempts *bar*.

The WCRT includes the maximum time that each task may be delayed by higher-priority tasks (*interference* time). It may also include the maximum time that each task may have to wait for uninterruptible sections of lower-priority tasks (*blocking* time), and the overhead of switching between tasks, which is carried out by the *real-time operating system* (RTOS). For Figure 4, we assume that there is no blocking and context-switching is free, so *bar* has a WCRT of 9ms and *foo* has a WCRT of 4ms.

2.4. Real-time Systems Sharing the Cache

The CPU is not the only resource that is shared by all tasks. Today, caches are ubiquitous. A cache is a small CPU-local memory that stores recently-accessed code and data [Hennessy and Patterson 2006]. When a task accesses memory, the cache is checked first. If the required information is already in cache (a *cache hit*), the access completes immediately. If it is not in cache, then a *cache miss* occurs, and the information is fetched from external memory. Caches are highly beneficial, greatly reducing the WCET of a task in comparison to its WCET if executed directly from external memory, provided they are well-designed [Heckmann et al. 2003; Wilhelm et al. 2009].

The cache must be considered during schedulability analysis if one task can cause cache misses within another. This can easily occur when two tasks use the same storage space within the cache. In Figures 2-4, the two tasks were given independent caches of identical size. When the two share a cache and CPU, the results are as shown in Figure 5. There is an increase in the execution time of *bar* because *foo* reused *bar*'s

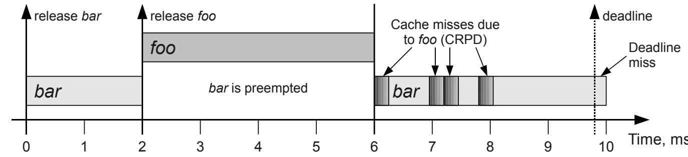

Fig. 5. Example tasks sharing a CPU and cache. *foo* evicts the cache blocks used by *bar*, creating a *cache-related preemption delay* (CRPD) of 1ms. This delay is sufficient to cause *bar* to miss the deadline.

cache blocks, forcing *bar* to fetch the blocks again after resuming. This delay (total 1ms) is the *cache-related preemption delay* (CRPD) [Busquets-Mataix et al. 1996].

The CRPD cost can be bounded by CRPD analysis. The crudest approach to CRPD analysis is to assume that every preemption causes a complete flush of the entire cache; a safe but often imprecise assumption. Successively more precise estimates have been achieved by researchers [Busquets-Mataix et al. 1996; Lee et al. 1998; Staschulat et al. 2005; Tan and Mooney 2007; Altmeyer and Maiza 2010; Altmeyer et al. 2011; 2012]. Estimates are generated by examining which cache blocks are definitely reused by lower-priority tasks (*useful* cache blocks) and which cache blocks are accessed by higher-priority tasks (*evicted* cache blocks).

2.5. Reducing CRPD

Sharing a cache reduces the available utilization of a real-time system [Altmeyer and Maiza 2010]. CRPD may increase the WCRT of any task that is preempted. In some cases, a high-utilization task set may not be schedulable because of CRPD. Therefore, it is worth considering more efficient ways to share the cache which incur a lower CRPD and allow higher-utilization task sets to be executed.

The impact of CRPD may be reduced in at least four ways, none of which are mutually exclusive. Firstly, we may improve CRPD analysis (section 2.6), which does not actually reduce CRPD, but does reduce task WCRTs as determined by schedulability analysis, which is equally important when demonstrating the system's schedulability.

Secondly, we may use static cache partitioning to give each task a dedicated area of cache (section 2.7). This means that there will be no CRPD. We may also choose the memory addresses of the code and data used by a task to reduce CRPD (section 2.8).

Finally, we may reduce the cost of reloading the useful cache blocks by pipelining the cache fill operations (section 2.9). This means that there is still some CRPD, but the worst-case cost is smaller.

2.6. Reducing CRPD by Better Analysis

In previous work [Altmeyer et al. 2011; 2012], the approach taken improved the quality of CRPD analysis rather than reducing CRPD itself. CRPD analysis is potentially *pessimistic*, i.e. the true CRPD is overestimated. To carry out CRPD analysis, we look at pairs of tasks within the task set, where one task may preempt the other. We determine the *cache footprint* of the preempting task. This is the set of cache blocks that the preempting task might evict: the *evicting cache blocks* or ECBs. We also determine the set of cache blocks that will be reused by the preempted task (the *useful cache blocks* or UCBs) [Altmeyer and Maiza 2010].

The earliest CRPD analyses used only ECB sets to determine a pessimistic upper bound on the CRPD [Busquets-Mataix et al. 1996]. Subsequent analyses introduced UCB sets to the analysis [Tan and Mooney 2007; Altmeyer and Maiza 2010], which can be used in different ways (e.g. ECB-Union, UCB-Union) to produce a more accurate CRPD estimate [Altmeyer et al. 2011]. Recent analyses reduce the pessimism in

these approaches in terms of the way they account for nested preemptions, but do not eliminate it entirely [Altmeyer et al. 2012].

To take advantage of this method of reducing the effective CRPD, we would need to find and eliminate a source of overestimation remaining in the most recent CRPD analysis research.

2.7. Reducing CRPD by Cache Locking and Partitioning

We can reduce CRPD to zero by preventing any task causing cache misses in another. This may be achieved by *cache locking* [Vera et al. 2007].

A locked cache is not updated by cache misses during application execution. Instead, it is explicitly filled by the OS [Liu et al. 2012]. Liu et al. classify locking techniques as *static* (filled once at boot time), *semi-dynamic* (filled at each context switch) and *dynamic* (filled at each context switch and at predetermined reload points during each task). The static approach grants a fixed area of cache space to each task; the dynamic approaches allow each task to use different areas of cache space according to their needs. Algorithms have been specified to determine cache allocations to minimize task WCETs [Arnaud and Puaut 2006; Falk et al. 2007] and to maximize schedulability of all tasks [Campoy et al. 2002; Liu et al. 2012]. The static approach prevents all CRPD, while the dynamic approaches replace it with the cache fill penalty incurred during each context switch.

Cache partitioning is similar to locking in that each task gets a preallocated area of cache space and cannot update any other part. However, the preallocated area can be updated by cache misses [Mueller 1995]. Therefore, there is no need to add reload points to a task in order to use the cache efficiently. CRPD applies only if there is a shared area of cache [Kirk 1989].

Static cache locking and partitioning are only effective when every task can be given a cache budget that is sufficient to meet its needs. The amount of available cache space has a major impact on WCET when the available space is insufficient for the task's working set, because the number of cache misses is dramatically increased [Hill and Smith 1989]. In some cases, the task effectively runs from external memory, because no cache blocks can be reused. This is very slow.

For the purposes of our example, we may assume that any reduction in the cache space available to *foo* and *bar* will cause such a slowdown, and deadlines will not be met, even though the CRPD will be zero.

Dynamic cache locking is potentially better, but because it involves partitioning a task by adding reload points, it relies on partitioning algorithms to minimize the WCET [Liu et al. 2012; Arnaud and Puaut 2006]. As yet, there is no evidence that such algorithms produce results that are competitive with cache when state-of-the-art WCET analysis is used.

To take advantage of static cache partitioning or locking as a method of reducing the CRPD, we would need a task set (or a task subset) that could fit entirely in cache without significantly increasing the WCET of any task.

2.8. Reducing CRPD by Optimizing Task Memory Layout

ECB and UCB sets depend on the code and data addresses used by each task. Code and static data addresses are determined by the linker in the final stage of compilation; dynamic data addresses are determined at runtime but may still be constrained by a suitable memory allocator [Herter et al. 2011].

It is possible to reduce CRPD by choosing the addresses making up the task's *memory layout* so as to minimize the overlap determined by CRPD analysis. Recent work used simulated annealing to search for task layouts that minimized CRPD [Lunniss

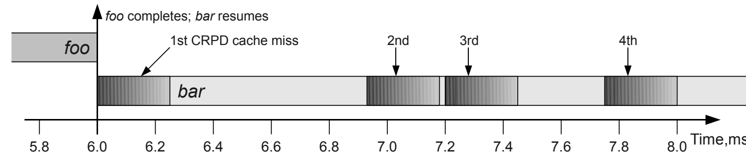

Fig. 6. Detail of Figure 5 showing that the CRPD of 1ms is actually composed of 4 separate cache misses, each occurring at sporadic intervals and each requiring 250μs to complete.

Fig. 7. If the addresses to be fetched are known in advance, and the memory subsystem is pipelined, the total time to restore the cache state of *bar* is 490μs, less than half the combined cost of individual cache misses in Figure 6.

et al. 2012]. The most significant benefits come from changing the order that the tasks are placed in memory. For n tasks, there are $n!$ possible orderings.

To take advantage of this method of reducing the effective CRPD, we would need to search a large number of possible task layouts.

2.9. Reducing CRPD by Pipelining Cache Fills

Figure 5 shows the 1ms CRPD imposed on *bar* by *foo*. We now look more closely at the events shortly after the completion of *foo* (Figure 6). The cache misses do not occur together. They are sporadic events triggered by *bar*'s execution, with total time 1ms.

The total time can be reduced if these cache miss operations are carried out together, because each subsequent cache fill can be started while the previous one is still running. A pipelined bus and pipelined memory make this possible, provided that the addresses to be fetched are known in advance. Each miss requires a total of 250μs, but the second, third etc. can be initiated after 80μs. Figure 7 shows what happens when pipelining is used.

To take advantage of this method of reducing the CRPD, we would require a pipelined memory subsystem, and perfect knowledge of the addresses of the cache blocks to be fetched from external memory by the preempted task as a result of CRPD. However, both of these are available. Pipelined access to memory is now a common hardware feature, and the cache blocks to be fetched are the cache blocks in use at the time the task was preempted.

3. EXPLICIT RESERVATION OF CACHE MEMORY

In this paper, we apply an *explicit reservation* approach to reduce CRPD. When a task is preempted, the state of the cache is saved on a stack. When the task continues execution, the state of the cache is restored from that stack. The restoration process takes advantage of pipelined access to memory and therefore completes earlier (in the worst case) than the series of separate cache misses which would be required with a conventional *non-reserved* cache.

3.1. Overview of the Approach

Each task τ_i has a *cache budget* S_i, which is a number of cache blocks ranging from 1 to the cache size S^{cmax}. Cache blocks are of equal size.

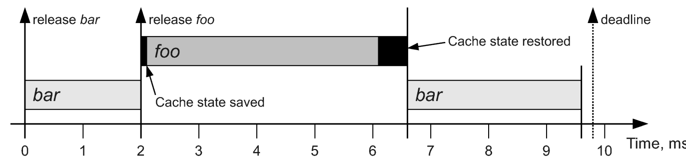

Fig. 8. Explicit memory reservation example. *foo* has a cache budget of 4 blocks. 100μs are spent saving the state of these blocks just after preemption. 490μs are spent restoring that state after completion. The result is a 410μs reduction in CRPD in comparison to sharing a cache implicitly (Figure 5). The deadline is now met by both tasks.

On each context switch to a new task τ_i, cache space is *explicitly reserved* by the RTOS to meet the cache budget for τ_i. This means saving the state of S_i cache blocks on a stack. The remaining $S^{\text{cmax}} - S_i$ blocks are locked so that they cannot be updated.

On completion of τ_i, the cache space is restored to its previous state. Each of the S_i blocks is refilled with its earlier contents, taken from the stack. The previous lock state of the S_i blocks is also restored.

The state of a cache block is (1) its lock state (locked/unlocked) and (2) the address in external memory that it was loaded from.

Figure 8 illustrates the idea when applied to our example task set. *bar* and *foo* are each given a cache budget of 4 blocks.

3.2. Related Approaches

As the restore step loads S_i blocks in quick succession, the memory operations can be pipelined. Each cache fill operation may be requested before the previous one completes. This brings similar advantages to the well-known technique of *cache prefetching* [Hennessy and Patterson 2006]. This is typically driven by heuristics, such as fetching cache blocks sequentially, though special CPU instructions may be used to provide hints to the prefetching process [Cepeda 2009]. Indeed, explicit reservation of cache memory could be regarded as a special case of cache prefetching, and cache prefetching instructions could be used to implement the restore process.

But cache prefetching is very different to explicit reservation of cache memory. Prefetching is not intended to improve the worst case for real-time systems. It is not aware of context switches. Being an heuristic process, it may make mistakes which could increase task WCETs or WCRTs, such as evicting useful cache blocks under rare circumstances. Conventional prefetching suffers from a potential to misuse resources, slowing a system down [Tse and Smith 1998].

The restore process does not rely on heuristics to determine likely future cache states, because it conservatively restores the cache state exactly as it was at preemption. It cannot introduce any CRPD beyond the time taken to save and restore. There will be no additional cache misses as a result of preemption.

3.3. Weaknesses of the Approach

The main disadvantage of explicit reservation of cache is that any preemption by task τ_i always imposes the same CRPD regardless of the preempted task τ_j. This could be regarded as a strength, making execution more predictable, but it is possible that this CRPD may be greater than the CRPD for a conventional cache.

In the conventional *non-reserved* arrangement, CRPD depends on both the preempted and the preempting task, and may be zero if the two have no cache blocks in common. Furthermore, if the preempted task does not reuse a block in cache, it is never loaded at all. Therefore, the technique may increase a task's WCRT by pes-

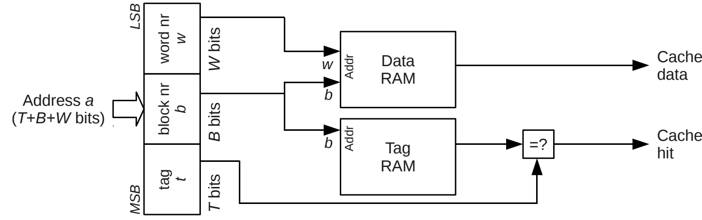

Fig. 9. Direct-mapped Cache [Xilinx 2008].

simistically assuming that all the cache blocks need to be restored, whether evicted or not, and whether reused or not.

Furthermore, new hardware is required to implement an appropriate *explicitly reservable* cache. There is no practical way to work around this requirement using software, because of the need to save the state of the cache upon preemption.

Finally, any cache budget in the range $1 \leq S_i \leq S^{\text{cmax}}$ can be implemented in principle, but only integer powers of two can be supported efficiently, because other values would require a hardware division unit within each cache. This would introduce an unacceptably high access latency as well as significantly increasing hardware costs.

4. EXPLICITLY RESERVABLE CACHE HARDWARE

In this section we describe cache hardware that supports the *explicit reservation protocol*. This protocol consists of the save and restore operations to be used at each context switch, plus a locking protocol for use with low-priority tasks. The protocol cannot be usefully implemented in software as the save/restore overhead must be minimized.

The cache is a direct-mapped, write-through data cache. This is a very simple type of cache design [Hennessy and Patterson 2006], but it can be extremely effective [Hill 1988]. Direct-mapped caches are often used in simple embedded systems due to their relatively low area and power costs [Xilinx 2008].

4.1. Conventional Direct-mapped Cache

A conventional direct-mapped cache design forms the basis for the implementation of the explicit reservation protocol. Like any direct-mapped cache, two RAMs are used: a data RAM, and a tag RAM (Figure 9).

Figure 9 shows some of the internal components of a direct-mapped cache. The CPU sends an address a to the cache, which consists of $T + B + W$ bits: a tag t (size T bits), a block number b (size B bits) and a word number w (size W bits). The data RAM receives address (b, w) and the tag RAM receives address b. On the next clock cycle, the output of the tag RAM will be tag[b] - the current value of the tag for block b.

If tag[b] $= t$, then the cache access is a hit. A signal is sent to the CPU, along with the requested data (data[b, w]). If tag[b] $\neq t$, then the cache access is a miss. Additional hardware, not shown in Figure 9, sends the address a to the external memory. After a delay (the *cache miss time*) the external memory responds with the requested data. tag[b] is updated to t, and data[$b, 0$] through data[$b, 2^W - 1$] are filled.

The cache is *no-write allocating*, i.e. cache misses cannot be generated by store operations. However, store operations may update (*write-through*) cache blocks if the accessed address is already in cache.

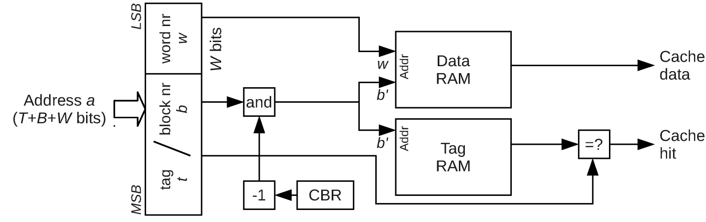

Fig. 10. Direct-mapped Cache with support for Explicit Reservation.

4.2. Direct-mapped Cache with Explicit Reservation

To adapt the basic direct-mapped cache design for the explicit reservation protocol, we make two modifications to the subsystem that decodes address a as shown in Figure 10. t and b are combined into a single bitfield (t, b) to be used as the tag input.

Firstly, a *cache budget register* (CBR) is added. The CBR holds the value of the cache budget S_i for the current task τ_i. The CBR value must be an integer power of two in the range $[1, S^{\text{cmax}}]$. The initial state of the CBR is S^{cmax}.

Secondly, a logical function is added to decode the bitfield (t, b) into the effective block address b'. The decoding operation is efficiently implemented in hardware as it involves just one fast operation (logical AND):

$$b' = (t, b) \text{ AND } (\text{CBR} - 1) \tag{1}$$

This makes data RAM with a block address outside [0, CBR -1] inaccessible.

Two further additions are the *save/restore stack* (SRS) and *control logic*. The SRS stores the tag values used by lower-priority tasks while higher-priority tasks are running. The control logic implements a state machine to control cache filling and a *control port* to allow software running on the CPU to initiate operations, such as save and restore. The control port receives instructions in the form of operation codes (opcodes).

4.3. Opcodes

The explicit reservation protocol has two opcodes, which can be sent by software to the cache control logic. The *Save* opcode has an operand S_i which specifies the cache budget for the next task. It initiates the following operation sequence:

— Push the current CBR value onto the SRS.
— For i from CBR -1 to 0: push tag[i] (the value of tag i) onto the SRS.
— Set the CBR to S_i.

The *Restore* opcode undoes the effects of the previous *Save* opcode. It initiates the following operation sequence:

— For i from 0 to CBR -1: pop tag[i] from the SRS.
— For i from 0 to CBR -1: fetch cache block i from external memory.
— Pop the new CBR value from the SRS.

4.4. Example

Suppose *foo* preempts *bar* as in Figure 8. The sequence of events is shown in Figure 11. The sequence includes the context switch time imposed by the OS (CS^{to}, CS^{from}) and the *Save* and *Restore* operations. The result of the *Restore* operation is that the cache returns to the state it was in at the beginning of the *Save* operation.

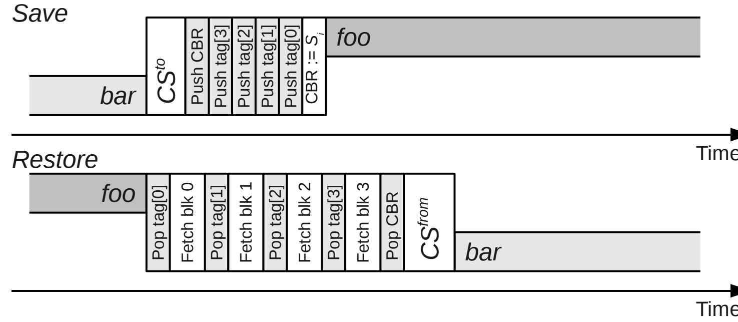

Fig. 11. Sequences of sub-operations executed for *Save* and *Restore* when *foo* preempts *bar* in Figure 8. For clarity, the *Restore* operations are shown here without pipelining: in reality, they overlap.

4.5. Prototype Implementation

In order to evaluate the explicitly reservable cache, we need realistic data about the cost of various operations, such as the latency of cache misses and the time taken to *Restore* a specific number of blocks. Tiny measurement errors in this data can lead to seriously flawed results, making explicit reservation seem better (or worse) than it should be. This is a very high risk when deriving results from simulators, which are rarely perfectly accurate. To minimize this risk, we implemented the explicit reservation cache using *field-programmable gate array* (FPGA) hardware.

Our implementation of the explicitly-reservable cache is built upon a Xilinx Spartan-6 FPGA on the Digilent Atlys prototyping board [Digilent 2013]. The cache is built as a component with three external bus connections:

— A *local memory bus* (LMB) for connection to a CPU such as Microblaze [Xilinx 2008]. The CPU uses this bus to access the cache. Addresses sent via this bus are decoded into the (t, b, w) tuple (section 4.2).
— A *Fast Simplex Link* (FSL) bus for sending opcodes to the cache (section 4.3).
— An *Advanced Extensible Interface* (AXI) bus connection to the memory controller [Xilinx 2012]. The AXI bus provides high-speed pipelined access to the DDR2 RAM on the Atlys prototyping board, via a Xilinx memory controller IP core.

Our hardware design is an embedded system capable of running a preemptive multitasking RTOS (Figure 12). The CPU is a Xilinx Microblaze IP core [Xilinx 2008]: the CPU is connected to two copies of the explicitly reservable cache, one for instructions and another for data. The hardware design also features two timers, one for measurements and another for generating interrupts. There is a small scratchpad memory (SPM) which is used to store the RTOS kernel.

Each cache is implemented as shown in Figure 10. The block size is 32 bytes ($W = 3$), and the valid range of B is 1 to 6, so $S_i \in \{1, 2, 4, 8, 16, 32, 64\}$. This gives a maximum size of 2048 bytes (64 blocks) for the instruction cache and data cache, and a minimum size of 32 bytes (1 block).

As an indication of area overheads, an instruction/data pair of 32kbyte direct-mapped caches requires 330 FPGA slices. Replace these with explicitly reservable caches of the same size, and the area requirement is 710 slices, a two-fold increase. This is primarily because of the size of the save/restore control logic. It includes debugging features, and we have not made any attempt to reduce the size.

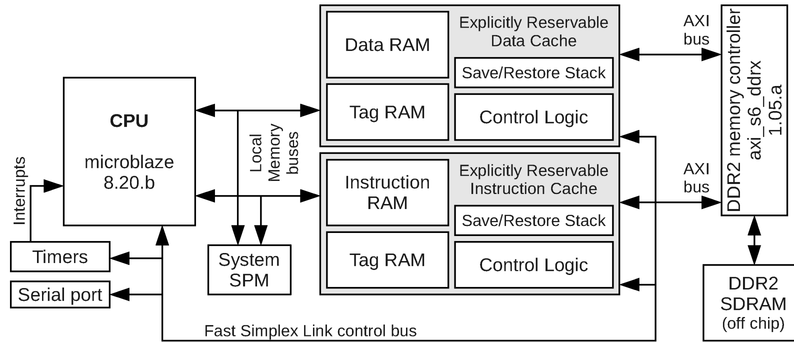

Fig. 12. Components and interconnections in our FPGA design.

4.6. Applications

Our experiments are based on the integer benchmarks from the MRTC collection [Gustafsson et al. 2010]. The benchmark collection contains 35 programs, all written in C, all single path, and all intended as examples of real-time tasks.

We used the Microblaze C compiler, which is based on GCC, to compile each benchmark. Our RTOS and our hardware does not currently support floating-point operations, so we omitted the 11 MRTC programs that use floating-point[1] but kept the remaining 24 programs that use only integer operations.

We modified each source file to rename the `main()` function to include the benchmark name (e.g. `matmult_main()`). This allows the benchmark to be invoked as an RTOS task rather than a standalone program.

Some benchmarks have different behavior if invoked more than once. For instance, the *crc* benchmark generates a lookup table when invoked for the first time. We wanted the same (worst-case) behavior on each invocation, so we added code to reset the "initialized" flag on each execution.

Some benchmarks have symbol names that conflict when the benchmarks are linked together. We modified the code so that these were declared `static`, making them local to a single C source file.

4.7. Task Timings

In subsequent experiments, a task τ_i is created from a benchmark program. For each τ_i, we require a WCET C_i^{nr} for execution with a conventional non-reserved cache, and a WCET C_i^{er} with explicit reservation.

We determined the WCET C_i^{nr} for each task τ_i by running the task on our FPGA platform without preemption, starting in a "cold" state in which the cache is completely empty. This produces a valid approximation for the WCET because the benchmarks are single path. The maximum execution time was measured across 100 runs.

C_i^{er} was determined in a similar way, but because this is the WCET with explicit reservation, it depends on the cache budget allocated to the task. Therefore, we searched all possible cache budgets in order to find the arrangement that minimized $C_i^{\text{er}} + C_i^{\text{save}} + C_i^{\text{restore}}$, being the total running time for task τ_i. Two cache budgets S^I and S^D apply to each task, because we have both an instruction cache and a data cache (Figure 12). There are 49 configurations to be tested, because S^D and S^I can each take 7 possible values (1, 2, 4, 8, 16, 32, 64) and there are two caches ($7^2 = 49$).

[1] *fft1*, *lms*, *ludcmp*, *minver*, *qsort-exam*, *qurt*, *select*, *sqrt*, *statemate*, *st* and *ud*.

Table I. Benchmarks used in this paper, with WCET data

Task τ_i	C_i^{nr} (ns)	C_i^{er} (ns)	S_i^I	S_i^D	C_i^{save} (ns)	C_i^{restore} (ns)
adpcm	3565897	3565817	64	64	1813	17612
binarysearch	6839	6666	8	1	226	1746
bsort100	1098612	1098332	8	16	426	3746
cnt	120170	120876	8	1	226	1746
compress	170115	170529	16	32	746	6946
cover	53651	54865	8	1	226	1746
crc	499160	504574	16	16	533	4813
duff	24986	25479	8	1	226	1746
edn	907577	908017	32	64	1386	13346
expint	118503	118903	4	1	173	1213
fac	5799	5626	4	1	173	1213
fdct	45358	46425	16	16	533	4813
fibcall	7293	7119	4	1	173	1213
fir	55491	55891	8	8	319	2679
insertsort	20572	20506	4	4	213	1613
janne	7466	7399	4	1	173	1213
jfdctint	91717	92557	16	32	746	6946
lcdnum	6226	6053	8	1	226	1746
matmult	2913393	2912593	16	64	1173	11213
ndes	971295	971229	64	64	1813	17612
ns	156849	156676	8	1	226	1746
nsichneu	704809	704609	1	8	226	1746
prime	325258	325085	8	4	266	2146
recursion	45465	45358	8	8	319	2679

Note: C_i^{nr} is the *non-reserved* cache WCET, assuming the entire cache is shared between all tasks. C_i^{er} is the WCET assuming that a cache budget is imposed. Values are accurate to the nearest clock cycle (13.3ns).

We executed each task on our FPGA platform with each possible cache budget configuration, capturing C_i^{er} for each, again taking the maximum across 100 runs.

Table I gives C_i^{nr}, C_i^{er}, C_i^{save} and C_i^{restore} for each task.

4.8. System Timings

Our schedulability calculations require upper-bound timings for various important operations, which we obtained either from architectural parameters, or by running test applications within our RTOS. These timings are:

— A clock cycle – 13.3ns (75MHz clock).
— A cache miss – 547ns.
— Context switch *to* a task – $CS^{\text{to}} = 14\mu$s.
— Context switch *from* a task – $CS^{\text{from}} = 14\mu$s.

4.9. Software Downloads

The source code and hardware designs referenced in this paper may be downloaded from `http://www.cs.york.ac.uk/rts/rtslab/`. They can be used immediately on a Digilent Atlys FPGA board, and can be adapted to other hardware using the Bluespec System Verilog compiler [Bluespec 2013].

5. EXPLICIT RESERVATION DEMONSTRATION

We now present experimental evidence to show the benefits of the explicitly reservable cache. In this section, all data is obtained by measurements from our FPGA prototype.

Being intended for use within hard real-time systems, the explicit reservation approach is intended to provide a benefit to the worst case only. This worst case may only

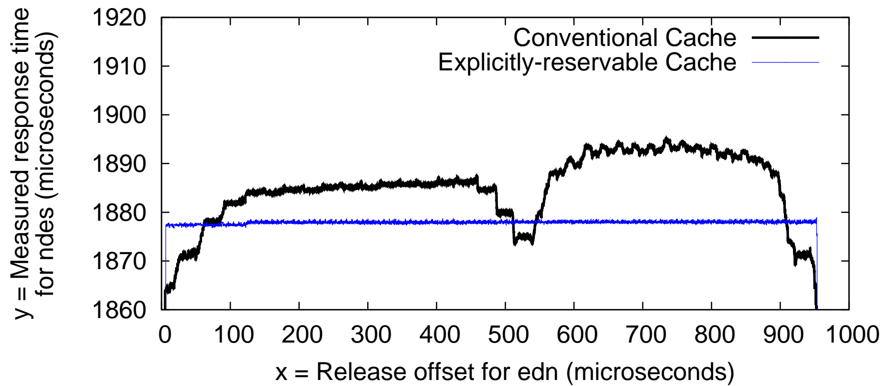

Fig. 13. The measured response time y for *ndes* when preempted by *edn* at time x is highly variable using a conventional cache, but almost constant using an explicitly-reservable cache.

be observable through WCRT analysis, and may not be observable through execution and measurement.

However, it is possible to demonstrate the benefits of explicit reservation using just two tasks. Suppose that we pick *ndes* and *edn* from Table I and assign *edn* a higher priority than *ndes*.

We now set the period of both tasks to a large value, so that the two never miss deadlines. The release offset of *ndes* is set to zero, so *ndes* begins executing immediately.

Let the release offset of *edn* be a variable x, so *edn* will preempt *ndes* at time x. Furthermore, let y be the measured response time for the first execution of *ndes*, i.e. *ndes* completes at time y.

Figure 13 shows how x and y are related, for a conventional cache and an explicitly-reservable cache. The *ndes* task is preempted once when x is between 6.2μs and 953μs. For $x < 6.2\mu$s, *edn* runs before *ndes* starts and there is no preemption. For $x > 953\mu$s, *edn* runs after *ndes* finishes.

5.1. Discussion

With a conventional cache, the response time of *ndes* is highly dependent on x, the point at which *ndes* is preempted (Figure 13).

If preemption occurs at an early point during execution (e.g. $x < 62\mu$s) then the conventional CRPD cost is relatively low, yielding a measured response time $y < 1877\mu$s. This is because relatively few cache lines used by *ndes* are evicted by *edn*. However, if preemption occurs at a later time, then many of the cache lines in use by *ndes* will be evicted. The measured response time increases accordingly: $y > 1895\mu$s. *ndes* takes longer to run because it must reload cache lines that were evicted by *edn*.

With an explicitly-reservable cache, the measured response time is dependent only upon the number of preemptions, and not upon the offset x. This is observed in Figure 13 where the data for the explicitly-reservable cache appears as a flat line (with some noise due to jitter from memory refresh cycles and the interrupt timer). The response time is almost constant: $y \approx 1878\mu$s.

Preemption with the explicitly-reservable cache may be faster or slower than with a conventional cache. In Figure 13, the conventional cache is preferable for $x < 62\mu$s, for $x \approx 522\mu$s, and for $x > 909\mu$s.

In a real-time system, there is no choice about when a task is preempted, as the preempting task may be triggered by an external event. If deadlines are tight, and the preemption occurs at a “bad” time, then the CRPD may be enough to cause a deadline

miss in the preempted task. This is why the CRPD must always be an upper bound, and why techniques to reduce this upper bound are worthwhile.

To better understand the properties of explicitly-reservable caches, we make use of schedulability analysis to identify and evaluate more complex versions of worst-case scenarios in which differences in CRPD may cause deadline misses. These worst-case scenarios are likely to be extremely difficult to reproduce in our prototype system since they rely on unlikely combinations of events, so analysis is used for the comparisons.

6. SCHEDULABILITY ANALYSIS WITH TASK-DEPENDENT CONTEXT SWITCH TIME

In this section, we describe exact and sufficient schedulability analyses for systems using explicitly-reservable caches. The analysis determines *worst-case response times* (WCRTs) for tasks executed with an explicitly-reservable cache.

In the classical model of a preemptive real-time system, context-switch costs are constant [Burns and Wellings 2009]. Context switches are not interruptible and can block execution of other tasks. But with explicit reservation of cache memory, context switches include C_i^{save} and C_i^{restore}: the time taken to save and restore the memory state. These are dependent on the number of cache blocks used by tasks (section 4.8).

6.1. System Model, Terminology and Notation

We consider the fixed priority preemptive scheduling of a set of sporadic tasks on a single CPU. Each task set comprises a static set of n tasks $\tau_1, \tau_2, ..., \tau_n$, where n is a positive integer. We assume that task τ_i has priority i, hence τ_1 has the highest priority, and τ_n the lowest.

We use the notation $\text{hp}(i)$ (and $\text{lp}(i)$) to mean the set of tasks with priorities higher than (lower than) i, and the notation $\text{hep}(i)$ (and $\text{lep}(i)$) to mean the set of tasks with priorities higher than or equal to (lower than or equal to) i.

Each task τ_i is characterized by its bounded worst-case execution time C_i, minimum inter-arrival time or period T_i, and deadline D_i (relative to the release time). Each task τ_i therefore gives rise to a potentially infinite sequence of invocations (or jobs), each of which has an execution time upper bounded by C_i, an arrival time at least T_i after the arrival of its previous job, and an absolute deadline that is D_i after its arrival. Task utilization U_i is defined as $U_i = \frac{C_i}{T_i}$.

In an *implicit-deadline* task set, all tasks have $D_i = T_i$. In a *constrained-deadline* task set, all tasks have $D_i \leq T_i$, while in an *arbitrary-deadline* task set, task deadlines are independent of their periods.

The tasks may block each other from executing by accessing mutually exclusive shared resources. The blocking factor at priority level i, due to mutually exclusive access to resources, is given by B_i and is equal to the length of the longest resource access by a task of lower priority than i to a resource that is shared with task τ_i or a task of higher priority.

In order to model C_i^{save} and C_i^{restore}, we assume that there are three phases of execution related to each task τ_i. Before task τ_i can begin normal execution, we assume that a non-preemptable phase of execution of maximum length C_i^{pre} must first be completed. C_i^{pre} represents the cost of saving the cache state of any preempted task C_i^{save}, plus any other context-switching costs needed to begin executing τ_i.

Similarly, once task τ_i has completed its normal execution C_i (during which time it may be preempted), then a non-preemptable phase of execution of maximum length C_i^{post} must be completed. C_i^{post} represents the time required to restore the context of any preempted task C_i^{restore}, plus any other context-switching costs needed to switch back to the preempted task.

The *worst-case response time* (WCRT) R_i of a task τ_i is given by the longest possible time from release of the task until it completes its normal execution. Thus task τ_i is schedulable if and only if $R_i \leq D_i$, and a task set is schedulable if and only if $\forall i. R_i \leq D_i$.

The following assumptions are made about the behavior of the tasks: The arrival times of the tasks are independent and unknown a priori; hence the tasks may share a common release time. Each task is released (i.e. becomes ready to execute) as soon as it arrives. The tasks do not voluntarily suspend themselves.

6.2. Exact Schedulability Test

In this section, we extend the schedulability analysis given by [Tindell et al. 1994] for fixed priority preemptive scheduling of tasks with arbitrary deadlines to account for the initial and final non-preemptive phases in our task model. We provide an exact test valid for tasks with arbitrary deadlines.

The worst-case scenario for task τ_i occurs following a *critical instant* where τ_i is released simultaneously with all higher priority tasks[2], and subsequent releases of task τ_i and higher priority tasks then occur after the minimum permitted time intervals. Further, immediately prior to the start of this busy period, a lower priority task either locks a resource shared with task τ_i or a higher priority task, or enters a non-preemptive phase of execution. The length L_i of the longest priority level-i busy period can be found via the following fixed point iteration:

$$L_i^{m+1} = B_i^{CS} + \sum_{\forall j \in \text{ hep}(i)} \left\lceil \frac{L_i^m}{T_j} \right\rceil (C_j^{\text{pre}} + C_j + C_j^{\text{post}}) \tag{2}$$

where B_i^{CS} is given by:

$$B_i^{CS} = \max(B_i, \max_{\forall k \in \text{ lp}(i)} (C_k^{\text{pre}}, C_k^{\text{post}})) \tag{3}$$

Iteration starts with an initial value guaranteed to be no larger than the minimum solution, for example $L_i^0 = C_i$, and ends when $L_i^{m+1} = L_i^m$. Convergence is guaranteed provided that the adjusted task set utilization $U^* = \sum_{\forall i} \frac{1}{T_i}(C_i^{\text{pre}} + C_i + C_i^{\text{post}})$ does not exceed 1. The number of jobs Q_i of task τ_i in the busy period is given by:

$$Q_i = \left\lceil \frac{L_i}{T_i} \right\rceil \tag{4}$$

In general it is necessary to compute the response times of all jobs of a task τ_i within the longest priority level-i busy period in order to determine the task's worst-case response time. The *completion time* $W_{i,q}$ of the qth job (where $q = 0$ is the first job) of task τ_i, with respect to the start of the busy period, is given by the following fixed point iteration:

$$\begin{aligned} w_{i,q}^{m+1} = & B_i^{CS} + q(C_i^{\text{pre}} + C_i + C_i^{\text{post}}) + C_i^{\text{pre}} + C_i + \\ & \sum_{\forall j \in \text{ hp}(i)} \left\lceil \frac{w_{i,q}^m}{T_j} \right\rceil (C_j^{\text{pre}} + C_j + C_j^{\text{post}}) \end{aligned} \tag{5}$$

Note that we are interested in the time at which the qth job completes its *normal execution*. Therefore we include $q(C_i^{\text{pre}} + C_i + C_i^{\text{post}})$ for the execution of q previous

[2]This is the worst case because the C_i^{pre} and C_i^{post} costs are incurred regardless of which tasks are preempted (including no tasks).

jobs of task τ_i in the busy period and similarly for interference from higher priority tasks, but only $C_i^{\textbf{pre}} + C_i$ for the qth job of task τ_i.

Iteration starts with an initial value $w_{i,q}^0$, for example:

$$w_{i,q}^0 = B_i^{CS} + q(C_i^{\textbf{pre}} + C_i + C_i^{\textbf{post}}) + C_i^{\textbf{pre}} + C_i \tag{6}$$

and ends when either $w_{i,q}^{m+1} = w_{i,q}^m$, in which case $W_{i,q} = w_{i,q}^{m+1}$, or when $w_{i,q}^{m+1} - qT_i > D_i$, in which case job q (and hence task τ_i) is unschedulable. We note that the number of iterations can be reduced by the use of appropriate initial values [Davis et al. 2008].

To find the WCRT of task τ_i, completion times $W_{i,q}$ need to be calculated for jobs $q = 0, 1, 2, ..., Q_i - 1$. The WCRT of task τ_i is then given by:

$$R_i = \max_{q \in [0, Q_i - 1]} (W_{i,q} - qT_i) \tag{7}$$

Task τ_i is schedulable provided that $R_i \leq D_i$. We note that for our task model, this form of analysis needs to be used even for constrained-deadline task sets. This is because the non-preemptive phase $C_i^{\textbf{post}}$ after the completion of the normal execution of a job of task τ_i may cause push-through blocking on the next job of task τ_i, even if the job finishes before its deadline $D_i \leq T_i$. This occurs when the busy period at priority level i extends beyond the release of the next job. This has similarities to fixed priority scheduling with deferred preemption [Bril et al. 2009].

6.3. Sufficient Schedulability Test

We now provide a sufficient schedulability test for task sets with constrained deadlines ($D_i \leq T_i$) based on the approach used in section 3.4 of [Davis et al. 2007].

The response time of the first job of task τ_i in the busy period is given by:

$$w_{i,0}^{m+1} = B_i^{CS} + C_i^{\textbf{pre}} + C_i + \sum_{\forall j \in \textbf{hp}(i)} \left\lceil \frac{w_{i,0}^m}{T_j} \right\rceil (C_j^{\textbf{pre}} + C_j + C_j^{\textbf{post}}) \tag{8}$$

Assuming that this job completes its normal execution at or before its deadline, and hence before the end of its period, then we have two scenarios to consider.

(1) If the priority level-i busy period ends by the time the next job of task τ_i is released, then (8) gives the correct WCRT.
(2) Alternatively, if the busy period does not end by the time the next job of task τ_i is released, then we must consider the response times of the next and subsequent jobs of task τ_i.

First, we derive an upper bound on the maximum length of the interval between the times $f_{i,q}$ and $f_{i,q+1}$ at which two arbitrary but consecutive jobs q and $q+1$ of task τ_i finish their normal execution. We then show that this upper bound is also an upper bound on the response time for job $q+1$, and can therefore be used as the basis for a sufficient schedulability test. We assume that:

(1) all $q+1$ jobs fall within the same busy period,
(2) the first q jobs are schedulable – we will return to this point later.

We observe that at time $f_{i,q}$, when job q finishes, there can be no tasks of higher priority than i that were released prior to $f_{i,q}$ and have any execution time remaining (otherwise they would have preempted task τ_i). Thus, an upper bound on the length of the time interval $[f_{i,q}, f_{i,q+1})$ can be found by making the potentially pessimistic assumption that all higher priority tasks are released just after job q finishes its normal

execution. The smallest solution to (9) thus provides an upper bound on the length of this interval.

$$w_{i,q}^{m+1} = C_i^{\textbf{post}} + C_i^{\textbf{pre}} + C_i + \sum_{\forall j \in \textbf{hp}(i)} \left\lceil \frac{w_{i,q}^m}{T_j} \right\rceil (C_j^{\textbf{pre}} + C_j + C_j^{\textbf{post}}) \tag{9}$$

Here, $C_i^{\textbf{post}}$ is the time taken to execute the non-preemptable phase after job q completes its normal execution and $C_i^{\textbf{pre}}$ is the time to execute the non-preemptable phase prior to the start of the $(q+1)$th job of task τ_i. The summation term represents the interference from higher priority tasks, released during the interval, and executed before the $(q+1)$th job of task τ_i.

Given the assumption that the first q jobs in the busy period are schedulable, and the constraint that $D_i \leq T_i$, then the normal execution of job q must finish before the end of its period, and hence before the $(q+1)$th job is released. This means that the length of the interval $[f_{i,q}, f_{i,q+1})$ and hence also the response time for the $(q+1)$th job is bounded by the solution to (9).

We now return to the assumption that the first q jobs are schedulable. Schedulability of the $q = 0$ job can be determined using (8); while the schedulability of the second and all subsequent jobs within the busy period can be determined, by induction, using (9).

Intuitively, we might say that the second and subsequent jobs of task τ_i are subject to push-through blocking of at most $C_i^{\textbf{post}}$ due to the previous job of the same task.

This result suggests a simple sufficient but not necessary schedulability test, formed by combining (8) and (9) into a single equation:

$$w_i^{m+1} = \max(B_i^{CS}, C_i^{\textbf{post}}) + C_i^{\textbf{pre}} + C_i + \sum_{\forall j \in \textbf{hp}(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil (C_j^{\textbf{pre}} + C_j + C_j^{\textbf{post}}) \tag{10}$$

We observe that the schedulability analysis embodied in (10) equates to assuming that a job of task τ_i can be subject to blocking; either of B_i^{CS} due to lower priority tasks, or $C_i^{\textbf{post}}$ due to the previous job of the same task, but not both. Hence, a sufficient but not necessary definition of WCRT is:

$$R_i = \max(B_i^{CS}, C_i^{\textbf{post}}) + C_i^{\textbf{pre}} + C_i + \sum_{\forall j \in \textbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j^{\textbf{pre}} + C_j + C_j^{\textbf{post}}) \tag{11}$$

7. EXPERIMENTAL EVALUATION

In this section, we evaluate explicit reservation of cache memory using benchmark programs (Table I) by applying schedulability analysis (section 6). The comparisons use timing data captured from our prototype hardware (section 4.8) but due to the difficulty of reproducing the worst-case scenarios that are of interest, they otherwise rely entirely on analysis. Task sets found to be schedulable by experiments in this section *can* be executed on our prototype system, but even if a deadline miss is theoretically possible according to analysis, we may not find it by execution, since analysis deals with theoretical worst cases.

7.1. Generating Task Sets

Our task sets are generated randomly, but based on the benchmarks from Table I. The task set generator parameters are the task set size n and the task set utilization U.

We generate a task set of size n as follows. The first step is to choose n utilization values for the tasks τ_1, τ_2, ..., τ_n. This is done using the UUnifast algorithm [Bini

Table II. The ECB and UCB set sizes for each benchmark

Task τ_i	$\lvert\text{ECB}_i^I\rvert$	$\lvert\text{ECB}_i^D\rvert$	max $\lvert\text{UCB}_i^I\rvert$	max $\lvert\text{UCB}_i^D\rvert$	S_i^I	S_i^D
adpcm	64	35	64	32	64	64
binarysearch	5	3	5	0	8	1
bsort100	7	15	5	14	8	16
cnt	12	18	8	0	8	1
compress	32	35	22	22	16	32
cover	11	25	10	1	8	1
crc	19	22	18	22	16	16
duff	11	5	9	0	8	1
edn	64	45	58	43	32	64
expint	10	1	5	0	4	1
fac	4	2	3	0	4	1
fdct	30	7	26	6	16	16
fibcall	4	1	3	0	4	1
fir	11	10	7	8	8	8
insertsort	8	3	4	3	4	4
janne	5	1	4	0	4	1
jfdctint	30	11	28	10	16	32
lcdnum	5	1	5	0	8	1
matmult	11	64	11	64	16	64
ndes	64	42	60	41	64	64
ns	6	64	5	15	8	1
nsichneu	64	4	0	4	1	8
prime	9	3	7	3	8	4
recursion	6	13	5	8	8	8

and Buttazzo 2005] which generates U_1, U_2, ..., U_n such that the task set utilization $U = \sum_{i\in[1,n]} U_i$.

A benchmark program is then assigned to each one of the n tasks. Every row of Table I has an equal chance of selection for each task τ_i. This also gives the various WCETs: C_i^{nr}, C_i^{er}, C_i^{save} and C_i^{restore}.

For each task τ_i, we evaluate the ECB and UCB sets using techniques from [Altmeyer et al. 2011]. The sets are evaluated separately for instruction and data caches, so this produces ECB^I, ECB^D, UCB^I and UCB^D sets. We choose the largest UCB^I and UCB^D sets found at any point within the task and discard the others, introducing a minor source of pessimism which is also present in previous work [Altmeyer et al. 2011]. Table II shows the ECB and UCB set sizes for each task.

We rotate the ECB and UCB sets by a random number of cache sets. This effectively changes the assumed placement of the task in memory, so that particularly bad (or good) placements are equally likely. A bad task placement would be one that causes a higher CRPD when preemption takes place; experiments with manipulating task placements to reduce CRPD are described in [Lunniss et al. 2012].

Finally, we assign a period T_i to each task based on the task utilization: $T_i = \frac{1}{U_i} C_i^{\text{nr}}$. We note that tasks have implicit deadlines ($D_i = T_i$) and that U_i does not include the processor utilization contributed by C_i^{pre} and C_i^{post}.

7.2. Schedulability Test: Non-reserved Configuration

Our comparisons use schedulability tests to determine if a particular task set is schedulable with a conventional non-reserved cache and with explicit reservation of cache. We expect to find task sets that are schedulable with one approach but not the other. Our schedulability tests are as described in section 6.

For the non-reserved approach, we use CRPD analysis to find the upper bound on the impact of each task on another, and then determine R_i^{nr}, the WCRT of task τ_i assuming non-reserved cache memory.

Without *Save/Restore* operations, $C_i^{\text{pre}} = CS^{\text{ to}}$ and $C_i^{\text{post}} = CS^{\text{ from}}$. This simplifies (3) to $B_i^{CS} = \max(CS^{\text{ to}}, CS^{\text{ from}})$. There is no other source of blocking in any task set.

CRPD analysis gives a value $\gamma_{i,j}$ to represent the CRPD imposed upon τ_i as a result of preemption by τ_j. The Combined ECB-Union/UCB-Union approach, described in full by [Altmeyer et al. 2011], is used to compute $\gamma_{i,j}$ as the maximum number of cache misses multiplied by the cache miss time (section 4.8). $\gamma_{i,j}$ is incorporated into a *sufficient* schedulability test alongside the other time costs of τ_j. The definition of R_i^{nr} is based on (11) which is extended as in [Altmeyer et al. 2011]:

$$R_i^{\text{nr}} = \max(B_i^{CS}, C_i^{\text{post}}) + C_i^{\text{pre}} + C_i^{\text{nr}} + \sum_{\forall j \in \text{ hp}(i)} \left\lceil \frac{R_i^{\text{nr}}}{T_j} \right\rceil (C_j^{\text{pre}} + C_j^{\text{nr}} + C_j^{\text{post}} + \gamma_{i,j}) \tag{12}$$

A task τ_i is known to be schedulable in the non-reserved case if $R_i^{\text{nr}} \leq D_i$. The task set is schedulable if every task is schedulable.

7.3. Schedulability Test: Explicitly Reserved Configuration

R_i^{er} is the the WCRT of task τ_i with explicit reservation of cache blocks.

The lowest-priority task τ_n never preempts, and therefore has no *Save* or *Restore* cost. For this task, $C_n^{\text{pre}} = CS^{\text{ to}}$ and $C_n^{\text{post}} = CS^{\text{ from}}$.

For any other task $\tau_i \neq \tau_n$, $C_i^{\text{pre}} = CS^{\text{ to}} + C_i^{\text{save}}$ and $C_i^{\text{post}} = CS^{\text{ from}} + C_i^{\text{restore}}$. There is no $\gamma_{i,j}$ term. The definition of R_i^{er} is based on (11):

$$R_i^{\text{er}} = \max(B_i^{CS}, C_i^{\text{post}}) + C_i^{\text{pre}} + C_i^{\text{er}} + \sum_{\forall j \in \text{ hp}(i)} \left\lceil \frac{R_i^{\text{er}}}{T_j} \right\rceil (C_j^{\text{pre}} + C_j^{\text{er}} + C_j^{\text{post}}) \tag{13}$$

A task τ_i is known to be schedulable with explicit reservation of memory if $R_i^{\text{er}} \leq T_i$.

7.4. Task Set Utilization

Figure 14 shows how the non-reserved and explicitly-reserved configurations benefit task sets of different utilization U. (The utilization of a task set U is the sum of the utilization of each task within it: $U = \sum_{i \in [1,n]} U_i$.)

We varied the task set utilization $U \in [0.01, 0.99]$ with a step size of 0.01. For each U, we generated 10000 task sets of size $n = 20$ and tested their schedulability in both configurations. The schedulability tests used equations (12) and (13).

Figure 14 tells us that the explicitly-reservable approach increases the ability of a system to schedule task sets. The number of schedulable task sets is often higher when an explicitly-reservable cache is used. The difference is greatest for $U \in [0.3, 0.7]$.

Figure 15 gives an alternate view of the same data: this is the number of task sets that were schedulable with one approach and not the other. In higher-utilization systems (e.g. $U > 0.3$) a significant proportion of task sets are not schedulable with a conventional cache, but can still be scheduled if we use an explicitly-reservable cache.

7.5. Number of Tasks

The success of explicit reservation depends on the CRPD incurred by the task set. CRPD is typically lower for explicit reservation when the cache memory is heavily overutilized, i.e. the number of tasks requiring each cache block is large. This situation is most likely with larger task set sizes, e.g. $n > 5$, though it can occur with any task set containing at least two tasks.

Figure 16 shows how the total number of schedulable task sets is affected by the number of tasks, n. To compute each point on this graph, we generated 10000 task

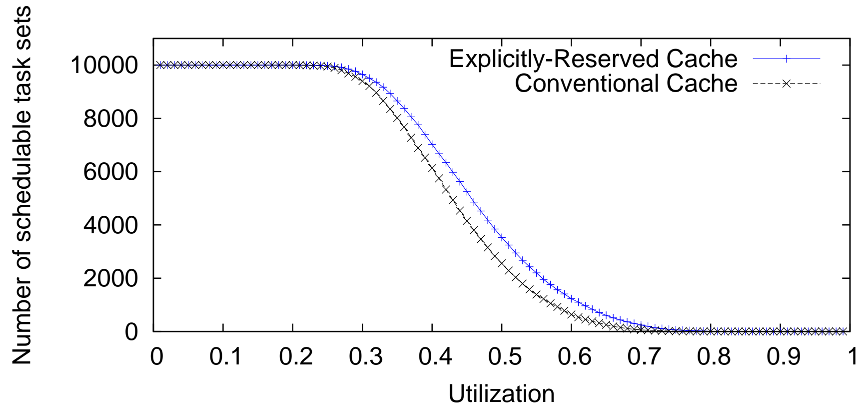

Fig. 14. Comparison of the non-reserved and explicitly-reserved configurations for task sets of size $n = 20$. The data points on this graph are discrete, but we find that a continuous line through them aids presentation, making trends more readily visible, so we have opted to add such lines to this graph and those that follow it.

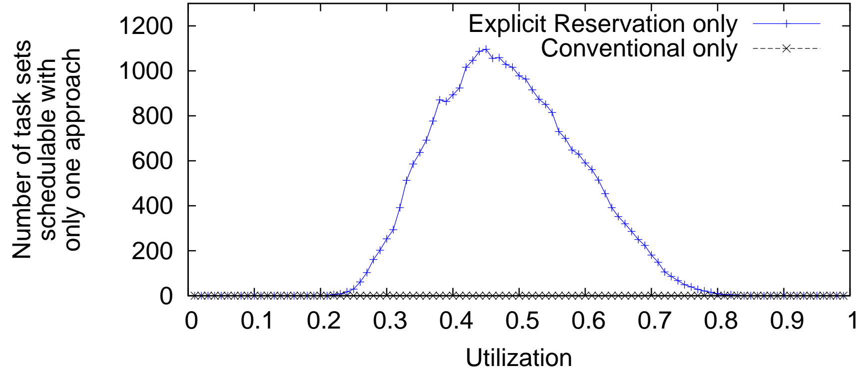

Fig. 15. Further comparison of the non-reserved and explicitly-reserved configurations for task sets of size $n = 20$. Here, we plot the number of task sets that were only schedulable with one approach or the other at each utilization.

sets of size n for each utilization $U \in [0.01, 0.99]$ in steps of 0.01. This gave a total of $980,000$ task sets, which were then tested for schedulability with equations (12) and (13), giving a total number of schedulable tasks for both the explicitly-reserved and conventional caches.

Figure 16 clearly shows that the benefits of the explicitly-reservable cache are related to the size of the task set. The benefit is less significant for small task sets (e.g. size $n < 5$). Figure 16 also suggests that both curves approach some limit as the number of tasks increases towards infinity. Our experiments with very large task sets (not shown) suggest that this is indeed what happens. The limit value is related to the cost of CRPD, and similar to the value for $n = 50$, so explicit reservation is also preferable for scheduling very large task sets.

7.6. Incomparability

Explicit reservation of cache memory is quite different to conventional CRPD analysis. There is no dominance relationship; some task sets are schedulable with one but not the other, and neither approach is certain to be preferable. Figures 14 to 16 show

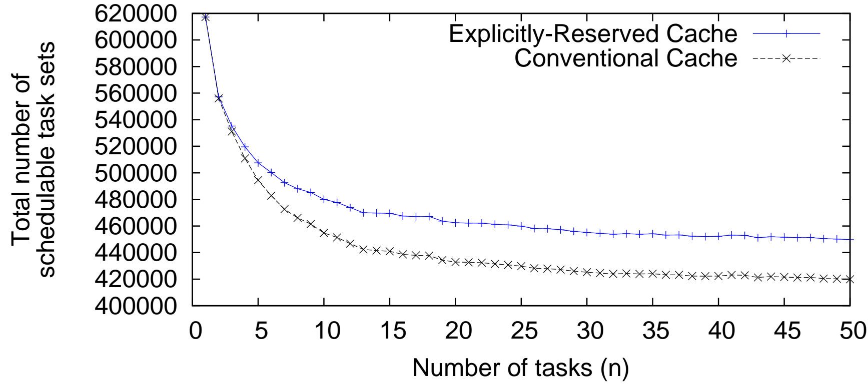

Fig. 16. Schedulability comparison for task sets of size n.

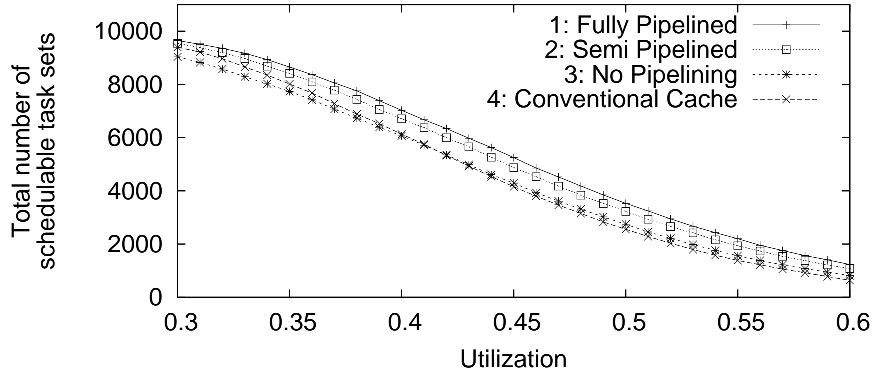

Fig. 17. Comparison of different explicitly-reserved cache implementations for task size $n = 20$.

many cases where explicit reservation is preferable, but it is possible to find task sets that are schedulable with a conventional cache and not schedulable with an explicitly-reservable cache. These cases are more likely to occur for small task sets (e.g. $n < 5$).

One example task set consists of $\tau_1 =$ *fibcall* (high priority) and $\tau_2 =$ *fir* (low priority). Both of these tasks have a small footprint in cache, and so it is likely that their ECB and UCB sets will have no elements in common. If this occurs, conventional CRPD analysis determines a CRPD of zero. However, the combined *Save* and *Restore* cost is 1386ns (Table I). When using an explicitly-reservable cache, each preemption of *fir* incurs a CRPD of 1386ns. For cases like this, the conventional cache is always preferable.

7.7. Dependence on Pipelining

Explicit reservation of cache memory reduces CRPD because the *Restore* operation is more efficient than the series of separate cache misses that would otherwise be required. This is clearly illustrated by adjusting the time required for *Restore*.

For Table I, C_i^{restore} was determined by measurements on our hardware prototype. But the relationship between C_i^{restore} and $S_i^I + S_i^D$ is linear and can be accurately modeled as $C_i^{\text{restore}} = (S_i^I + S_i^D)a + b$, where $a = 133$ns and $b = 547$ns.

Roughly speaking, $\frac{1}{a}$ is the bus bandwidth (cache blocks per second) and b is the bus latency. a and b are properties of the implementation, just like the cache miss time

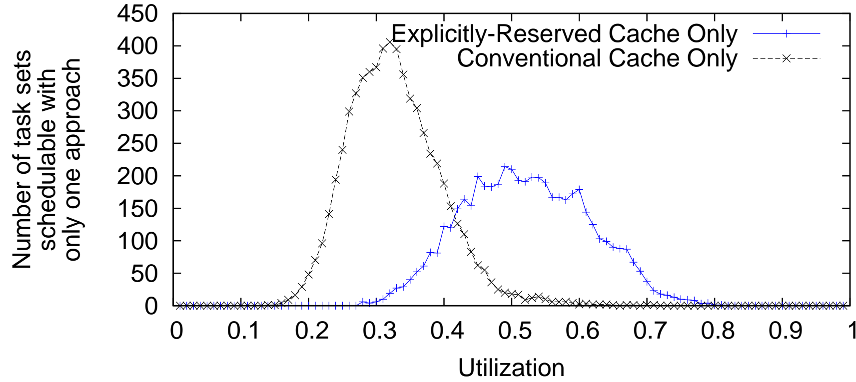

Fig. 18. Comparison of non-reserved and explicitly-reserved configurations without pipelining *Restore* operations. Due to cache footprint effects (section 7.8) the non-pipelined explicitly-reserved cache still improves on the conventional cache in some cases.

(547ns for our prototype). Figure 17 shows the effect of adjusting a and b. We show four scenarios for reducing CRPD:

(1) with a fully pipelined explicitly-reserved cache implementation (same as Figure 14);
(2) with a semi-pipelined implementation where $a = 267$ns and $b = 547$ns;
(3) with a non-pipelined implementation where $a = 547$ns and $b = 0$; and
(4) with a conventional non-reserved cache (same as Figure 14).

As Figure 17 shows, both pipelined implementations improve upon the conventional non-reserved cache and the fully pipelined implementation is best. The non-pipelined implementation is usually worse than the non-reserved cache. This highlights the importance of pipelining.

7.8. Cache Footprint Effect

Even without pipelining *Restore* operations, the explicitly reservable cache approach has some benefit. Figure 18 gives an alternate view of scenarios 3 and 4 in Figure 17. It shows the number of task sets that were schedulable with the non-pipelined implementation, and not schedulable with a conventional non-reserved cache.

These task sets are a small minority of those tested. One example consists of τ_1 = *cover* (high priority) and τ_2 = *matmult* (low priority). The *cover* benchmark accesses many different data elements as part of the implementation of a `switch` statement, but each of these elements is only accessed once. This means that *cover* has a large $|\text{ECB}^D|$ set, containing 25 elements, and a small $|\text{UCB}^D|$ set, containing at most 1 element (Table II). Consequently the benchmark requires no more than 1 data cache block, but it can nevertheless evict up to 25 blocks belonging to lower-priority tasks such as *matmult*.

The explicitly-reserved cache implementation restricts each task τ_i to use no more than S_i^D blocks of data cache. For *cover*, $S^D = 1$. The *Restore* operation only transfers 1 block, whereas with the conventional cache, the CRPD may be as high as 25.

The apparent success of the non-pipelined explicitly-reserved cache (Figure 18) is entirely due to this *cache footprint effect*. If we reduce the cache footprint of tasks running on the conventional cache, the apparent benefit disappears. One way to do this would be to introduce dynamic cache locking with the aim of restricting the size of the ECB sets for each task.

As far as we know, this approach has not yet been attempted as a way to reduce CRPD, other than as a part of the explicitly-reservable cache. However, it may be fruitful for systems where tasks have large ECB sets and small UCB sets, since caches may be locked to allow access to no more than $|\text{UCB}|$ blocks. No hardware support will be required beyond the capability to lock caches on a per-block basis. According to our experiments, the benefits are not as great as those obtained from the explicitly-reservable cache, but CRPDs may still be reduced.

7.9. Exact Schedulability Tests

If we substitute the exact schedulability test (section 6.2) for the sufficient test (section 6.3) we get results that are visually indistinguishable to Figures 14-16.

The difference between the exact test and the sufficient test is greatest for smaller task sets. An examination of all the data gathered for Figures 14-16 revealed that the greatest difference was observed for task set size $n = 9$, and this was tiny in relation to the total number of experiments, affecting only 12 out of $980,000$ task sets. This tells us that the sufficient test is close enough to the exact test to be useful in practice. This is a good thing as the exact test requires significantly more computation time.

8. COMMENTARY

Explicit reservation of cache memory can be extremely beneficial. In the worst case, the *Restore* process requires less time than cache misses carried out separately, as shown in Figure 13. When explicit reservation is applied to large task sets, it typically improves schedulability (Figures 14-16). Task sets that are not schedulable with a conventional non-reserved cache may nevertheless be schedulable with explicit reservation. This is primarily due to the improved efficiency of the *Restore* process (section 7.7). There is a smaller, secondary effect due to restricting the cache footprints of the tasks (section 7.8).

It is possible to find examples where a conventional cache is preferable because the ECB and UCB sets of the tasks do not coincide and therefore the CRPD is zero (section 7.6). The explicit reservation approach is not universal and a conventional cache may be preferable for some task sets.

Explicit reservation of cache blocks depends critically on the speed of the hardware that implements it. It is very important that this hardware allows the *Restore* operation to be pipelined, so that a *Restore* operation for m cache blocks can be achieved in less time than m cache misses, at least for $m > 1$.

We note some approximations in our experiments. The periodic memory refresh is ignored on the grounds that it applies equally to all experiments. We use only the largest UCB set in every task for CRPD analysis of non-reserved caches, a technique which has been used in earlier work [Altmeyer et al. 2011]. This is not entirely accurate. We assume that large numbers of schedulability tests will cause the impact of inaccuracy to tend towards 0. Improved CRPD analysis techniques might improve the results for the non-reserved case.

The *Save* and *Restore* operations require task execution to be strictly nested, i.e. tasks must preempt each other in a strict *last-in first-out* (LIFO) order. This is a requirement for stack-based scheduling, which is widely accepted for hard real-time systems [Baker 1991].

The *save/restore stack* (SRS) stores the tag values used by cache blocks that have been *Save*d: we assume that the stack is large enough to accommodate all the tasks apart from the highest-priority task. This is an additional local memory requirement on top of the usual memory required for a direct-mapped cache. This paper has not considered mechanisms for storing the stack in external memory, which might be important for very large task sets.

9. CONCLUSION

This paper has presented explicit reservation of cache memory as a means to reduce the *cache-related preemption delay* (CRPD) which is observed when multiple tasks share a cache. A reduction in CRPD is achieved by pipelining cache fills when returning to lower-priority tasks, and restricting the cache footprint of each task if possible.

Our experiments use benchmark tasks and timings captured from a hardware implementation to evaluate explicit reservation of cache memory. We have shown that explicit reservation is not worthwhile for all task sets. Sometimes the CRPD is increased by the technique. However, an arbitrarily-chosen task set is more likely to be schedulable with explicit reservation if the number of tasks or the total utilization are large. Explicit reservation is likely to be most useful in heavily-loaded systems, where it can permit larger task sets to be scheduled.

REFERENCES

ALTMEYER, S., DAVIS, R., AND MAIZA, C. 2011. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*. RTSS '11. IEEE Computer Society, Washington, DC, USA, 261–271.

ALTMEYER, S., DAVIS, R. I., AND MAIZA, C. 2012. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems 48*, 499–526.

ALTMEYER, S. AND MAIZA, C. 2010. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture 57*, 707–719.

ARNAUD, A. AND PUAUT, I. 2006. Dynamic instruction cache locking in hard real-time systems. In *Proceedings of the 14th International Conference on Real-Time and Network Systems (RTNS)*. Poitiers, France.

AUDSLEY, N., BURNS, A., RICHARDSON, M., TINDELL, K., AND WELLINGS, A. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal 8,* 5, 284–292.

BAKER, T. P. 1991. Stack-based scheduling of real-time processes. *Real-Time Syst. 3,* 1, 67–100.

BINI, E. AND BUTTAZZO, G. C. 2005. Measuring the performance of schedulability tests. *Real-Time Syst. 30,* 1-2, 129–154.

BLUESPEC. 2013. About the synthesizable modeling company. `http://www.bluespec.com/about/index.htm`.

BRIL, R. J., LUKKIEN, J. J., AND VERHAEGH, W. F. 2009. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Syst. 42,* 1-3, 63–119.

BURNS, A. AND WELLINGS, A. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX* 4th Ed. Addison-Wesley Educational Publishers Inc, USA.

BUSQUETS-MATAIX, J. V., SERRANO, J. J., ORS, R., GIL, P., AND WELLINGS, A. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. RTAS '96. IEEE Computer Society, Washington, DC, USA, 204–218.

CAMPOY, A. M., IVARS, A. P., AND MATAIX, J. V. B. 2002. Dynamic Use Of Locking Caches In Multitask, Preemptive Real-Time Systems. In *Proceedings of the International Federation of Automatic Control*.

CEPEDA, S. 2009. What you need to know about prefetching. `http://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching`.

DAVIS, R., BURNS, A., BRIL, R. J., AND LUKKIEN, J. J. 2007. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst. 35,* 3, 239–272.

DAVIS, R., ZABOS, A., AND BURNS, A. 2008. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. Comput. 57,* 9, 1261–1276.

DIGILENT. 2013. Atlys Spartan-6 FPGA Development Board. `http://www.digilentinc.com/ATLYS/`.

FALK, H., PLAZAR, S., AND THEILING, H. 2007. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '07. ACM, New York, NY, USA, 143–148.

GUSTAFSSON, J., BETTS, A., ERMEDAHL, A., AND LISPER, B. 2010. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th Workshop on Worst-Case Execution Time Analysis*.

HECKMANN, R., LANGENBACH, M., THESING, S., AND WILHELM, R. 2003. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE 91,* 7, 1038–1054.

HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

HERTER, J., BACKES, P., HAUPENTHAL, F., AND REINEKE, J. 2011. CAMA: A Predictable Cache-Aware Memory Allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. 23–32.

HILL, M. D. 1988. A case for direct-mapped caches. *Computer 21,* 12, 25–40.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput. 38,* 12, 1612–1630.

KIRK, D. B. 1989. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proceedings of the 1989 IEEE Real-Time Systems Symposium*. 229–237.

LEE, C.-G., HAHN, J., SEO, Y.-M., MIN, S. L., HA, R., HONG, S., PARK, C. Y., LEE, M., AND KIM, C. S. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput. 47,* 6, 700–713.

LIU, T., LI, M., AND XUE, C. J. 2012. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Syst. 48,* 2, 166–197.

LUNNISS, W., ALTMEYER, S., AND DAVIS, R. 2012. Optimising task layout to increase schedulability via reduced cache related pre-emption delays. In *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS)*.

MUELLER, F. 1995. Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*. LCTES '95. ACM, New York, NY, USA, 125–133.

PUSCHNER, P. 2005. Experiments with WCET-oriented programming and the single-path architecture. In *Proceedings of the Workshop on Object-Oriented Real-Time Dependable Systems*.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. ECRTS '05. IEEE Computer Society, Washington, DC, USA, 41–48.

TAN, Y. AND MOONEY, V. 2007. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embed. Comput. Syst. 6,* 1.

TINDELL, K. W., BURNS, A., AND WELLINGS, A. J. 1994. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Syst. 6,* 2, 133–151.

TSE, J. AND SMITH, A. J. 1998. CPU Cache Prefetching: Timing Evaluation of Hardware Implementations. *IEEE Trans. Comput. 47,* 5, 509–526.

VERA, X., LISPER, B., AND XUE, J. 2007. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys. 7,* 1, 1–38.

WHITHAM, J. AND AUDSLEY, N. C. 2012. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. RTAS '12. IEEE Computer Society, Washington, DC, USA, 3–12.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys. 7,* 3, 1–53.

WILHELM, R., GRUND, D., REINEKE, J., SCHLICKLING, M., PISTER, M., AND FERDINAND, C. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems 28,* 7, 966–978.

XILINX. 2008. MicroBlaze Processor Reference Guide. Manual UG081.

XILINX. 2012. Spartan-6 FPGA Memory Interface Solutions User Guide (AXI). Manual UG416.