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

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

***Abstract*—This paper proposes Carousel, a mechanism to manage local memory space, i.e. cache or scratchpad memory (SPM), such that inter-task interference is completely eliminated. The cost of saving and restoring the local memory state across context switches is explicitly handled by the preempting task, rather than being imposed implicitly on preempted tasks. Unlike earlier attempts to eliminate inter-task interference, Carousel allows each task to use as much local memory space as it requires, permitting the approach to scale to large numbers of tasks.**

Carousel is experimentally evaluated using a simulator. We demonstrate that preemption has no effect on task execution times, and that the Carousel technique compares well to the conventional approach to handling interference, where worst-case interference costs are simply added to the worst-case execution times (WCETs) of lower-priority tasks.

I. INTRODUCTION

A key issue for predictable preemptive multitasking system implementations is bounding the cost of *context switches*. A context switch occurs when one task stops executing and another begins [1]. Simple real-time scheduling models assume that context switching only imposes a small, fixed cost. In a sufficiently simple system this is true, but when task execution times are affected by the state of CPU components such as caches, it is not [2]. Preemptive multitasking allows any task to be interrupted by a higher-priority task during execution, and when the original task resumes, stateful components including cache will be in a different state. In this paper, we refer to this phenomenon as *inter-task interference*. It may change the execution times of preempted tasks. Figure 1 illustrates a preemption resulting in some inter-task interference.

For the remainder of the paper, we concentrate exclusively on local RAM as the stateful component. Local RAM can be implemented as either cache or *scratchpad memory* (SPM) and is high-speed, low-latency memory, physically located close to the CPU (Figure 2). Other components such as branch prediction units can cause similar inter-task interference effects [3], [4], but these are outside the scope of this paper.

Three possible solutions can be used to deal with inter-task interference. Firstly, we can *statically partition* the stateful components so that each task gets a small, reserved portion of the local RAM (Figure 3). Cache partitioning is an example [5] which eliminates inter-task interference by locking cache lines used by other tasks. Secondly, we can *explicitly save and restore* the component state when tasks begin and end. This is the approach used for the CPU register file. Finally, we can ignore the component state at runtime and instead statically *bound* the worst-case effects of inter-task interference, which is possible for caches under some circumstances [6]–[8].

These solutions can be compared by considering both their limitations and their effect on the *schedulability* of the real-time system as a whole. A real-time system is schedulable if every task is guaranteed to meet its deadline [1]. *Schedulability analysis* is used to check this property. It relies on task properties such as their priorities and *worst-case execution times* (WCETs): the maximum amount of CPU time required to complete execution, determined by WCET analysis [9].

The *static partitioning* approach leads to larger WCETs and hence schedulability is reduced. This is because each task has an optimal memory requirement [10]. If this is not available within its partition, the WCET is increased. Partitioning is therefore an impractical approach for large numbers of tasks, because the local RAM space is not shared efficiently.

The *bounding* approach is necessarily pessimistic as it is not generally possible to know *exactly* which cache blocks will be evicted or needed again. The supersets of useful and/or evicted cache blocks can be computed for tasks [6] as a safe but inexact upper bound. Again, this leads to larger WCETs, or at least, larger WCET *estimates*: because while the WCET may not be increased in reality, the safe upper bound determined by analysis certainly is. Hence, schedulability is reduced. Bounding is also only suitable for *timing-compositional* systems in which interference can be accounted by simple addition [4], and it is only suitable for cache, not SPM.

The *explicit save/restore* approach can also reduce schedulability, because additional time is required for saving and restoring the local RAM state. If this activity is performed within the context switch (i.e. along with saving and restoring the CPU register values) then the schedulability analysis will consider it a global, static cost applying to every context switch. This would be undesirable because of the size of local RAM, with a typical minimum of several kilobytes.

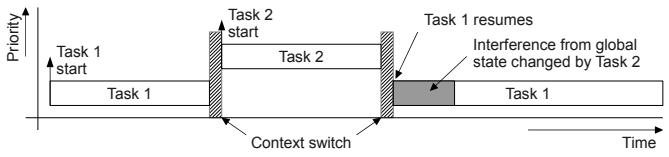

Fig. 1. The impact of inter-task interference. The execution time of Task 1 is increased because Task 2 preempted it and changed the state of the cache.

Schedulability would be dramatically reduced by incorporating such a large cost.

However, if the save/restore process is explicit, it can be carried out *within* a task. Figure 4 illustrates this: by the time Task 1 resumes, Task 2 has restored the local RAM state.

Carousel, proposed in this paper, is based on the principle that each task should pay the *reservation cost* of any local RAM that it wishes to use. This is the cost of saving the local RAM state (the first step carried out by any task) and the cost of restoring it (the last step). Carousel is a practical implementation of the explicit save/restore approach.

With Carousel, the inter-task interference is zero, much like a partitioning approach. But Carousel allows each task to use as much (or as little) local RAM space as it wants. The reservation cost is *internalized* as part of the WCET of the task that actually uses the memory, rather than being treated as an *externality* affecting the WCET of other tasks, as in bounding approaches.

Carousel is useful for any priority-based real-time scheduling paradigm in which task execution is strictly nested, i.e. once a task starts to execute, no task of lower priority can run until the higher priority task completes its execution. In particular, it may be used with Baker's *stack resource protocol* (SRP) [11], and for the stack-based function call protocol used by the C language. This is because the SRP ensures that a task is *never* blocked after it commences execution (unlike the *priority ceiling protocol* [12]). This property means that all tasks can share a common stack. The SRP can be used with fixed-priority and *earliest deadline first* (EDF) schedulers. Furthermore, Carousel blocks can be used as cache or SPM, which may be useful in mixed-criticality systems mixing hard real-time and non real-time tasks.

After presenting related work in section II, this paper gives a problem analysis (section III) leading to the Carousel hardware and software (section IV). A suitable schedulability analysis technique is given (section V) followed by some experiments (section VI) which demonstrate that task execution times are completely unaffected by preemption. Section VII describes some possible improvements, and section VIII concludes.

II. RELATED WORK

Inter-task interference may occur whenever tasks share a stateful resource such as a cache [13] (e.g. Figure 1).

The problem of *intra*-task interference due to cache state is now quite well-known and has been studied thoroughly. Cache WCET analyses model the state of a cache at each point within a task in order to estimate the worst-case miss

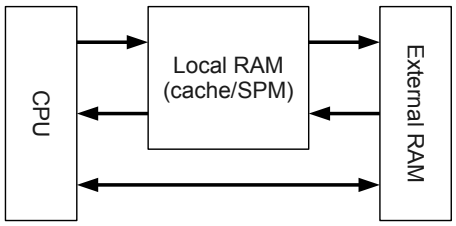

Fig. 2. Local RAM in relation to the CPU and external RAM. Local RAM may be implemented as a cache or SPM.

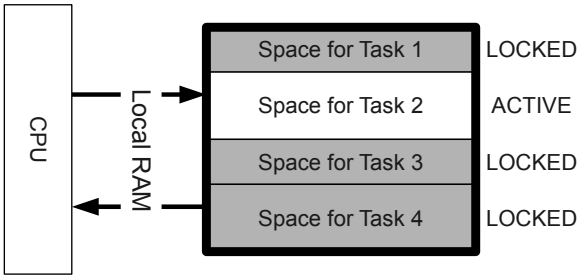

Fig. 3. Local RAM partitioning. Each task gets a small, fixed slice of the local RAM. Other areas are inaccessible or read-only.

count, and hence the maximum execution time [4], [9], [14]. Each cache state is dependent on earlier cache states. Earlier task activity may result in subsequent activity producing either a hit or a miss. This dependence is a form of interference, but *intra*-task interference, since it occurs between one part of a task and another.

Inter-task interference has also been examined [5], [13]. This occurs between two or more tasks in a multitasking system. When two or more tasks share a cache, activity in one task can disturb data used by the other, producing hits or misses at unpredictable times. Inter-task analysis is pessimistic because the exact set of evicted and/or useful cache blocks of tasks cannot usually be computed offline.

Earlier work has prevented inter-task interference entirely by *static partitioning*: reserving local RAM space for each task [5], [15]. Each task is only permitted to update its own partition (Figure 3). The remainder is locked [13]. While the size of the partitions can vary between tasks, and non real-time tasks can share a single partition, the assignment is static. Tasks cannot use more than their fixed share of local RAM, not even temporarily. This is a problem, because tasks may have a suboptimal local RAM allocation [10]. This situation becomes a near-certainty as the number of tasks increases.

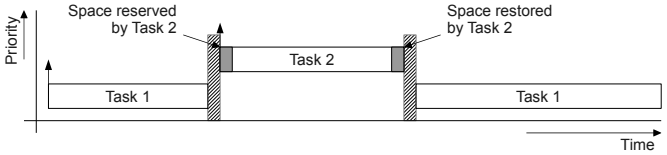

Fig. 4. Interference can be avoided by explicitly saving and restoring the local RAM state as part of the new task. That way, a task only needs to save and restore the parts of the local RAM that it actually needs to use.

Consequently, some researchers have suggested allowing inter-task interference, but *bounding* its impact. A number of approaches are evaluated in [6]. They are suitable for timing-compositional systems where interference does not cause any timing anomalies [4]. They incorporate the cost of reloading evicted cache blocks into the worst-case response time equation [7].

A third approach involves *explicitly saving and restoring* the state of local RAM. This idea has been successfully applied *within* tasks for SPM [16], [17]. It avoids intra-task interference by ensuring that the state of local RAM is known at each point within the task. But it is not trivial to expand the idea to multitasking. The large size of local RAM (several kilobytes or more) prevents saving and restoring the entire state on each context switch.

III. PROBLEM ANALYSIS

Dynamic interference acts in the "*wrong direction*", in that the *preempted* task pays the additional cost due to interference (Figure 1). In order to bound interference, we must assume the worst-case number of preemptions, and also assume the worst results from each preemption. Pessimism is essentially certain.

The other direction, where the *preempting* task pays the additional cost, would be preferable (Figure 4). In this model, the preempting task must save and restore the state of any local RAM it wishes to use. The cost is incurred once per preemption, the preempted task's execution time is unaffected, and pessimism is avoidable. In this way, inter-task interference can be *eliminated entirely* while still allowing each task to use *any amount* of local RAM.

This idea is simple, but its implementation is tricky. Firstly, we are obliged to manage scheduling with a stack policy, so that tasks start and complete in *last-in first-out* (LIFO) order. If we use any other policy, we will need a way to resume *any* task after *any* preemption. In that arrangement, when a task τ_i completes, we cannot just restore the state of the local RAM as it was before τ_i started, because this assumes that the previous task *and* the next task are both τ_j. With a non-LIFO task ordering, the next task may be $\tau_k \neq \tau_j$, requiring a wholly different state to be restored, potentially of *any size*. We absolutely require the swapping cost to be constant for each task, so this will not do. Fortunately, scheduling policies that assure the required LIFO ordering are already well-known and in common use, having been described by Baker as the *stack resource protocol* (SRP) as early as 1991 [11].

A second problem is apparent if tasks need to communicate via shared memory, which is very likely in any practical system. If one task may access memory updated by another, then we have to ensure that both tasks have a coherent view of that memory. We must ensure this even if one task is "swapped out", and if one or both tasks copy the relevant data into local RAM. The issues are tricky, but they have already been investigated during our earlier work on the *scratchpad memory management unit* (SMMU) [18]. The SMMU gave a logical address space to both local RAM and external RAM, so that blocks could be swapped between the two without changing their logical address. This proved an effective solution, dealing with many edge cases related to pointer aliasing.

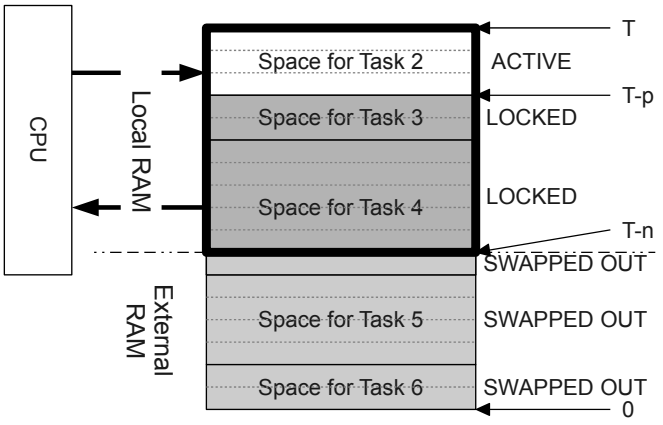

Fig. 5. Carousel divides RAM space into blocks of size 2^x bytes, organized as a stack. The top n blocks are always resident in local RAM, and the top p blocks ($p \leq n$) are used by the current task.

IV. CAROUSEL

We propose an extension to local RAM named Carousel. With Carousel, inter-task interference is eliminated. But unlike a partitioning approach, each task can use any amount of local RAM.

In memory, Carousel acts as a stack (Figure 5). The stack elements are *blocks* of fixed size 2^x bytes. The top n blocks are stored in local RAM. The remaining blocks are stored in external RAM.

Starting a new task τ_i involves allocating p_i blocks on Carousel. These blocks are pushed onto the top of the stack, reserving them for τ_i, while p_i blocks are swapped out to external RAM (Figure 6). As always, the top n blocks remain in local RAM, and the remainder are in external RAM.

When a task ends, the process is reserved. p_i blocks are popped from the top of the stack, and p_i blocks are swapped in from external RAM (Figure 7). Again, the top n blocks remain in local RAM.

p_i is task-specific. Each task τ_i will have an optimal p_i that minimizes either the average execution time or the *worst-case execution time* (WCET), noting that the time required to reserve p_i blocks (i.e. swap in and out) is incorporated into the execution time.

A. Task Invocation Procedure

When a task τ_i is invoked on a Carousel-architecture machine, the following steps are taken by the *real-time operating system* (RTOS) in order to execute τ_i (Figure 8):

1) Save registers used by previous task τ_j (context switch)
2) Swap out Carousel blocks T through $(T + p_i) \bmod n$, where T is the top of Carousel's stack (Figure 5) and n is the total number of blocks
3) Set $T = (T + p_i) \bmod n$
4) Call task τ_i
5) Set $T = (T - p_i) \bmod n$

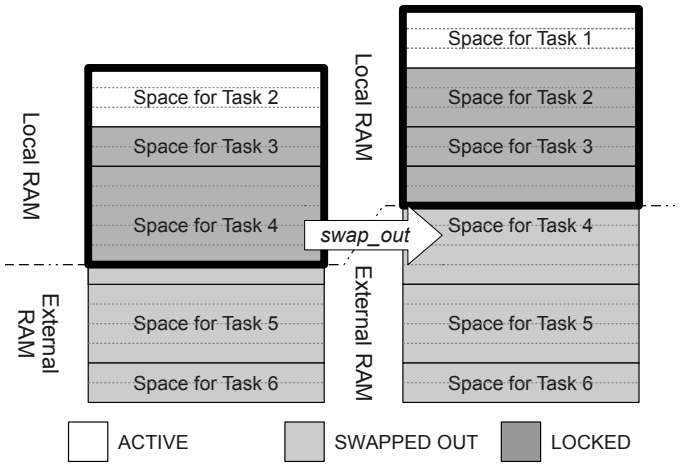

Fig. 6. Starting a new task (Task 1) with Carousel involves reusing local RAM blocks originally allocated by lower-priority tasks by swapping their contents out to external RAM.

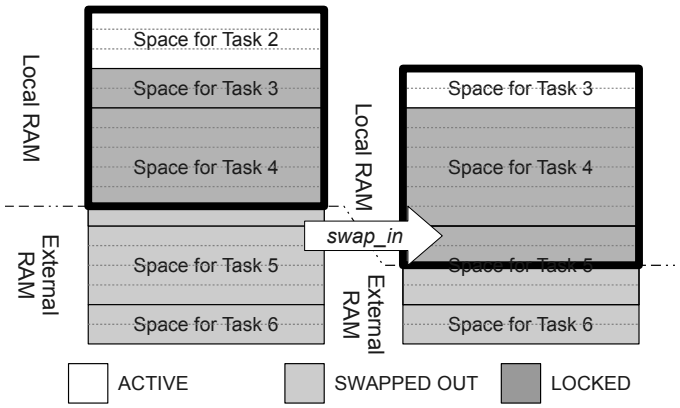

Fig. 7. Ending a task (Task 2) with Carousel involves restoring the contents of local RAM blocks swapped out earlier (Figure 6).

6) Swap in Carousel blocks T to $(T + p_i) \bmod n$
7) Restore registers used by τ_j (context switch)

The process is re-entrant: a further task τ_k can be started during step 4. This may happen any number of times; τ_k may also be preempted, or initiate subtasks (i.e. call methods).

B. Overhead of Carousel

As Carousel shifts the cost of reserving local RAM for task τ_i onto task τ_i itself, the execution time of any task preempted by τ_i is unaffected by that preemption. There is no inter-task interference: the execution time of the lower-priority task is the same regardless of how many times τ_i preempts it[1].

The cost of reserving the local RAM is proportional to the amount of space required. Steps 2 and 6 (Figure 8) operate in $O(p_i)$ time, while steps 1, 3, 5, and 7 operate in $O(1)$ time. Therefore, the overhead of Carousel can be expressed precisely as a linear expression of the form $ap + b$.

[1]Here, "execution time" is defined conventionally for real-time systems theory as the CPU time used by one task [1]. By definition, τ_i's execution time explicitly excludes any time when any other task is running. The minimum possible execution time for a task is its *best-case execution time* (BCET), while the maximum possible execution time is its *worst-case execution time* (WCET).

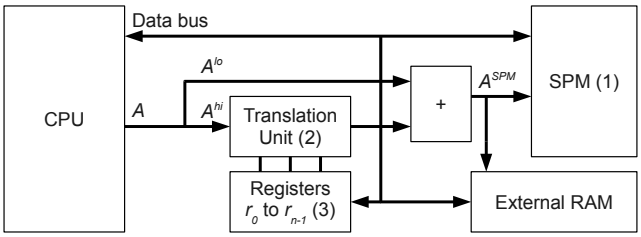

Fig. 9. Basic Carousel components: (1) an SPM, (2) a translation unit, and (3) n memory mapped registers (one per block).

By incorporating this cost into the execution of τ_i, Carousel's model differs from earlier approaches that considered interference as separate from the execution times of tasks [6] or as part of preempted tasks [8]. It is also unlike the approach of considering the whole local RAM as context, which increases the cost of every context switch. Though a Carousel task could use the entire local RAM, there is no requirement to do so, and tasks can choose their memory usage to minimize execution time. Lastly, it is unlike a partitioning approach, which would avoid both interference and swap-in/swap-out costs but prevent any task using more than a small, statically reserved area of local RAM space [5], [15].

The reserved space for τ_i, $2^x p_i$ bytes, is typically larger than the space that is actually required because of the need to round up to the nearest multiple of blocks. Some SPM space is therefore unused; some data is transferred unnecessarily. There is a tradeoff between block size and system performance which is not examined within this paper.

C. Hardware Design

In its most basic form, Carousel is implemented using three components (Figure 9), which we describe for a generic 32-bit CPU[2]. They are (1) an SPM of size $2^x n$ (the local memory), (2) n memory-mapped registers of width $32 - x$ named r_0 to r_{n-1}, and (3) a translation unit. The SPM is logically divided into n blocks, each of size 2^x bytes.

The translation unit receives addresses, A, from the CPU. These are addresses for accessing code or data, produced in the course of instruction fetches and load or store operations. The addresses are split at bit x, so we have:

$$A^{\text{hi}} = \frac{A}{2^x} \qquad A^{\text{lo}} = A \bmod 2^x \tag{1}$$

Each A^{hi} is compared against all n registers in parallel. If some $r_i = A^{\text{hi}}$, then the memory access is a *hit*, and is redirected to SPM. The new address is:

$$A^{\text{SPM}} = A^{\text{lo}} + 2^x i + S \tag{2}$$

(Where S is the base address of the SPM.)

[2]Carousel is not a CPU-specific technology, but certain CPU designs may themselves be a source of inter-task interference because of stateful components such as branch predictors. [3] gives an overview of problematic CPU designs.

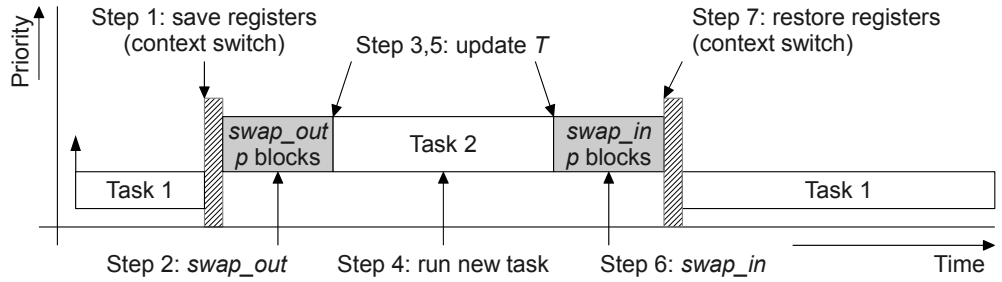

Fig. 8. The task invocation procedure for Carousel. Every task τ_i begins by swapping out p_i blocks in order to reuse that space for its own code and data. The blocks are swapped in before the task completes.

If $\forall i.r_i \neq A^{\text{hi}}$, then the memory access is a *miss*, and it passes to external RAM. This is slow, and uses more energy than an SPM access, so tasks achieve better performance by mapping code and data to SPM.

The purpose of the translation unit is to ensure that code and data retains the same logical address A whether it is in SPM or external memory. This simplifies software design [18].

Carousel allows software to access the SPM and r_0 to r_{n-1} directly, as well as via the translation unit. This is used to swap blocks in and out of local memory at the beginning and end of each task. Tasks can also swap blocks during execution if necessary.

D. Software Design

Carousel's supporting software needs to be able to copy data between SPM and external RAM. We define a *dma_copy*(A_d, A_s, s) method which copies s bytes from $[A_s : A_s + s]$ to $[A_d : A_d + s]$.

The *open*(A, i) method assigns the data at address A to block i. Two steps are required:

1) Set $r_i = A$
2) *dma_copy*($S + 2^x i$, A, 2^x)

The *close*(i) method writes back the data in block i to its associated external memory address r_i.

1) *dma_copy*(r_i, $S + 2^x i$, 2^x)
2) Set $r_i =$ UNUSED

The *swap_out*(p) method swaps out Carousel blocks T through $T + p - 1$ so that they can be used by a new task (Figure 6). It is necessary to store the values of r_T through r_{T+p-1} so that they can be restored later, so stack operations *push* and *pop* are introduced:

1) Set $j = 0$
2) *push*($r_{j+T \bmod n}$)
3) *close*($j + T \bmod n$)
4) Set $j = j + 1$
5) If $j < p$ goto step 2

The *swap_in*(p) method restores Carousel blocks to their former positions, undoing an earlier *swap_out* (Figure 7):

1) Set $j = p$
2) Set $j = j - 1$
3) *open*(*pop*(), $j + T \bmod n$)
4) If $j > 0$ goto step 2

System	Cost
ARM PB11MPcore	79
StrongARM-110	17
PPC 405 (FX12)	33
Microblaze (ML505)	31

TABLE I
Measured worst-case execution time for a load operation on four embedded systems in clock cycles when cache is disabled. The cost of a cache hit is 1 clock cycle. This table is based on data published in [18].

These system methods are enough to implement the Carousel functionality, but a further method-invoking method is extremely useful. This executes an application method after *open*ing its code and stack space. *call_method* takes three arguments: A_i (the address of the target method for τ_i), y_i (the code size of τ_i), and z_i (the stack size of τ_i). The method invoking process is shown in Figure 10.

Two simple improvements to this basic design are used within our experiments. Firstly, since code is read-only, the *close* operation for code blocks does not need to write them back to external RAM. They can simply be discarded. Secondly, since stack data is only valid for addresses above the current stack pointer, the *open* and *close* operations for stack blocks do not need to access external RAM at all. However, *swap_in* and *swap_out* must always copy blocks, regardless of their contents.

E. Prototype System Architecture

We constructed a prototype of the Carousel system within a simulator. Based on our existing FPGA experience, we know that the simulated system can be translated to FPGA hardware, but a simulator can be built more quickly, which is valuable for experimentation. Furthermore a simulator provides an easy way to tweak design parameters such as n (the number of Carousel blocks) and 2^x (the size of each Carousel block) during experiments. The simulator includes a Microblaze CPU [19], an interrupt timer, an external RAM, a small SPM for the OS, a DMA controller, and Carousel.

The OS is Carousel OS, a prototype RTOS implementing the functions listed in section IV-D and specifically designed for the Carousel architecture. This RTOS cannot easily be stored in Carousel because it must remain in local RAM at all times.

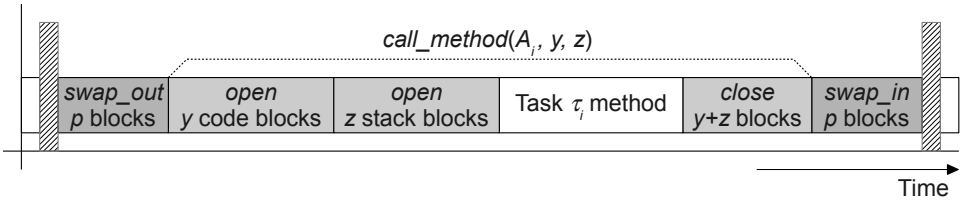

Fig. 10. The method invocation procedure for Carousel, implemented by *call_method*.

Rather than attempt to arrange for *swap_in*, etc., to skip blocks used by the RTOS, it is simply placed in a separate SPM.

The simulator includes some assumptions about timing, based on real FPGA hardware. The simulated Microblaze CPU uses the same execution times as the real CPU as documented in [19]. Bus operation times are chosen to be typical of embedded systems (Table I). The bus used in our experiments imposes an overhead of 49 clock cycles for every transaction, plus a further overhead of 1 clock cycle for every 4 bytes in the transaction (rounding up). The maximum transaction size is 64 bytes: the same limit as that imposed by PLB, the Microblaze system bus [19].

These settings mean that it takes 50 clock cycles to load or store a single word in external RAM, and a single clock cycle to load or store a word in local RAM. Transferring 32 bytes with *dma_copy* takes $\frac{32}{4} + 49 = 57$ clock cycles; transferring 64 bytes takes $\frac{64}{4} + 49 = 65$.

V. SCHEDULABILITY ANALYSIS

Schedulability analysis is straightforward with Carousel. We consider steps 2 through 6 within Figure 8 as part of the task τ_i. This means that the cost of *swap_in* and *swap_out* are incorporated into C_i, along with the whole of *call_method* if used (Figure 10).

If there were no static context-switch cost at all, i.e. the RTOS scheduler and interrupt handler operate in zero time, then the basic task scheduling equation for the worst-case response time R applies [1]:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \tag{3}$$

In that equation, R_i represents the worst-case response time for τ_i, $hp(i)$ is the set of tasks with higher-priority than τ_i, and T_i represents task period. B_i is the blocking time: the longest time for which a lower priority task locks a resource that is shared with τ_i or any task $j \in hp(i)$. Each task τ_i also has a deadline D_i where $D_i \leq T_i$.

As described in [1], the equation is easily refined to incorporate context switch costs *to* a task (CS^1) and *from* a task (CS^2), provided that these are constants. For Carousel, they are indeed constant, because all variable task-switching costs are incorporated into the cost of running the higher-priority task. The final equation is:

$$R_i = CS^1 + CS^2 + C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil (CS^1 + CS^2 + C_j) \tag{4}$$

The system is considered schedulable if $\forall i . R_i \leq D_i$,

The blocking time B_i in equation 4 is modified to account for the fact that Carousel is effectively a resource shared by all tasks. The Carousel operations (*open*, *swap_in*, etc.) are critical sections: they cannot be interrupted, not even by the RTOS. The worst-case blocking time for task preemption is incurred when the scheduling event occurs right at the beginning of either the longest resource access or the longest critical section created by Carousel operations, i.e. the point marked on Figure 11. That critical section incorporates a complete context switch (from and to, CS^2 and CS^1), plus *close*, *swap_in*, *swap_out*, and *open*. For all τ_i except the lowest-priority task τ_N, B_i is greater than or equal to the execution time of this critical section.

VI. EXPERIMENTS

Our experiments use the prototype system architecture described in section IV-E. Source code for experiments may be downloaded from http://www.jwhitham.org/pubs/.

A. Benchmark Tasks

We took some sample applications from the Mälardalen Real-time Technology Center (MRTC) benchmarks [20]. These have the useful property of being single-path, so any execution always produces the same execution time *provided that* there is no interference from other tasks.

We excluded benchmarks that use floating-point, and benchmarks requiring more than a combined 4kbytes of code and data memory for efficient (in-SPM) execution (e.g. adpcm, edn). Furthermore, we excluded benchmarks that can be optimized to nothing (e.g. loop3, fac) as they take no input and have no effect on RAM. The list of remaining benchmarks is shown in Table II. In general, larger benchmarks could be accommodated by dynamic SPM allocation, e.g. [16], [17]. Note that a cache/SPM partitioning strategy would require more than 16kb of local RAM to accommodate all the tasks, even in this small-scale example!

Each benchmark was manually adapted for use with SPM, and hence Carousel. The SPM usage scheme is very simple: all of the code is stored in SPM along with the call stack, and frequently-used global data is stored in SPM where possible.

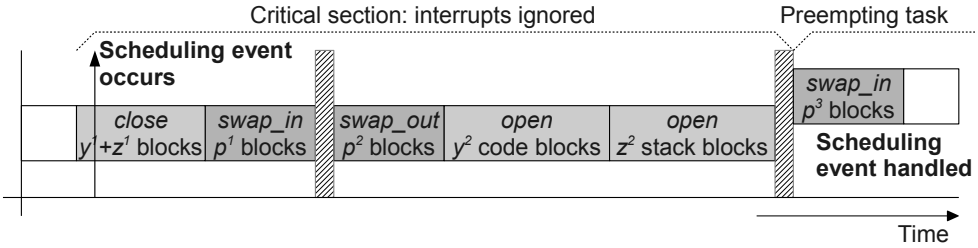

Fig. 11. The worst-case blocking time for a task is always greater than or equal to the WCET of the critical section created by Carousel operations.

Task Name	Code Size	Data Size	Stack Size
binarysearch	140	120	56
bsort100	208	408	56
cnt	280	544	56
compress	1660	1757	36
crc	252	1168	128
duff	304	0	56
expint	44	0	0
fdct	868	128	84
fir	840	701	112
insertsort	272	44	56
janne	100	0	0
jfdctint	928	256	84
matmult	156	5056	128

TABLE II
Benchmark programs used as application tasks. Sizes are in bytes.

Task Name	Code Blocks	Data Blocks	Stack Blocks
binarysearch	2	1	1
bsort100	2	4	1
cnt	3	4	1
compress	13	0	1
crc	2	8	1
→ icrc	4	0	1
duff	3	2	1
expint	1	0	0
fdct	7	1	1
fir	7	6	1
insertsort	3	1	1
janne	1	0	0
jfdctint	8	2	1
matmult	2	0	1
→ Initialize	2	13	1
→ Multiply	5	10	1

TABLE III
Carousel block count for tasks and subtasks (methods called by tasks).

Most of the tasks consist of a single method once optimized for size, inlining methods that are only called from one place.

More complex techniques were only needed in three cases. These reallocate the Carousel blocks dynamically within the task, changing the code and data during execution. An exact description is outside the scope of this paper, where experiments require only that tasks make use of Carousel. However, a short description follows.

The crc and matmult benchmarks used library subroutines more than once. These could not be inlined without increasing the code size, and were therefore invoked as subtasks using the method-invoking method (section IV-D). The matrices multiplied within the matmult benchmark also proved too large to be loaded into Carousel, but it was possible to load a few rows at a time into SPM through a loop tiling optimization of the sort described in [21]. Though necessary changes were applied by hand in this case, an dynamic SPM allocator (e.g. [16], [17]) could have done some of the required work.

B. Assumptions

Using the simple strategy of mapping all code and stack data to SPM, at least 2kbytes of space are required for some tasks (e.g. compress, Table II). We set the total size of the Carousel blocks ($n2^x$) accordingly.

Experience with evaluating the SMMU tells us that we cannot set n, the number of blocks, too high because all comparisons must be performed in parallel and this can affect the maximum frequency of the CPU. $n = 16$ was found to be practical [18]. This leaves $x = 7$ for 128-byte blocks. The resulting allocation of Carousel blocks is as shown in Table III. Notice that there is no way to fit all tasks in Carousel simultaneously. Furthermore, some tasks (crc, matmult) are themselves split across more than one method.

C. Carousel Overheads

On Microblaze, Carousel OS requires 3.2 kbytes of SPM space for its code and data tables. This includes all device drivers and the scheduler. The (observed) worst-case value for CS^1 is 401 clock cycles, and the (observed) worst-case value for CS^2 is 387 clock cycles.

To measure the swapping overhead for different numbers of Carousel blocks, we introduced a trivial task that immediately returns - but nevertheless requires y blocks for code and z blocks for stack. Table IV shows how the execution time of this task varies with different values of y and z.

D. Task Set Generation

Carousel is intended to eliminate inter-task interference, and this property is easily tested by looking for the main effect of interference, specifically a change in execution time due to preemption.

We generated one thousand task sets by assigning random priorities, periods and offsets to each task listed in Table II. The priority of each τ_i was assigned so that no two tasks have equal priority. The offset of each τ_i was assigned a value between 0 and T_i using a uniform pseudorandom number

Code Blocks	Stack Blocks	Overhead
1	0	864
1	1	1157
1	2	1443
2	0	1280
2	1	1573
2	2	1859
3	0	1696
3	1	1989
3	2	2275
4	0	2112
4	1	2405
4	2	2691
y	z	$416y+$ $286z+455$

TABLE IV
Carousel system overheads: the combined cost of *swap_in*, *swap_out*, *open* and *close* for tasks requiring different numbers of code blocks (y) and stack blocks (z). Each overhead value is in clock cycles. Every execution time given in this paper incorporates these system overheads. The final row of the table gives an upper bound on the overhead, expressed in terms of y and z.

Task Name	Observed BCET	Observed WCET	Preemption count
binarysearch	2086	2086	57
bsort100	94647	94647	1538
cnt	9328	9328	309
compress	76851	76851	1351
crc	40399	40399	824
duff	3874	3874	149
expint	1838	1838	50
fdct	5657	5657	202
fir	61835	61835	1167
insertsort	3392	3392	112
janne	1143	1143	59
jfdctint	9230	9230	289
matmult	728391	728391	7981

TABLE V
Observed best-case and worst-case execution times (BCET, WCET) for the tasks (clock cycles) obtained by experiment on a Carousel-architecture machine (section VI-E).

generator. The period T_i was also assigned using the same generator, with a range from $2C_i$ to $max(4C_i, \frac{SZ}{4})$, where SZ is the execution time for the entire task set, not including bootup time. $SZ = 15 \times 10^6$ clock cycles for our experiments. Each deadline $D_i = T_i$. We checked the schedulability of each task set using equation 4, and rejected and regenerated each task set found to be unschedulable.

E. Results

We observed that every execution of a specific task always results in the same execution time (Table V). This is expected given the single-path nature of the benchmark programs, *provided that* preemption has no effect on the execution time. Table V also shows how many times each task was preempted across all task sets, demonstrating that task execution times are indeed unaffected by preemption.

Figure 12 illustrates the relationship between the observed WCETs of each task and the Carousel system overheads. It is plain that the very short tasks (janne, expint, binarysearch) are dominated by the overhead. However, the overhead is only a very small part of the execution time of longer tasks.

F. Comparison with Cache

Carousel should also be competitive with techniques used in previous work, and in order to examine this aspect, we compared Carousel with an equally-sized cache (Table VI). The cache provides 1kbyte of code space and 1kbyte of data space, organized as 16 byte blocks. A simulation of Microblaze's cache, it is a direct-mapped Harvard-architecture cache with a write-through policy and no allocate on write.

There is now some variation in execution time, clearly visible in Figure 13. Although the tasks are single-path, their execution times are now dependent on the state of the cache, which is affected by the execution of other tasks. This is the effect of inter-task interference, as shown in Figure 1. Sometimes, the variation is quite small (e.g. 1.2% for matmult). This is because the benchmark task runs for a relatively long time,

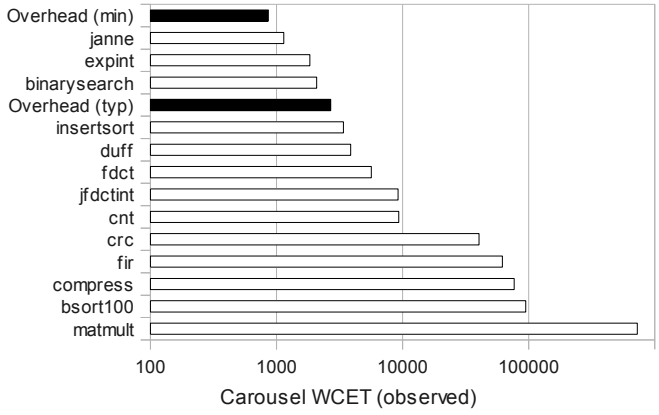

Fig. 12. The relationship between the observed WCETs of each task and the Carousel system overheads (marked in black). A logarithmic scale is used to accommodate the wide range of execution times. 2691 (Table IV) is taken as a typical overhead.

and the effects of any interference are amortized across this time. (The true WCET, considering all possible interference, may be much greater than the observed value given in Table VI.)

Four tasks see an increase in WCET for Carousel, i.e. the task is slower on a Carousel architecture. In three cases (binarysearch, expint, janne) this is because the tasks are short and complete very quickly. Thus, the overhead of using Carousel is not made up by any improvement in execution time. For instance, *expint* requires only one code block (Table III), so an overhead of 864 clock cycles is imposed by Carousel (Table IV). This accounts for almost half of the task's execution time (1838). The cache has a lower overhead and completes in 1046 clock cycles. For compress, the difference is marginal, and improvements could be obtained by better use of SPM.

Nine tasks see an improvement in WCET for Carousel versus a cache. Sometimes the improvement is very large. For instance, the bsort100 benchmark requires over 590,000 clock cycles with a cache, but under 95,000 clock cycles with Carousel! Carousel has improved performance.

Task Name	Observed BCET	Observed WCET	Comparison factor
binarysearch	143	468	0.22
bsort100	591766	596251	6.30
cnt	10921	11961	1.28
compress	67409	70074	0.91
crc	57104	59314	1.47
duff	8238	8888	2.29
expint	981	1046	0.57
fdct	8384	9944	1.76
fir	90981	107296	1.74
insertsort	6083	6603	1.95
janne	286	481	0.42
jfdctint	14325	16275	1.76
matmult	1030453	1043323	1.43

TABLE VI
Observed execution times (clock cycles) obtained by experiment on a cache-architecture machine. Comparison factor is the cache observed WCET divided by the Carousel observed WCET.

Task Name	Observed BCET	Observed WCET	Comparison factor
binarysearch	93	613	0.29
bsort100	91406	94591	1.00
cnt	5701	7976	0.86
compress	15699	19664	0.26
crc	44314	48084	1.19
duff	1038	2208	0.57
expint	981	1046	0.57
fdct	1584	3404	0.60
fir	55736	83036	1.34
insertsort	983	1633	0.48
janne	286	481	0.42
jfdctint	4325	6860	0.74
matmult	611263	642983	0.88

TABLE VII
Observed execution times (clock cycles) obtained by experiment on a cache-architecture machine with a write-back policy.

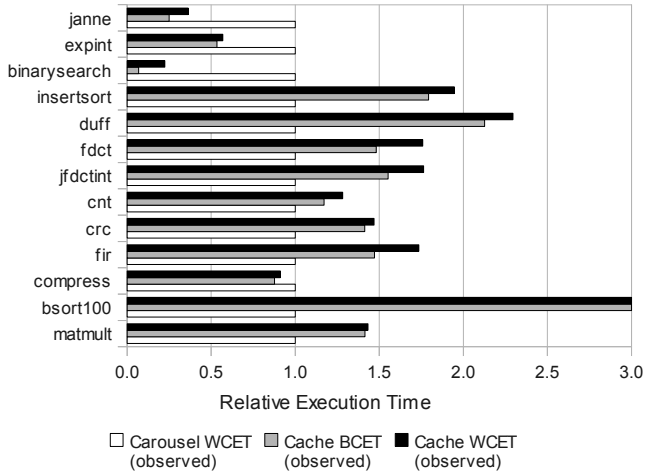

Fig. 13. The relationship between the observed BCETs and WCETs for cache (Table VI) and Carousel (Table V). Tasks are sorted into order of Figure 12 (i.e. the shortest tasks appear at the top) and normalized to the Carousel observed WCET.

Carousel has also improved schedulability versus a cache. Of the one thousand task sets used to generate the data in Table V, only 366 were schedulable with cache for Table VI. The remaining 75% missed at least one deadline.

G. Causes of Improvement

The improved schedulability is not due to the avoidance of inter-task interference. There are two causes, both pleasant consequences of Carousel's design.

Firstly, Carousel permits a task-specific split between code and data, but the cache split is fixed, with exactly 1kb for code and 1kb for data. This is a problem for matmult, where the working set is larger than 1kb. If we double the data cache size, the matmult execution time drops to around 697,000. A cache does not allow any dynamic flexibility of this sort; Carousel does.

Secondly, the write-through policy of the Microblaze cache is causing a significant slowdown. bsort100 performs over 10,000 stores, and the overwhelming majority of these use addresses that are already in cache. However, they must be sent to the external RAM as well, and subsequent operations must wait for them to complete. If we were to substitute a write-back policy [22], the execution time would drop to 94,000 clock cycles.

SPM technology (and thus, Carousel) already operates in write-back mode. This is a natural consequence of requiring tasks to explicitly copy data between external RAM and SPM. It requires no extra hardware. The same is not true for write-back mode in a cache, which adds significant complexity due to the need to track which data in the cache has changed. This data will have to be written back to external RAM if replaced in cache; thus, replacement is a multi-step process, perhaps requiring copying in two directions. This is why the simplest cache policy is used for Microblaze.

Write-back caching also makes WCET analysis more difficult [14]. Intuitively, the timing-related state space of the cache is greatly expanded by the possibility that data may not only be in cache (or not) but also modified (or not).

That said, if a write-back cache were introduced, the results would appear as shown in Table VII. This brings the cache and Carousel results closer together. Carousel remains a great option for some benchmarks (e.g. bsort100, crc), but the gap is closed. Smaller benchmarks are now much better with cache, because the overhead is much lower.

H. Worst-case Blocking

The task sets executed on our prototype system do not share any resources other than Carousel itself. This means that blocking is limited to that incurred by Carousel (section V). Measurements from our prototype system indicate that $B_i = 4321$ for all τ_i except the lowest-priority task τ_N, as $B_N = 0$.

For a task set where resources other than Carousel are shared, $\forall i \neq N.B_i$ is the maximum of 4321 and whatever worst-case blocking time is imposed by the other resources used by τ_i.

VII. IMPROVEMENTS TO CAROUSEL

The basic design of Carousel can be improved in a number of ways. Two simple improvements, already used within our experiments, are discarding read-only data rather than writing it back, and *not* reading uninitialized data (section IV-D).

Another obvious improvement is to place code and data into two separate Carousels. This fits well with the Harvard architecture of many embedded systems CPUs [22]. Furthermore, the code-side Carousel can be explicitly read-only, meaning that *swap_out* never needs to write back its contents. Finally, dividing Carousel in this way allows more blocks to be used, as the number of parallel comparisons is halved.

Carousel blocks are used with an SPM within our experiments, but they can also be used with a cache, by substituting cache for SPM within Figure 9. Rather than *dma_copy*ing during *open*, the *open* method just invalidates all cache blocks within the Carousel block, and code or data is loaded on demand. Instruction and data cache analyses will be needed to determine the WCET, but with the advantage that there is still no inter-task interference. However, *swap_out* will need to save the state of the cache tag store in addition to the block contents. This will increase the cost of *swap_in* and *swap_out*.

This technique is likely to be most advantageous in mixed-criticality systems with both hard real-time and non real-time tasks. But caches are usable in hard real-time systems given appropriate hardware [4], so even hard real-time software development may benefit from the technique.

VIII. CONCLUSION

This paper has presented Carousel, a mechanism to manage local RAM space. Carousel eliminates inter-task interference by requiring each task to save and restore any local RAM that it wishes to use. Thus, the costs of preemption are handled explicitly by the preempting task, instead of being imposed on the preempted tasks.

Our experiments show that Carousel does indeed eliminate inter-task interference. It provides substantially better performance than a cache-architecture machine with a write-through policy, and a comparison with a write-back policy also gives good results. And unlike earlier approaches for eliminating interference, such as cache partitioning, Carousel allows each task to use as much of the local RAM as required. One consequence of this improvement is that task sets are more likely to be schedulable; for instance, of one thousand task sets schedulable with Carousel, only 366 were schedulable with an cache-architecture machine (section VI).

However, Carousel does have unavoidable limitations. Because it swaps memory to and from a stack, it cannot support any scheduling paradigm where task execution is not strictly nested. It requires something similar to Baker's *stack resource protocol* (SRP). There is also a somewhat higher overhead for invoking tasks. Nevertheless, the elimination of inter-task interference brings plenty of advantages.

IX. ACKNOWLEDGMENTS

This work was supported by EPSRC project TEMPO, no. EP/G055548/1. Thanks to Martin Schoeberl and Ian Gray for their comments on drafts, to Robert Davis for his greatly helpful advice on scheduling theory and response time equations, and to the reviewers for their suggestions.

References

[1] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages, 4th Edition*. Addison Wesley, 2009.

[2] J. Stärner and L. Asplund, "Measuring the cache interference cost in preemptive real-time systems," in *Proc. LCTES*, 2004, pp. 146–154.

[3] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools." *Proc. IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

[4] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[5] R. Reddy and P. Petrov, "Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems," in *Proc. CASES*, 2007, pp. 198–207.

[6] S. Altmeyer, R. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proc. RTSS*, 2011, pp. 261–271.

[7] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *Proc. RTAS*, 1996, pp. 204–.

[8] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *Proc. RTAS*, 2006, pp. 71–80.

[9] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[10] O. Temam, "An algorithm for optimally exploiting spatial and temporal locality in upper memory levels," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 150–158, 1999.

[11] T. P. Baker, "Stack-based scheduling of real-time processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–100, 1991.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[13] I. Puaut, "Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems," in *Proc. WCET*, Vienna, Austria, June 2002.

[14] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," in *Proceedings of Embedded Real Time Software and Systems*, 2010.

[15] F. Mueller, "Compiler support for software-based cache partitioning," in *Proc. LCTES*. New York, NY, USA: ACM Press, 1995, pp. 125–133.

[16] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511.

[17] J.-F. Deverge and I. Puaut, "WCET-Directed Dynamic Scratchpad Memory Allocation of Data," in *Proc. ECRTS*, 2007, pp. 179–190.

[18] J. Whitham and N. Audsley, "Implementing Time-Predictable Load and Store Operations," in *Proc. EMSOFT*, 2009, pp. 265–274.

[19] Xilinx, "Microblaze processor reference guide," http://www.xilinx.com/bvdocs/userguides/ug081.pdf, Manual UG081, 2005.

[20] MRTC, "WCET Benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[21] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proc. DAC*, 2001, pp. 690–695.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.