A Predictable Execution Model for COTS-based Embedded Systems

Rodolfo Pellizzoni†, Emiliano Betti‡, Stanley Bak‡, Gang Yao♯, John Criswell‡, Marco Caccamo‡ and Russell Kegley⋆
† University of Waterloo, Canada, rpellizz@uwaterloo.ca
‡ University of Illinois at Urbana-Champaign, USA, {ebetti, sbak2, criswell, mcaccamo}@illinois.edu
♯ Scuola Superiore Sant'Anna, Italy, g.yao@sssup.it
⋆ Lockheed Martin Corp., USA, russell.b.kegley@lmco.com

***Abstract*—Building safety-critical real-time systems out of inexpensive, non-real-time, COTS components is challenging. Although COTS components generally offer high performance, they can occasionally incur significant timing delays. To prevent this, we propose controlling the operating point of each shared resource (like the cache, memory, and interconnection buses) to maintain it below its saturation limit. This is necessary because the low-level arbiters of these shared resources are not typically designed to provide real-time guarantees. In this work, we introduce a novel system execution model, the PRedictable Execution Model (PREM), which, in contrast to the standard COTS execution model, coschedules at a high level all active components in the system, such as CPU cores and I/O peripherals. In order to permit predictable, system-wide execution, we argue that real-time embedded applications should be compiled according to a new set of rules dictated by PREM. To experimentally validate our theory, we developed a COTS-based PREM testbed and modified the LLVM Compiler Infrastructure to produce PREM-compatible executables.**

I. Introduction

Real-time embedded systems are increasingly being built using commercial-off-the-shelf (COTS) components such as mass-produced CPUs, peripherals and buses. Overall performance of mass produced components is often significantly higher than custom-made systems. For example, a PCI Express bus [16] can transfer data three orders of magnitude faster than the real-time SAFEbus [9]. Unfortunately, COTS components are typically designed with little or no attention to worst-case timing guarantees required by real-time systems. Modern COTS-based embedded systems include multiple active components (such as CPU cores and I/O peripherals) that can independently initiate access to shared resources, which, in the worst case, cause contention leading to timing degradation.

Computing precise bounds on timing delays due to contention is difficult. Even though some existing approaches [18], [24] can produce safe upper bounds, they need to be very pessimistic due to the unpredictable behavior of arbiters of physically shared resources (like caches, memories, and buses). As a motivating example, we have previously shown that the computation time of a task can increase linearly with the number of suffered cache misses due to contention for access to main memory [20]. In a system with three active components, a task's worst-case computation time can nearly triple. To exploit the high average performance of COTS components without occasionally experiencing long delays suffered by real-time tasks, we need to control the operating point of each shared resource and maintain it below saturation limits. This work aims at showing that this is indeed possible by carefully rethinking the execution model of real-time tasks and by enforcing a high-level coscheduling mechanism among all active components in the system. Briefly, the key idea is to coschedule active components so that contention for accessing shared resources is implicitly resolved by the high-level coscheduler without relying on low-level, non-real-time arbiters. In particular, in this work we focus on contention at the level of bus communication and main memory. Several challenges had to be overcome to realize the PRedictable Execution Model (PREM):

- I/O peripherals with DMA master capabilities contend for physically shared resources, including memory and buses, in an unpredictable manner. To address this problem, we expand upon on our previous work [1] and introduce hardware to put the I/O subsystem under the discipline of real-time scheduling.
- The bus and memory access patterns of tasks executed on COTS CPUs can exhibit high variance. In particular, predicting a precise pattern of cache fetches in main memory is very difficult, forcing the designer to make very pessimistic assumptions when performing schedulability analysis. To address this problem, PREM uses a novel program execution model with three main features: (1) jobs are divided into a sequence of non-preemptive scheduling intervals; (2) some of these scheduling intervals (named **predictable intervals**) are executed *predictably* and *without cache-misses* by prefetching all required data at the beginning of the interval itself; (3) the execution time of **predictable intervals** is kept constant by monitoring CPU time counters at run-time.
- Low-level COTS arbiters are usually designed to achieve fairness instead of real-time performance. To address this problem, we enforce a coscheduling mechanism that serializes arbitration requests of active components (CPU cores and I/O peripherals). During the execution of a task's predictable interval, a scheduled peripheral can

access the bus and memory without experiencing delays due to cache misses caused by the task's execution.

Our PRedictable Execution Model (PREM) can be used with a high level programming language like C by setting some programming guidelines and by using a modified compiler to generate predictable executables. Based on annotations provided by the programmer, the compiler generates programs which perform cache prefetching and enforce a constant execution time in each predictable interval. The generated executable becomes highly predictable in its memory access behavior, and when run with the rest of the PREM system, shows significantly reduced worst-case execution time. Some work is needed to code an application according to our programming guidelines. However, we argue that our requirements are not significantly stricter with respect to those of state-of-the-art timing analysis, which requires detailed modeling of both the software application and hardware platform to produce tight bounds. While our code annotations are still dependent on architectural features such as the size and associativity of the cache, task execution under PREM becomes independent from low-level bus and memory arbiters, simplifying system design and verification.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes our main contribution: a co-scheduling mechanism that schedules I/O interrupt handlers, task memory accesses and I/O peripheral data transfers in such a way that access to shared resources is serialized, achieving zero or negligible contention during memory accesses. Sections IV and V present our solutions to hardware architecture and code organization challenges that make creating predictable real-time tasks difficult. Section VI presents our schedulability analysis. In Section VII, we detail our prototype testbed, including our compiler implementation based on the LLVM Compiler Infrastructure [10], and provide an experimental evaluation. We conclude with future work in Section VIII.

II. Related Work

Prior real-time research has proposed several solutions to address different sources of unpredictability in COTS components, including real-time handling of peripheral drivers, real-time compilation, and analysis of contention for memory and buses. For peripheral drivers, Facchinetti et al. [5] proposed using a non-preemptive interrupt server to better support the reusing of legacy drivers. Additionally, analysis can be done to model worst-case temporal interference caused by device drivers [12]. For real-time compilation, a tight coupling between compiler and worst-case execution time (WCET) analyzer can optimize a program's WCET [6]. Alternatively, a compiler-based approach can provide predictable paging [21]. For analysis of contention for memory and buses, existing timing analysis techniques can analyze the maximum delay caused by contention for a shared memory or bus under various access models [18], [24]. All these works attempt to analyze or control a single resource and obtain safe bounds that are often highly pessimistic. Instead, PREM is based on a global coschedule of all relevant system resources.

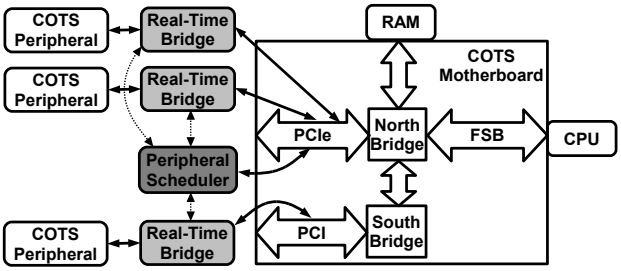

Fig. 1. Real-Time I/O Management System.

Instead of using COTS components, other researchers have discussed new architectural solutions that can greatly increase system predictability by removing significant sources of interference. Instead of a standard cache-based architecture, a real-time scratchpad architecture can be used to provide predictable access time to main memory [28]. The Precision Time (PRET) machine [4] promises to simultaneously deliver high computational performance together with cycle-accurate estimation of program execution time. Unfortunately, these solutions require extensive redesign of existing components (in particular, the CPU). While our PREM execution model borrows some ideas from these works, it is compatible with available COTS platforms: all existing components can be reused, albeit some new devices must be connected to the motherboard and the COTS peripherals. This approach allows PREM to leverage the advantage of the economy of scale of COTS systems, and support the progressive migration of legacy systems.

III. System Model

We consider a typical COTS-based real-time embedded system composed of a CPU, main memory and multiple DMA peripherals. While this paper restricts our discussion to single-core systems with no hardware multithreading, we believe that our predictable execution model is also applicable to multicore systems. The CPU can implement one or more cache levels. We focus on the last cache level, which typically employs a write-back policy. Whenever a task suffers a cache miss in the last level, the cache controller must access main memory to fetch the newly referenced cache line and possibly write-back a replaced cache line. Peripherals are connected to the system through COTS interconnects such as PCI or PCIe [16]. DMA peripherals can autonomously initiate data transfers on the interconnect. We assume that all data transfers target main memory, that is, data is always transferred between the peripheral's internal buffers and main memory. Therefore, we can treat main memory as a single resource shared by all peripherals and by the cache controller[1].

[1]Note that while using a dual-port memory could potentially reduce contention between a single core and peripheral interconnect, implementing a large external memory as a dual-port SRAM is impractical. Furthermore, such a solution does not scale to multiple interconnects and/or cores.

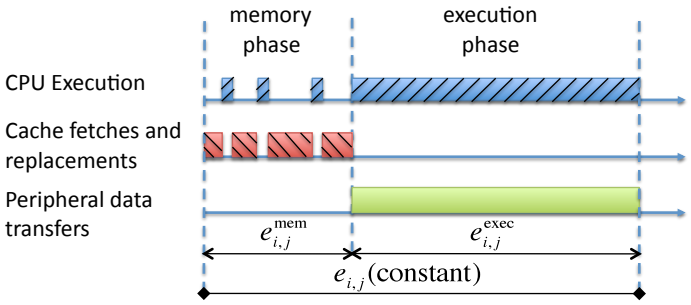

Fig. 2. Predictable Interval with constant execution time.

The CPU executes a set of N real-time periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_N\}$. Each task can use one or more peripherals to transfer input or output data to or from main memory. We model all peripheral activities as a set of M periodic I/O flows $\Gamma^{\text{I/O}} = \{\tau_1^{\text{I/O}}, \ldots, \tau_M^{\text{I/O}}\}$ with assigned timing reservations, and we want to schedule them in such a way that only one flow is transferred at a time. Unfortunately, COTS peripherals do not typically conform to the described model. As an example, consider a task receiving input data from a Network Interface Card (NIC). Delays in the network could easily cause a burst of packets to arrive at the NIC. Since a high-performance COTS NIC is designed to autonomously transfer incoming packets to main memory as soon as possible, the NIC could potentially require memory access for significantly longer than its expected periodic reservation. In [1], [2], we introduced a solution to this problem consisting of a real-time I/O management scheme. A diagram of our proposed architecture is depicted in Figure 1. A minimally intrusive hardware device, called the *real-time bridge*, is interposed between each peripheral and the rest of the system. The real-time bridge buffers all incoming traffic from the peripheral and delivers it predictably to main memory according to a global I/O schedule. Outgoing traffic is also retrieved from main memory in a predictable fashion. Bounds on the necessary buffer sizes so that data loss is avoided can be computed [1]. Our implemented real-time bridges are fully compatible with the PCIe standard and require no modification to either the system chipset or the controlled peripherals. Furthermore, real-time bridges provide traffic virtualization and isolation: a single peripheral can support multiple I/O flows with different timing reservations, each servicing a different task [2]. To maximize responsiveness and avoid CPU overhead, the I/O schedule is computed by a separate *peripheral scheduler*, a hardware device based on the previously-developed [1] reservation controller, which controls all real-time bridges.

Notice that our previously-developed I/O management system [1] does not solve the problem of memory interference between peripherals and CPU tasks. When a typical real-time task is executed on a COTS CPU, cache misses are unpredictable, making it difficult to avoid low-level contention for access to main memory. To overcome this issue, we propose a set of compiler and OS techniques that enable us to predictably schedule all cache misses during a given portion of a task execution. The code for each task τ_i is divided into a set of N_i scheduling intervals $\{s_{i,1}, \ldots, s_{i,N_i}\}$, which are executed sequentially at run-time. The timing requirements of τ_i can be expressed by a tuple $\{\{e_{i,1}, \ldots, e_{i,N_i}\}, p_i, D_i\}$, where p_i, D_i are the period and relative deadline of the task, with $D_i \leq p_i$, and $e_{i,j}$ is the maximum execution time of $s_{i,j}$, assuming that the interval runs in isolation with no memory interference. A job can only be preempted by a higher priority job at the end of a scheduling interval. This ensures that the cache content can not be altered by the preempting job during the execution of an interval. We classify the scheduling intervals into *compatible intervals* and *predictable intervals*.

Compatible intervals are compiled and executed without any special provisions (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt handlers of associated peripherals. We assume that the maximum execution time $e_{i,j}$ for a compatible interval is computed based on static analysis techniques. However, to reduce the pessimism in the analysis, we prohibit peripheral traffic from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially compiled to execute according to the PREM model shown in Figure 2: they are divided into two different phases and exhibit three main properties. First, during the initial *memory phase*, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache. Second, during the following *execution phase*, the task performs useful computation without suffering any last level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. This property allows peripheral traffic to be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory. Third, at run-time, we force the time length of a predictable interval to always be equal to $e_{i,j}$. Let $e_{i,j}^{\text{mem}}$ be the maximum time required to complete the memory phase and $e_{i,j}^{\text{exec}}$ to complete the execution phase. Then offline we set $e_{i,j} = e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$ and at run-time, if the predictable interval completes in less than $e_{i,j}$, we busy-wait until $e_{i,j}$ time units have elapsed since the beginning of the interval. This property ensures that peripherals can transmit for at least $e_{i,j}^{\text{exec}}$ time units in a time window of length $e_{i,j}$. If we did not enforce a constant interval length, then the execution phase could potentially complete in zero time, resulting in no peripheral traffic being sent during that predictable interval. In Section VI we will formally show how we can provide hard real-time guarantees to I/O flows based on the constant interval length property.

Figure 3 shows a concrete example of a system-level predictable schedule for a task set with two tasks τ_1, τ_2 together

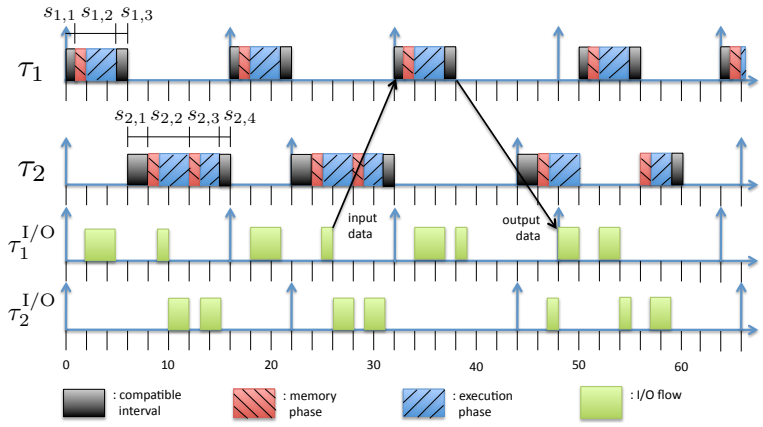

Fig. 3. Example System-Level Predictable Schedule

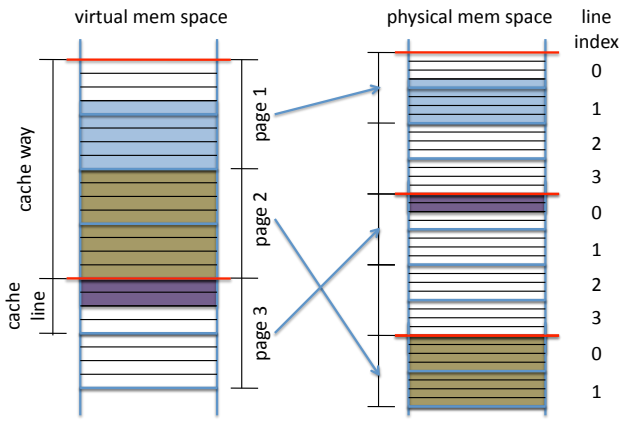

Fig. 4. Cache organization with one memory region

with two I/O flows $\tau_1^{I/O}, \tau_2^{I/O}$ which service τ_1 and τ_2, respectively. Both tasks and I/O flows are scheduled according to fixed priority, with τ_1 having higher priority than τ_2 and $\tau_1^{I/O}$ higher priority than $\tau_2^{I/O}$. We set $D_i = p_i$ and assign to each I/O flow the same period and deadline as its serviced task and a transmission time equal to 4 time units. As shown in Figure 3 for task τ_1, this means that the input data for a given job is transmitted in the period before the job is executed, and the output data is transmitted in the period after. Task τ_1 has a single predictable interval of length $e_{1,2} = 4$ while τ_2 has two predictable intervals of lengths $e_{2,2} = 4$ and $e_{2,3} = 3$. The first and last interval of both τ_1 and τ_2 are special compatible intervals. These intervals are needed to execute the associated peripheral driver (including interrupt handlers) and set up the reception and transmission buffers in main memory (i.e. read and write system calls). More details are provided in Section VII. I/O flows can be scheduled both during execution phases and while the CPU is idle. As we will show in Section VI, the described scheme can be modeled as a hierarchical scheduling system [26], where the CPU schedule of predictable intervals supplies available transmission time to I/O flows. Therefore, existing tests can be reused to check the schedulability of I/O flows. However, due to the characteristics of predictable intervals, a more complex analysis is required to derive the supply function.

Executing a task according to the PREM model reduces its overall execution time because PREM ensures that peripheral traffic in main memory cannot contend with and stall cache fetches; in the worst case, the peripheral-induced delay can be very significant. A possible issue in our approach is that by deciding which cache lines to prefetch during the memory phase, we might need to prefetch more cache lines than the ones that are actually used at run-time in the execution phase. Our experimental evaluation in Section VII shows that for several embedded benchmarks, this increase in memory load is not significant. A second possible issue is that by blocking I/O flows during compatible intervals, we risk reducing peripheral bandwidth significantly. However, in Section VII, we show that for a significant category of benchmarks, the execution time of predictable intervals dominates the total length of the job.

IV. Architectural Constraints and Solutions

Predictable intervals are executed in a radically different way compared to the speculative execution model that COTS components are typically designed to support. In this section, we detail the challenges and solutions to implement the PREM execution model on top of a COTS architecture.

Caching and Prefetch: Our general strategy to implement the memory phase consists of two steps: (1) we determine the complete set of memory regions that are accessed during the interval. Each region is a continuous area in virtual memory. In general, its start address can only be determined at run-time, but its size A is known at compile time. (2) During the memory phase, we prefetch all cache lines that contain instructions and data for required regions; most instruction sets include a `prefetch` instruction that can be used to load specific cache lines in last level cache. Step (1) will be detailed in Section V. Step (2) can be successful only if there is no cache *self-eviction*, that is, prefetching a cache line never evicts another line loaded during the same memory phase. In the remainder of this subsection, we describe self-eviction prevention.

Most COTS CPUs implement the last-level cache as a write-back, N-way set associative cache. Data is loaded into and written-back from cache in units known as *cache lines*. Let B be the total size of the cache and L be the size of each cache line in bytes. An N-way set associative cache is divided into N *cache ways*, each with a size of $W = B/N$ bytes. An *associative set* is the set of all cache lines, one for each of the N cache ways, which have the same index in cache; there are W/L associative sets. Last level cache is typically physically tagged and physically indexed, meaning that cache lines are accessed based on physical memory addresses only. Each cache line in main memory is associated with a single associative set based on its index in the physical addess space, but can be loaded into any of the N ways; the specific way is chosen at run-time by the cache *replacement policy*. We also assume that last level cache is not exclusive, that is, when a cache line is copied to a higher cache level, it is not removed

from the last level. Figure 4 shows an example of indexing in main memory where $L = 4, W = 16$ (parameters are chosen to simplify the discussion and are not representative of typical systems). The main idea behind our conflict analysis is as follows: we compute the maximum number of entries in each associative set that are required to hold the cache lines prefetched for all memory regions in a scheduling interval. Based on the cache replacement policy, we then derive a safe lower bound on the number of entries that can be prefetched in an associative set without causing any self-eviction.

Consider a memory region of size A. As shown in Figure 4 for a region with $A = 15, K = 5$, the worst case number of occupied cache lines is produced when the region uses a single byte in its first cache line. The remaining $A-1$ bytes will then occupy $\lceil \frac{A-1}{L} \rceil$ other cache lines. Hence, the region requires at most $K = 1 + \lceil \frac{A-1}{L} \rceil$ cache lines. Assume now that virtual memory addresses coincide with physical addresses. Since the region is contiguous and there are W/L cache lines in each way, the maximum number of entries used in any associative set by the region is $\lceil \frac{K}{W/L} \rceil$. For example, the region in Figure 4 requires two entries in the set with index 0. We then derive the maximum number of entries for the entire interval by summing the entries required for each memory region. Unfortunately, this is not generally true if the system employs paged virtual memory. If the page size P is smaller than the size W of each way, the index of each cache line inside the cache way is different for virtual and physical addresses. In the example of Figure 4 with $P = 8$, the number of entries for the memory region is increased from 2 to 3. We consider two solutions: 1) if the system supports it, we can select a page size multiple of W just for our specific process. This solution, which we employed in our implementation, solves the problem because the index in cache for virtual and physical addresses is the same no matter the page allocation. 2) We use a modified page allocation algorithm in the OS. Note that a suitable allocation algorithm could decrease the required number of associative entries by controlling the allocation in physical memory of multiple regions. We will pursue this solution in future work.

Due to space constraints, a thorough discussion of cache replacement policies is provided in [17]. Let Q be the maximum number of entries required by the scheduling interval in any N-way associative set. Based on the results in [8], [23], in our companion technical report [17] we show the following:

Theorem 1. *A memory phase will not suffer any cache self-eviction if Q is at most equal to:*

- *N: for FIFO or LRU replacement policy;*
- *$\log_2 N + 1$: for pseudo-LRU replacement policy;*
- *1: for random replacement policy.*

Computing Phase Length: We assume that traditional static analysis can be employed to derive bounds on the maximum time required for an execution phase $e_{i,j}^{\text{exec}}$, as well as on the execution time $e_{i,j}$ for a compatible interval. Obtaining tight bounds on WCET can be challenging in COTS cores exploiting architectural features such as deep pipelines and intermediate cache levels. We believe that the problem of intra-core execution analysis is fundamentally orthogonal to our approach; the role of PREM is to simplify timing analysis by making cache miss patterns predictable. Research in static timing analysis for complex core architectures is very active. For example, analyses that derive patterns of cache misses for split level 1 data and instruction caches have been proposed in [14], [22], while the work in [13] derives tight execution bounds by analyzing the pipeline status.

The length $e_{i,j}^{\text{mem}}$ of a memory phase depends on the time required to prefetch all accessed memory regions. An analysis to compute upper bounds for read/write operations using a COTS DRAM memory controller is detailed in [15]. Note that DRAM access times are significantly dependent on both space and time locality of data in main memory. Therefore, the current analysis must make pessimistic guesses. However, since the number and relative virtual addresses of prefetched cache lines is known under PREM, the analysis could potentially be extended to consider features such as burst read and parallel bank access; we plan to investigate this direction in future work. Similarly, we expect that static analysis will significantly overestimate the execution time of compatible intervals; however, as previously mentioned, our experimental results in Section VII show that compatible intervals are typically short.

An important note is relative to the write-back mechanism of last level cache. Since the task can be preempted at the boundary of scheduling intervals, the cache state is unknown at the beginning of the memory phase. Hence, in the analysis we must account that each prefetch could require both a cache line fetch and a replacement. An alternative solution would be to add a second memory phase after each execution phase and invalidate the whole cache, forcing write-back of all dirty cache lines. We have not pursued this solution because our implemented testbed does not allow full-cache invalidation.

Finally, for systems implementing paged virtual memory, we employ the following three assumptions: 1) the CPU supports hardware Translation Lookaside Buffer (TLB) management; 2) all pages used by predictable intervals are locked in main memory; 3) the TLB is large enough to contain all page entries for a predictable interval without suffering any conflict. Under such assumptions, each page used in a predictable interval can cause at most one TLB miss during the memory phase, which requires a number of fetches in main memory equal to at most the level of the page table.

Scheduling synchronization: In our model, peripherals are only allowed to transmit during a predictable interval's execution phase or while the CPU is idle. To compute the peripheral schedule, the peripheral scheduler must thus know the status of the CPU schedule. Synchronization is achieved by connecting the peripheral scheduler to a peripheral interconnection as shown in Figure 1; scheduling messages containing the amount of consecutive time in which peripherals are allowed to transmit are then sent by either a task or the OS to the peripheral scheduler. In particular, at the end of each memory phase the task sends to the peripheral scheduler the remaining amount of time until the end of the current predictable interval.

Note that to simplify the discussion we do not consider issues of message propagation delay[2] and clock drift in this paper, but the described scheme together with the schedulability analysis of Section VI could be easily modified by suitably reducing the time allowed for peripheral traffic compared to the value contained in the scheduling message.

Finally, to avoid executing interrupt handlers during predictable intervals, a peripheral should only raise interrupts to the CPU during compatible intervals of its serviced task. As we describe in Section VII, in our I/O management scheme, peripherals raise interrupts through their assigned real-time bridge. Since the peripheral scheduler communicates with each real-time bridge, it is used to block interrupt propagation outside the desired compatible intervals. Scheduling messages are again used to notify the peripheral scheduler of the length of interrupt-enabled intervals. Note that blocking real-time bridge interrupts to the CPU will not cause any loss of input data because the real-time bridge is capable of independently acknowledging the peripheral and storing all incoming data in the bridge local buffer.

V. Programming Model

Our system supports existing applications written in standard high-level languages such as C. Unmodified code can be executed within one or more compatible intervals. We extend the source language with a `predictable` block construct that defines a single-entry, single-exit region of code that should execute as a single predictable interval. In C, we define the construct as the keyword `predictable` followed by a compound block of statements.

During compilation, the PREM real-time compiler transforms code within a predictable block so that it first prefetches any data and code required in the predictable block into the cache; it also adds a busy-wait loop at the end of the predictable block to ensure that every execution of the predictable interval takes the same amount of time. This ensures that no cache misses occur during the execution phase and that the interval itself has a constant execution time.

In order to create a predictable interval, the programmer should first profile the code to determine the portions in which the task spends most of its execution time. The programmer should then perform the cache and architecture analysis as presented in Section IV for these portions of ”hot” code and then, based on the results and task information, place portions of the code into predictable blocks.

Since it is difficult to compile arbitrary code so that it does not induce cache misses, there are several constraints that the compiler must place on code within predictable blocks. These constraints are:

1) Predictable code blocks should only access memory objects, arrays, and scalar values that are capable of being referenced at the entry of the predictable block. There should be no traversal of link-based data structures (e.g., a binary tree) since the compiler cannot infer the memory that would need to be prefetched.
2) The code can use data structures, in particular arrays, that are allocated outside the predictable code block[3]. For global or heap allocated arrays, the programmer must specify the first and last address that is accessed within the predictable code block and (if necessary) the maximum difference between these two addresses if the compiler cannot infer this information via static analysis. This must be done for the code in the predictable block and for code within functions that are called (either directly or transitively) by the code in the predictable block. The compiler needs this information to add correct prefetching code to the predictable code block.
3) Code within a predictable block should not contain system calls, calls to heap allocators, or stack allocations within loops. System calls enter the kernel and execute code not generated by the PREM compiler, and the heap allocator executes code that can generate cache misses. Stack allocations cannot occur in loops because the compiler must insert code to prefetch the stack frame at the beginning of a predictable interval; stack allocations in loops make the stack frame size unpredictable.
4) Code within a predictable block should not make recursive function calls. Recursive function calls can grow the stack frame to an unpredictable size, making it impossible for the compiler to prefetch the stack frame.
5) Code within a predictable block may use both direct and indirect function calls. The compiler can use points-to analysis combined with call-graph construction [11] to find all the targets of indirect function calls. Since points-to analysis may yield conservative results, the compiler may find more function targets than are actually possible. If too many function targets are found (making construction of a predictable interval impossible), the compiler may ask the programmer to use annotations to specify the valid function targets.

The compiler employs several transforms to ensure that code within predictable blocks does not cause cache misses during the execution phase. First, the compiler inlines all functions called (either directly or transitively) by the predictable block into the predictable block (link-time optimization can inline functions across compilation units). This ensures that all code used by the predictable block is contiguous within virtual memory and uses a single stack frame. Second, the compiler inserts code at the beginning of the predictable block to prefetch the code and data needed to execute the interval; this prefetching code is the memory phase of the predictable interval. Based on the described constraints, this includes three types of contiguous memory regions: (1) the code for the function; (2) the actual parameters passed to the function and the stack frame (which contains local variables and register spill slots); and (3) the global and heap memory objects

[2]In our implementation we measured an upper bound to the message propagation time of 1μs, while we envision scheduling intervals with a length of 100-1000μs.

[3]Note that heap objects must have been allocated during a compatible interval.

accessed within the predictable block. Third, the compiler inserts code to send scheduling messages to the peripheral scheduler as described in Section IV. Finally, the compiler emits code at the end of predictable block to enforce its constant, predictable execution time.

A. Porting Legacy Applications

Converting existing code to predictable intervals clearly requires some work. In particular, adding annotations to correctly split the code into predictable blocks requires some knowledge of cache parameters. While this might seem an undue limit on code portability, the type of data-intensive, real-time applications that we target in this work are typically already optimized based on hardware architecture. In this sense, we believe that the benefits of a more predictable behavior for program hot-spots, decoupled from the low-level details of bus and memory arbiters, outweigh the burden of code annotations. Additionally, we can design our compiler to help the programmer create predictable code blocks. The compiler can verify when the aforementioned restrictions are violated in a predictable block (e.g., it can use static analysis to find irregular data structure usage or use of system calls) and issue warnings to aid the programmer in correcting them. Furthermore, given cache size information, the compiler could verify that all prefetched memory regions fit in last level cache and issue warnings otherwise.

A second possible concern regards code constraints. In general, we believe that our constraints are not significantly more restrictive than those imposed by state-of-the-art static timing analysis. Typical real-time applications avoid recursive calls, stack or heap allocation within loops, and indirect function calls that are not decidable at compile time. Furthermore, we are not aware of any timing analysis tool that can provide WCET bounds if Constraints 3 or 4 are violated.

Constraints 1-2 are more severe because they prevent using complex pointer-based data structures. However, existing code that is too complex to be compiled into predictable intervals can still be executed inside compatible intervals. An alternative solution is presented in [25]: the cache is statically partitioned, for example using the OS page allocator, into an area for predictable code and data and a second area for complex, unpredictable data structures. During a predictable interval, the predictable area is prefetched while unpredictable data is handled by the caching logic. Static analysis can then be used to derive a (pessimistic) upper bound to the number of cache misses in the unpredictable area; ideally, most of the data would be placed in the predictable area, resulting in a very small number of unpredictable misses. While we do not discuss it in detail, the PREM model could be amended to tolerate a small number of cache misses in the predictable phase by using the analysis in [18], [19] to compute the (limited) contention delay on both the task and I/O flows.

VI. Schedulability Analysis

PREM allows us to enforce strict timing guarantees for both CPU tasks and their associated I/O flows. By setting timing parameters as shown in Figure 3, the task schedule becomes independent of I/O scheduling. Therefore, task schedulability can be checked using available schedulability tests. As an example, assume that tasks are scheduled according to fixed priority scheduling as in Figure 3. For a task τ_i, let $e_i = \sum_{j=1}^{N_i} e_{i,j}$ be the sum of the execution times of its scheduling intervals, or equivalently, the execution time of the whole task. Furthermore, let $\mathrm{hp}_i \subset \Gamma$ be the set of higher priority tasks than τ_i, and lp_i the set of lower priority tasks. Since scheduling intervals are executed non-preemptively, τ_i can suffer a blocking time due to lower priority tasks of at most $B_i = \max_{\tau_l \in \mathrm{lp}_i} \max_{j=1\ldots N_l} e_{l,j}$. The worst-case response time of τ_i can then be found [3] as the fixed point r_i of the following iteration, starting from $r_i^0 = e_i + B_i$:

$$r_i^{k+1} = e_i + B_i + \sum_{l \in \mathrm{hp}_i} \left\lceil \frac{r_i^k}{p_l} \right\rceil e_l, \tag{1}$$

Task set Γ is schedulable if $\forall \tau_i : r_i \leq D_i$.

We now turn our attention to peripheral scheduling. Note that due to space limitations, proofs and a more detailed discussion are provided in [17]. Assume that each I/O flow $\tau_i^{\mathrm{I/O}}$ is characterized by a maximum transmission time $e_i^{\mathrm{I/O}}$ (with no interference in both main memory and the interconnect), period $p_i^{\mathrm{I/O}}$ and relative deadline $D_i^{\mathrm{I/O}}$, where $D_i^{\mathrm{I/O}} \leq p_i^{\mathrm{I/O}}$. The schedulability analysis for I/O flows is more complex because the scheduling of data transfers depends on the task schedule. To solve this issue, we extend the hierarchical scheduling framework proposed by Shin and Lee in [26]. In this framework, tasks/flows in a *child scheduling model* execute using a timing resource provided by a *parent scheduling model*. Schedulability for the child model can be tested based on the *supply bound function* $\mathrm{sbf}(t)$, which represents the minimum resource supply provided to the child model in any interval of time t. In our model, the I/O flow schedule is the child model and $\mathrm{sbf}(t)$ represents the minimum amount of time in any interval of time t during which the execution phase of a predictable interval is scheduled or the CPU is idle. Define the *service time bound function* $\mathrm{tbf}(t)$ as the pseudo-inverse of $\mathrm{sbf}(t)$, that is, $\mathrm{tbf}(t) = \min\{x | \mathrm{sbf}(x) \geq t\}$. Then if I/O flows are scheduled according to fixed priority, in [26] it is shown that the response time $r_i^{\mathrm{I/O}}$ of flow $\tau_i^{\mathrm{I/O}}$ can be computed according to the iteration:

$$r_i^{\mathrm{I/O},k+1} = \mathrm{tbf}\Big(e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \left\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \right\rceil e_l^{\mathrm{I/O}}\Big), \tag{2}$$

where $\mathrm{hp}_i^{\mathrm{I/O}}$ has the same meaning as hp_i. In the remainder of this section, we detail how to compute $\mathrm{sbf}(t)$.

For the sake of simplicity, let us initially assume that tasks are strictly periodic and that the initial activation time of each task is known. Furthermore, notice that using the solution described in Section IV, we could enforce interval lengths not just for predictable intervals but also for all compatible intervals. Finally, let h be the hyperperiod of task set Γ, defined as the least common multiple of all tasks' periods.

Under these assumptions, it is easy to see that if Γ is feasible, the CPU schedule can be computed offline and repeats itself identically with period h after an initial interval of $2h$ time units (h time units if all tasks are activated simultenously). Therefore, a tight $\mathrm{sbf}(t)$ can be computed as the minimum amount of supply (time during which the CPU is idle or in execution phase of a predictable interval) during any interval of time t in the periodic task schedule, starting from the initial two hyperperiods. More formally, let $\{t^1, \ldots, t^K\}$ be the set of start times for all scheduling intervals activated in the first two hyperperiods; $\mathrm{sbf}(t)$ is derived in the following theorem.

Theorem 2. *Let* $\mathrm{sf}(t', t'')$ *be the amount of supply provided in the periodic task schedule during interval* $[t', t'']$. *Then:*

$$\mathrm{sbf}(t) = \min_{k=1\ldots K} \mathrm{sf}(t^k, t^k + t). \tag{3}$$

Unfortunately, the proposed $\mathrm{sbf}(t)$ derivation can only be applied to strictly periodic tasks. If Γ includes any sporadic task τ_i with minimum interarrival time p_i, the schedule cannot be computed offline. Therefore, in [17], we detail an alternative analysis that is independent of the CPU scheduling algorithm and computes a lower bound $\mathrm{sbf}_L(t)$ to $\mathrm{sbf}(t)$. Note that since $\mathrm{sbf}(t)$ is the minimum supply in any interval t, using Equation 2 with $\mathrm{sbf}_L(t)$ instead of $\mathrm{sbf}(t)$ will still result in a sufficient schedulability test. The analysis first computes $\mathrm{sbf}_L(t)$ in a finite set of time points by solving a linear optimization problem and then derives $\mathrm{sbf}_L(t)$ for all values of t using interpolation. The overall time complexity of the analysis is $O\big(N^2 \max_i D_i^{\mathrm{I/O}} / \min_i p_i\big)$.

VII. Evaluation

In order to verify the validity and practicality of PREM, we implemented the key components of the system. In this section, we first describe our implemented testbed. We then discuss our compiler implementation and analyze its effectiveness on several embedded benchmarks. Finally, using synthetic tasks, we measure the effectiveness of the PREM system as a function of cache stall time.

A. PREM Hardware/Software Testbed

To support I/O flow scheduling, we developed a real-time bridge and peripheral scheduler prototype as described in Section III. Due to space limitations, a detailed description of the implemented hardware components, based on FPGA prototyping boards, is provided in [17]. Compared to the previous work in [1], [2], the new components exhibit two main differences: (1) an additional `interrupt_block` wire between the real-time bridge and peripheral scheduler is used to control interrupt propagation as detailed in Section IV; (2) the peripheral scheduler is connected to the PCIe bus and exposes a set of registers accessible from the main CPU. In particular, the `yield` register is used to receive scheduling messages as described in Section IV. Both the real-time bridge and the peripheral scheduler require software drivers to be controlled from the main CPU and interact with each peripheral. The driver for the peripheral scheduler is extremely simple, exposing to the CPU the peripheral scheduler's registers. The driver for each real-time bridge is more difficult, since each unique COTS peripheral requires a unique driver. However, since, in our implementation, we employ Linux (version 2.6.31), we can reuse existing, thoroughly tested Linux drivers to drastically reduce the driver creation effort [1]. The presence of a real-time bridge is not apparent in user space, and software programs using the peripherals require no modification.

For our experiments, we use an Intel Q6700 CPU with a 82975X system controller; we set the CPU frequency to 1Ghz obtaining a measured memory bandwidth of 1.8Gbytes/s to configure the system in line with typical values for embedded systems. We also disable the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q6700 has four CPU cores and each pair of cores shares a common level 2 (last level) cache. Each cache is 16-associative with a total size of $B = 4$ Mbytes and a line size of $L = 64$ bytes; reloading the whole cache requires roughly 2.2 ms. Since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads, and other processes, which must be removed for our measurements to be meaningful. For this reason, in order to best emulate a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The *system partition*, running on the first pair of cores, receives all interrupts for non-critical devices (e.g., the keyboard) and runs all the system activities and non real-time processes (e.g., the shell we use to run the experiments). The *real-time partition* runs on the second pair of cores. One core in the real-time partition runs our real-time tasks together with the drivers for real-time bridges and the peripheral scheduler; the other core is turned off. Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses or raise rare inter-processor interrupts (IPI) that cannot be easily prevented. However, in our experiments, we found these sources of noise to be negligible. Finally, to solve the paging issue detailed in Section IV, we used a large 4MB page size just for the real-time tasks using the HugeTLB feature of the Linux kernel for large page support.

B. Compiler Evaluation

We built a PREM real-time C compiler prototype using the LLVM Compiler Infrastructure [10]. We extended LLVM by writing self-contained analysis and transformation passes which were then loaded into the compiler. For simplicity, in the current compiler prototype, interval partitioning is performed by putting each predictable interval into a separate function. Within the predictable interval function, the programmer adds macros that 1) indicate that non-local data should be prefetched (`PREFETCH_DATA(start_address, size)`) and 2) indicate that the execution phase is beginning and send scheduling messages (`START_EXECUTION(WCET)`).

Our new LLVM compiler pass performs all remaining

Input bytes	4K	8K	32K	128K	512K	1M
Non-PREM miss	151	277	1046	4144	16371	32698
PREM prefetch	255	353	1119	4185	16451	32834
PREM exec-miss	1	1	1	1	1	104

TABLE I
DES BENCHMARK CACHE MISSES.

	PREM			Non-PREM	
	prefetch	exec-miss	time(μs)	miss	time(μs)
JPEG(1 Mpix)	810	13	778	588	797
JPEG(8 Mpix)	1736	19	3039	1612	3110
qsort	3136	3	2712	3135	2768
susan_smooth	313	2	7159	298	7170
susan_edge	680	4	3089	666	3086
susan_corner	3286	3	341	598	232

TABLE II
MIBENCH RESULTS WITHOUT PERIPHERAL TRAFFIC.

operations needed to transform the interval to be predictable. When a function representing a predictable interval is found, our transform first inlines all functions called within the predictable interval function. This ensures that there is only a single stack frame and segment of code that needs to be prefetched into the cache. Second, our transform inserts code to read and record the processor's cycle counter at the beginning of the interval. Third, it inserts code to prefetch the stack frame and function arguments by prefetching memory between the stack pointer and slightly beyond the frame pointer (to include function arguments) using the `prefetcht2` instruction. Fourth, the transform prefetches the code of the function by deriving pointers to the beginning and end of the predictable function and then using the `prefetcht2` instruction. Finally, the pass identifies all return instructions inside the predictable interval function and adds a special function epilog before them. The epilog performs interval-length enforcement by looping until the cycle counter reaches the worst-case cycle count based on the time value saved at the beginning of the interval and the predictable interval length (`WCET`) provided in `START_EXECUTION`.

To verify the correctness of the PREM compiler and to test its applicability, we used LLVM to compile several benchmarks mostly taken from MiBench [27], a commercially-representative embedded benchmark suite. First, we elaborate on a DES cypher benchmark because its pattern of cache misses in the various PREM phases is representative of the other benchmarks we tested. Second, we discuss a JPEG benchmark because it has a larger, more complicated code base, and we believe the workflow to make it PREM-compliant will match realistic PREM applications. Lastly, we discuss the entire automotive program group of MiBench (6 benchmarks) to evaluate the broader necessity and feasibility of PREM compilation. We ran all benchmarks with multiple input data sizes and have shown here representative measurements using the "input_small" files from the benchmarks. No peripheral traffic occurs during execution. To capture worst-case behavior, the cache was invalidated prior to the start of each measured function by using a hand-written `cache_trash` function.

The **DES Cypher Benchmark** is composed of one scheduling interval which encrypts a variable amount of data. We compiled it both in the standard, non-PREM way and also with PREM prefetching, and measured the number of cache misses and prefetches which occurred by using a CPU performance counter. Adapting the interval required no modification to any cypher functions and a total of 11 `PREFETCH_DATA` macros.

As shown in Table I, non-PREM execution results in a significant number of cache misses throughout the interval, which increases roughly proportionally to the amount of processed data. If I/O peripherals were to transmit to main memory concurrently, the task's execution time would increase. Conversely, the execution phase (after prefetch) of the predictable interval has almost zero cache misses, only suffering a small increase when a large amount of data is being processed. This demonstrates the key result: with PREM, I/O peripherals can communicate with main memory freely during the execution phase *without affecting the timing of the executing task*.

The reason the number of cache misses is not exactly zero in the PREM execution phase is that the Q6700 CPU core used in our experiments uses a random cache replacement policy, meaning that with more than one contiguous memory region, the probability of self-eviction is non-zero. In our experiments, we observed that the number of self-evictions is usually small. However, we still recommend that timing-critical applications avoid CPUs with a random cache replacement policy. Many embedded platforms used in safety-critical markets such as avionics use processors like the Freescale PowerPC family [7] with more predictable policies like pseudo-LRU [23].

A typical PREM code augmentation workflow was exemplified by the **JPEG Image Encoding Benchmark**. In this benchmark, we first used `gprof` to find that around 80% of the execution time is spent in the `compress_data()` function which performs DCT transformation, quantization and Huffman encoding. We made `compress_data()` PREM-compliant by replacing constant function pointers with direct calls, adding 18 `PREFETCH_DATA` macros, and removing `fwrite` system calls from the predictable interval. The results for two image sizes are shown in Table II, where `time(`μ`s)` represents the execution time of the whole interval.

We also went through the complete **Automotive Program Group** of MiBench and evaluated each of the six benchmarks to determine the broader necessity and feasibility of PREM compilation. Two of the benchmarks (basicmath and bitcount) were not data intensive, so PREM was not necessary. Of the remaining four benchmarks, three (qsort, susan_smooth, susan_edge) were found to be well-suited for PREM, and we were able to perform all computation inside predictable intervals. The final benchmark (susan_corner) had variable-size output so PREM would typically need to prefetch more buffer space than was actually used. The results are again shown in Table II. Note that, except for susan_corner, the number of prefetched cache lines is only slightly higher than the number of cache misses suffered in the non-PREM way.

By evaluating several benchmarks, common strategies emerged to meet the PREM coding constraints (from Section V). Similar to the JPEG benchmark, function pointers often pointed to a single function for a specific benchmark execution and could be replaced with a direct call (as with the qsort benchmark's `compare` function). In the susan benchmarks, function calls such as `malloc`, `memcpy`, and `memset` were moved before the predictable interval function. Furthermore, since our current compiler implementation cannot inline library functions, the implementation of selected libc functions was copied or rewritten to be local functions.

C. WCET Experiments

In this section, we evaluate the effects of PREM on the execution time of a task. To quickly explore different execution parameters, we developed two synthetic applications. In our `linear_access` application, each scheduling interval operates on a 256-kilobyte global data structure. Data is accessed sequentially, and we vary the amount of computation performed between memory references. The `random_access` application is similar, except that references inside the data structure are nonsequential. Both applications access all cache lines of the global data structure at run-time. For each application, we measured the execution time after compiling the program in two ways: into predictable intervals which prefetch the accessed memory, and into standard, compatible intervals. For each type of compilation, we ran the experiment in two ways, with and without I/O traffic transmitted by an 8-lane PCIe peripheral with a measured throughput of 1.2Gbytes/s. In the case of compatible intervals, we transmitted traffic during the entire interval to mirror the worst case according to the traditional execution model.

Figures 5 and 6 show the observed execution time for the scheduling interval as a function of the cache stall time of the application, averaged over 10 runs. The cache stall time represents the percentage of time required to fetch cache lines out of an entire compatible interval, assuming a fixed (best-case) fetch time based on the maximum measured main-memory throughput. Only a single line is shown for predictable intervals because experiments confirmed that injecting traffic during the execution phase does not increase execution time. In all cases, the computation time decreases with an increase in stall time. This effect is an artifact of our implementation: stall time is controlled by varying the amount of computation between memory references while the overall number of cache misses is kept constant. Hence, for lower stall time values, the task must execute for longer amounts of time. Furthermore, execution times should not be compared between the two figures because the two applications execute different code. Finally, note that reported execution times should be interpreted as average execution times rather than worst case measures; each scheduling interval comprises thousands of cache misses, and capturing the worst case interference among all cache fetches and peripheral transactions in main memory is overly difficult.

In the `random_access` case, predictable intervals outperform compatible intervals (without peripheral traffic) by up to 28%, depending on the cache stall time. We believe this effect is primarily due to the behavior of DRAM main memory. Specifically, accesses to adjacent addresses can be served quicker in burst mode than accesses to random addresses. Thus, using PREM can actually decrease the total execution time by loading all the accessed memory into cache, in order, at the beginning of each predictable interval. Furthermore, note that transmitting peripheral traffic during a compatible interval can increase execution time by more than 60% in the worst case. In Figure 6, predictable intervals perform worse than compatible intervals (without peripheral traffic). We believe this is mainly due to out-of-order execution in the Q6700 core. In compatible intervals, while the core performs a cache fetch, instructions in the pipeline that do not depend on the fetched data can continue to execute. When performing linear accesses, fetches require less time and this effect is magnified. Furthermore, the gain in execution time for the case with peripheral traffic is decreased: this occurs because bursting data on the memory bus reduces the amount of blocking time suffered by a task due to peripheral interference (this effect has been previously analyzed in detail [18]). In practice, we expect the effect of PREM on an application's execution time to be between the two figures, depending on the specific application's memory access pattern.

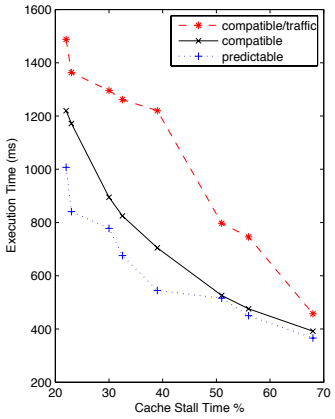

Fig. 5. `random_access`

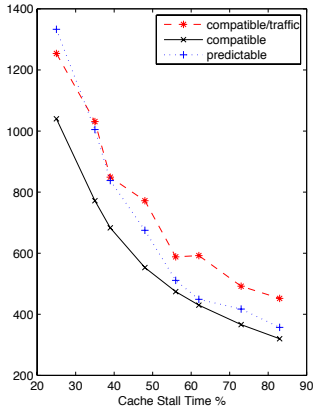

Fig. 6. `linear_access`

Additionally, we have also performed extensive experiments to validate the correctness of the scheduling message and interrupt blocking mechanisms, as well as obtained bus-level traces of PREM running on our testbed [17].

VIII. Conclusions

We have discussed the concept and implementation of a novel task execution model, PRedictable Execution Model (PREM). Our evaluation, based on embedded benchmarks, proves that by enforcing a high-level co-schedule among CPU tasks and peripherals, PREM can greatly reduce or outright eliminate low-level contention for shared resource access. In particular, experiments based on synthetic applications reveal that removing contention for access to main memory can

prevent dangerous spikes of up to 60% in the execution time of critical tasks.

We plan to further develop our solution in two main directions. First of all, the current PREM programming model requires a non-trivial effort by the programmer. While we believe that for a significant class of real-time applications, the advantages in terms of predictability and ease of verification overshadow the required effort in porting an application to PREM, this is nonetheless a source of concern. Ease of programmability could be improved in several different ways. As detailed in Section V, the compiler could be extended to both automatically check the correct application of the PREM rules and to perform cache analysis to ensure that all data in a predictable interval can be prefetched without causing self-eviction. Furthermore, based on our analysis of the MiBench suite, we believe that the compiler could be instrumented to find targets for all indirect function calls without needing to manually change all function pointers. We also believe that a set of C language constructs and (formal) coding rules could be established to simplify some aspects of development, in particular regarding the management of data buffers.

As a second research direction, it is important to recall that contention for shared resources becomes more severe as the number of active components increases. In particular, worst-case execution time can greatly degrade in multicore systems [20]. Since PREM can make the system contentionless, we predict that the benefits of our approach will become even more significant when applied to multiple processor systems. The key idea is to apply PREM to each processor, synchronizing their execution such that no two concurrent tasks access the same shared resource at the same time. Realizing such a vision will require overcoming several challenges and devising new solutions, including efficient synchronization mechanisms, novel multiprocessor scheduling algorithms and memory partitioning schemes.

ACKNOWLEDGMENT

The authors would like to thank Vikram Adve for his insightful suggestions on the programming model and PREM compiler. The material presented in this paper is based upon work supported by Lockheed Martin and NSF under Award No. CNS-0720512 and CNS-0720702. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF or Lockheed Martin.

REFERENCES

[1] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time control of I/O COTS peripherals for embedded systems. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, Washington DC, Dec 2009.

[2] E. Betti. *Satisfying hard real-time constraints using COTS components*. PhD thesis, University of Rome "Tor Vergata", June 2010.

[3] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[4] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proc. of the 44th annual Design Automation Conference*, 2007.

[5] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *ECRTS '05: Proc. of the 17th Euromicro Conf. on Real-Time Systems*, pages 98–105, 2005.

[6] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a wcet-aware c compiler. In *ESTMED '06: Proc. of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 121–126, 2006.

[7] Freescale. *MPC8560 PowerQUICCIII Communication Processor Reference Manual*, 2004. http://cache.freescale.com/files/32bit/doc/ref_manual/MPC8560RM.pdf.

[8] D. Grund and J. Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, July 2010.

[9] K. Hoyme and K. Driscoll. Safebus(tm). *IEEE Aerospace Electronics and Systems Magazine*, pages 34–39, Mar 1993.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the International Symposium of Code Generation and Optimization*, Mar 2004.

[11] Chris Lattner, Andrew D. Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, 2007.

[12] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability: Study of a network driver. In *Proc. of the* 13^{th} *IEEE Real Time Application Symposium*, Apr 2007.

[13] S. Mohan and F. Mueller. Merging state and preserving timing anomalies in pipelines of high-end processors. In *Proc. of the* 29^{th} *IEEE Real-Time System Symposium*, December 2008.

[14] F. Mueller. Timing analysis for instruction caches. *Real Time Systems Journal*, 18(2/3):272–282, May 2000.

[15] M Paolieri, E Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded System Letter*, 1(4), Dec 2009.

[16] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. http://www.pcisig.com.

[17] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, and M. Caccamo. Predictable execution model: concept and implementation. Technical report, University of Illinois at Urbana-Champaign, June 2010. http://www.ideals.illinois.edu/handle/2142/16605.

[18] R. Pellizzoni and M. Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Trans. on Computers*, 59(3):400–415, Mar 2010.

[19] R. Pellizzoni, M. Nam, R. Bradford, and L. Sha. A network calculus based analysis for the PCI bus. Technical report, University of Illinois at Urbana-Champaign, January 2009.

[20] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, Mar 2010.

[21] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *ECRTS '07: Proc. of the 19th Euromicro Conf. on Real-Time Systems*, pages 169–178, 2007.

[22] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proc. of the IEEE Real-Time Embedded Technology and Application Symposium*, Apr 2006.

[23] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2), 2007.

[24] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *CODES/ISSS*, 2008.

[25] M. Schoeberl, W. Puffitsch, and B. Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, Nov 2009.

[26] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of Proceedings of the 23th IEEE Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[27] University of Michigan at Ann Arbor. *MiBench Version 1.0*, 2001. http://www.eecs.umich.edu/mibench/.

[28] J. Whitham and N. Audsley. Implementing time-predictable load and store operations. In *Proc. of the Intl. Conf. on Embedded Systems (EMSOFT)*, Grenoble, France, Oct 2009.