Predictable Execution Model: Concept and Implementation

Article · May 2012

Source: OAI

CITATION
1

READS
76

6 authors, including:

Emiliano Betti
Epigenesys s.r.l., Rome, Italy
11 PUBLICATIONS **370** CITATIONS
SEE PROFILE

Gang Yao
Scuola Superiore Sant'Anna
17 PUBLICATIONS **803** CITATIONS
SEE PROFILE

Marco Caccamo
University of Illinois, Urbana-Champaign
128 PUBLICATIONS **3,998** CITATIONS
SEE PROFILE

Predictable Execution Model: Concept and Implementation

Rodolfo Pellizzoni†, Emiliano Betti†, Stanley Bak†, Gang Yao♯, John Criswell† and Marco Caccamo†
† University of Illinois at Urbana-Champaign, IL, USA,
{rpelliz2, ebetti, sbak2, criswell, mcaccamo}@illinois.edu
♯ Scuola Superiore Sant'Anna, Italy, g.yao@sssup.it

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 COTS 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 COTS 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 need to 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.

1. 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 [14] can transfer data three orders of magnitude faster than the real-time SAFEbus [8]. However, the main drawback of using COTS components within a real-time system is the presence of unpredictable timing anomalies since the individual components are typically designed paying little or no attention to worst-case timing behavior. Additionally, 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 can produce safe upper bounds, they need to be very pessimistic due to the unpredictable behavior of arbiters of physically shared COTS 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 [16]. 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 experiencing the long delays occasionally suffered by real-time tasks, we need to control the operating point of each COTS shared resource and maintain it below saturation limits. This is necessary because the low-level arbiters of the shared resources are not typically designed to provide real-time guarantees. 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 COTS components in the system. Briefly, the key idea is to coschedule active components so that contention for accessing COTS shared resources is implicitly resolved by the high-level coscheduler without relying on low-level, non-real-time arbiters. Several challenges had to be overcome to realize the PRedictable Execution Model (PREM):

- Task execution times suffer high variance due to internal CPU architecture features (caches, pipelines, etc.) and unknown cache miss patterns. This source of temporal unpredictability forces 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.

- 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 COTS I/O subsystem under the discipline of real-time scheduling.

- 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. The programmer provides some information, like beginning and end of each predictable execution interval, and the compiler generates programs which perform cache prefetching and enforce a constant execution time in each predictable interval. In light of the above discussion, we argue that real-time embedded applications should be compiled according to a new set of rules dictated by PREM. At the price of minor additional work by the programmer, the generated executable becomes far more predictable than state-of-the-art compiled code, and when run with the rest of the PREM system, shows significantly reduced worst-case execution time.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3 we describe 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 COTS resources is serialized achieving zero or negligible contention during memory accesses. Then, in Sections 4 and 5 we discuss the challenges in term of hardware architecture and code organization that must be met to predictably compile real-time tasks. Section 6 presents our schedulability analysis. Finally, in Section 7 we detail our prototype testbed, including our compiler implementation based on the LLVM Compiler Infrastructure [9], and provide an experimental evaluation. We conclude with future work in Section 8.

2. Related Work

Several solutions have been proposed in prior real-time research 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. [4] 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 [10]. For real-time compilation, a tight coupling between compiler and worst-case execution time (WCET) analyzer can optimize a program's WCET [5]. Alternatively, a compiler-based approach can provide predictable paging [17]. For analysis of contention for memory and buses, existing techniques can analyze the maximum delay caused by contention for a shared memory or bus under various access models [15, 20]. 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.

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 [22]. The Precision Time (PRET) machine [3] promises to simultaneously deliver high computational performance together with cycle-accurate estimation of program execution time. While our PREM execution model borrows some ideas from these works, it exhibits one key difference: our model can be applied to existing COTS-based systems, without requiring significant architectural redesign. This approach allows PREM to leverage the advantage of the economy of scale of COTS systems, and support the progressive migration of legacy systems.

3. System Model

We consider a typical COTS-based real-time embedded system comprising of a CPU, main memory and multiple DMA peripherals. While in this paper we restrict our discussion to single-core systems with no hardware multithreading, we believe that our predictable execution model is also applicable to multicore systems. We will present a predictable execution model for multicore systems as part of our planned future work. 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 suf-

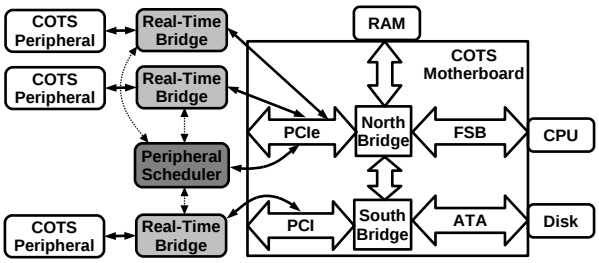

Figure 1: Real-Time I/O Management System.

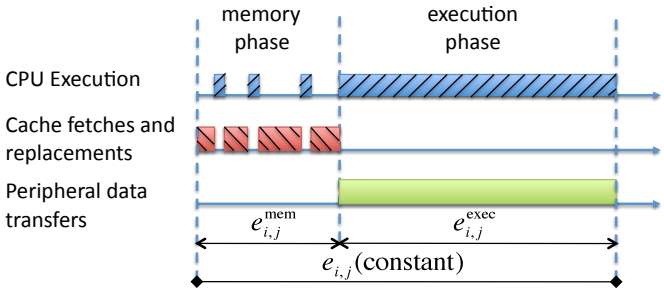

Figure 2: Predictable Interval with constant execution time.

fers 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 interconnect such as PCI or PCIe [14]. 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.

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], we first 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 *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. 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. For simplicity, in the rest of our discussion we assume that each peripheral services a single task. However, this assumption can be lifted by supporting peripheral virtualization in real-time bridges. More details about our hardware/software implementation are provided in Section 7.

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 runtime. 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 can be computed based on traditional 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, and exhibit three main properties. First, each predictable interval is divided into two different phases. During the initial *memory phase*, the CPU accesses main memory to perform a set of

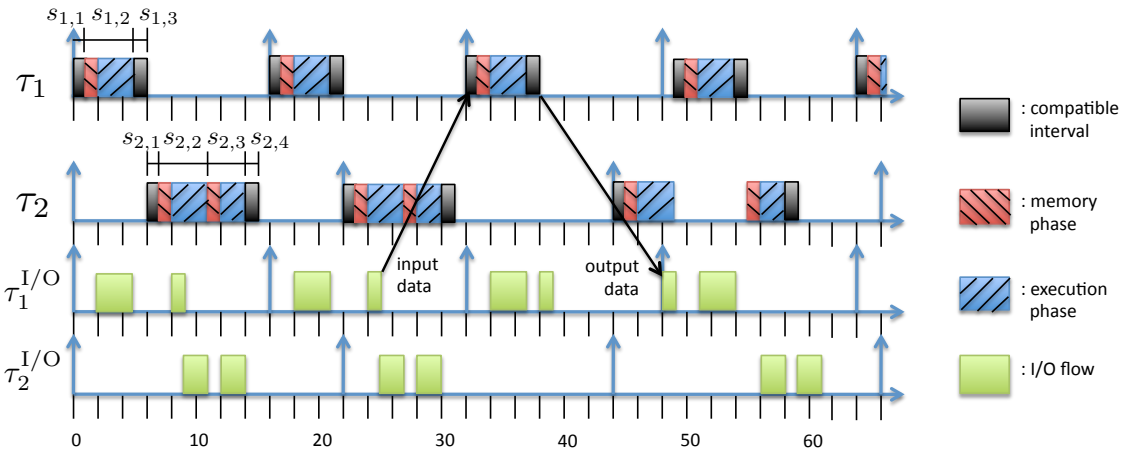

Figure 3: Example System-Level Predictable Schedule

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, the second phase is know as the *execution phase*. During this 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. Due to this property, peripheral traffic can 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 execution time of a predictable interval to be always 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, even if the memory phase lasts for less than $e_{i,j}^{\text{mem}}$ time units, the overall interval still completes in exactly $e_{i,j}$. This property greatly increases task predictability without affecting CPU worst-case guarantees. In particular, as we show in Section 6, it ensures that hard real-time guarantees can be extended to I/O flows.

Figure 3 shows a concrete example of a system-level predictable schedule for a task set comprising two tasks τ_1, τ_2 together with two I/O flows $\tau_1^{\text{I/O}}, \tau_2^{\text{I/O}}$ 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 $\tau_1^{\text{I/O}}$ higher priority than $\tau_2^{\text{I/O}}$. 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 7. I/O flows can be scheduled both during execution phases and while the CPU is idle. As we will show in Section 6, the described scheme can be modeled as a hierarchical scheduling system [21], 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.

4. 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.

4.1. 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 COTS 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 5. Step (2) can be successful only if there is no cache *self-eviction*, that is, prefetching a cache line never evicts another line that has already been accessed (prefetched) dur-

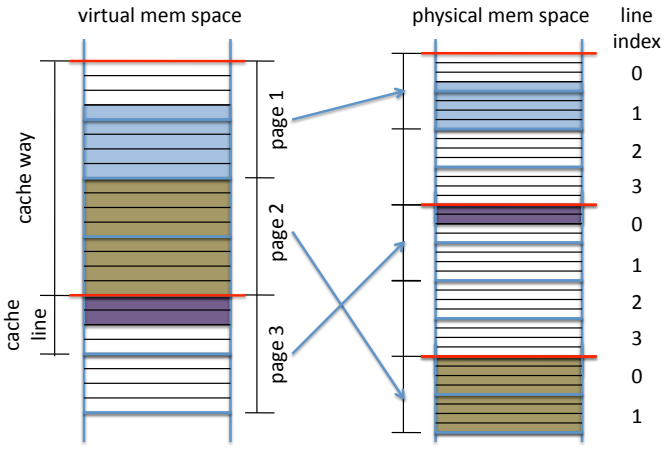

Figure 4: Cache organization with one memory region

ing the same memory phase. In the remainder of this section, we describe self-eviction prevention.

Most COTS CPUs implement the last-level cache as an N-way set associative cache. Let B be the total size of the cache and L be the size of each cache line in bytes. Then the byte size of each of the N cache ways is $W = B/N$. An associative set is the set of all cache lines, one for each way, 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. 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 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 amount 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 amount of entries that can be prefetched in an associative set without causing any self-eviction.

Consider a memory region of size A. The region will occupy at most $K = \lceil \frac{A-1}{L} \rceil + 1$ cache lines. As shown in Figure 4 for a region with $A = 15, K = 5$, the worst case is produced when the region uses a single byte in its first cache line. Assume now that virtual memory addresses coincide with physical addresses. Then 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. Cache-aware allocation algorithms have been proposed in the literature, for example in [11] for cache partitioning. 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 plan to pursue this solution in our future work.

We now consider the cache replacement policy. We consider four different replacement policies: random, FIFO, LRU and pseudo-LRU, which cover most implemented cache architectures. Let Q be the maximum number of entries in any associative set required by the predictable interval. Furthermore, let Q' be the number of such entries relative to cache lines that are accessed multiple times during the memory phase; in our implementation, this only includes the cache lines that contain the small amount of instructions of the memory phase itself, so $Q' = 1$. A detailed analysis of the four replacement policies, based on the work in [7, 19], is provided in Appendix A; in particular, based on the replacement policy and possibly Q', we show how to compute a lower bound on Q such that no self-eviction can happen if Q is less than or equal to the derived bound. It is important to notice that the bound for some policies depends on the state of the cache before the beginning of the predictable interval. Therefore, we consider two different *cache invalidation models*. In the *full-invalidation* model, none of the cache lines used by a predictable interval are already available in cache at the beginning of the memory phase. Furthermore, the replacement policy is not affected by the presence of invalidated cache lines. As an example, this model can be enforced by fully invalidating the cache either at the end or at the beginning of each predictable interval, which in most architectures has the side effect of resetting all replacement queues/trees. While this model is more predictable, several COTS CPUs do not support it, including our testbed described in Section 7. In the *partial-invalidation* model, selected lines in last level cache can be invalidated, for example to handle data coherency between the CPU and peripherals in compatible intervals. Furthermore, the replacement policy always selects invalidated cache lines before valid lines, independently of the state of the queue/tree. Results in Appendix A are summarized by the following theorem.

Theorem 1. *If cache lines are prefetched in a consistent or-*

der, then at the end of the memory phase all Q cache lines in an associative set will be available in cache, requiring at most Q fetches to be loaded, if Q is at most equal to:

- 1 *for random;*
- N *for FIFO and LRU;*
- $\log_2 N + 1$ *for pseudo-LRU with partial-invalidation semantic;*
- $$\begin{cases} \frac{N}{2^{Q'}} + Q' & \text{if } Q' < \log_2 N; \\ \log_2 N + 1 & \text{otherwise,} \end{cases}$$

  for pseudo-LRU with full-invalidation semantic.

4.2. Computing Phase Length

To provide predictable timing guarantees, the maximum time required for memory phase $e_{i,j}^{\text{mem}}$ and for execution phase $e_{i,j}^{\text{exec}}$ must be computed. We assume that an upper bound to $e_{i,j}^{\text{exec}}$ can be derived based on static analysis of the execution phase. Note that our model only ensures that all required instructions and data are available in last level cache; cache misses can still occur in higher cache levels. Analyses that derive patterns of cache misses for split level 1 data and instruction caches have been proposed in [12,18]. These analyses can be safely executed because according to our model, level 1 misses during the execution phase will not require the CPU to perform any external access. $e_{i,j}^{\text{mem}}$ depends on the time required to prefetch all accessed memory regions. Note that 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, each prefetch could require both a cache line fetch and a replacement in last cache level. An analysis to compute upper bounds for read/write operations using a COTS DRAM memory controller is detailed in [13]. Note that since the number and relative addresses of prefetched cache lines is known, the analysis could potentially exploit features such as burst read and parallel bank access that are common in modern memory controllers.

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.

4.3. Interval Length Enforcement

As described in Section 3, each predictable interval is required to execute for exactly $e_{i,j}$ time units. If $e_{i,j}$ is set to be at least $e_{i,j}^{\text{mem}} + e_{i,j}^{\text{exec}}$, the computation in the execution phase is guaranteed to terminate at or before the end of the interval. The interval can then enter active wait until $e_{i,j}$ time units have elapsed since the beginning of its memory phase. In our implementation, elapsed time is computed using a CPU performance counter that is directly accessible by the task; therefore, neither OS nor memory interaction are required to enforce interval length.

4.4. 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 can be achieved by connecting the peripheral scheduler to a peripheral interconnection as shown in Figure 1. Scheduling messages can then be 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 propagating a message through the interconnection takes non-zero time. Since this time is typically negligible compared to the size of a scheduling interval[1], we will ignore it in the rest of our discussion. However, the schedulability analysis of Section 6 could be easily modified to take such overhead into account.

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 7, 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. 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.

5. Programming Model

Our system supports COTS applications written in standard high-level languages such as C. Unmodified code can be executed within one or more compatible intervals. To

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

create predictable intervals, programmers add source code annotations as C preprocessor macros. The PREM real-time compiler creates code for predictable intervals so that it does not incur cache misses during the execution phase and the interval itself has a constant execution time.

Due to current limitations of our static code analysis, we assume that the programmer manually partitions the task into intervals. In particular, compatible intervals can be handled in the conventional way while each predictable interval is encapsulated into a single function. All code within the function, and any functions transitively called by this function is executed as a single predictable interval. To correctly prefetch all code and data used by a predictable interval, we impose several constraints upon its code:

1. Only scalar and array-based memory accesses should occur within a predictable interval; there should be no use of pointer-based data structures.
2. The code can use data structures, in particular arrays, that are not local to functions in the predictable intervals, e.g. they are allocated either in global memory or in the heap[2]. However, the programmer should specify the first and last address that is accessed within the predictable interval for each non-local data structure. In general, it is difficult for the compiler to determine the first and last access to an array within a piece of code. The compiler needs this information to be able to load the relevant portion of each array into the last-level cache.
3. The functions within a predictable interval should not be called recursively. As described below, the compiler will inline callees into callers to make all the code within an interval contiguous in virtual memory. Furthermore, no system calls should be made by code within a predictable interval.
4. Code within a predictable interval may only have direct calls. This alleviates the needs for pointer-analysis to determine the targets of indirect function calls; such analysis is usually imprecise and would bloat the code within the interval.
5. No stack allocations should occur within loops. Since all variables must be loaded into the cache at function entry, it must be possible for the compiler to safely hoist the allocations to the beginning of the function which initiates the predictable interval.

While these constraints may seem restrictive, some of these features are rarely used in real-time C code e.g., indirect function calls, and the others are met by many types of functions. We believe that the benefit of faster, more predictable behavior for program hot-spots outweighs the restrictions imposed by our programming model. Furthermore, existing code that is too complex to be compiled into predictable intervals can still be executed inside compatible intervals. Therefore, our model permits a smooth transition for legacy systems.

Notice that, the compiler can be used to verify that all of the aforementioned restrictions are met. Simple static analysis can determine whether there is any irregular data structure usage, indirect function calls, or system calls. During compilation, the compiler employs several transforms to ensure that code marked as being within a predictable interval does not cause a cache miss. First, the compiler inlines all functions called (either directly or transitively) during the interval into the top-level function defining the interval. This ensures that all program analysis is intra-procedural and that all the code for the interval is contiguous within virtual memory. Second, the compiler can transform the program so that all cache misses occur during the memory phase, which is located at the beginning of the predictable scheduling interval. To be specific, it inserts code after the function prologue to prefetch the code and data needed to execute the 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 data structures marked by the programmer as being accessed by the interval. Third, the compiler inserts code to send scheduling messages to the peripheral scheduler as will be described in Section 7. Fourth, the compiler emits code at the end of the predictable interval to enforce its constant length. In particular, the compiler identifies all return instructions within the function and adds the required code before them. Finally, based on the information on prefetched memory regions, we assume that an external tool, such as a static timing analyzer used to compute maximum phase length in Section 4.2, can check the absence of cache self-evictions according to the analysis provided in Section 4.1.

6. 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

2 Note that data structures in the heap must have been previously allocated during a compatible interval.

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 [2] as the fixed point r_i of the iteration:

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

starting from $r_i^0 = e_i + B_i$. Task set Γ is schedulable if $\forall \tau_i : r_i \leq D_i$.

We now turn our attention to peripheral scheduling. 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 [21]. 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 [21] 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:

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

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 4, 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; the following theorem shows how to compute $\mathrm{sbf}(t)$.

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

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

Proof. Let t', t'' be two consecutive interval start times in the periodic schedule. We show that $\forall \bar{t}, t' \leq \bar{t} \leq t'' : \min\big(\mathrm{sf}(t', t' + t), \mathrm{sf}(t'', t'' + t)\big) \leq \mathrm{sf}(\bar{t}, \bar{t} + t)$. This implies that to minimize $\mathrm{sbf}(t)$, it suffices to check the start times of all scheduling intervals.

Assume first that $\bar{t}$ falls inside a compatible interval or a memory phase. Then the schedule provides no supply in $[t', \bar{t}]$. Therefore: $\mathrm{sf}(t', t' + t) = \mathrm{sf}(\bar{t}, t' + t) \leq \mathrm{sf}(\bar{t}, \bar{t} + t)$. Now assume that $\bar{t}$ falls in an execution phase or in an idle interval before the start time t'' of the next scheduling interval. Then $\mathrm{sf}(\bar{t}, t'') = t'' - \bar{t}$. Therefore: $\mathrm{sf}(t'', t'' + t) \leq \mathrm{sf}(t'', t'' + t - (t'' - \bar{t})) + (t'' - \bar{t}) = \mathrm{sf}(t'', \bar{t} + t) + \mathrm{sf}(\bar{t}, t'') = \mathrm{sf}(\bar{t}, \bar{t} + t)$.

To conclude the proof, it suffices to note that since the schedule is periodic, if $t' \geq 2h$, then $\mathrm{sf}(t', t' + t) = \mathrm{sf}(t^k, t^k + t)$ where $t^k = (t' \mod h) + h$. □

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 can not be computed offline. Therefore, we now propose 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.

Let $\overline{\mathrm{sbf}}(t)$ be the maximum amount of time in which the CPU is executing either a compatible interval or the memory phase of a predictable interval in any time window of length t. Then by definition, $\overline{\mathrm{sbf}}(t) = t - \mathrm{sbf}(t)$. Our analysis derives an upper bound $\overline{\mathrm{sbf}}_U(t)$ to $\overline{\mathrm{sbf}}(t)$. Therefore, $\mathrm{sbf}_L(t) = t - \overline{\mathrm{sbf}}_U(t)$ is a valid lower bound for $\mathrm{sbf}(t)$. For a given interval size t, we compute $\overline{\mathrm{sbf}}(t)$ using the following key idea. For each task τ_i, we determine the minimum and maximum amount of time $E_i^{\min}(t), E_i^{\max}(t)$ that the task can execute in a time window of length t while meeting its deadline constraint. Let $t_i, E_i^{\min}(t) \leq t_i \leq E_i^{\max}(t)$, be the amount of time that τ_i actually executes in the time window. Since the CPU is a single shared resource, it must hold $\sum_{i=1}^{N} t_i \leq t$ independently of the task scheduling algorithm. Finally, for each task τ_i, we define the *memory bound function* $\mathrm{mbf}_i(t_i)$ to be the maximum amount of time

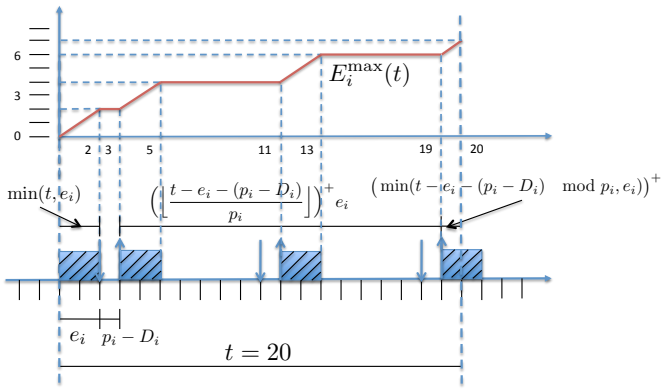

Figure 5: Derivation of $E_i^{\max}(t)$, periodic or sporadic task.

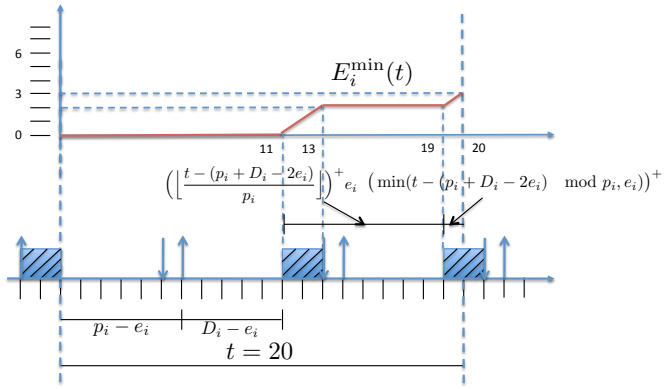

Figure 6: Derivation of $E_i^{\min}(t)$, periodic task.

that the task can spend in compatible intervals and memory phases, assuming that it executes for t_i time units in the time window. We can then obtain $\overline{\text{sbf}}_U(t)$ as the maximum over all feasible $\{t_1, \ldots, t_N\}$ tuples of $\sum_{i=1}^N \text{mbf}_i(t_i)$. In other words, we can compute $\overline{\text{sbf}}_U(t)$ by solving the following optimization problem:

$$\overline{\text{sbf}}_U(t) = \max \sum_{i=1}^{N} \text{mbf}_i(t_i), \tag{4}$$

$$\sum_{i=1}^{N} t_i \leq t, \tag{5}$$

$$\forall i, 1 \leq i \leq N : E_i^{\min}(t) \leq t_i \leq E_i^{\max}(t). \tag{6}$$

$E_i^{\max}(t)$ and $E_i^{\min}(t)$ can be computed according to the scenarios shown in Figures 5, 6 where the notation $(x)^+$ is used for $\max(x, 0)$.

Theorem 3. *For a periodic or sporadic task:*

$$E_i^{\max}(t) = \min(t, e_i) + \left(\left\lfloor \frac{t - e_i - (p_i - D_i)}{p_i} \right\rfloor\right)^+ e_i + \left(\min(t - e_i - (p_i - D_i) \bmod p_i, e_i)\right)^+. \tag{7}$$

For a sporadic task, $E_i^{\min}(t) = 0$, while for a periodic task:

$$E_i^{\min}(t) = \left(\left\lfloor \frac{t - (p_i + D_i - 2e_i)}{p_i} \right\rfloor\right)^+ e_i + \left(\min(t - (p_i + D_i - 2e_i) \bmod p_i, e_i)\right)^+. \tag{8}$$

Proof. As shown in Figure 5, the relative distance between successive executions of a task τ_i is minimized when the task finishes at its deadline in the first period within the time window of length t, and as soon as possible (e.g., e_i time units after its activation) for each successive period. Equation 7 directly follows from this activation pattern, assuming that the time window coincides with the start time of the first job of τ_i. In particular:

- term $\min(t, e_i)$ represents the execution time for the first job, which is activated at the beginning of the time window;
- term $\left(\left\lfloor \frac{t-e_i-(p_i-D_i)}{p_i} \right\rfloor\right)^+ e_i$ represents the execution time of jobs in periods that are fully contained within the time window; note that the first period starts $e_i + (p_i - D_i)$ time units after the beginning of the time window;
- finally, term $\left(\min(t - e_i - (p_i - D_i) \bmod p_i, e_i)\right)^+$ represents the execution time for the last job in the time window.

Similarly, as shown in Figure 6, the relative distance between successive executions of a periodic job τ_i is maximized when the task finishes as soon as possible in its first period and as late as possible in all successive periods. Equation 8 follows assuming that the time window coincides with the finishing time of the first job of τ_i, noticing that periodic task activations start $(p_i - e_i) + (D_i - e_i) = p_i + D_i - 2e_i$ time units after the beginning of the time window. Finally, $E_i^{\min}(t)$ is zero for a sporadic task since by definition the task has no maximum interarrival time. □

Lemma 4. *The optimization problem of Equations 4-6 admits solution if task set Γ is feasible.*

Proof. The optimization problem admits solution if the set of t_i values that satisfy Equations 5 and 6 is not empty. Note that by definition $E_i^{\max}(t) \geq E_i^{\min}(t)$, therefore there is at least one admissible solution iff $\sum_{i=1}^N E_i^{\min}(t) \leq t$. Define $E_{Ui}^{\min}(t) = \left(t - (D_i - e_i)\right)^+ \frac{e_i}{p_i}$. Then it is easy to see that $\forall t, E_i^{\min}(t) \leq E_{Ui}^{\min}(t)$: in particular, both functions are equal to e_i for $t = p_i + D_i - e_i$ and increase by e_i every p_i afterwards. Now note that $E_{Ui}^{\min}(t) \leq t\frac{e_i}{p_i}$, therefore it also holds: $\sum_{i=1}^N E_i^{\min}(t) \leq t \sum_{i=1}^N \frac{e_i}{p_i}$. The proof follows by noticing that since Γ is feasible, it must hold $\sum_{i=1}^N \frac{e_i}{p_i} \leq 1$. □

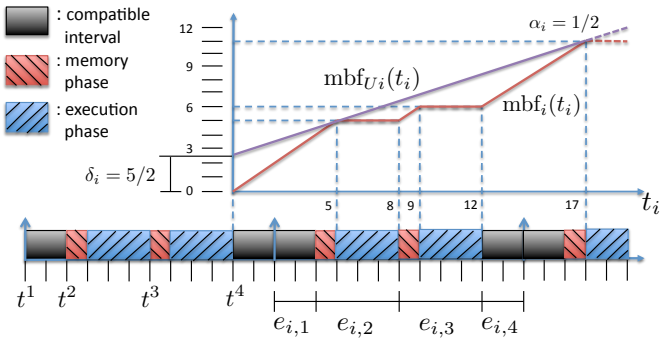

Figure 7: Derivation of $\mathrm{mbf}_i(t_i)$ and $\mathrm{mbf}_{Ui}(t_i)$.

$\mathrm{mbf}_i(t_i)$ can be computed using a strategy similar to the one in Theorem 2. Since t_i represents the amount of time that τ_i executes, we consider a schedule in which τ_i is executed continuously being activated every e_i time units, as shown in Figure 7. The start time of the first scheduling interval in the first job of τ_i is $t^1 = 0$, the start time of the second scheduling interval is $t^2 = e_{i,1}$, and so on and so forth until the first interval of the second job which has start time equal to e_i. Then $\mathrm{mbf}_i(t_i)$ can be computed as the maximum amount of time that the task spends in a compatible interval or memory phase in any time window t_i.

Theorem 5. *Let $\mathrm{mf}_i(t', t'')$ be the amount of time that task τ_i spends in a compatible interval or memory phase in the interval $[t', t'']$, in the schedule in which τ_i is executed continuously. Then:*

$$\mathrm{mbf}_i(t_i) = \max_{k=1\ldots N_i} \mathrm{mf}_i(t^k, t^k + t_i). \tag{9}$$

Proof. Note that the schedule in which τ_i executes continuously is periodic with period p_i. The same strategy as in Theorem 2 can be used to show that if t', t'' are consecutive interval start times, then $\forall \bar{t}, t' \leq \bar{t} \leq t'' : \max\left(\mathrm{mf}_i(t', t' + t_i), \mathrm{mf}_i(t'', t'' + t_i)\right) \geq \mathrm{mf}_i(\bar{t}, \bar{t} + t_i)$. □

As an example, in Figure 7 $\mathrm{mbf}_i(t_i)$ is maximized in the time window that starts at $t^4 = 10$.

Since $\mathrm{mbf}_i(t_i)$ is a nonlinear function, computing $\overline{\mathrm{sbf}}_U(t)$ according to Equations 4-6 requires solving a nonlinear optimization problem. To simplify the problem, we consider a linear upper bound approximation $\mathrm{mbf}_{Ui}(t_i) = \alpha_i t_i + \delta_i$, as shown in Figure 7, where

$$\alpha_i = \Big(\sum_{\forall s_{i,j} \text{ compatible}} e_{i,j} + \sum_{\forall s_{i,j} \text{ predictable}} e^{\mathrm{mem}}_{i,j} \Big) / e_i, \tag{10}$$

and δ_i is the minimum value such that $\forall t_i, \mathrm{mbf}_{Ui}(t_i) \geq \mathrm{mbf}_i(t_i)$. Using $\mathrm{mbf}_{Ui}(t_i)$, Equations 4-6 can be efficiently solved in $O(N)$ time. Furthermore, we can show that $\overline{\mathrm{sbf}}_U(t)$ can be computed for all relevant values of t by solving the optimization problem of Equations 4-6 in a finite number of points, and using linear interpolation to derive the remaining values. Due to the complexity of the resulting algorithm, the full derivation is provided in Appendix B.

7. Evaluation

In order to verify the validity and practicality of PREM, we implemented the key components of the system. In this section, we describe our evaluation, first introducing the new hardware components followed by the corresponding software driver and OS calibration effort. We then discuss our compiler implementation and analyse its effectiveness on a DES benchmark. Finally, using synthetic tasks we measure the effectiveness of the PREM system as a function of cache stall time, and show traces of PREM when running on COTS hardware.

7.1. PREM Hardware Components

When introducing PREM in Section 3, two additional hardware components were added to the COTS system (recall Figure 1). Here, we briefly review our previous work which describes these hardware components in more detail [1], and describe the additions to these components which we made to provide the mechanism for PREM execution. First we describe the real-time bridge component, then we describe the peripheral scheduler component.

Our **real-time bridge** prototype is implemented on an ML507 FPGA Evaluation Platform, which contains a COTS TEMAC Ethernet hardware block (the I/O peripheral under control), and a PCIe edge connector (the interface to the main system). Interposed between these hardware components, our real-time bridge uses a System-on-Chip design running Linux. The real-time bridge is also wired to peripheral scheduler, which allows direct communication between these components. Three wires are used to transmit information between each real-time bridge and the peripheral scheduler, `data_ready`, `data_block`, and `interrupt_block`. The `data_ready` wire is an output signal sent to the peripheral scheduler which is asserted whenever the COTS peripheral has buffered data in the real-time bridge. The peripheral scheduler sends two signals, `data_block` and `interrupt_block`, to the real-time bridge. The `data_block` signal is asserted to block the real-time bridge from transferring data in DMA mode over the PCIe bus from its local buffer to main memory or viceversa. The `interrupt_block` signal is a new signal in the real-time bridge implementation, added to support PREM, and instructs the real-time bridge to not further raise interrupts. In previous work [1], we provide a more detailed description and buffer analysis of

the real-time bridge design, and demonstrate the component's effectiveness and efficiency.

The **peripheral scheduler** is a unique component in the system, connected directly to each real-time bridge. Unlike our previous work where the peripherals were scheduled asynchronously with the CPU [1], PREM requires the CPU and peripherals to coordinate their access to main memory. The peripheral scheduler provides a hardware implementation of the child scheduling model used to schedule peripherals, with each peripheral given a sporadic server to schedule its traffic. Meanwhile, the OS real-time scheduler runs the parent scheduling model for CPU tasks to complete the hierarchical PREM scheduling model described in Section 6. The peripheral scheduler is connected to the PCIe bus, and exposes a set of registers accessible from the main CPU. In the `configuration` register, constant parameters such as the maximum cache write-back time are stored. Writing a value to the `yield` register indicates that the CPU will not access main memory for the given amount of time, and I/O peripherals should be allowed to read to and write from RAM. The value written to the `yield` register contains a 14 bit unsigned integer indicating the number of microseconds to permit peripheral traffic with main memory, as described in Section 4. The CPU can also use the `yield` register to allow interrupts to be raised by peripherals. Another 14 bit unsigned integer indicates the number of microseconds to allow peripheral interrupts, and a 3 bit interrupt mask selects which peripherals should be allowed to raise the interrupts. In this way, different CPU tasks can predictably service interrupts from different I/O peripherals.

7.2. Software Evaluation

The software effort required to implement PREM execution involves two aspects: (1) creating the drivers which control the custom PREM hardware discussed in the previous section, and (2) calibrating the OS to eliminate undesired execution interference. We now discuss these, in order.

The two custom hardware components, the real-time bridge and the peripheral scheduler, each require a software driver to be controller from the main CPU. Additionally, each peripheral requires a driver running on the real-time bridge's CPU to control the COTS peripheral. The driver for the peripheral scheduler is straightforward, mapping the bus addresses corresponding to the exposed registers to user space where a PREM-compiled process can access them. The driver for each real-time bridge is more difficult, since each unique COTS peripheral requires a unique driver. However, since in our implementation both the main CPU and the real-time bridge's CPU are running 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 COTS peripherals require no modification.

For our experiments, we use a Intel Q6700 CPU with a 975X system controller; we set the CPU frequency to 1Ghz obtaining a measured memory bandwidth of 1.8Ghz/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. 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 emulate at our best 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 (ex: the keyboard) and runs all the system activities and non real-time processes (ex: 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 not used. 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 can not 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 4, 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.

7.3. Compiler Evaluation

We built the PREM real-time compiler prototype using the LLVM Compiler Infrastructure [9], targeting the compilation of C code. LLVM was extended by writing self-contained analysis and transformation passes, which were then loaded into the compiler.

In the current PREM real-time compiler prototype, we rely on the programmer to partition the task into predictable and compatible intervals . The partitioning is done by putting each predictable interval into its own function. The beginning and end of the scheduling interval correspond to the entry and exit of the function, and the start of execution phase (the end of memory access phase) is manually marked by the programmer. We assume that non-local data accessed during a predictable interval exists in continuous memory spaces, which can be prefetched by a set of

PREFETCH_DATA(start_address, size) macros that must be placed by the programmer during the memory phase. The implementation of this macro does the actual prefetching of the data into level 2 cache by prefetching every cache line in the given range with the i386 prefetcht2 instruction. After the memory phase, the programmer adds a STARTEXECUTION(wcet) macro to indicate the beginning of the execution phase. This macro measures the amount of time remaining in the predictable interval using the CPU performance counter, and writes the time remaining to the yield register in the peripheral scheduler.

All remaining operations needed to transform the interval are performed by a new LLVM function pass. The pass iterates over all the functions in a compilation unit. When a function representing a predictable interval is found, the pass performs code transformation. First, our transform inlines called functions using preexisting LLVM inlining functions. This ensures that there are only a single stack frame and segment of code that need to be prefetched into the cache. Second, our transform inserts code to read the CPU performance counter at the beginning of the interval and save the current time. Third, it inserts code to prefetch the stack frame and function arguments. Bringing the stack frame into the cache is done by inserting instructions into the program to fetch the stack pointer and frame pointer. Code is then inserted to prefetch the 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. This is done by transforming the program so that the function is placed within a unique ELF section. We then use a linker script to define variables pointing to the beginning and end of this unique ELF section. The compiler then adds code that prefetches the memory inside the ELF section. 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 performance counter reaches the worst-case cycle count based on the time value saved at the beginning of the interval. It may also enable peripheral interrupts by writing the worst-case interrupt processing time to the peripheral scheduler's yield register.

To verify the correctness of the PREM real-time compiler prototype and to test its applicability, we used LLVM to compile a DES cypher benchmark. The DES benchmark was selected because it represents a typical real-time data flow application. The benchmark comprises one scheduling interval which encrypts a variable amount of data. We compiled it as both a predictable and a compatible interval (e.g. with and without prefetching), and measured number of cache misses with a performance counter. Adapting the interval required no modification to any cypher functions and a total of 11 PREFETCH_DATA macros.

Data size	4K	8K	32K	128K	512K	1M
Compatible	138	254	954	3780	15k	31k
Predictable	2	2	4	2	1	81

Table 1: DES benchmark.

Results are shown in Table 1 in terms of the number of cache misses suffered in the execution phase of the predictable interval (after prefetching), and in the entire compatible interval. Data size is in bytes. The compatible interval suffers an excessive number of cache misses, which increases roughly proportionally with the amount of processed data. Conversely, the execution phase of the predictable interval has almost zero cache misses, only suffering a small increase when large amounts of data are being processed. The reason the number of cache misses is not zero 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 all the following experiments, we observed that the number of self-evictions is typically so small that it can be considered negligible.

7.4. WCET Experiments with Synthetic Tasks

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. 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 8 and 9 show the observed worst case execution time for any 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

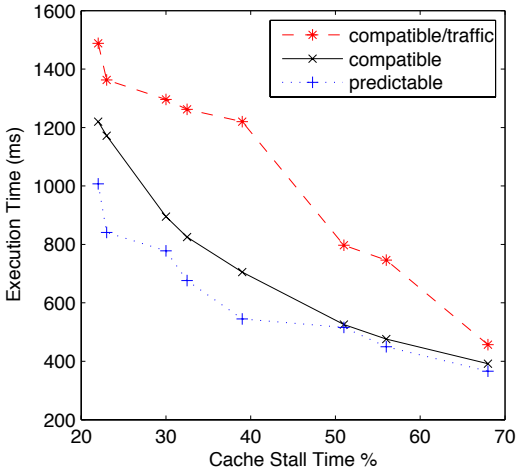

Figure 8: `random_access`

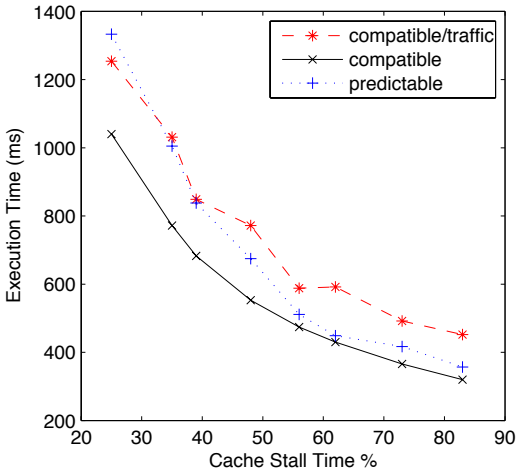

Figure 9: `linear_access`

does not increase execution time. In all cases, the computation time decreases with an increase in stall time. This is because stall time is controlled by varying the amount of computation between memory references. Furthermore, execution times should not be compared between the two figures because the two applications execute different code.

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, we can decrease the 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 9, 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 [15]). In practice, we expect the effect of PREM on an application's execution time to be between the two figures, based on the specific memory access pattern.

7.5. System-wide Coscheduling Traces

We now present execution traces of the implemented system which demonstrate the advantage of the PREM coscheduling approach. The traces are obtained by using the peripheral scheduler as a logic analyzer for the various signals which are being sent to or from the real-time bridges, `data_block`, `data_ready`, and `interrupt_block`. Additionally, the peripheral scheduler has a `trace` register which allows timestamped trace information to be recorded with a one microsecond resolution when instructed by the main CPU, such as at the start and end of an execution interval. An execution trace is shown for a task running the traditional COTS execution model in Figure 10, and the same task running within the PREM model is shown in Figure 11[3].

In the first trace (Figure 10), although the execution is divided into unpreemptable intervals, there is no memory phase prefetch or constant execution time guarantees. When the scheduling intervals of task T_1 finish executing, an I/O peripheral begins to access main memory, which may happen if T_1 had written output data into RAM. Task T_2 then executes, suffering cache misses that compete for main memory bandwidth with the I/O peripheral. Due to the cold cache and peripheral interference, the execution time of T_2 grows from 0.5 ms (the execution time with warm cache and no peripheral I/O), to 2.9ms as shown in the figure, an increase of about 600%.

In the second trace (Figure 11), the system executes according to the PREM execution model, where peripherals only access the bus when permitted by the peripheral scheduler. The predictable interval is divided into a memory phase and an execution phase. Instead of competing for main memory access, task T_2 gets contentionless access to

3 The (compatible) intervals at the end of T_1 and at the start of T_2 were measured as 0.107ms and 0.009ms, respectively, and have been exaggerated in the figures to be visible.

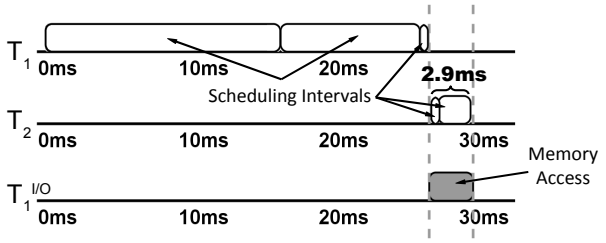

Figure 10: An unscheduled bus-access trace (without PREM)

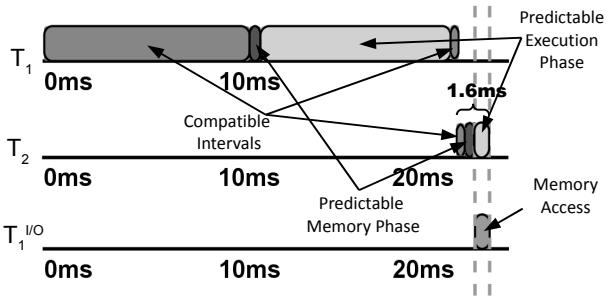

Figure 11: A scheduled trace using PREM

main memory during the memory phase. After all to-be-accessed data is loaded into the cache, the execution phase begins which incurs no cache misses, and the peripheral is allowed to access the data in main memory. The constant execution time for the predictable interval in the PREM execution model is 1.6ms, which is significantly lower than the worst-case observed for the unscheduled trace (and is about the same as the execution time of the scheduling interval of T_2 in the unscheduled trace with a cold cache and no peripheral traffic).

8. Conclusions

We have discussed the concept and implementation of a novel task execution model, PRedictable Execution Model (PREM). Our evaluation shows 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. We plan to further develop our solution in two main directions. First, we will study extensions to our compiler infrastructure to lift some of the more restrictive code assumptions and compile and test a larger set of benchmarks. Second, 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 [16]. 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1997.

[7] 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.

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

[9] 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*, San Jose, CA, USA, Mar 2004.

[10] 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.

[11] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2), 207.

[20] 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.

[21] 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.

[22] 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.

A. Analysis of Cache Replacement Policy

We first provide a brief overview of each analyzed policy. Under *random* policy, whenever a new cache line is loaded in an associative set, the line to be evicted is chosen at random among all N lines in the set. While this policy is implemented in some modern COTS CPUs with very large cache associativity, such as Intel Core architecture, it is clearly ill suited to real-time systems. In *FIFO* policy, a FIFO queue is associated with each associative set as shown in Figure 12(a). Each newly fetched cache line is inserted at the head of the queue (position q_1), while the cache line which was previously at the back of the queue (position q_N) is evicted. In the figure, $l_1, \ldots, l_4$ represent cache lines used in the predictable interval, while dashes represent other cache lines available in cache but unrelated to the predictable interval. Finally, grayed boxes represent invalidated cache lines. The Least Recently Used (*LRU*) policy also uses a replacement queue, but a cache line is moved at the head of the queue whenever it is accessed. Due to the complexity of implementing LRU, an approximated version of the algorithm known as *pseudo-LRU* can be employed where N is a power of 2. A binary replacement tree with $N-1$ internal nodes and N leaves $q_1, \ldots, q_N$ is constructed as shown in Figure 13 for $N = 4$. Each internal node encodes a "right" or "left" direction using a single bit of data, while each leaf encodes a fetched cache line. Whenever a replacement decision must be taken, the encoded directions are followed starting from the root to the leaf representing the cache line

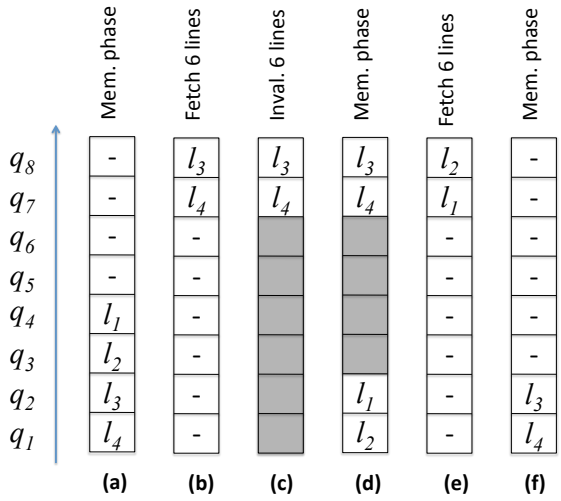

Figure 12: FIFO policy, example replacement queue.

to be replaced. Furthermore, whenever a cache line is accessed, all directions on the path from the root to the leaf of the accessed cache line are set to be the opposite of the followed path. Figure 13 shows an example where four cache lines are accessed in the order l_1, l_2, l_3, l_2, l_4, causing a self-eviction.

Without loss of generality, in all proofs in this section we focus on the analysis of a single associative set with Q cache lines accessed during the predictable interval. Furthermore, let $l_1, \ldots, l_Q$ be the set of Q cache lines in the order in which they are first accessed during the memory phase (note that this order can be different from the order in which the lines are prefetched due to the Q' lines that are accessed multiple times). Results for LRU and random are simple and well-known, see [6] for example; we detail the bound derivation in the following theorem to provide a better understanding of the replacement policies.

Theorem 6. *A memory phase will not suffer any cache self-eviction in an associative set requiring Q entries, if Q is at most equal to:*

- 1*: for random replacement policy;*
- N*: for LRU replacement policy.*

Proof. Clearly no self-eviction is possible if $Q = 1$, since after the unique cache line is fetched no other cache line can be accessed in the associative set.

Now consider LRU policy. When a cache line l_i is first accessed, it is fetched (if it is not already in cache) and moved to the beginning of the replacement queue. Since no cache line outside of $l_1, \ldots, l_Q$ is accessed in the associative set, when l_i is first accessed, cache lines $l_1, \ldots, l_{i-1}$ must be at the head of the queue in some order. But $Q \leq N$ implies $i - 1 \leq N - 1$, hence the algorithm always picks an unrelated line at the back of the queue to be replaced. □

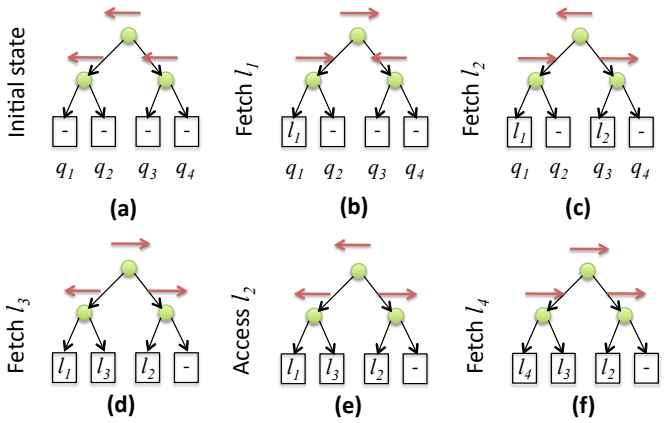

Figure 13: Pseudo-LRU policy, full-invalidation, $Q' = 1$.

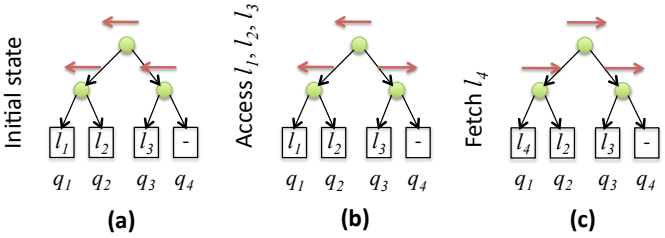

Figure 14: Pseudo-LRU policy, partial-invalidation.

Note that for random the bound of 1 is tight, since in the worst-case fetching a second cache line in the associative set can cause the first cache line to be evicted. Furthermore, note that if the cache is not invalidated between successive activations of the same predictable interval, then some of the $l_1, \ldots, l_Q$ cache lines can still be available in cache. Assume that at the beginning of the predictable interval l_2 is available but not l_1. Then even with LRU policy, prefetching l_1 can potentially cause l_2 to be evicted. According to our definition, this is not a self-eviction: l_2 is evicted before it is accessed during the memory phase, and is then reloaded when it is first accessed, thus guaranteeing that all required lines are available in cache at the end of the memory phase. In fact, Theorem 6 proves that the bounds for random and LRU are independent of the state of the cache before the beginning of the predictable interval.

Unfortunately, the same is not true of FIFO and pseudo-LRU. Consider FIFO policy. If a cache line l_i is already available in cache, then when l_i is accessed during the memory phase no modification is performed on the replacement queue. In some situations this can cause l_i to be evicted by another line fetched after l_i in the memory phase, causing a self-eviction. Due to this added complexity, when analyzing FIFO and pseudo-LRU we need to consider the invalidation model. A detailed analysis of FIFO replacement is provided in [7]; here we summarize the results and intuitions relevant to our system.

Theorem 7 (**Directly follows from Lemma 1 in [7]**). *Assume FIFO replacement with full-invalidation semantic. Then no self-eviction is possible for an associative set with $Q \leq N$.*

The analysis for partial-invalidation FIFO is more complex. Figure 12 (analogous to Figure 2 in [7]) depicts a clarifying example, where the "mem. phase", "fetch" and "inval." labels show the state of the replacement queue after the memory phase and after some cache lines have been either fetched or invalidated outside the predictable interval, respectively. Note that after Step (e), lines l_1, l_2 remain in cache but not l_3, l_4. Hence, when l_3 and l_4 are fetched during the next memory phase in Step (f), they evict l_1 and l_2. The solution is to prefetch $l_1, \ldots, l_Q$ multiple times in the memory phase, each time in the same order.

Theorem 8 (**Directly follows from Theorem 3 in [7]**). *Assume FIFO replacement with partial-invalidation semantic. If the Q cache lines in an associative set, with $Q \leq N$, are prefetched at least Q times in the same access-order $l_1, \ldots, l_Q$, then all Q cache lines are available in cache at the end of the memory phase. Furthermore, no more than Q fetches are required.*

Note that while executing the prefetching code Q times might seem onerous, in practice the majority of the time in the memory phase is spent fetching and writing back cache lines, and Theorem 8 ensures that no more than Q fetches and write backs are required for each associative set.

While LRU and FIFO allow to prefetch up to N cache lines with N fetches, the same is not true of pseudo-LRU, even in the full-invalidation model. Consider again Figure 13, where no relevant lines are available in cache at the beginning of the memory phase but $Q' = 1$, meaning that line l_2 can be accessed multiple times. Then it is easy to see that no more than 3 cache lines can be prefetched without causing a self-eviction. The key intuition is that up to Step (d), and with respect to l_1, l_2, l_3, the replacement tree encodes the LRU order exactly. However, after l_2 is accessed again at Step (e), the LRU order is lost, and the next replacement causes l_1 to be evicted instead of the remaining unrelated cache line. Furthermore, note that the strategy employed in Theorem 8 does not work in this case, because l_1 has already been fetched during the memory interval before being evicted. A similar situation can happen even if $Q' = 0$ in the partial-invalidation model, as shown in Figure 14. Assume that due to partial replacements and cache accesses, at the beginning of the memory phase the cache state is as shown in Figure 14(a), with l_1, l_2 and l_3 being available in cache but not l_4. Then after the first three cache lines are prefetched in order, l_1 will be replaced when l_4 is next prefetched.

A lower bound that is independent of Q' and of the invalidation model was first proven in [19].

Theorem 9 (**Theorem 10 in [19]**). *Under pseudo-LRU replacement, no self-eviction is possible for an associative set with* $Q \leq \log_2 N + 1$.

We now show in Theorem 12 that under the full-invalidation model, a better bound can be obtained based on the value of Q'. In the theorem, a subtree of a pseudo-LRU replacement tree is a tree rooted at any internal node of the replacement tree. For example, the replacement tree in Figures 13, 14 has three subtrees: the whole tree, which has height 2 and leaves $q_1, \ldots, q_4$, and its left and right subtrees with height 1 and leaves q_1, q_2 and q_3, q_4, respectively. For each subtree of height k, we compute a lower bound Lb_k^p on the number of cache lines in $l_1, \ldots, l_Q$ that can be allocated in the subtree (either because they are already available in cache at the beginning of the predictable interval or because they are fetched during the memory phase) without causing any self-eviction, assuming that none of the lines were available in cache at the beginning of the predictable interval and that up to p lines can be accessed multiple times during the memory phase. Note that by definition, $Lb_k^i \leq Lb_k^j$ if $i \geq j$. Furthermore, $\forall p, Lb_1^p = 2$ by Theorem 9. The following Lemmas 10, 11 prove some important facts on Lb_k^p.

Lemma 10. $Lb_k^0 = 2Lb_{k-1}^0$.

Proof. Consider once again the left and right subtrees with height $k-1$. By contradiction and without loss of generality, assume that a self-eviction happens in the left subtree; then at least $Lb_{k-1}^0 + 1$ cache lines must be allocated in it. Since no relevant cache line is originally available in cache and furthermore no cache line is accessed more than once, it follows that every access results in a fetch and replacement. Furthermore, whenever a cache line is fetched in the left subtree, the root of the height-k subtree is changed to point to the right subtree and viceversa. Hence, to fetch Lb_{k-1}^0+1 lines in the left subtree, at least Lb_{k-1}^0 lines must be fetched in the right subtree. Therefore, at least $2Lb_{k-1}^0 + 1$ total cache lines must be allocated in the whole height-k subtree to cause a self-eviction, implying that $Lb_k = 2Lb_{k-1}^0$ is a valid lower bound. □

Note that from Lemma 10 and $Lb_1^0 = 2$ it immediately follows that $Lb_k^0 = 2^k$.

Lemma 11. *For any subtree of height* $k > 1$ *with* $p > 0$, $Lb_k^p = \min(Lb_{k-1}^{p-1} + 1, 2Lb_{k-1}^p)$.

Proof. Consider the left and right subtrees with height $k-1$, and assume that out of the p cache lines that can be accessed multiple times, i are allocated in the left subtree and j are allocated in the right subtree, with $i+j = p$. By contradiction and without loss of generality, assume that a self-eviction happens in the left subtree; then at least $Lb_{k-1}^i + 1$ cache lines must be allocated in it. We distinguish two cases.

Case (1): $j > 0$. Then following the same reasoning as in Theorem 9, it is sufficient to allocate a single cache line in the right subtree, resulting in a total of $Lb_{k-1}^i + 2$ lines required to cause a self eviction; note that this value is minimized by maximizing i, e.g. when $j = 1, i = p-1$, resulting in $Lb_{k-1}^{p-1} + 2$ required cache lines.

Case (2): $j = 0$. Then following the same reasoning as in Lemma 10, at least Lb_{k-1}^i cache lines must be allocated in the right subtree, for a total of $2Lb_{k-1}^i + 1 = 2Lb_{k-1}^p + 1$ cache lines.

Combining case (1) and (2), it follows that no eviction is possible if the number of allocated lines is at most equal to $\min(Lb_{k-1}^{p-1} + 1, 2Lb_{k-1}^p)$, which concludes the proof. □

We can finally prove Theorem 12.

Theorem 12. *Under pseudo-LRU replacement with full-invalidation semantic, no self-eviction is possible if:*

$$Q \leq \begin{cases} \frac{N}{2^{Q'}} + Q' & \text{if } Q' < \log_2 N; \\ \log_2 N + 1 & \text{otherwise.} \end{cases}$$

Proof. The proof proceeds by induction on the pair (p, k) ordered by p first, e.g. in the sequence $(0,1), \ldots, (0,Q), (1,1), \ldots, (1,Q), \ldots, (Q',1), \ldots, (Q',Q)$. In particular, we show that the following property holds $\forall p, k$:

$$Lb_k^p = \begin{cases} 2^{k-p} + p & \text{if } p < k; \\ k+1 & \text{otherwise.} \end{cases}$$

Since a pseudo-LRU replacement tree for a N-associative set has height $\log_2 N$, the theorem then follows.

By Lemma 10, the property is verified for $p = 0$ since $Lb_k^0 = 2^k$. By Theorem 9, the property is verified for $k = 1$ since $Lb_1^p = 2$. Therefore, it remains to complete the induction step by showing that the property holds at step (p, k) with $p > 0, k > 1$. We do so by applying Lemma 11, assuming that the property also holds at previous steps $(p-1, k-1)$ and $(p, k-1)$. We consider three cases based on the relative value of p, k.

Case (1): $p < k-1$. We have to show that $Lb_k^p = 2^{k-p} + p$. Then:

$$Lb_k^p = \min\left(2^{(k-1)-(p-1)} + (p-1) + 1, 2(2^{(k-1)-p} + p)\right) = \\ = \min\left(2^{k-p} + p, 2^{k-p} + 2p\right) = 2^{k-p} + p.$$

Case (2): $p = k-1$. We have to show that $Lb_k^p = 2^{k-p} + p = k+1$. Then:

$$Lb_k^p = \min\left(2^{(k-1)-(p-1)} + (p-1) + 1, 2((k-1)+1)\right) = \\ = \min\left(2^{k-p} + p, 2k\right) = k+1.$$

Case (3): $p \geq k$. We have to show that $Lb_k^p = k+1$. Then:

$$Lb_k^p = \min\left((k-1) + 1 + 1, 2((k-1)+1)\right) = \\ = \min\left(k+1, 2k\right) = k+1.$$

□

B. Solving The Optimization Problem

Using $\mathrm{mbf}_{Ui}(t_i) = \alpha_i t_i + \delta_i$, the optimization problem of Equations 4-6 can be rewritten as follows:

$$\overline{\mathrm{sbf}}_U(t) = \sum_{i=1}^{N}(\delta_i + \alpha_i E_i^{\min}(t)) + \max \sum_{i=1}^{N} \alpha_i x_i, \quad (11)$$

$$\sum_{i=1}^{N} x_i \leq t - \sum_{i=1}^{N} E_i^{\min}(t), \quad (12)$$

$$\forall i, 1 \leq i \leq N : 0 \leq x_i \leq E_i^{\max}(t) - E_i^{\min}(t), \quad (13)$$

where we substituted $x_i = t_i - E_i^{\min}(t)$. Without loss of generality, assume that the N tasks are ordered by non-increasing values of α_i. Then Algorithm 1 computes the solution val to Equations 4-6. The algorithm first computes the time bound $t - \sum_{i=1}^{N} E_i^{\min}(t)$ on the sum of the variables x_i. Then, starting from x_1, it assigns to each variable the minimum between the remaining time t_{rem} and the upper constraint of Equation 13. Since the tasks are ordered by non-increasing values of α_i, it is trivial to see that Algorithm 1 computes the maximum of Equation 11. Furthermore, the while loop at Lines 5-10 is executed at most N times, and each iteration requires constant time. Hence, the algorithm has complexity $O(N)$.

Algorithm 1 Compute $\overline{\mathrm{sbf}}_U(t)$ for a given t.

```
1:  procedure COMPUTEINSTANT(t, τ_1, ..., τ_N ordered by non-increasing α_i)
2:      t_rem := t − Σ_{i=1}^{N} E_i^min(t)
3:      val := Σ_{i=1}^{N} (δ_i + α_i E_i^min(t))
4:      i := 1
5:      while t_rem > 0 and i ≤ N do
6:          x_i = min(t_rem, E_i^max(t) − E_i^min(t))
7:          t_rem := t_rem − x_i
8:          val := val + α_i x_i
9:          i := i + 1
10:     end while
11:     return (val, i, {x_1, ..., x_N})
12: end procedure
```

Algorithm 1 computes $\overline{\mathrm{sbf}}_U(t)$ for a specific value of t. To test the schedulability condition of Equation 2 using the upper bound $tbf_U(t) = \min\{x | \mathrm{sbf}_L(x) \geq t\}, \overline{\mathrm{sbf}}_U(t)$ must be computed for all t in the interval $[0, \max_i D_i^{\mathrm{I/O}}]$. The reason is as follows. Note that if for any i:

$$\mathrm{tbf}_U\Big(e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \Big\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \Big\rceil e_l^{\mathrm{I/O}}\Big) > \max_i D_i^{\mathrm{I/O}}, \quad (14)$$

then the system is not schedulable. By definition of $\mathrm{tbf}_U(t)$, if $e_i^{\mathrm{I/O}} + \sum_{l \in \mathrm{hp}_i^{\mathrm{I/O}}} \Big\lceil \frac{r_i^{\mathrm{I/O},k}}{p_l^{\mathrm{I/O}}} \Big\rceil e_l^{\mathrm{I/O}} > \mathrm{sbf}_L(\max_i D_i^{\mathrm{I/O}})$, then Equation 14 is verified. Hence, it suffices to compute $\mathrm{sbf}_L(t), \overline{\mathrm{sbf}}_U(t)$ in $[0, \max_i D_i^{\mathrm{I/O}}]$.

Luckily, we do not need to run Algorithm 1 for all values of t in $[0, \max_i D_i^{\mathrm{I/O}}]$. Since functions $E_i^{\max}(t), E_i^{\min}(t)$ are piecewise linear, we can obtain $\overline{\mathrm{sbf}}_U(t)$ by interpolation of a finite number of values as shown in Algorithm 2. The algorithm returns $\overline{\mathrm{sbf}}_U(t)$ as a set p_{set} of points $\{\ldots, (t', val'), (t'', val''), \ldots\}$, where for $t' \leq t \leq t''$, $\overline{\mathrm{sbf}}_U(t) = val' + (t - t')\frac{val'' - val'}{t'' - t'}$, e.g. $\overline{\mathrm{sbf}}_U(t)$ is the linear interpolation between (t', val') and (t'', val''). The algorithm first computes in Line 4 the set $t^1, \ldots, t^{M-1}$ of angular points of $E_i^{\max}(t), E_i^{\min}(t)$ in interval $[0, \max_i D_i^{\mathrm{I/O}})$, with t^M being the maximum value of interest $\max_i D_i^{\mathrm{I/O}}$; note that this implies that in every open interval (t^j, t^{j+1}), each function $E_i^{\max}(t), E_i^{\min}(t)$ has a constant slope equal to either 0 or 1. In particular, function $\mathrm{EMaxAdd}(l, t^j)$ returns the slope of $E_l^{\max}(t)$ in interval (t^j, t^{j+1}), while $\mathrm{EMinAdd}(l, t^j)$ returns the slope of $E_l^{\min}9(t)$ in interval (t^j, t^{j+1}). For each point t^j, Algorithm 1 is applied to compute the value $val = \overline{\mathrm{sbf}}_U(t^j)$, as well as the index i of the last task for which variable x_i has been assigned a non-zero value. (t^j, val) is then inserted in p_{set} at Line 11.

Finally, the algorithm iterates in Lines 10-26, increasing the current time value t_{cur} starting at $t_{cur} = t^j$, until t_{cur} exceeds the next angular point t^{j+1}. Note that when Algorithm 1 is run to compute the solution to Equations 11-13 for $t_{cur} = t^j$, variables x_l will be assigned as follows:

$$x_l = E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur}) \quad \forall l, 1 \leq l \leq i-1, \quad (15)$$

$$x_l = 0 \quad \forall l, i < l \leq N, \quad (16)$$

$$x_i = t^j - \sum_{l=1}^{N} E_l^{\min}(t_{cur}) - \sum_{l \neq i} x_l = \quad (17)$$

$$= t^j - \sum_{l=1}^{i-1} E_l^{\max}(t_{cur}) - \sum_{l=i}^{N} E_l^{\min}(t_{cur}).$$

Now consider the solution $\overline{val}$, with variable assignment $\bar{x}_1, \ldots, \bar{x}_N$, computed by Algorithm 1 for time instant $\bar{t} = t_{cur} + \Delta$, where Δ is a small enough value. Note that as long as $\bar{t} \leq t^{j+1}$, then $E_l^{\max}(\bar{t}) = E_l^{\max}(t_{cur}) + \Delta\mathrm{EmaxAdd}(l, t_{cur})$ and similarly $E_l^{\min}(\bar{t}) = E_l^{\min}(t_{cur}) + \Delta\mathrm{EminAdd}(l, t_{cur})$. Furthermore, define $div \equiv \sum_{l=1}^{i-1} \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \mathrm{EMinAdd}(l, t^j)$ as computed in Line 12. Then based

on Equations 15-17, we obtain:

$$\bar{x}_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t}) \quad \forall l, 1 \le l \le i-1, \tag{18}$$

$$\bar{x}_l = 0 \quad \forall l, i < l \le N, \tag{19}$$

$$\begin{aligned}\bar{x}_i &= \bar{t} - \sum_{l=1}^{i-1} E_l^{\max}(\bar{t}) - \sum_{l=i}^{N} E_l^{\min}(\bar{t}) = \\ &= t^j + \Delta(1-div) - \sum_{l=1}^{i-1} E_l^{\max}(t_{cur}) - \sum_{l=i}^{N} E_l^{\min}(t_{cur}) = \\ &= x_i + \Delta(1-div),\end{aligned} \tag{20}$$

$$\begin{aligned}\overline{val} &= \sum_{l=1}^{N}(\delta_l + \alpha_l E_l^{\min}(\bar{t})) + \sum_{l=1}^{N}\alpha_l \bar{x}_l = \\ &= \sum_{l=1}^{N}\delta_l + \sum_{l=1}^{i-1}\alpha_l E_l^{\max}(\bar{t}) + \sum_{l=i}^{N}\alpha_l E_l^{\min}(\bar{t}) + \alpha_i \bar{x}_i = \\ &= \Big(\sum_{l=1}^{N}\delta_l + \sum_{l=1}^{i-1}\alpha_l E_l^{\max}(t_{cur}) + \sum_{l=i}^{N}\alpha_l E_l^{\min}(t_{cur}) + \\ &+ \alpha_i x_i\Big) + \sum_{l=1}^{i-1}\alpha_l \Delta\mathrm{EMaxAdd}(l, t^j) + \\ &+ \sum_{l=i}^{N}\alpha_l \Delta\mathrm{EMinAdd}(l, t^j) + \alpha_i\Delta(1-div) = \\ &= val + \sum_{l=1}^{i-1}\alpha_l \Delta\mathrm{EMaxAdd}(l, t^j) + \\ &+ \sum_{l=i}^{N}\alpha_l \Delta\mathrm{EMinAdd}(l, t^j) + \alpha_i\Delta(1-div),\end{aligned} \tag{21}$$

and such solution is valid as long as $\Delta \le t^{j+1} - t_{cur}$ and furthermore Δ is small enough that it holds: $0 \le \bar{x}_i \le E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$; note that based on Equation 20, the latter condition can be rewritten as:

$$x_i + \Delta(1-div) \ge 0, \tag{22}$$

$$\begin{aligned}x_i + \Delta(1-div) \le E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) + \\ + \big(\mathrm{EMaxAdd}(i, t^j) - \mathrm{EMinAdd}(i, t^j)\big)\Delta.\end{aligned} \tag{23}$$

Since $\bar{x}_i$ can be increasing or decreasing with Δ based on the value of div, we have to consider several different cases. In all cases, note that Equation 21 is linear in Δ, hence as long as the Equation holds in an interval (t', t''), the solution to Equations 11-13 can be obtained by linear interpolation of $(t', val'), (t'', val'')$.

Case (1): $div > 1$ (Lines 14-17). The constraint in Equation 23 is verified for all Δ. Solving Equation 22 yields $\Delta \le x_i/(div-1)$. We have two subcases **(a)** and **(b)**. **(a)** Assume that $x_i/(div-1) < t^{j+1} - t_{cur}$. Then Equations 20, 21 are valid in the interval $[t_{cur}, t_{cur} + x_i/(div-1)]$. Note that for $\Delta = x_i/(div-1)$, $\bar{x}_i = 0$. Furthermore, we have $\alpha_i\Delta(1-div) = -\alpha_i x_i$. Hence, $\overline{val}$ can be computed as in Line 16, new values for t_{cur}, val are assigned as $t_{cur} := t_{cur} + x_i/(div-1)$, $val := \overline{val}$ and the new point (t_{cur}, val) is inserted in Line 11 at the next iteration. Finally, note that at the next iteration, all variables x_l with $l < i-1$ are still assigned their maximum value $E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur})$ and all $x_l, l \ge i$ are set to 0. Hence, we can apply the same reasoning as in Equations 18-21 after setting $i := i-1$ as in Line 17. In particular, note that i can never be assigned the invalid value 0. By contradiction, assume that in the current iteration $i = 1$. Then all $x_l = 0$, meaning that $\bar{t} = \sum_{l=1}^{N} E_l^{\min}(\bar{t})$ according to Equation 12; this in turn implies $\sum_{l=1}^{N} \mathrm{EMinAdd}(l, t^j) = 1$, which contradicts $div > 1$. **(b)** Assume that $x_i/(div-1) \ge t^{j+1} - t_{cur}$. Then Equations 20, 21 are valid in the interval $[t_{cur}, t^{j+1}]$, the condition of the while loop in Line 10 becomes false and the point (t^j, val) is added to p_{set} in the next iteration of the for loop at Line 7.

Case (2): $div = 0$ and $\mathrm{EMaxAdd}(i, t^j) = 0$ (Line 19-22). Note $div = 0$ implies $\mathrm{EMinAdd}(i, t^j) = 0$. The constraint in Equation 22 is verified for all Δ. Solving Equation 23 yields $\Delta \le E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$. Once again we have two subcases. **(a)** Assume that $E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i < t^{j+1} - t_{cur}$. Then Equations 20, 21 are valid in the interval $[t_{cur}, t_{cur} + E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i]$. Note that for $\Delta = E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$, $\bar{x}_i$ is set to the maximum value $E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$. Also note that in Equation 21, $\alpha_i\Delta(1-div) = \alpha_i\Delta$ and $\sum_{l=1}^{i-1}\alpha_l\Delta\mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N}\alpha_l\Delta\mathrm{EMinAdd}(l, t^j) = 0$, hence $\overline{val}$ can be computed as in Line 21. The same considerations as in Case (1a) then apply except that we set $i := i+1$ since in the next iteration, all variables x_l with $l \le i$ are assigned their maximum value $E_l^{\max}(t_{cur}) - E_l^{\min}(t_{cur})$ and all $x_l, l > i+1$ are still set to 0. Note that it is possible to set i to the invalid value $N+1$; this is covered in Case (6). **(b)** Assume that $E_i^{\max}(t^j) - E_i^{\min}(t^j) - x_i \ge t^{j+1} - t_{cur}$. Then the same considerations as in Case (1b) apply.

Case (3): $div = 1$ and $\mathrm{EMaxAdd}(i, t^j) = 0, \mathrm{EMinAdd}(i, t^j) = 1$ (Line 19-22). As in Case (2), the constraint in Equation 22 is verified for all Δ, while solving Equation 23 yields $\Delta \le E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$. Furthermore, note that for $\Delta = E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$, $\bar{x}_i$ is again set to the maximum value $E_i^{\max}(\bar{t}) - E_i^{\min}(\bar{t})$ and from Equation 21 we obtain $\overline{val} = val + \alpha_i\Delta$. Hence, this case is equivalent to Case (2a)-(2b) and can be handled by the same Lines 19-22.

Case (4): $div = 0$ and $\mathrm{EMaxAdd}(i, t^j) = 1$ (Line 24). As in Case (2), note $div = 0$ implies $\mathrm{EMinAdd}(i, t^j) = 0$. Then both Equations 22 and 23 are verified for all Δ. Hence, as in Case (1b), Equations 20, 21 are valid in the interval $[t_{cur}, t^{j+1}]$ and the point (t^j, val) is added to p_{set} in the next iteration of the for loop.

Case (5): $div = 1$ and either $\mathrm{EMaxAdd}(i, t^j) = 1$ or

Algorithm 2 Compute $\overline{\mathrm{sbf}}_U(t)$ in $[0, \max_i D_i^{\mathrm{I/O}}]$.

1: **procedure** COMPUTEINTERVAL($\max_i D_i^{\mathrm{I/O}}, \tau_1, \ldots, \tau_N$)
2: $\forall i, 1 \leq i \leq N$: compute α_i, δ_i.
3: Order $\tau_i, \ldots, \tau_N$ by non-increasing values of α_i.
4: Compute $t^1, \ldots, t^{M-1}$ as the set of angular points for any $E_i^{\max}(t), E_i^{\min}(t)$ in $[0, \max_i D_i^{\mathrm{I/O}})$.
5: $t^M := \max_i D_i^{\mathrm{I/O}}$
6: $p_{set} := \emptyset$
7: **for** $j = 1 \ldots M-1$ **do**
8: $(val, i, \{x_1, \ldots, x_N\}) = \mathrm{ComputeInstant}(t^j, \tau_1, \ldots, \tau_N)$
9: $t_{cur} := t^j$
10: **while** $t_{cur} < t^{j+1}$ **do**
11: add (t_{cur}, val) to p_{set}
12: $div := \sum_{l=1}^{i-1} \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \mathrm{EMinAdd}(l, t^j)$
13: **if** $div > 1$ **then**
14: $\Delta := x_i/(div - 1)$
15: $t_{cur} := t_{cur} + \Delta$
16: $val := val + \Delta\big(\sum_{l=1}^{i-1} \alpha_l \mathrm{EMaxAdd}(l, t^j) + \sum_{l=i}^{N} \alpha_l \mathrm{EMinAdd}(l, t^j)\big) - \alpha_i x_i$
17: $i := i - 1$
18: **else if** $(div = 0$ & $\mathrm{EMaxAdd}(i, t^j) = 0) \mid (div = 1$ & $\mathrm{EMaxAdd}(i, t^j) = 0$ & $\mathrm{EMinAdd}(i, t^j) = 1)$ **then**
19: $\Delta := E_i^{\max}(t_{cur}) - E_i^{\min}(t_{cur}) - x_i$
20: $t_{cur} := t_{cur} + \Delta$
21: $val := val + \alpha_i \Delta$
22: $i := i + 1$
23: **else**
24: break
25: **end if**
26: **end while**
27: **end for**
28: **return** p_{set}
29: **end procedure**

$\mathrm{EMinAdd}(i, t^j) = 0$ (Line 24). Then again both Equations 22 and 23 are verified for all Δ, hence this case is equivalent to Case (4).

Case (6): $i > N$ (Line 24). This can only happen if in the previous iteration either Case (2a) or (3a) is executed with $i = N$. Note that in both cases, in the current iteration $div = \sum_{l=1}^{N} \mathrm{EMaxAdd}(l, t^j) = 0$, hence Line 24 is executed. We show that the assignment where each variable is maximal, e.g. $x_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})$, is optimal for all $\bar{t}$ in the interval $[t_{cur}, t^{j+1}]$. Equation 12 is equivalent to: $\sum_{l=1}^{N} \big(E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})\big) \leq \bar{t} - \sum_{l=1}^{N} E_l^{\min}(\bar{t})$, which is in turn equivalent to: $\sum_{l=1}^{N} E_l^{\max}(\bar{t}) \leq \bar{t}$. But since $\mathrm{EMaxAdd}(l, t^j) = 0$ for all l and furthermore at t_{cur} it must hold $\sum_{l=1}^{N} E_l^{\max}(t_{cur}) \leq t_{cur}$, Equation 12 is verified for all $\bar{t}$ in $[t_{cur}, t^{j+1}]$. Hence, the assignment $x_l = E_l^{\max}(\bar{t}) - E_l^{\min}(\bar{t})$ must result in the optimum. Computing $\overline{val}$ according to Equation 21 results in $\overline{val} = val$, hence $\overline{\mathrm{sbf}}_U(t^j)$ is constant between point (t_{cur}, val) and point (t^j, val), which is added to p_{set} in the next iteration of the for loop.

We can now prove our main theorem.

Theorem 13. *A correct upper bound $\overline{\mathrm{sbf}}_U(t)$ to $\overline{\mathrm{sbf}}(t)$ in the interval $[0, \max_i D_i^{\mathrm{I/O}}]$ can be computed by linear interpolation of the point set returned by Algorithm 2.*

Proof. Based on the discussion above, whenever Algorithm 2 inserts two consecutive points $(t', val'), (t'', val'')$ in p_{set}, $\overline{\mathrm{sbf}}_U(t)$ can be computed by linear approximation of $(t', val'), (t'', val'')$ for all t in the interval $[t', t'']$. Since furthermore points are inserted for the beginning and end of the interval $[0, \max_i D_i^{\mathrm{I/O}}]$, to conclude the proof it remains to show that the algorithm terminates; this is not obvious, since in the while loop in Lines 10-26, Δ can be set to 0 and furthermore i can be either incremented or decremented. We show that if i is incremented in the first iteration of the while loop, then it can not be decremented in further iterations; similarly, if i is decremented in the first iteration, it can not be incremented afterwards. Since $i \geq 1$ by Case (1a), and furthermore the while loop is terminated whenever $i > N$, it follows that the loop is executed at most $N + 1$ times.

By contradiction, assume that Lines 14-17 are executed in one iteration, and Lines 19-22 in the following. Note that

if i is incremented or decremented by 1, the value of div can only change by 1. Hence, it must hold $div = 2$ in the first iteration and $div = 1$ in the second. However, this implies $\mathrm{EMaxAdd}(i, t^j) = 1$ in the second iteration, hence Lines 19-22 can not be executed. Similarly, assume that Lines 19-22 are executed in one iteration, and Lines 14-17 in the following; it must hold $div = 1$ in the first iteration and $div = 2$ in the second. This implies $\mathrm{EMinAdd(i, t^j)} = 0$ in the first iteration, hence Lines 19-22 can not be executed. □

It remains to discuss the computational complexity of Algorithm 2. Note that in a time window of length t, the number of angular points for $E_i^{\max}(t)$ or $E_i^{\min}(t)$ are not more than $2 + 2\lceil \frac{t}{p_i} \rceil$. Hence, an upper bound to the number of iterations of the for loop at Line 7 can be computed as $N\Big(4 + 4\lceil \frac{\max_i D_i^{\mathrm{I/O}}}{\min_i p_i} \rceil\Big) + 1$, where the 1 term accounts for $t^M = \max_i D_i^{\mathrm{I/O}}$. Lines 8-9 can be executed in $O(N)$. Based on the proof of Theorem 13, the while loop at Line 10 is repeated at most $N + 1$ times. Finally, note that Lines 11-25 can be optimized to run in constant time rather than linear time. This is because the value of i changes by 1 between successive iterations. Hence, the summations in Lines 12, 16 can be computed based on their values at the previous step in constant time. In conclusion, Algorithm 2 has a pseudo-polynomial complexity of $O\big(N^2 \max_i D_i^{\mathrm{I/O}} / \min_i p_i\big)$.