Coscheduling of Real-Time Tasks and PCI Bus Transactions

Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, Lui Sha
Department of Computer Science, University of Illinois at Urbana-Champaign
{rpelliz2, bachbui2, mcaccamo, lrs}@cs.uiuc.edu

Abstract

Integrating COTS components in critical real-time systems is challenging. In particular, we show that the interference between cache activity and I/O traffic generated by COTS peripherals can unpredictably slow down a real-time task by up to 44%. To solve this issue, we propose a framework comprised of three main components: 1) a COTS-compatible device, the peripheral gate, that controls peripheral access to the system; 2) an analytical technique that computes safe bounds on the I/O-induced task delay; 3) a coscheduling algorithm that maximizes the amount of allowed peripheral traffic while guaranteeing all real-time task constraints. We implemented the complete framework on a COTS-based system using PCI peripherals, and we performed extensive experiments to show its feasibility.

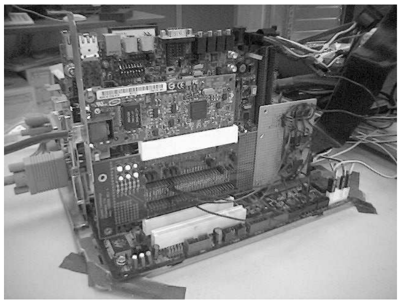
1. Introduction

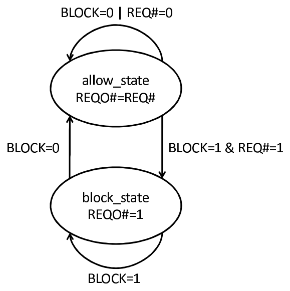
Modern embedded systems are increasingly built by using Commercial Off-The-Shelf (COTS) components in an attempt to reduce costs and time-to-market. This trend is true even for companies in the safety-critical avionic market such as Lockheed Martin Aeronautics, Boeing and Airbus [3]: it is becoming difficult to rely on completely specialized hardware and software solutions since development time and costs raise dramatically while performance is often lower when compared to equivalent COTS components commonly used for general purpose computers. For example, the specialized SAFEbus backplane [5] used in the Boing777 is capable of transferring data up to 60 Mbps, while a modern COTS interconnection such as PCI Express 2.0 [14] can reach transfer speeds over three orders of magnitude greater at 16 Gbyte/s.

Unfortunately, the predictable integration of COTS components in real-time systems poses significant challenges from a timing perspective. In particular, in this paper we focus on the interaction between the CPU cache and COTS peripherals contending for shared main memory access. We assume that peripherals are connected to the system through a peripheral interconnection such as the PCI bus, and that they have master (also called DMA) capabilities: they can directly initiate read/write transactions towards either other peripherals or the main memory. Bus master mode is essential to avoid overloading the processor, especially in the case of fast I/O interfaces that could otherwise produce millions of interrupts per second. However, since the memory is a shared resource in the system, peripheral transactions can interfere with cache line fetches produced by the CPU memory controller whenever a task experiences a cache miss. This interaction can slow down task execution tremendously: our experiments in Section 5 show that task execution time in the presence of heavy I/O load is increased up to 44%.

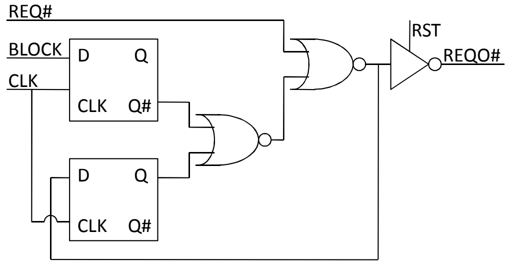
The described effect is potentially dangerous for real-time embedded systems that employ a partitioned architecture, such as the ARINC 653 avionic standard [2]: different computational components are put into isolated partitions, each of which is assigned a fixed, cyclic time slice of the CPU. However, the standard does not offer any isolation from the effects of bus traffic: COTS peripherals on the market do not typically provide any form of time protection/virtualization. Hence, a peripheral assigned to one partition is free to transmit and interfere with cache fetches while another partition is executing on the CPU.

In general, two types of solutions can be feasibly applied to this problem. The first is to account for the effect of all peripheral traffic in the worst case computation time (wcet) of each task. Note that since different partitions and peripherals are typically integrated very late in the development cycle, testing is not enough. Instead, we need an analysis that can compute the increase in wcet given design-time bounds on peripheral traffic. The problem of this solution is that, as already mentioned, the wcet increment can be very large, up to 44%. The approach that we propose in this paper is to *coschedule* CPU tasks and I/O transactions: in fact, assuming we find a way to control when peripherals are allowed to transmit, we can create a bus transmission schedule and synchronize it with the CPU task schedule. We can then formulate our coscheduling objective as follows: maximize the traffic transmitted by each peripheral, while guaranteeing that each task meets its original deadline.


(a) ***P-Gate controlling a Network Card***



(b) ***State Machine***



(c) ***Schematic***

Figure 1: Peripheral Gate

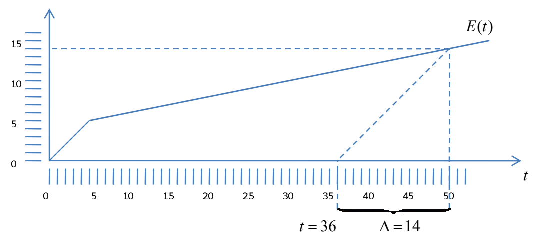
Previous work. In a previous paper [15], we introduced an analysis to compute wcet for a task subject to peripheral interference given a trace of the task's cache activity and an upper bound function on the amount of traffic transmitted by peripherals. However, the analysis makes a fairly restrictive assumption: it must know the exact time at which each cache miss is produced for a specific task run. This information can not be gathered by running the task on real hardware; rather, a CPU simulator must be employed. Writing such simulator is difficult for an embedded application, as all used peripherals must also be simulated.

Key contributions. We provide three main contributions. First of all, in Section 2 we introduce a device for the PCI/PCI-X bus, called a *peripheral gate* (or *p-gate* for short), that allows us to control peripheral access to the bus. The implemented p-gate is compatible with COTS devices: no modification to either the peripheral or the motherboard is required. Second, in Section 3 we provide a new wcet analysis that removes our previous restrictive assumption. The main idea is to divide each task into a series of *superblocks*; each superblock can include branches and loops, but superblocks must be executed in sequence. By running the task, the CPU can collect information on the number of cache misses in each superblock. We can then compute a safe wcet bound by determining a worst case arrival pattern of cache misses in each superblock. As we show in Section 3.3, the result is rather counterintuitive when compared to the classic critical instant theorem [11]; in fact, spreading the cache misses throughout the superblock results in higher wcet than when all cache misses happen at its beginning. Furthermore, while the computed bound is more pessimistic than the one in [15] since the available information is coarser, in practice for many applications of interest the difference is negligible. Finally, our third contribution described in Section 4 is a run-time coscheduling heuristic that builds on top of the described p-gate and wcet analysis. We implemented a new peripheral (based on an FPGA board), the *reservation controller*, which executes the coscheduling algorithm and controls all p-gates. The reservation controller uses run-time information provided by the OS to compute available task slack and it dynamically opens the p-gates when it is safe to do so. In Section 5, we show that our heuristic performs well compared to the best possible run-time, adaptive and predictive algorithm. We conclude by discussing related work in Section 6 and future work in Section 7.

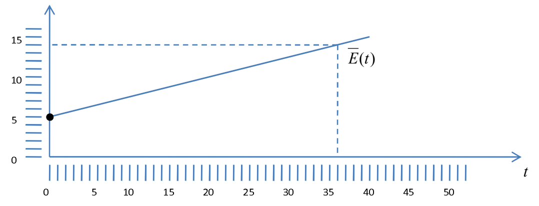
2. Peripheral Gate

In this section, we first provide a brief overview of the Peripheral Component Interconnect (PCI) standard and then describe our p-gate implementation. PCI is the current standard family of architectures for motherboard - peripheral interconnection in the personal computer market; it is also widely popular in the embedded domain [14]. The standard can be divided in two parts: a *logical* specification, which details how the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. Several widely different physical specifications have been published; here we focus on the PCI/PCI-X physical specification, which uses a shared bus architecture with support for multiple bus segments connected by bridges. To gain access to the shared bus, each peripheral must first obtain permission from the bus segment arbiter using a standard handshake with two point-to-point, active-low wires, `REQ#` and `GNT#`. The peripheral first lowers `REQ#` to signal a request for the bus, and the arbiter grants permission by lowering `GNT#`. The peripheral then waits for the bus to become free and starts a data transfer (also called a bus *transaction*). The handshake finishes after both the peripheral and the arbiter raise `REQ#` and `GNT#` in succession; if the peripheral wants to initiate another transaction, it must reacquire the grant.

We implemented the p-gate based on a PCI extender card, a debug card that is interposed between the peripheral card and the motherboard and provides easy access to all signals. We modified the card to intercept the `REQ#` signal and to control it based on an input `block` signal coming from the reservation controller. The main idea is to force



(a) ***Upper Bound Function*** $E(t)$**.**



(b) ***Modified Upper Bound Function*** $\bar{E}(t)$**.**

Figure 2: Peripheral Load Functions.

REQ# to remain high whenever block is active; in this way, the peripheral is not able to get the grant from the arbiter and thus can not transmit. The actual implementation is more complex: if block is raised while REQ# is active low, we could violate the PCI specification by immediately deactivating REQ#. Instead, we must allow the current request to finish and then we can block all further requests. A corresponding synchronous state machine is shown in Figure 1(b) and an optimized schematic in Figure 1(c), where REQ# is the input from the peripheral and REQO# is the controlled output to the arbiter. Our implementation uses discreet components: two positive-edge-triggered D flip-flops, two nor gates and an inverting tri-state buffer. The output buffer is required by the specification to set the output to high impedance whenever the bus is reset. We measured a worst case propagation delay for the circuit of 7ns, which allowed us to run the bus at a frequency up to 66Mhz.

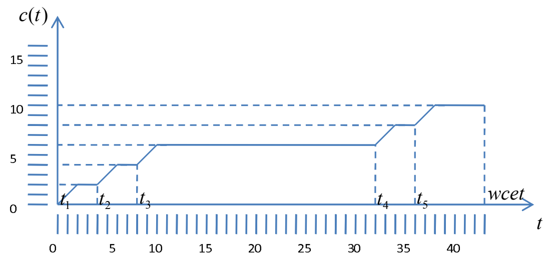
The reservation controller outputs a block signal for each p-gate in the system. We implemented a prototype reservation controller based on a Xilinx ML505 board. The board is connected to the system using a PCI-E motherboard slot, and uses a Virtex-5 FPGA to implement a custom peripheral. All registers used by the peripheral are memory mapped; a PCI driver is used to allocate the registers in the CPU virtual memory space, hence tasks running in user mode can communicate with the peripheral performing memory reads/writes. The reservation controller can run in two different modes: in *data acquisition mode* (see Section 3.2) it simply collects statistics about the task execution while keeping all p-gates closed. In *execution mode* (see Section 4) it runs the coscheduling algorithm and dynamically controls the p-gates. This solution moves as much computation as possible in hardware on the FPGA, thus minimizing the overall CPU overhead.

3. Wcet Analysis

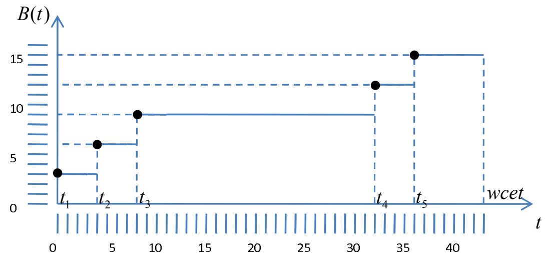
In [15], we first presented an analysis to compute the worst case delay suffered by a task due to peripheral interference. We consider a typical COTS architecture where the processor is connected to the rest of the system through a dedicated bus known as the Front Side Bus (FSB). The system memory is connected to the FSB as a slave device. Similarly, other peripherals can be either connected directly on the FSB [1] or located on a separate interconnection such as PCI and connected to the FSB through a bridge [14]. The analysis computes bounds on task delay induced by contention at the FSB level under the following assumptions: all FSB transactions initiated by the CPU consist of cache replacement/fetches from main memory to the last cache level, and all other FSB transactions consist of DMA reads/writes in main memory initiated by peripherals. The CPU is stalled whenever waiting for a cache miss (i.e. no hyperthreading is used), and the FSB protocol is known, in particular regarding the arbitration used for accessing the bus and the maximum length of peripheral transactions.

Information about both the task under analysis and the load imposed by peripherals on the bus is needed. For each peripheral we consider an upper bound on the amount of time that the FSB is busy executing read/write transactions initiated by that peripheral. We can then sum the bounds for all peripherals to obtain a cumulative function $E(t)$: for every $t > 0$, the total time that the FSB is occupied executing peripheral transactions in any interval of length t is at most equal to $E(t)$. An example of bound $E(t)$ is shown in Figure 2(a). For tasks, the analysis in [15] assumes that a precise pattern of cache misses is available. The pattern can be used to produce a *cache access function* $c(t)$ for the task. An example of cache access function is plotted in Figure 3, where the x axis represents the task execution time and the y axis represents the cumulative cache replacement/fetch time on the FSB, assuming no interference from peripheral activity. At each time t, a slope of 1 indicates that the CPU is stalled waiting for a cache line replacement/fetch, while a slope of 0 indicates that the CPU is executing task code. Each cache access function corresponds to a specific execution of the task. In general, if a task is tested with M different input vectors, then M different functions must be derived and the analysis is run M times.

We now briefly describe the main results obtained in [15]. For simplicity, we assume that each cache miss results in a single cache line fetch with no write-back, taking a constant time L to complete the read transaction (in all figures, $L = 2$). The maximum length of a peripheral transaction is $L' = 3$, and the FSB uses round-robin, non-preemptive scheduling (in [15] it is shown how each of these



(a) ***Access Function*** $c(t)$**.**



(b) ***Blocking Function*** $B(t)$**.**

Figure 3: Cache Functions.

assumptions can be lifted by slightly modifying the analysis). Let N be the number of cache fetches for the task under analysis. We denote the set of N fetches, in temporal order, as $\{f_1, f_2, \ldots, f_N\}$. For each fetch f_i, let t_i be the time at which the cache fetch operation is initiated in the cache access function $c(t)$ (see Figure 3(a)), i.e. the fetch start time in the schedule unmodified by peripheral activity. Then $\forall i, j : 1 \leq i \leq j \leq N$, $D(f_i, f_j)$ represents the worst case cumulative delay suffered by all fetches in $\{f_i, \ldots, f_j\}$. The main idea of the analysis is to iteratively derive bounds on $D(f_i, f_j)$ in order to finally obtain an upper bound Ub on the overall delay $D(f_1, f_N)$ suffered by the task.

To simplify the analysis, we introduce new representations for peripheral load and cache misses. We define the *cache blocking function* $B(t)$ as follows:

$$\begin{cases} \forall t, t < t_1 : & B(t) = 0 \\ \forall i, 1 \leq i < N, \forall t, t_i \leq t < t_{i+1} : & B(t) = iL' \\ \forall t, t \geq t_N : & B(t) = NL' \end{cases} \tag{1}$$

The $B(t)$ function associated to the $c(t)$ function of Figure 3(a) is shown in Figure 3(b). Intuitively, $B(t)$ represents the maximum cumulative time that fetch operations can be delayed by the effect of peripheral transactions: since the bus arbitration is round-robin, each fetch can be delayed for at most the maximum length L' of any one peripheral transaction. We prove the following lemma:

Lemma 1 (1 in [15]) *For each* $i, j : i \leq j$, $D(f_i, f_j) \leq B(t_j) - B(t_i^-)$, *where* $B(t^-) = \lim_{x \to t^-} B(x)$.

Lemma 1 provides a delay bound based on the trace of cache misses. A bound based on the peripheral load is shown in the next lemma.

Lemma 2 (2 in [15]) *Let* $\bar{E}(t) = \sup\{\Delta | \Delta \leq E(t + \Delta)\}$. *Then* $\bar{E}(t_j - t_i)$ *is an upper bound for* $D(f_i, f_j)$.

A graphical representation of the *modified peripheral load function* $\bar{E}(t)$ is shown in Figure 2(b). The main intuition behind this result is that there is a circular dependency between the amount of peripheral load that interferes with $\{f_i, \ldots, f_j\}$ and the delay $D(f_i, f_j)$: when peripheral traffic is injected on the FSB, the start time of each fetch is delayed. In turn, this increases the time interval between f_i and f_j and therefore more peripheral traffic can now interfere with those fetches. Our key idea is that we do not need to modify the start times $\{t_i, \ldots, t_j\}$ of fetches when we take into account the peripheral traffic injected on the FSB: instead, we can take this effect into account using the equation that defines $\bar{E}(t)$, where Δ represents both the maximum delay suffered by fetches and the increase in the time interval for interfering traffic. Lemmas 1, 2 can be combined resulting in the following main Theorem:

Theorem 3 (3 in [15]) *For each* $i, j : i \leq j$, $\min(B(t_j) - B(t_i^-), \bar{E}(t_j - t_i))$ *is an upper bound for* $D(i, j)$.

Algorithm 1 Compute $D(f_1, f_N)$

```
1: $Ub := 0$
2: $Q := \{\}$
3: for $j = 1 \ldots N$ do
4:   add $(t_j, y_i = 0)$ to $Q$
5:   $u_j = \min\left(B(t_j) - B(t_j^-), \min_{z_i=(t_i,y_i) \in Q}\{\bar{E}(t_j - t_i) - y_i\}\right)$
6:   for all $z_i = (t_i, y_i)$ in $Q$ do
7:     $y_i := y_i + u_k$
8:   $Ub := Ub + u_k$
9: return $Ub$
```

While Theorem 3 expresses an upper bound Ub on the delay $D(i, j)$, unfortunately Ub is not tight, since it could be refined by splitting $\{f_i, \ldots, f_j\}$ into subintervals and recomputing the bound on each subinterval. Using the strategy of iteratively computing bounds on multiple subintervals, Algorithm 1 was first first introduced in [15] to compute a tight bound on the overall delay $D(f_1, f_N)$ in $O(N^2)$ time. Algorithm 1 iteratively computes a delay term u_j for each fetch f_j; the delay terms are then added together to obtain the final bound on $D(f_1, f_N)$. In the algorithm description, Q is a list of pairs $z_i = (t_i, y_i)$ ordered by increasing values of t_i. For each pair, t_i is the time associated with fetch f_i, while y_i represents the accumulated delay, i.e. at step k of the algorithm, $y_i = \sum_{k=i}^{j-1} u_k$. The following theorem provides the main result of [15]:

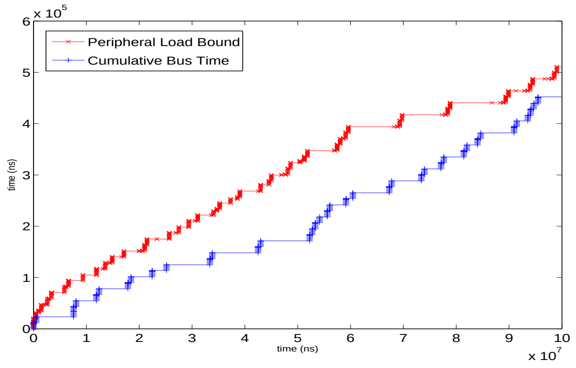


Figure 4: Measured Peripheral Load.

Theorem 4 (7 in [15]) *Algorithm 1 computes an upper bound Ub to $D(f_1, f_N)$. The bound is tight, in the sense that there exists a schedule of peripheral transactions consistent with $E(t)$ for which $D(f_1, f_N) = Ub$.*

The main problem of the described analysis is that obtaining a precise cache access function is very hard. Both running the task on real hardware and using static analysis [16] only provides imprecise information, i.e. number of cache misses in an interval. While it is theoretically possible to use a cycle-accurate CPU emulator, in practice it is very difficult: the exact architecture of the CPU and cache must be known, which is often not the case in COTS systems, and all used peripherals must be simulated as well. Therefore, in this paper we use an alternative solution: we assume that at compile time, a control flow graph for the task can be derived comprised of a series of S *superblocks* $\{s_1, \ldots, s_S\}$. Each superblock can include branches and loops, but superblocks must be executed in sequence. For each s_i, we measure the worst case execution time $wcet_i$ without peripheral interference and the worst case number of cache misses CM_i, either through static analysis or making use of CPU self-measures. We can then obtain a safe bound on task delay in the following way: for each superblock s_i, we consider the worst case pattern of CM_i cache misses in an interval of length $wcet_i$, i.e. the pattern that results in the highest possible delay. In the remaining of this section, we first detail how to obtain the described measurement in a concrete setting and then we provide our new analysis results.

3.1. Peripheral Load Evaluation

The peripheral load function can be obtained in two ways. If the peripheral is an I/O interface and the node is part of a distributed system using a real-time communication protocol, then a bound on the peripheral activity can be derived analytically. Otherwise, we propose a testing methodology that is well suited to the analysis of COTS peripherals[1]. A trace of activity for a PCI/PCI-X peripheral can be gathered monitoring the bus with a logic analyzer. For example, Figure 4 shows the first 100ms of a measured trace for a 100Mb/s ethernet network card in term of cumulative time taken by peripheral transactions on a 32bit, 33Mhz PCI bus segment; the whole recorded trace consisted of 1000 transactions. We developed a simple algorithm that computes the peripheral load function $E(t)$ from a trace in quadratic time in the number of bus transactions. The algorithm performs a double iteration over all transactions, computing at each step the amount of peripheral traffic in an interval between the beginning of any transaction and the end of any other successive transaction. The computed values are inserted into a list ordered by interval length, and all non maximal values are culled. Figure 4 shows the resulting $E(t)$ function in the interval $[0, 100ms]$. If multiple traces are recorded, then an upper bound can be computed by merging the computed load functions for each trace and again removing all non maximal values. Finally, note that the computed $E(t)$ expresses a load bound for the bus segment on which the peripheral is located. In the case of the PCI/PCI-X architecture, the segment is connected to the FSB through one or multiple bridges, each of which has buffering capabilities. In this case, a safe bound on the generated FSB load can be obtained summing a factor B/C to the computed $E(t)$ function, where B is the sum of the sizes of all traversed buffers and C is the speed of the FSB.

3.2. Cache Miss Measurement

We devised a testing methodology to experimentally obtain the worst case execution time and worst case number of cache misses for each superblock. Our implementation uses the Intel Core Microarchitecture architectural performance counters [7], but other CPU architectures such as IBM PowerPC provide similar support for CPU self-measures. Support was added by modifying the Linux/RK kernel [13]. The Core Microarchitecture specifies support for three architectural performance counters, each of which can be configured to count a variety of internal events. In particular, we used Counter 0 to count the number of elapsed CPU clock cycles and Counter 1 to count the number of level-2 cache misses. To accurately measure task execution without the effects of OS overhead, we configured both counters to be active only when the CPU is executing in user mode. Finally, we allowed reading the counter values from user mode with the `rdpmc` instruction (the counters can still be written and configured only in kernel mode) to reduce measurement overhead.

1 While testing can fail to reveal the real worst case, we argue that it is nevertheless an accepted and commonly used methodology in the industry.

```
cpuid; //synchronization barrier
mov ECX, 0000 0000H;
rdpmc; //read Counter 0
//move value from DL:EAX to reservation controller
mov [RESCON_COUNTER0_H], DL;
mov [RESCON_COUNTER0_L], EAX;
mov ECX, 0000 0001H;
rdpmc; //read Counter 1
mov [RESCON_COUNTER1_H], DL;
mov [RESCON_COUNTER1_L], EAX;
```

Figure 5: Checkpoint Assembler Code.

Counters are read inside each task by adding the checkpoint code in Figure 5 at the end of each superblock. The `cpuid` instruction inserts a synchronization barrier, i.e. it makes sure that all instructions fetched before `cpuid` are completed before the counters are read; this is required to cope with out-of-order execution. The counter values are then sent to the reservation controller running in data collection mode; this ensures that no write to system memory is performed at the checkpoint, which could cause an additional cache miss. The reservation controller determines the execution time and number of cache misses in each superblock computing the difference between the values obtained in successive checkpoints. After the task has finished, the computed values are read back from the reservation controller and $wcet_i$ and CM_i can be determined as the worst case over several task runs. Note that performance counters are not task-specific, so we had to modify the kernel to support reading the counters in a multitasking environment. We added two new fields to the task descriptor, `counter_extime` and `counter_cmisses`, to store the counter information. When a task is created, the fields are set to zero. When a task is preempted, the kernel first reads the counter values and saves them in the preempted task's descriptor. Then, it writes the values in the preempting task's descriptor back in the counters. Finally, the kernel writes the id of the preempting task in a register of the reservation controller, so that the controller can correctly associate the received information with the running task.

We implemented a compiler pass using the LLVM compiler infrastructure [9] to automatically add checkpoint code to the task. In the current implementation, the designer must manually identify the superblock boundaries selecting an initial and final basic block for each superblock. The choice involves a tradeoff, as smaller superblocks provide better information and tighter wcet bounds but at the price of increased measurement overhead.

3.3. Analysis Results

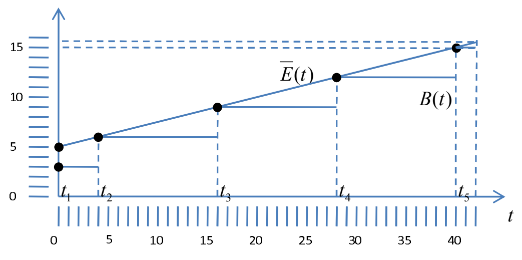
Given load function $E(t)$ and values $wcet_i, CM_i$ for superblock s_i, we first tackle the problem of determining the worst case delay $D(s_i)$ suffered by all cache misses in s_i. The key idea is that since Lemma 2 implies that the delay depends on the amount of incoming traffic during the interval between successive fetches, a worst case pattern can be produced by "spreading out" the cache misses over the length $wcet_i$ of the superblock. Figure 6(a) provides a clarifying example, where $wcet_i = 42, CM_i = 5$, and $E(t)$ is the same as the previous examples. The worst case is produced when fetch f_1 starts at $t = 0$, and each successive fetch starts at time t_j, such that $\bar{E}(t_j - 0)$ provides just enough traffic to cause maximum delay for $\{f_1, \ldots, f_j\}$ (remember that t_i is the start time of fetch f_i in the schedule without traffic, as the effect of delay is fully captured in $\bar{E}(t)$). Since all fetches must be completed before each checkpoint, it must hold $t_{CM_i} \leq wcet_i - L$. Furthermore, since there are CM_i cache misses in the interval, the maximum delay is also bounded by $L' \cdot CM_i$. We can then obtain an upper bound on the delay $D(s_i, s_j)$ suffered by fetches in superblocks $\{s_i, \ldots, s_j\}$ using the following theorem, where $t_i^s = \sum_{j<i} wcet_j$ is intuitively the start time of superblock s_i.

Theorem 5 *For each* $1 \leq i \leq j \leq S : D(s_i, s_j) \leq \min(L' \sum_{k=i}^{j} CM_k, \bar{E}(t_j^s - t_i^s + wcet_j - L))$.

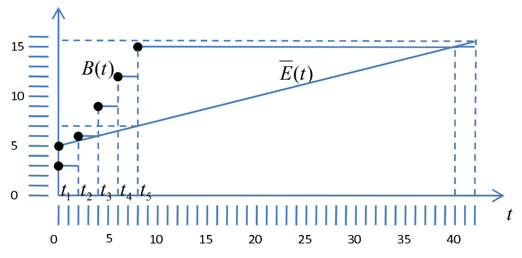
Proof.
Let f_l be the first fetch in s_i and f_q be the last fetch in s_j. We determine the start times $t_l, \ldots, t_q$ for fetches $\{f_l, \ldots, f_q\}$ that produce the maximal delay. Clearly $t_l \geq t_i^s, t_q \leq t_j^s + wcet_j - L$ since all fetches must be initiated and finish within $\{s_i, \ldots, s_j\}$. Furthermore, under the assumption of round-robin schedule it holds: $B(t_q) - B(t_l^-) = L' \sum_{k=i}^{j} CM_k$. The proof follows by applying Theorem 3 to $D(f_l, f_q)$. □

Note that for a single superblock s_i it holds $D(s_i) \equiv D(s_i, s_i) \leq \min(L' \cdot CM_i, \bar{E}(wcet_i - L))$. Compared to classical real-time bounds following the critical instant theorem [11], it is worth to notice that the bound of Theorem 5 only follows if cache misses are spread out in the superblock. As a counterexample, in Figure 6(b) we show the case where all cache misses happen at the beginning of the superblock. Since $\bar{E}(t)$ can only provide 7 units of traffic in the interval $[0, 8]$, the delay in this case is 7, while the worst case obtained in Figure 6(a) and Theorem 5 is 15.

An upper bound Ub on the overall delay $D(s_1, s_S)$ for the task can then be computed using Algorithm 2, which uses an iteratively approach similar to Algorithm 1. The algorithm iteratively computes a delay term u_j^s for each su-



(a) ***Worst Case.***



(b) ***All Misses at the Beginning.***

Figure 6: Cache Arrival Patterns.

Algorithm 2 Compute $D(s_1, s_S)$

```
1: $Ub := 0$
2: $Q := \{\}$
3: for $j = 1 \ldots S$ do
4:     add $(t^s_j, y_j = 0)$ to $Q$
5:     $u^s_j = \min\left(L' \cdot CM_j, \min_{z_i=(t^s_i,y_i)\in Q}\{\bar{E}(t^s_j - t^s_i + wcet_j - L) - y_i\}\right)$
6:     for all $z_i = (t^s_i, y_i)$ in $Q$ do
7:         $y_i := y_i + u^s_j$
8:     $Ub := Ub + u^s_j$
9: return $Ub$
```

perblock s_j; the delay terms are then added together to obtain the final bound Ub. In the algorithm description, Q is a list of pairs $z_i = (t^s_i, y_i)$ ordered by increasing values of t^s_i; y_i represents the accumulated delay, i.e. at step j of the algorithm, $y_i = \sum_{k=i}^{j-1} u^s_k$. The key intuition is that u^s_j can be computed as the minimum of a bound on cache misses in superblock s_j, and a bound on peripheral load for each pair z_i. In particular, for (t^s_i, y_i) the peripheral bound can be rewritten as:

$$\sum_{k=i}^{j} u^s_k = u^s_j + y_i \leq D(s_i, s_j) \leq \bar{E}(t^s_j - t^s_i + wcet_j - L), \tag{2}$$

which follows from Theorem 5.

Theorem 6 *Algorithm 2 computes an upper bound Ub to* $D(s_1, s_S)$.

Proof.
The proof is similar to Theorem 5 in [15], which shows that Algorithm 1 computes an upper bound to the delay $D(f_1, f_N)$. Let $Ub(s_i, s_j) = \sum_{k=i}^{j} u^s_j$. By definition $Ub = Ub(s_1, s_S)$. The proof proceeds by induction on j, proving the following property: $\forall j \geq 1, Ub(f_1, f_j)$ is an upper bound to $D(s_1, s_j)$.

Base Case: We need to prove that $Ub(s_1, s_1)$ is an upper bound to $D(s_1)$. Following the algorithm, $Ub(s_1, s_1) = u_1 = \min\left(L' \cdot CM_1, \bar{E}(t_1 - t_1 + wcet_1 - L) - y_1\right)$ with $t_1 = t_1, y_1 = 0$. Therefore, $Ub(f_1, f_1)$ is equal to the bound computed by Theorem 5 for $D(s_1)$, concluding the proof obligation.

Induction Step: Assume that $\forall k < j, Ub(s_1, s_k)$ is an upper bound to $D(s_1, s_k)$. We need to prove that $Ub(s_1, s_j)$ is an upper bound to $D(s_1, s_j)$. It is sufficient to prove that $Ub(s_1, s_j)$ is maximal, in the sense that increasing $Ub(s_1, s_j)$ would lead to a violation of Theorem 5. By contradiction, assume that $Ub(s_1, s_j)$ is not an upper bound to $D(s_1, s_j)$. Then since $Ub(s_1, s_j) = Ub(s_1, s_{j-1}) + u^s_j$, it follows than at least one of the following two assertions is true for any pattern of cache fetches and schedule of bus transactions that produce a delay $D(s_1, s_j)$: the delay suffered by superblocks $\{s_1, \ldots, s_{j-1}\}$ is strictly greater than $Ub(s_1, s_{j-1})$; or the delay suffered by superblock s_j is strictly greater than u^s_k. However, the first assertion is impossible due to the induction hypothesis. Hence, let the delay suffered by s_k be $u^s_k + \Delta$, with $\Delta > 0$. We consider two cases, relative to whether in the expression:

$$u^s_j = \min\left(L' \cdot CM_j, \min_{z_i=(t^s_i,y_i)\in Q}\{\bar{E}(t^s_j - t^s_i + wcet_j - L) - y_i\}\right) \tag{3}$$

u^s_j is constrained by $L' \cdot CM_j$ (case 1) or by $\bar{E}(t^s_j - t^s_i + wcet_j - L) - y_i$ for some z_i (case 2).

1. $u_j = L' \cdot CM_j$: then following Theorem 5 applied to $D(s_j)$, if follows $\Delta = 0$, a contradiction.
2. We show that the delay for $\{s_1, \ldots, s_{j-1}\}$ is at most equal to $Ub(s_1, s_{j-1}) - \Delta$, which contradicts the fact that $Ub(s_1, s_j)$ is not an upper bound to $D(s_1, s_j)$. By definition of Algorithm 2 and $Ub(s_i, s_j)$, it follows $y_i = Ub(s_i, s_{j-1})$, and therefore $u^s_j = \bar{E}(t^s_j - t^s_i + wcet_j - L) - Ub(s_i, s_{j-1})$, or equivalently $\bar{E}(t^s_j - t^s_i + wcet_j - L) = Ub(s_i, s_j)$. Since from Theorem 5: $D(s_i, s_j) \leq \bar{E}(t^s_j - t^s_i + wcet_j - L)$, it follows that if the delay suffered by s_j is $u^s_j + \Delta$, then the delay suffered by $\{s_i, \ldots, s_{j-1}\}$ is at most equal to $Ub(s_i, s_{j-1}) - \Delta$. To conclude the proof, it now suffices to note that the delay suffered by the remaining superblocks $\{s_1, \ldots, s_{i-1}\}$ can not be greater than $Ub(s_1, s_{i-1})$ due to the induction hypothesis.

□

Unfortunately, for general load functions the bounds computed by Theorem 5 and Algorithm 2 are not tight. An

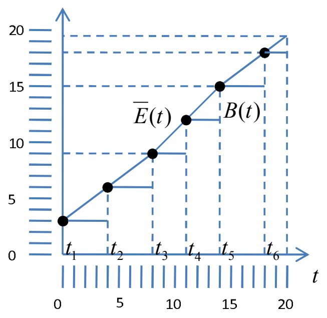


Figure 7: Example Non-Concave Load Function.

example using a different load function $E(t)$ is provided in Figure 7, which shows a worst case pattern similar to Figure 6(a) with $wcet_i = 20, CM_i = 6$. According to Theorem 5, the maximum delay for the superblock is 18. However, if we apply Lemmas 1, 2 to fetches f_3, f_4, f_5, we obtain a delay $D(f_3, f_5) = \min(9, \bar{E}(6)) = 7.5$ which is less than the maximum blocking time of $3L' = 9$ for the three fetches. In fact, in this simple example it can be shown that the real delay bound is $D(f_1, f_6) = 16.5$ time units. Intuitively, this effect is caused by the fact that $E(t)$, and consequently $\bar{E}(t)$, are not concave. However we are able to prove the following:

Theorem 7 *If $E(t)$ is concave, then Algorithm 2 computes a tight upper bound Ub to $D(s_1, s_S)$.*

The proof of Theorem 7 is detailed in the next subsection, as it is quite involved.

Assuming that function $\bar{E}(t)$ can be evaluated in constant time, Algorithm 2 has a complexity $O(S^2)$. Whether a polynomial-time algorithm exists that computes a tight bound for general load functions is left as on open problem. However, we argue that in many cases $E(t)$ can be well approximated with a concave upper bound, hence we do not incur a significant penalty using Algorithm 2. We performed a simulation to validate this claim. We generated 1000 synthetic tasks with $S = 10$ superblocks and randomized $wcet_i$ and CM_i extracted from uniform distributions, with an average cache stall time (the percentage of time that the CPU is stalled waiting for a fetch) of 25%. For each task, we computed an upper delay bound Ub using Algorithm 2 applied to the measured $E(t)$ function of Section 3.1. We also computed a lower bound Ul on the worst case delay as follows: we first assigned start times for all fetches according to Figures 6(a), 7; then, we ran the algorithm described in Section 3 using the assigned start times as input. Since the pattern of cache misses constructed by the start time assignment is consistent with the measured $wcet_i$, CM_i, and the algorithm computes a tight bound, Ul is a valid lower delay bound. Hence, the real worst case delay for each task falls in the interval $[Ul, Ub]$. We finally computed the ratio $\frac{Ub-Ul}{Ul}$ and averaged it over all tasks. Considering we obtained a value of 0.2033%, we conclude that the pessimism introduced by Algorithm 2 is indeed negligible.

3.4. Proof of Theorem 7

Since according to Theorem 6, Algorithm 2 computes an upper bound Ub to $D(s_1, s_S)$, we only need to prove that the bound is tight, i.e. there exists a pattern of cache fetches consistent with $wcet_i, CM_i$ and a schedule of peripheral transactions consistent with $E(t)$ that results in a cumulative delay of Ub time units for superblocks $\{s_1, \ldots, s_S\}$. We will use the following approach: we first define a cache fetch pattern that is a generalization of the example in Figure 6(a). We then apply Algorithm 1 to compute the delay bound Ub for all fetches in the pattern, and we show that the bound is equal to the one computed by Algorithm 2. Since according to Theorem 4, Algorithm 1 computes a tight bound (in the sense that the bound is consistent with $E(t)$), this concludes the proof.

It is easy to see that if $E(t)$ is concave, then $\bar{E}(t)$ is concave as well [15]. We can use this concavity condition to simplify Algorithm 1 at step j by removing a pair $z_i = (t_i, y_i)$ whenever there is another pair $z_q = (t_q, y_q), t_q < t_i$ that constrains u_j more than z_i. More formally, we can remove z_i if the following condition holds at step j, like Algorithm 3 shows below: $\bar{E}(t_j - t_i) - y_i \geq \bar{E}(t_j - t_q) - y_q$.

Algorithm 3 Compute $D(f_1, f_N)$

```
1:  $Ub := 0$
2:  $Q := \{\}$
3:  **for** $j = 1 \ldots N$ **do**
4:      add $(t_j, y_j = 0)$ to $Q$
5:      **for all** $(t_i, y_i)$ in $Q$ **do**
6:          **if** $\exists (t_q, y_q)$ in $Q$: $t_q < t_i \wedge \bar{E}(t_j - t_i) - y_i \geq \bar{E}(t_j - t_q) - y_q$ **then**
7:              remove $(t_i, y_i)$ from $Q$
8:      $u_j = \min\big(B(t_j) - B(t_j^-), \min_{z_i=(t_i,y_i)\in Q}\{\bar{E}(t_j - t_i) - y_i\}\big)$
9:      **for all** $z_i = (t_i, y_i)$ in $Q$ **do**
10:         $y_i := y_i + u_k$
11:     $Ub := Ub + u_k$
12: return $Ub$
```

Lemma 8 *If $E(t)$ is concave, Algorithm 3 computes the same bound Ub on $D(f_1, f_N)$ as Algorithm 1.*

Proof.
Algorithm 3 is a special case of Algorithm 2 in [15]; hence, the proof follows directly from Theorem 8 in [15]. □

We now formally describe the worst case fetch pattern. For ease of exposition, we shall first prove the theorem under two simplifying assumptions: $\bar{E}(0) \le 2L'$, and $L = \epsilon$, where $\epsilon > 0$ is an arbitrarily small number; these two assumptions are introduced because they substantially simplify the fetch pattern. We will then remove both assumptions to prove the theorem in the general setting. In the definition below, we use f_i^k to denote the k-th fetch in superblock s_i and t_i^k to denote the start time of f_i^k.

Definition 1 (Infinitesimal-Size Fetch Pattern) *Let u_j^s be the delay for superblock s_j computed by Algorithm 2. The worst case pattern is as follow: we allocate $N_j = \lceil u_j^s/L' \rceil$ fetches in each superblock s_j, with:*

$$t_j^k = \max(t_j^s, \{t : L' = \min_{t_i^s, i \le j}\{\bar{E}(t-t_i^s) - \sum_{q=i}^{j-1} u_q^s - L'(k-1)\}\}) \tag{4}$$

for all $k, 1 \le k \le \lfloor u_j/L' \rfloor$, and $t_j^{N_j} = t_j^s + wcet_j - L$ if $N_j = \lfloor u_j^s/L' \rfloor + 1$.

Note that Definition 2 generalizes the example of Figure 6(a) in two ways. First, to determine the start time f_j^k of fetch f_j^k we need to consider both the peripheral load $\bar{E}(t_j^k - t_j^s)$ between the beginning of the current superblock s_j and the fetch start time t_j^k and the loads $\bar{E}(t_j^k - t_i^s)$ between the beginning of all previous superblocks s_i and t_j^k. Second, if the delay u_j^s is not an integer multiple of L', we need to add a cache miss at the last valid time instant $t_j^s + wcet_j - L$ in the superblock. Note that in Equation 6 we need to compute t_j^k as the maximum between t_j^s and the time t determined by the load bounds, because in some cases t does not exist (in particular, for f_j^1 when $\bar{E}(0) > L'$ and the bound for all other superblock starting times t_i^s is greater than L' at $t = t_j^s$).

Now that we introduced Algorithm 3 and Definition 2 we can describe the main intuition behind our subsequent proofs: the load bounds based on function $\bar{E}(t)$ used in Equation 6 in Definition 2, Line 8 of Algorithm 3, and Line 5 of Algorithm 2, all express the same condition. In particular, given an Infinitesimal-Size Fetch Pattern, for each pair $z_i = (t_i, y_i)$ in Algorithm 3 there is an equivalent pair (t_i^s, y_i) in Algorithm 2, and the expression $\sum_{q=i}^{j-1} u_q^s + L'(k-1)$ is equivalent to y_i. Following this intuition, we will prove two fundamental properties: 1) Equation 6 implies that every fetch f_j^k suffers a delay $u_j^k = L'$ in Algorithm 3 (with the exception of the last fetch $u_j^{N_j}$); 2) Line 5 in Algorithm 2 will then imply that the total delay u_j^s in superblock s_j is equal to the sum of the delays u_j^k computed by Algorithm 3 for all fetches in s_j. This in turn is enough to complete the proof.

Lemma 9 *The Infinitesimal-Size Fetch Pattern in Definition 2 is a consistent pattern under the assumptions $\bar{E}(0) \le 2L', L = \epsilon$.*

Proof.
For the pattern to be consistent, we need to show two properties: $N_j \le CM_j$ for each superblock s_j, and the distance between any two successive start times f_j^k, f_j^{k+1} must be at least $L = \epsilon$.

The first property follows immediately from the fact that u_j^s is at most $L' \cdot CM_j$ according to Algorithm 2, and thus: $N_j \le \lceil CM_j \rceil = CM_j$. By contradiction, assume that the second property is false. Then since ϵ is arbitrarily small, there must exist fetches f_j^k, f_j^{k+1} such that $t_j^k = t_j^{k+1}$. We distinguish three cases: 1) $k = 1$; 3) $k = N_j - 1$ with $N_j = \lfloor u_j/L' \rfloor + 1$, and 2) otherwise. In case 1, according to Equation 6 for $t_j^1 = t_j^2$ to be possible it must hold: $\bar{E}(0) \ge 2L'$, but this violates our assumption. In case 2, according to Equation 6 for $t_j^k = t_j^{k+1}$ to be possible it must exist a time t such that: $\bar{E}(t) - \bar{E}(t^-) \ge L'$, but this again is impossible since $\bar{E}(t)$ is concave and therefore continuous for $t > 0$. Finally, consider case 3. Let (t_q^s, y_q) be the pair that constraints u_j^s in Algorithm 2. Then it is easy to see that for $t_j^{N_j-1}$, the bound relative to t_q^s in Equation 6 can be rewritten as $\bar{E}(t-t_q^s) - y_q - L'(N_j-2)$, and for $t = t_j^s + wcet_j - L$ it holds: $\bar{E}(t_j^s - t_q^s + wcet_j - L) - y_q - L'(N_j-2) = u_j^s - L'(N_j-2) \ge \bar{E}(t - t_i^s) - y_i - L'(N_j-2)$ for every other $t_i^s, i \le j$. Since $\bar{E}$ is continuous for $t > 0$ and furthermore $u_j^s - L'(N_j-2) > L'$, it follows $t_j^{N_j-1} < t_j^s$, which concludes the proof. □

Lemma 10 *At each step of Algorithm 3 for fetch f_j^k using the Infinitesimal-Size Fetch Pattern, the following two properties apply: A) before computing u_j^k, the only remaining pairs in Q are of the form (t_i^s, y_i); B) if $k = N_j \wedge N_j = \lfloor u_j^s/L' \rfloor + 1$, then $u_j^{N_j} = u_j^s - L'(N_j - 1)$, otherwise $u_j^k = L'$.*

Proof.
The proof proceeds by induction on the fetch indexes k, j, using the obvious fetch order (formally, we can define a bijective mapping that associates each fetch f_j^k with natural number $k + \sum_{q=1}^{j-1} N_q$). We consider three cases: 1) $k = 1$, which covers the base case of the induction for $j = 1$; 3) $k = N_j \wedge N_j = \lfloor u_j^s/L' \rfloor + 1$; and 2) otherwise. For all cases, we use the induction hypothesis that properties A and B hold for all fetches previous to f_j^k; hence, for property A we only need to focus on the additional pair $(t_j^k, 0)$ that can be inserted into Q before computing u_j^k in Algorithm 3.

Case 1: Assume $k = 1$, and let (t_i^s, y_i) be the remaining pairs in Q in Algorithm 3 at the previous step. By the in-

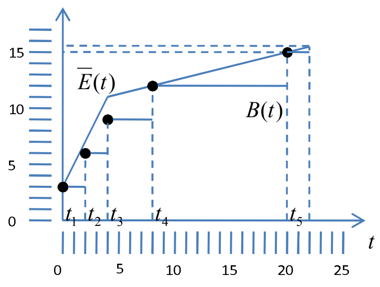


Figure 8: Example Size-Constrained Fetch Pattern.

duction hypothesis of Property B it directly follows that for each u_i^s with $i < j$: $u_i^s = \sum_{q=1}^{N_i} u_i^q$. Also note that when evaluating u_j^1, the only pair that can be added to Q in Algorithm 3 is $(t_j^s, 0)$, and thus property A trivially holds. For property B, consider that Equation 6 can then be rewritten as:

$$t_j^1 = \max(t_j^s, \{t : L' = \min_{t_i^s, i \leq j} \{\bar{E}(t - t_i^s) - y_i\}\}), \quad (5)$$

since the term $L'(p-1)$ is equal to zero. We distinguish two cases. Assume that $\bar{E}(t_j^s - t_i^s) - y_i \geq L'$ for each $i < l$. Then t_j^1 is assigned value t_j^s (note that for $i = l$, it must hold $\bar{E}(0) \geq L'$ since L' is the maximum transaction size). Therefore, since only pairs (t_i^s, y_i) remains in Q in Algorithm 3, u_l^1 is limited by the term $B(t_l^1) - B(t_l^{1-}) = L'$. If the assumption is not true, let (t_i^s, y_i) be the pair for which $L' = \bar{E}(t - t_i^s) - y_i$ in Equation 6. Since again only pairs (t_i^s, y_i) remains in Q in Algorithm 3, u_l^1 is limited by $\bar{E}(t_l^0 - t_i^s) - y_i = L'$.

Case 2: Let (t_i^s, y_i) be the remaining pairs in Q in Algorithm 3 before evaluating u_j^1. Then after computing delay for fetches $f_j^1, \ldots, f_j^{k-1}$, all terms y_i will be increased by the same amount $L'(p-1)$. The above discussion implies that Equation 6 can still be rewritten as Equation 5; hence, the same argument as in the base case can be applied to prove $u_j^k = L'$ and thus property B holds. Furthermore, since $\bar{E}(0) \geq L'$, the new pair $(t_j^k, 0)$ will be removed from Q, thus property A also holds.

Case 3: Since $\forall\ i < j : u_i^s = \sum_{q=1}^{N_i} u_q^i$, it follows that all pairs (t_i^s, y_i) remaining in Q in Algorithm 3 before computing u_j^1, are also in the pair queue of Algorithm 2 when evaluating u_j^s, with the same y_i values. From Algorithm 2 it must hold: $\bar{E}(t_j^s - t_i^s + wcet_j - L) - y_i \geq u_j^s$. But since y_i is increased by $L'(N_j - 1)$ for fetches $f_j^1, \ldots, f_j^{N_j - 1}$ in Algorithm 3, and furthermore $t_j^{N_j} = t_j^s + wcet_j - L$, we can derive that Algorithm 3 will compute a delay bound $u_j^{N_l}$ greater or equal than $u_j^s - L'(N_j - 1)$, as now for each pair: $\bar{E}(t_j^s - t_i^s + wcet_j - L) - y_i \geq u_j^s - L'(N_j - 1)$. To prove property B, it is sufficient to note that according to Theorem 6, Algorithm 2 computes an upper bound on u_j^s, hence $u_j^{N_j}$ can not be greater than $u_j^s - L'(N_j - 1)$. Furthermore, by definition $u_j^s - L'(N_j - 1) < L'$, while again $\bar{E}(0) \geq L'$, hence pair $(t_j^{N_j}, 0)$ is immediately removed from Q proving property A. □

Theorem 11 *Under the assumptions that $E(t)$ is concave, $\bar{E}(0) \leq 2L'$ and $L = \epsilon$, Algorithm 2 computes a tight upper bound Ub to $D(s_1, s_S)$.*

Proof.
Under the assumptions, the Infinitesimal-Size Fetch Pattern is valid. Following Lemma 10, for the superblock delay u_j^s computed by Algorithm 2 and the fetch delays u_j^k computed by Algorithm 3 using the Infinitesimal-Size Fetch Pattern, it holds: $u_j^s = \sum_{k=1}^{N_j} u_j^k$. Hence, the overall delay $Ub = D(s_1, s_S)$ computed by Algorithm 2 is the same as the delay $Ub = D(f_1^1, f_S^{N_S})$ computed by Algorithm 3 using the Infinitesimal-Size Fetch Pattern, which by Lemma 8 is the same as the delay computed by Algorithm 1 since $E(t)$ is concave. Since according to Theorem 4, the bound computed by Algorithm 1 is tight, it follows that Ub is indeed the worst case delay for the Infinitesimal-Size Fetch Pattern (with respect to all valid peripheral transaction schedules according to $E(t)$). Since the Infinitesimal-Size Fetch Pattern is valid and furthermore according to Theorem 5, Algorithm 2 computes an upper bound, it follows that Ub is a tight upper bound to $D(s_1, s_S)$. □

We now show how the assumptions of Theorem 11 can be relaxed, thus obtaining a proof for Theorem 7. When we remove the two assumptions, the that fetch pattern of Definition 2 is no longer valid: a clarifying example is provided in Figure 8 for a single superblock. After assigning $t_1 = 0$, we obtain $\bar{E}(L - 0) - L' = 7 - 3 = 4 > L'$: hence, the start time for fetch f_2 is constrained not by the peripheral load, but by the fact that fetch start times must be spaced apart at least L time units. Intuitively, $\bar{E}(t)$ raises "too fast", i.e. with a derivative greater than L'/L. This creates a problem in Lemma 10: at step 2 of Algorithm 3, we obtain $\bar{E}(t_2 - t_2) - 0 = 3 < \bar{E}(t_2 - t_1) - L' = 4$, hence pair $(t_2, 0)$ is not removed immediately from Q and Property A is violated. The trick we use to solve this problem is to show that even if we have to keep pair (t_2, y_2) in Q, it is never the case that the delay u_j^k for a successive fetch f_j^k is constrained by $\bar{E}(t_j^k - t_2) - y_2$ in Algorithm 3.

Definition 2 (Size-Constrained Fetch Pattern) *Let u_j^s be the delay for superblock s_j computed by Algorithm 2. The worst case pattern is as follow: we allocate $N_j = \lceil u_j^s / L' \rceil$*

fetches in each superblock s_j, with:

$$t_j^k = \max(\bar{t}_j^k, \{t : L' = \min_{t_i^s, i \le j}\{\bar{E}(t - t_i^s) - \sum_{q=i}^{j-1} u_q^s - L'(k-1)\}\}) \quad (6)$$

for all $k, 1 \le k \le \lfloor u_j/L' \rfloor$ with $\bar{t}_j^1 = t_j^s, \bar{t}_j^k = t_j^{k-1} + L$, and $t_j^{N_j} = t_j^s + wcet_j - L$ if $N_j = \lfloor u_j^s/L' \rfloor + 1$.

Definition 3 (Size-Constrained Fetches) *Assume $t_j^k > t_j^{k-1} + L, t_j^q = t_j^k + L(q-k)$. Then fetches $f_j^{k+1}, \dots, f_j^q$ are said to be* size-constrained*, and f_j^k is the fetch constraining them.*

Lemma 12 *When executing Algorithm 3, if at step f_i^l, $\bar{E}(t_i^l - t_j^q) - y_j^q < L'$ for pair (t_j^q, y_j^q) and f_j^q is a size-constrained fetch, then $\bar{E}(t_i^l - t_j^k) - y_j^k < \bar{E}(t_i^l - t_j^q) - y_j^q$, where f_j^k is the fetch constraining f_j^q.*

Proof.
Since fetch f_j^q is size constrained, then following the same reasoning as Lemma 10 it is easy to see that fetches $f_j^k, \dots, f_j^{q-1}$ must suffer a delay equal to L' and therefore $y_j^k = y_j^q + L'(q-k)$ (formally, we would have to use a similar induction proof, but we omit it for the sake of conciseness). We then only need to prove the following:

$$\bar{E}(t_i^l - t_j^k) - L'(q-k) < \bar{E}(t_i^l - t_j^q) \quad (7)$$

To simplify the proof, let us define $x \equiv t_i^l - t_j^q, y \equiv t_j^q - t_j^k = L(q-k)$, from which: $t_i^l - t_j^k = x + y$. Then it must necessarily hold: $\bar{E}(x) < x\frac{L'}{L} + L'$, otherwise f_i^l would be size constrained and $\bar{E}(t_i^l - t_j^q) - y_j^q < L'$ could not hold at step f_i^l. Now note that since f_j^q is size constrained, according to Equation 6 it must hold: $L' \ge \bar{E}(y) - y\frac{L'}{L}$. By combining the two equations we obtain: $\bar{E}(x) - \bar{E}(y) < (x-y)\frac{L'}{L}$. Now consider interval $[x, x+y]$ for function $\bar{E}(t)$. Since $\bar{E}(t)$ is concave and both extremes of the interval $[x, x+y]$ are greater than the extremes of the interval $[y, x]$, it must hold: $\bar{E}(x+y) - \bar{E}(x) < (x+y-y)\frac{L'}{L} = y\frac{L'}{L}$. We can now write:

$$\bar{E}(x+y) = (\bar{E}(x+y) - \bar{E}(x)) + \bar{E}(x) < \bar{E}(x) - y\frac{L'}{L}, \quad (8)$$

from which Equation 7 directly follows. □

In essence, Lemma 12 implies that we do not need to keep track of pairs (t_j^q, y_j^q) for constrained fetches, since the value of all future delays u_i^l can be computed using the pair (t_j^k, y_j^k) for the fetch f_j^k that constrains f_j^q.

Lemma 13 *Consider a modification of Algorithm 3 where in Line 4 a pair $(t_j^q, 0)$ is not added to Q if f_j^q is a constrained fetch. Then this new modified algorithm computes the same bound Ub as Algorithm 3.*

Proof.
The proof follows immediately from Lemma 12, since no delay u_i^l for a future fetch f_i^l can be modified if the pair $(t_j^q, 0)$ is removed. □

Lemma 10 can then be proved on the modified algorithm of Lemma 13 instead of Algorithm 3. The proof is equivalent, with the only difference that pairs introduced by constrained fetches are removed thanks to the property of the modified algorithm. The proof of Theorem 7 then follows from Lemma 13 along the lines of the proof of Theorem 11.

4. Coscheduling Algorithm

It is important to note that even when the bound computed by Algorithm 2 is tight, it is rare that at run-time a task will suffer a delay equal to the bound: a particular pattern for both cache misses and peripheral transactions is required to produce the worst case. As such, accounting for the worst case delay inflicted by all peripherals in the computation time budget of each task can lead to a large waste of resources. We propose an alternative solution based on a run-time adaptive algorithm. The idea is to assign to a task the minimal time budget of $\sum_{1 \le i \le S} wcet_i$, and then to monitor the actual execution of the task and open the p-gates whenever possible. At run-time, information on the execution time consumed by the current job is sent to the reservation controller at each checkpoint. The controller uses this information to determine the actual execution time e_i of superblock s_i for the current job. The *accumulated slack* time after superblock s_i can then be computed as $\sum_{1 \le j \le i}(wcet_j - e_j)$; the slack time is the maximum delay that the task can suffer while still meeting its computation time budget. We can then design a coscheduling algorithm that strives to maximize the amount of time that the p-gates are opened under the constraint that the slack can never become negative. Our proposal is to integrate both the analysis and coscheduling techniques in a mixed-criticality system. Inspired by the avionic domain, we consider two types of guaranteed real-time tasks: *safety critical* tasks like flying control, that have stringent delay and verification requirements, and *mission critical* tasks that are still hard real-time but have lower criticality. We propose to schedule safety critical tasks blocking all I/O traffic except the one from peripherals used by the task, and to account the delay in the time budget. For *mission critical* tasks we instead use coscheduling. Finally, we also assume that the system runs best effort or soft real-time tasks for which all p-gates are opened.

Algorithm 4 is our main coscheduling heuristic. For simplicity, we describe the algorithm for a single controlled task and a single peripheral, but it can be easily extended to a multitasking environment with multiple p-gates. Com-

Algorithm 4 Adaptive Algorithm

```
JobStart() {
$slack := 0$
$i := 0$
CloseGate()
}
Checkpoint($e$) {
$i := i + 1$
$slack := slack + wcet_i - e$
**if** $D(s_{i+1}) \leq slack$ **then**
    OpenGate()
**else**
    CloseGate()
}
```

munication from the task to the reservation controller triggers the algorithm at the beginning of each job and at each checkpoint. The algorithm maintains two variables: `i` is the index of the last executed superblock, and `slack` represents the accumulated slack. At the end of each superblock s_i, the algorithm first recomputes the slack and then performs a check: if the slack is at least equal to the maximum delay $D(s_{i+1})$, then the p-gate is opened because we are sure that the slack will be non negative after the next superblock s_{i+1} is executed. Otherwise, the p-gate is kept closed.

Algorithm 5 Predictive Algorithm

```
JobStart() {
$slack := 0$
$pslack := \sum_{k=1}^{S}(wcet_k - avg_k)$
$i := 0$
CloseGate()
}
Checkpoint($e$) {
$i := i + 1$
$slack := slack + wcet_i - e$
$tmp := pslack := pslack + avg_i - e$
**for all** $k$ in ORDERED_LIST ($\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)}$) **do**
    **if** $k > i + 1 \wedge D^{avg}(s_k) \leq tmp$ **then**
        $tmp := tmp - D^{avg}(s_k)$
    **if** $k == i + 1$ **then**
        **if** $D(s_{i+1}) \leq slack \wedge D^{avg}(s_{i+1}) \leq tmp$ **then**
            OpenGate()
        **else**
            CloseGate()
        **return**
}
```

The limitation of Algorithm 4 is that it greedily "allocates" all slack to the next superblock by immediately opening the p-gate. This can lead to a suboptimal allocation, as superblock s_{i+1} could be short and have a lot of cache misses while superblock s_{i+2} could be longer with very few cache misses. If we have additional information on the task, we can potentially do better using a predictive heuristic. In particular, Algorithm 5 assumes that the average case computation time avg_i and average case delay $D^{avg}(s_i)$ for each superblock s_i is known. The algorithm keeps track of the *predicted slack*, i.e. the total slack assuming that all future superblocks will execute for $e_j = avg_j$. We can then compute a strategy that maximizes the amount of time that the p-gate is opened by allocating the predicted slack among all future superblocks: if we decide to open the p-gate during s_j, we consume an amount of slack equal to $D^{avg}(s_j)$ and the p-gate is opened for $avg_j + D^{avg}(s_j)$ time units. It is easy to see that this allocation problem is equivalent to the KNAPSACK problem [8], which is well known to be NP-hard. We therefore use a sub-optimal polynomial time greedy solver: off-line, we order all superblocks by non-increasing values of $\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)}$. At run-time, we perform the allocation by iterating through the list ignoring all superblocks already executed. When the iteration arrives to the next superblock s_{i+1}, the p-gate is opened if the remaining predicted slack is greater or equal than $D^{avg}(s_{i+1})$.

Note that Algorithm 5 is not the only possible predictive algorithm; in fact, no on-line algorithm can be optimal, since any optimal algorithm must known exactly the computation time of future superblocks, i.e. it must be clairvoyant. However, for the sake of comparison it is interesting to compute an upper bound on the best possible performance of any on-line predictive algorithm. Assume that for a specific run, e_i is the execution time of s_i assuming that the p-gate is closed, and $\bar{e}_i$ is the execution time assuming that the p-gate is opened. An upper bound can be computed by solving the following integer linear programming problem:

$$\max \sum_{i=1}^{S} x_i \bar{e}_i \tag{9}$$

$$\forall i, 1 \leq i \leq S : x_i D(s_i) \leq \sum_{j=1}^{i-1}(wcet_j - (1 - x_j)e_j - x_j \bar{e}_j) \tag{10}$$

$$\forall i, 1 \leq i \leq S : x_i \in \{0, 1\}, \tag{11}$$

where $\{x_1, \ldots, x_S\}$ are indicator variables (i.e., $x_i = 1$ if the p-gate is opened during s_i). Equation 9 maximizes the open time, while Equation 10 expresses the slack constraint.

5. Experimental Results

To validate our architecture, we performed experiments on a COTS PC platform comprised of an Intel Core2 CPU and an Intel 975X system controller. Using a PC platform allowed us easy access to all PC slots; however, to derive meaningful measures we changed the FSB clock frequency obtaining a speed of 900Mhz for the CPU and a theoretical bandwidth of 2.4Gbyte/s for the FSB, which is in line with typical values for embedded platforms.

We first performed an experiment to evaluate the maximum delay incurred by a task due to peripheral interference. To obtain repeatable measures, we implemented a custom traffic generator peripheral for the PCI-X bus based on a Xilinx ML455 board. The peripheral periodically initiates write transactions to main memory, and both the period and transactions length can be configured to produce a load up to the maximum of 1 Gbyte/s supported by PCI-X. We then designed a task to maximize cache stall time. The task allocates a memory buffer of double the size of the CPU level 2 cache, and then cyclically reads from the buffer, one word for each 128-byte cache line; a cache miss is thus generated for each memory read. We first ran the task without using the traffic generator and measured an execution time of 48.73ms and 580,227 cache misses. Using a memory benchmark, we evaluated a main memory throughput of $C = 1.8$Gbyte/s, which is slightly lower than the theoretical FSB speed. Since 128 bytes must be transferred for each cache miss, the task is actually stalled for $\frac{580,227 \cdot 128}{1.8e9} = 41.26$ms, resulting in the desired high cache stall time of 84.67%. We then ran the task again, enabling the traffic generator with maximum load, and we measured an average increase in computation time of 43.85%.

It is important to note that the measured value is an average case delay, since obtaining the worst case pattern (as depicted in Figure 6(a)) over more than 500,000 fetches is improbable. Our analysis is able to compute the worst case delay, but we need additional information on the system controller, in particular the type of arbitration used by the FSB and the maximum length values L and L'. These data are typically available for components used in embedded systems, see [12] for example. However, in the PC market manufacturers are often wary of revealing precise details for fear of losing competitive edge. We therefore used the following experimental methodology to obtain the required information: we first guessed values for L, L' and the arbitration type, and we built a simulator to predict average case delay for a variety of settings. We then performed extensive experiments and confronted the measured values with the predictions to determine if our guesses were acceptable.

Experimental results are shown in Figure **??**, where each point is an average over 5 runs. We measured the percentage increase in computation time for the aforementioned task varying the offered peripheral load and length of peripheral transactions. Note that for small lengths we are not able to significantly load the bus, as the period of the traffic generator is constrained by PCI-related overhead; hence, some points in Figure **??**(a) can not be generated on the bus and we show them as zero values. All results are within 5% of our predictions, assuming round-robin arbitration and

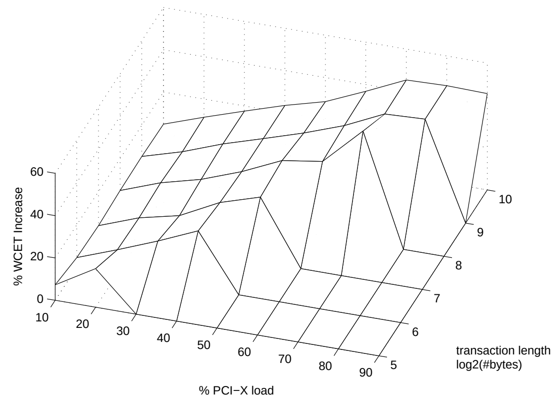


Figure 9: Measured Delay.

SLACKONLY	ADAPTIVE	PREDICTIVE	BOUND
4.89%	31.21%	36.65%	40.85%

Table 1: Benchmark Results.

$L = L' = 32\text{bytes}/C = 17.8$ns, which means that each fetch is broken down into 4 data transfers on the FSB. Unexpectedly, we found that delay is constant over 70% load. Investigation of the PCI-X bus using a logic analyzer revealed that it is an issue of the PCI bridge, which is not fast enough to buffer all peripheral data. We also performed additional experiments varying the cache stall time of the task; this can be achieved by inserting a variable number of instructions between each successive cache line read. The obtained wcet increases also matched our simulation results within a small deviation.

We then evaluated the performance of the described coscheduling algorithms on our platform. We chose a MPEG decoder [4] as our benchmark for two reasons: it is both a memory and I/O intensive application, and it is representative of the type of video computation that is becoming increasingly important for mission control in avionic systems. We collected average and worst case statistics on a test video clip after placing multiple checkpoints for each frame; the MPEG decoder is run as a periodic task, with 20 superblocks in each period. To mirror the behavior of a real application and increase the number of cache misses, we also ran a higher priority task that preempts the MPEG decoder every 1ms and replaces its cache content. Results averaged over 50 runs are shown in Table 1 in term of the percentage of time that the p-gate is opened in the task period. In the table, SLACKONLY represents a baseline solution where the p-gate is kept closed while the task is executing and is opened after the task has finished for its remaining time budget $\sum_{1 \leq i \leq S}(wcet_i - e_i)$; ADAPTIVE is Algorithm 4; PREDICTIVE is Algorithm 5;

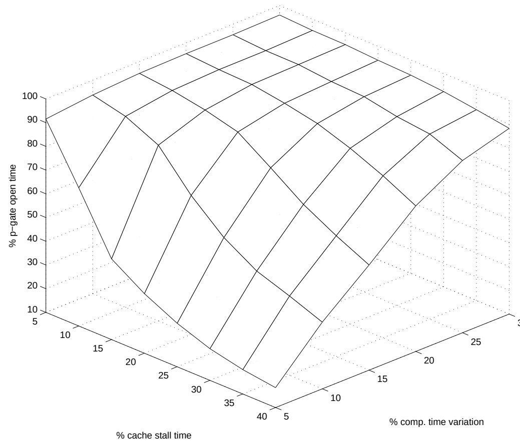


Figure 10: Synthetic Tasks, ADAPTIVE, $\sigma = 0.1$

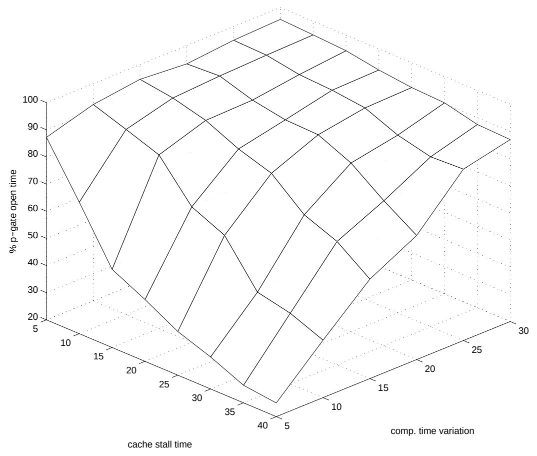


Figure 12: Synthetic Tasks, ADAPTIVE, $\sigma = 0.4$

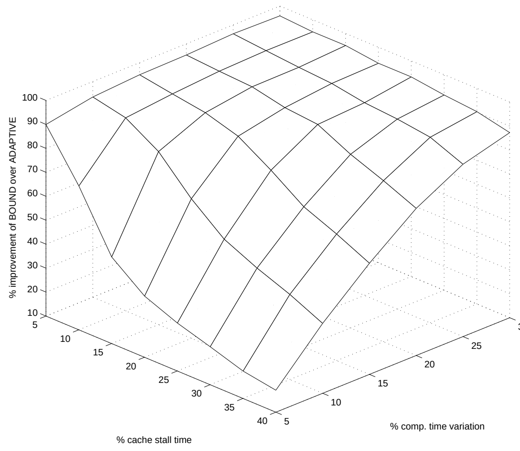


Figure 11: Synthetic Tasks, ADAPTIVE, $\sigma = 0.2$

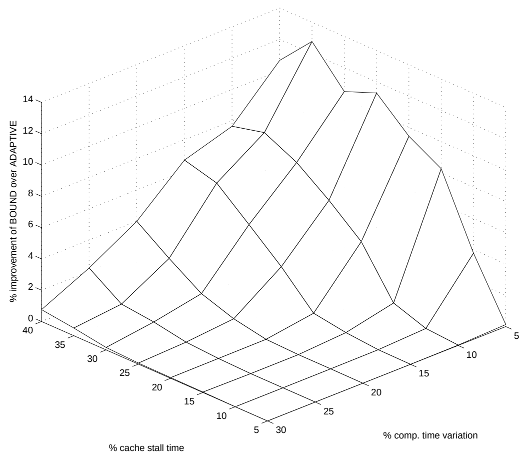


Figure 13: Synthetic Tasks, algorithm ratio, $\sigma = 0.1$

BOUND is the bound computed by solving the ILP problem of Equations 9-11. Note that since BOUND is not implementable at run-time, we computed the bound offline using measured values of computation times and number of cache misses. We can see that SLACK-ONLY tends to perform very poorly; ADAPTIVE is within 30% of BOUND and PREDICTIVE is roughly in between the two, which seems to suggest that prediction offers limited improvement.

To check whether the obtained results hold for more general settings, we also performed extensive simulations on synthetic tasks, each composed of 20 superblocks, varying three parameters σ, α, β. For each task and each superblock s_i, we first generated the average computation time avg_i from a uniform distribution with constant mean and coefficient of variation σ, and the average cache stall time $stall_i$ from a uniform distribution with mean β and coefficient of variation σ. We then simulated 10 task runs by extracting for each run and each superblock a computation time $e_i = avg_i(1 + \bar{\alpha})$ and a number of cache misses equal to $stall_i(1 + \bar{\alpha})$, where $\bar{\alpha}$ is extracted from a uniform distribution with mean 0 and maximum value α. Note that this implies $wcet_i = avg_i(1 + \alpha)$, i.e. α is the increase in computation time between the average and the worst case.

We focus on results for the ADAPTIVE and BOUND cases: Figure 10, 11, 12 show the value of ADAPTIVE for values of σ equal to $0.1, 0.2, 0.4$ respectively. Figure 13, 11 15 shows the competitive ratio of $\frac{\text{BOUND}-\text{ADAPTIVE}}{\text{ADAPTIVE}}$, again for $\sigma = 0.1, 0.2, 0.4$. All points are averages over 10 tasks (100 runs total of the simulator). We varied the average cache stall time β between $[0.05, 0.4]$ and the computation time variation α be-

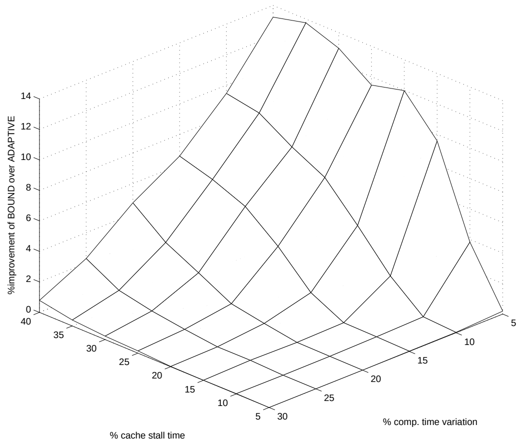


Figure 14: Synthetic Tasks, algorithm ratio, $\sigma = 0.2$

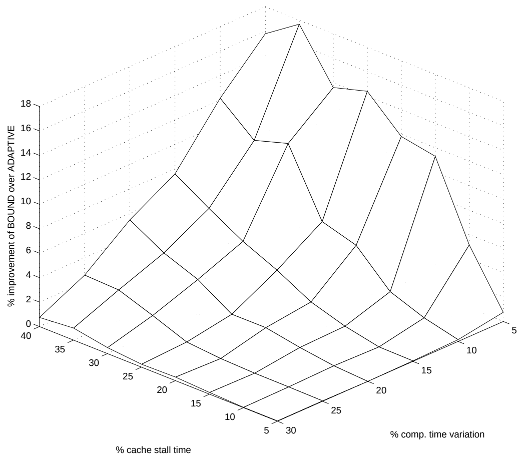


Figure 15: Synthetic Tasks, algorithm ratio, $\sigma = 0.4$

tween $[0.05, 0.3]$; axis direction is inverted between the two sets of figures for easier visualization. Note that the definition of α implies SLACKONLY=$\frac{\alpha}{1+\alpha}$ for all cases. First note that the obtained results are very close for the different values of σ, which seems to indicate that none of the tested algorithm is very sensitive to variations in superblock size. The second main observation is that the performance of the algorithms depends on the difference between α and β. For $\alpha > \beta$, both algorithms can open the p-gate almost all the time because the high wcet variability forces us to over-provision the computation time budget of the task; however, note that a coscheduling algorithm is still needed to guarantee safety, as there are times where the p-gate must be closed to ensure that the task meets its deadline. In the case $\alpha < \beta$, which is representative of more predictable, but memory intensive real-time tasks, the fraction of time the p-gate can be opened decreases as the delay $D(s_i)$ becomes significant compared to $wcet_i - avg_i$. The performance of ADAPTIVE degrades more rapidly than BOUND, but it remains with a competitive ratio of 18%, which compares even more favorably than the MPEG case.

6. Related Work

Apart from [15], to the best of our knowledge the only works that study the impact of I/O load on real-time scheduling are [18] and [6]. [18] uses a PCI-based testbed similar to ours, but its empirical approach can not derive safe wcet bounds. [6] uses an analytical approach but it assumes highly predictable cycle-stealing bus arbitration, which is not true of commodity systems.

Two other research areas are related to our work. First of all, peripheral activities impose an additional overhead on the CPU: device driver execution. Techniques to account for such overhead have been described in [10, 19] for network cards and hard disks based on experimentally-derived bounds. Second, there is a second potential source of interference at either the cache or FSB level: other CPUs. A methodology to compute cache access delay in multiprocessor systems has been proposed in [17] based on static analysis. However, we argue that this problem domain is essentially different from ours because synchronizing task schedule across multiple processors is easier than synchronizing CPU and peripheral execution.

7. Conclusions

The effects of peripheral traffic in a COTS-based system can not be ignored: our experiments using typical settings for an embedded system reveal that interference at the FSB level can increase the computation time of a task by almost half. In this paper, we proposed two ways to cope with this effect. The first is to account for the additional delay in the wcet of the task, using an analysis that is able to compute a safe delay bound based on real measurements of the task and of the interfering peripherals. The second is to control peripheral activities using a peripheral gate and a coscheduling algorithm that dynamically allows/disallows peripherals to transmit making sure that the task computation time does not exceed its wcet without traffic. Our experiments show that even a simple adaptive coscheduling heuristic can greatly improve the amount of allowed traffic compared to the baseline approach of blocking all peripherals while the task is executing. More complex predictive heuristics can do even better, but our experiments revealed that the improvement space is somehow limited.

More work remains to be done. First of all, our p-gate implementation assumes that peripherals have buffering capabilities; otherwise, data can be lost while the p-gate is closed. We are developing a second generation p-gate that will be able to both buffer and classify incoming data from the peripheral. Second, we plan to extend our framework to cover multicore systems, where multiple tasks can be executed simultaneously with master peripherals. Third, our current analysis abstracts away some complexities of PCI involving buffering in the $E(t)$ function. If master peripherals are allowed to communicate with each others, then a specific PCI bus analysis is required to compute new traffic bounds and delays for the peripherals.

References

[1] Advanced Micro Devices, Inc. *Torrenza Initiative*. `http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx`.

[2] Aeronautical Radio Inc. *ARINC 653 Specification*. http://www.arinc.com/.

[3] T. Baker. Lessons learned integrating COTS into systems. In *Proc. of the First International Conference on COTS-Based Software Systems (ICCBSS 2002)*, Feb 2002.

[4] FFMPEG project. *libavcodec multimedia library*. `http://ffmpeg.mplayerhq.hu/`.

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

[6] Tay-Yi Huang, Jane W. S. Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access I/O concurrent with hard-real-time programs. In *Int. Conf. on Parallel and Distributed Systems*, Tokyo, 1996.

[7] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, February 2008.

[8] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 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 *Proceedings of the* 13^{th} *IEEE Real Time Application Symposium*, Apr 2007.

[11] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[12] Marvell. *Discovery II PowerPC System Controller MV64360 Specifications*. `http://www.marvell.com/`.

[13] S. Oikawa and R. Rajkumar. Linux/RK: a portable resource kernel in linux. In *Proceedings of the* 19^{th} *IEEE Real-Time System Symposium*, Madrid, Spain, December 1998.

[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. Towards the predictable integration of real-time COTS based systems. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, Dec 2007.

[16] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proc. of the IEEE RTAS*, Apr 2006.

[17] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, December 2007.

[18] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[19] M. Stanovich, T. Baker, and A. Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Proc. of the* 14^{th} *IEEE RTAS*, St. Louis, Missouri, April 2008.