Protecting Real-Time Applications against Memory Induced Slowdown on a Small Multicore System

Antoine Blin[*†‡] Julien Sopena[†‡] Julia Lawall[‡†] Gilles Muller[‡†]
[*]*Renault* [†]*Sorbonne Universités, UPMC, LIP6* [‡]*Inria, Whisper team*
firstname.lastname@lip6.fr

Abstract

Complex embedded systems today commonly involve a mix of real-time and best-effort applications. The recent emergence of small low-cost commodity multicore processors raises the possibility of running both kinds of applications on a single machine, with virtualization ensuring that the best-effort applications cannot steal CPU cycles from the real-time applications. Nevertheless, memory pressure can introduce other sources of delay, that can lead to missed deadlines. In this paper, we present a combined offline/online memory bandwidth monitoring approach to estimate the impact of the memory pressure incurred by the best-effort applications on the execution time of the real-time application. Our approach is compatible with the hardware counters provided by current small commodity multicore processors. Our approach allows the system designer to limit the overhead on the real-time application to under 5% of its expected execution time, while still enabling progress of the best-effort applications.

1 Introduction

In many embedded system domains, such as the automotive industry, it is necessary to run applications with different levels of criticality. Some applications may need nearly hard real-time constraints, while others may need only best-effort access to the CPU and memory resources. A typical example is the car dashboard, which may display both critical real-time information, such as an alarm, and non critical information, such as travel maps and outsmarting traffic. To provide full isolation between application classes, that often run different OS and middleware software stacks, the traditional approach is to rely on separate computer systems. This approach, however, induces a high hardware costs.

Recent small commodity multicore systems, such as the Freescale SABRE Lite board [1], offer sufficient CPU power to run multiple applications on a single low-cost computer, and thus represent a promising solution for minimizing hardware cost. Nevertheless, running multiple classes of applications on a single computer raises two main challenges: (i) in terms of compatibility, how to use legacy software stacks with little or no adaptation, and (ii) in terms of dependability, how to isolate real-time applications so that their deadlines are not impacted by other applications. Hypervisors [8, 16, 31] address the first challenge, allowing legacy software stacks to be used without modification. Furthermore, recent hypervisors such as SeL4 [2] and PikeOS [3] that specifically target embedded systems address part of the second challenge, by making it possible to dedicate one or several cores to a class of applications, and thus provide CPU isolation. Still, for many small commodity multicore systems, the memory bus and non core-local caches are shared resources. Therefore, even if CPU isolation is provided, any overuse of these resources by the best-effort applications may impact the execution time of the real-time applications.

Existing solutions to the memory and cache sharing problem rely on specific hardware mechanisms. Mancuso et al. [20] propose a cache-coloring approach for partitioning memory accesses. Caccamo et al. [12] rely on a hardware mechanism for measuring the memory bandwidth consumption of each core. Each core is allocated a quota of the total memory bandwidth, and memory consumption is continuously monitored by the OS, which is able to suspend applications in case of overuse. However, most small commodity multicore systems provide only system-wide memory consumption accounting, which makes these existing approaches inapplicable. Alternatively, a baseline approach for sharing a computing system between real-time and best-effort tasks is to just suspend the best-effort applications whenever the real-time applications are running. This approach, however, would lead to a waste of CPU resources and long latencies for the best-effort tasks.

In this paper, we propose an approach that requires only system-wide memory accounting to prevent the memory usage of the best-effort applications from impacting the real-time application. Our solution is thus usable on current small commodity multicore systems that only provide this form of accounting. Our key observation is that, as long as there are not too many requests that are issued simultaneously, the memory subsystem serves memory requests with little latency, and is not a performance bottleneck. The challenge, then, is to determine how many requests is too many, with respect to the memory bandwidth requirements of the real-time application. To address this issue, we propose (i) an off-line analysis for characterizing the performance overhead induced by increases in memory bandwidth, (ii) a memory profile of the real-time application constructed off-line based on testing, analogous to approaches used for WCET estimation, and (iii) a run-time system, implemented within the operating system or the hypervisor, that samples the system-wide memory bandwidth and suspends the best-effort applications when the accumulated overhead exceeds the level at which the real-time application can be guaranteed to meet its timing requirements.

Concretely, the memory profile of the real-time application is constructed from a measured sequence of bandwidth samples from a series of runs, that are summarized as a set a plateaux that conservatively approximate the effect of possible variations in the execution path and any delay. The run-time system then periodically samples the global memory bandwidth usage. On each sample, it uses the memory system characterization to estimate the current sample overhead, given the current plateau and the observed global bandwidth. If the accumulated overhead becomes too high, the run-time system suspends all of the best-effort applications. When the real-time application ends its computation within the current period, the best effort applications are resumed. Therefore, the memory contention induced by the best-effort applications will impact the real-time application during at most one sampling period per real-time application period.

We have prototyped our approach on a SABRE Lite multicore system using Linux 3.0.35. One core runs the real-time application, while the other three cores run best-effort applications. We have modified the Linux kernel, to implement the memory bandwidth sampler using counters available in the memory subsystem, and to make it possible to suspend and resume the best-effort applications on the best-effort cores.

Our contributions are the following:

- We introduce a periodic microbenchmark for characterizing the impact of memory accesses on execution time overhead for a given multicore system.
- We experimentally quantify the overhead on the microbenchmark induced by the system-wide memory bandwidth. Indeed, on the SABRE Lite, the periodic microbenchmark can be slowed down by up to a factor of 230% under high memory-bandwidth conditions.
- We experimentally show that on the SABRE Lite there is a bandwidth threshold of 146 Mb/s under which the performance of the microbenchmark is never impacted. This means that any real-time application that has memory needs below that level will run without been impacted by the memory accesses of best-effort applications.
- We have implemented a profiler that determines the memory bandwidth usage profile of a real-time application, and an algorithm to approximate this profile into a set of plateaux. In practice, we found that 5 plateaux were sufficient to capture the main variations in the MiBench applications.
- We show that our approach is able to limit the overhead on MiBench applications to at most 4% under high memory-bandwidth conditions.
- We compare our approach with the aforementioned baseline solution. Using a best-effort application that we designed to stress memory, but to do so only periodically, we show that this application makes progress when it does not induce an overhead of more than 5% on a MiBench application running concurrently.

The rest of this paper is organized as follows. Section 2 presents our target hardware and illustrates the problem of overhead due to high memory bandwidth on the applications of the MiBench embedded benchmark [15] and a microbenchmark. Section 3 presents our approach, particularly focusing on our off-line and run-time profiling strategies. Section 4 evaluates our approach on the MiBench applications. Finally, Section 5 presents related work and Section 6 concludes.

2 Problem characterization

In this section, we first describe our target hardware, and then present a microbenchmark that illustrates the problem of overhead induced by high memory bandwidth usage in an artificial, controlled setting.

2.1 Architecture of the SABRE Lite

In this paper, we target embedded systems, as used in the automotive domain, that have strong hardware cost constraints. We choose the SABRE Lite multicore system (see Figure 1) [14] as an experimentation platform since it has already been adopted by some industry leaders.

The processor of the SABRE Lite is an i.MX 6, which is based on a quad-core Cortex A9 MPCore [7]. Each core has two 32-kilobyte 4-way set-associative L1 caches, one for data and the other for instructions. Each core is also connected to an external 1-megabyte 16-way set-associative L2 cache [6] that is shared across all the cores. The memory controller manages access to one gigabyte of DDR3 RAM that can be used by all the cores [14]. Each core contains six configurable hardware counters to gather statistics on the operation of the processor (number of cycles, etc.) and the memory system (L1 accesses, L1 misses, etc.) [4, 5]. The memory controller contains hardware counters that measure global memory traffic (read/write bytes, read/write access, etc.) on the platform [14].

The SABRE Lite's L2 cache consists of two equal-sized partitions that can be pinned to particular cores, thus potentially reducing the interference between applications. We have disabled cache pinning, however, because it does not eliminate the problem of memory contention, since when the applications run out of cache space their requests on the memory bus will again interfere. Pinning cache partitions to cores furthermore artificially limits the number of cache lines available to each application, independent of their actual, changing needs.

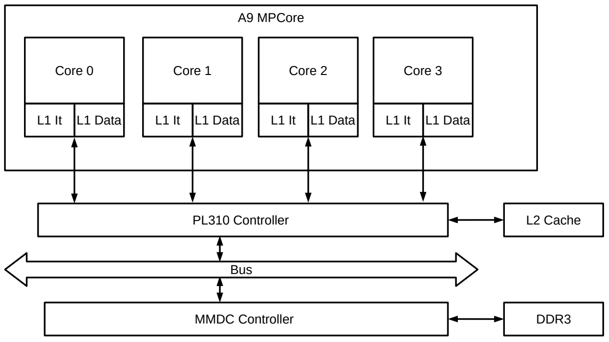


Figure 1: Architecture of the SABRE Lite board

2.2 Execution time overhead in the presence of memory contention

To illustrate the problem of memory contention, we measure the impact that is induced by memory-access intensive best-effort applications on the execution time of the applications of the MiBench embedded benchmark suite [15]. We have chosen this benchmark suite because it targets embedded systems, because the accompanying article has been cited almost 2400 times,[1] and because it has been used in many studies. MiBench comprises 35 applications covering a variety of embedded domains, including Automotive and Industrial Control, Networking, and Telecommunications. We exclude 19 applications that contain x86 code or that relate to long-running or office applications, leaving 16 applications. Most of the benchmarks are provided with "small" and "large" datasets. For most of the applications, we use only the small datasets, as these result in an execution time of at most 150ms when run in isolation that is more typical of real-time applications. For the susan benchmark, we used the large dataset since the small one is indeed too small in terms of execution time.

The experiment runs on the SABRE Lite, running a 3.0.35 Linux kernel that has been ported by Freescale to the i.MX 6 architecture. We run a MiBench application on one core and a *load* on each of the other three cores. As the load, we use the `add` kernel of the STREAM [21] memory bandwidth benchmark suite, which sums the elements of two very large arrays. All MiBench applications are compiled using GGC 4.6.3 with the option `-O2`. The loads are compiled with GGC 4.6.3 using the option `-O3` and several other options to maximize the generated memory bandwidth on the SABRE Lite hardware.[2] All processes run under the FIFO Linux scheduling policy with the maximum priority and are pinned to a core to avoid migration. Datasets are stored in an in-memory file system, to eliminate the cost of disk access. Each experiment involves 100 runs, from which we collect the maximum run time and overhead.

Table 1 shows the average run time for each MiBench application running alone, with its standard deviation, the maximum run time when the application runs alone, and the overhead induced by three identical instances of the `add` kernel, comparing the maximum run time of the application when run in parallel with the `add` kernel to the maximum run time of the application when running alone. The maximum overhead is 32%, with 8 of the 16 tests incurring an overhead of over 5%.

To study the memory contention problem in a more controlled setting, we have designed a real-time periodic microbenchmark that exhibits a constant rate of memory accesses. In each activation period, the microbenchmark copies the successive elements of an array, to generate memory traffic. This array copy generates the highest memory bandwidth that we measure; to create variants with lower bandwidth requirements, we optionally follow each element copy with a delay loop that increments the just-copied element a fixed number of times. The locations accessed in this delay loop should all be present in the L1 cache, which is local to the core, and thus they

[1]Google Scholar, January 2015

[2]-mthumb -mfpu=neon-fp16 -march=armv7-a -funsafe-math-optimizations -ffast-math -mtune=cortex-a9 -mcpu=cortex-a9 -mfloat-abi=softfp

Application	Description	Mean Runtime	Max Runtime	Ovd.
susan large -c	auto: image recognition	25.1 ± 0.07	25.5	32%
fft 4 8192 -i	telecomm: IFFT	24.2 ± 0.05	24.3	22%
qsort	auto: quick sort	51.5 ± 0.07	51.7	20%
fft 4 4096	telecomm: FFT	10.6 ± 0.09	10.7	18%
rijndael encode	security: block cipher	43.2 ± 0.42	43.9	16%
patricia	network: tree structure	58.1 ± 0.36	60.4	15%
susan large -e	auto: image recognition	56.6 ± 0.08	56.9	10%
rijndael decode	security: block cipher	40.8 ± 0.45	41.5	10%
dijkstra	network: shortest path	70.6 ± 0.24	71.6	5%
basicmath	auto: math calculations	17.1 ± 0.03	17.2	5%
adpcm encoder	telecomm: speech	34.2 ± 0.05	34.5	4%
sha	security: secure hash	9.4 ± 0.07	9.5	3%
adpcm decoder	telecomm: processing	95.7 ± 0.06	95.8	1%
crc32	telecomm: cyclic redundancy check	149.8 ± 0.23	150.2	1%
susan large -s	auto: image recognition	279.3 ± 22.01	316.5	0%
bitcount	auto: bit manipulation	31.2 ± 2.29	36.5	-9%

Table 1: Run time (in ms) in isolation and overhead for the MiBench applications. The overhead compares the maximum run time with three symmetric loads to the maximum run time of the application alone. All tests use the "small" dataset, except where noted.

do not involve memory accesses. The microbenchmark is compiled with GGC 4.6.3 using the `-O2` option. Its duration is 5 milliseconds and its period is 100 milliseconds. This execution time is small as compared to the period, and is thus not realistic for a real-time system in which one would like to get as much benefit as possible out of the processor. Nevertheless, we have chosen these parameters such that the overhead induced by the concurrent best-effort applications does not cause the microbenchmark to exceed its period, and disrupt the experiment process.

Using our microbenchmark, our second experiment simulates a real-time application, with various memory bandwidth requirements, that executes concurrently with best-effort applications (loads) running on a varying number of cores. As loads, we again use the `add` kernel, compiled with the same options used for the loads in the MiBench experiment. For the loads, we additionally vary the memory bandwidth requirements, and consider both symmetric and asymmetric loads among cores. Specifically, each core other than the one running the microbenchmark runs zero or one loads. Processes are pinned to their cores. If a core runs neither the microbenchmark nor a load, it runs an infinite loop that performs no memory accesses. This infinite loop is run at the maximum FIFO priority to avoid interference from OS tasks. We define the requested memory bandwidth of a load as the measured memory bandwidth when the load is run as a standalone process in the system. To be able to vary the requested memory bandwidth, we follow the same strategy as for the microbenchmark, introducing additional instructions that do not involve memory accesses uniformly throughout the source code.

Table 2 summarizes the maximum overhead incurred by the largest possible load that can be generated in each case. For 146 MB/s, we observe that the microbenchmark is essentially not impacted by the best-effort applications. Indeed, the observed 1.5% maximum overhead is within the margin of error of the measurement process. This is an important result for real-time application designers about the SABRE Lite system, since any real-time application that has memory bandwidth needs that are consistently below that threshold will be unaffected by best-effort applications. Table 2 also shows that the microbenchmark starts to be highly impacted when its bandwidth reaches 235 MB/s, with a maximum overhead of 13% in that case. The worst observed overhead is 230%, which is obtained for the microbenchmark with a memory bandwidth of 710 MB/s.

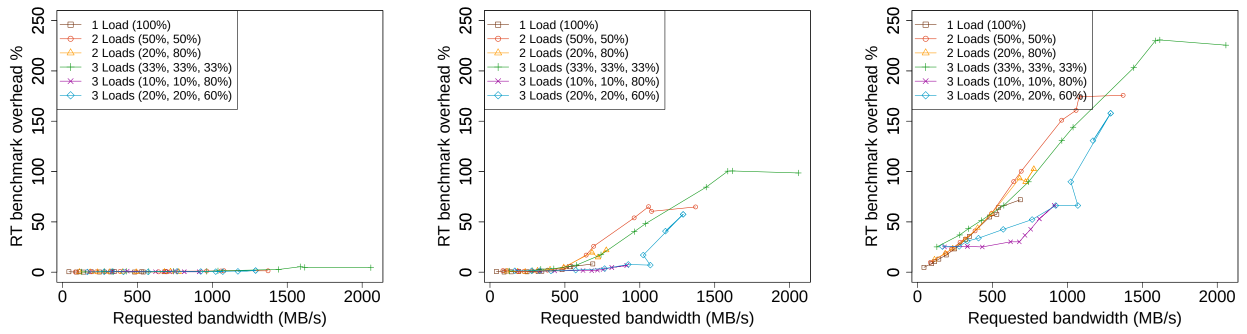
Real-time task bandwidth	Delay loop iterations	Maximum overhead
146 MB/s	10	1.5 %
204 MB/s	7	5 %
235 MB/s	6	13 %
281 MB/s	5	30 %
343 MB/s	4	57 %
438 MB/s	3	100 %
613 MB/s	2	180 %
710 MB/s	0	230 %

Table 2: Maximum RT benchmark overhead

We now explore these results in more detail. Figure 2 shows the overhead incurred by the 204, 438 and 710 MB/s microbenchmark instances for varying loads. These results show not only how the overhead increases, as the bandwidth requirement of the loads increases, but also the impact of varying ratios of load across the best effort cores. We observe that a symmetric stress load, *i.e.*, two identical loads or three identical loads, typically induces a larger overhead than an asymmetric load.

Memory bandwidth contention not only affects the microbenchmark, but also the loads themselves. As memory bandwidth contention increases, all of the applications are slowed down, reducing the rate of their memory accesses. Figure 3 shows the system-wide memory bandwidth measured depending on the bandwidth requested by the microbenchmark and the best-effort applications. We find that the memory system starts to saturate around a measured bandwidth of 1000 MB/s, regardless of the distribution of load.

The saturation of the memory system implies that the requested bandwidth, measured on the real-time application or the loads in isolation, is only meaningful as a characterization of the intent of the application, not of its actual run-time behavior in a specific execution context.



(a) RT benchmark 204 MB/s (b) RT benchmark 438 MB/s (c) RT benchmark 710 MB/s

Figure 2: Overhead induced by best-effort tasks on the real-time microbenchmark

Accordingly, Figure 4 relates the measured bandwidth, shown in Figure 3, to the overhead, shown in Figure 2. To illustrate this computation, consider the case of the real-time microbenchmark with bandwidth requirement 438 MB/s and three symmetric loads with requested bandwidth around 1500 MB/s. Figure 2b shows that this configuration leads to an overhead of around 80% and Figure 3b shows that this configuration leads to a measured bandwidth of around 1000 MB/s. Thus, in this configuration, a measured bandwidth of around 1000 MB/s implies an overhead of around 80%.

The results, in Figure 4, show that all loads, whether symmetric or asymmetric, follow essentially the same pattern, with an initial plateau of very low overhead, followed by a sharp increase, as the memory system begins to saturate, at a point depending on the requirements of the real-time application, and concluding with the largest overhead at around 1000 MB/s. Since the memory system is a black box, we do not know exactly how requests are served. Still, there are a few spikes that suggest that a burst of requests in a specific pattern may be served in a more efficient way.

3 Approach

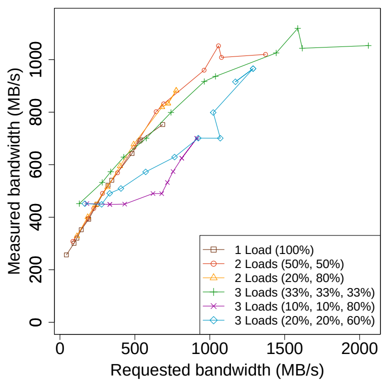
We target the setting of an *n*-core machine, with a real-time application running on one core, and best-effort applications running on some or all of the remaining cores. In this setting, we have two complementary goals: (i) we want to ensure that the memory induced overhead on the real-time application remains below a threshold specified by the system designer, and (ii) we want to avoid preempting the best-effort applications when the overhead that they incur on the real-time application is acceptable. As is standard for real-time computing, we assume that the real-time application is known in advance. The real-time application is periodic, and it can be profiled to determine its worst-case execution time and worst-case resource consumption properties during each execution activation. On the other hand, new best-effort applications can start and stop at any time, and we do not know any properties of their memory usage.

To achieve our goals, we propose an approach in three stages. The first stage involves an off-line characterization of the memory system. More precisely, we run the real-time microbenchmark for several bandwidth values and for each instance, we measure the maximum possible overhead that can be incurred for a given measured memory bandwidth. This characterization is only performed once for a given platform, and henceforth serves as a reference for system designers. The second stage involves off-line profiling of each real-time application, in isolation, to determine its memory bandwidth requirements at each given time within its period. This information is used to determine which microbenchmark bandwidth instance is the closest to the application at each moment. At run time, in the third stage, a run-time system integrated into the OS kernel or hypervisor samples the system memory bandwidth and uses the results obtained from the appropriate microbenchmark to accumulate a running sum of the maximum possible overhead for the current period. If the accumulated overhead is greater than the desired one, the run-time system suspends the best-effort applications. Suspended applications are allowed to run again when the real-time application completes its execution in the current period.

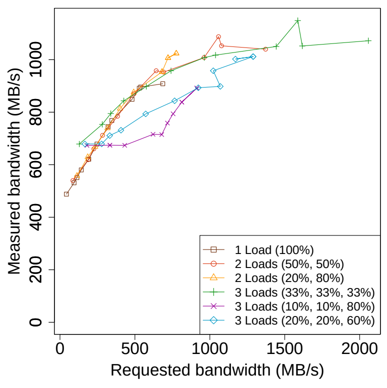
In the rest of this section, we describe the various mechanisms and analyses used to support this approach.

3.1 Measuring memory bandwidth

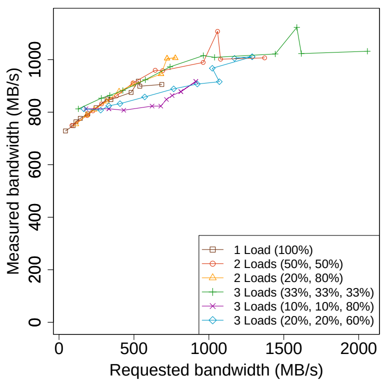
The three stages of our approach, off-line memory-system characterization, off-line profiling of the real-time application, and run-time scheduling, require information about the system memory bandwidth. We measure the bandwidth using a sampling technique, triggered by a timer interrupt on one of the cores dedicated to best-effort applications. On each timer interrupt, the interrupt



(a) RT benchmark 204 MB/s

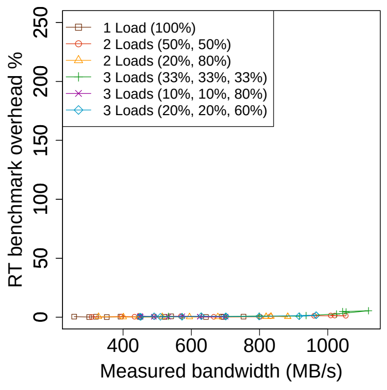


(b) RT benchmark 438 MB/s

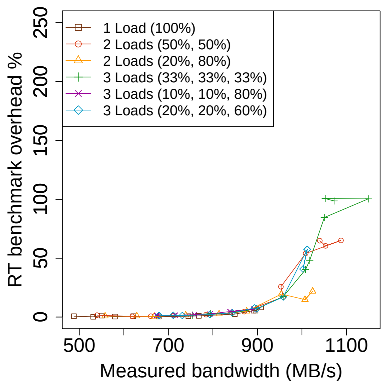


(c) RT benchmark 710 MB/s

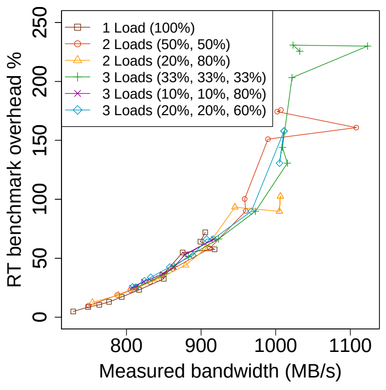
Figure 3: Requested bandwidth vs. measured bandwidth



(a) RT benchmark 204 MB/s



(b) RT benchmark 438 MB/s



(c) RT benchmark 710 MB/s

Figure 4: Overhead incurred on the RT microbenchmark depending on the measured bandwidth

reads the current value of the memory subsystem counter and resets it to 0. Sampling induces an overhead, since computations are interrupted regularly. Nevertheless, the sampling is never done on the core that is running the real-time application, and thus has no observable impact on its performance.

During our experiments, we initially observed random spikes that were due to device interrupts and clock ticks. When profiling the real-time application, for which we compute the bandwidth requirement at the granularity of a single sample, we disable the sources of these interrupts so that the measured bandwidth is only influenced by the real-time application.

3.2 Memory-system characterization

This stage performs an off-line characterization of the memory system that establishes a relationship between the measured bandwidth and the worst-case overhead that can be incurred. In Section 2.2, we already showed experimentally the relationship between the observed memory bandwidth and the overhead incurred by the microbenchmark (Figure 4). To use this information directly, however, requires that we know the current bandwidth requirements of the real-time and best effort applications. Neither is known precisely at run time: as is standard for resource requirement analysis, we only collect a conservative approximation of the bandwidth requirement of the real-time application, and we have no information about either the number of best-effort applications nor their bandwidth requirements.

To address these sources of imprecision, we use the mappings of observed bandwidth to overhead, collected as described in Section 2.2, to construct a conservative approximation. First, to account for the lack of information about the best-effort applications, for a given real-time bandwidth requirement and observed bandwidth, we consider all possible configurations of best-effort applications, as shown in the corresponding graph of Figure 4, and take the maximum of the overheads associated with each one. Second, to account for the fact that the real-time application's profile contains an upper bound

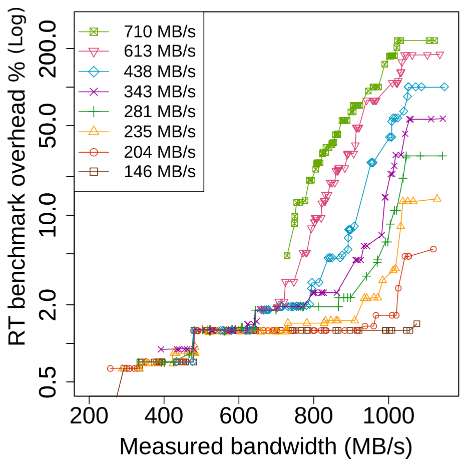


Figure 5: Memory system characterization

on the bandwidth requirement of the real-time application at each point in time, we consider not only the overhead values for that real-time bandwidth requirement, but also the values for all smaller real-time bandwidth requirements, for the given observed bandwidth. The result is a table that maps a real-time application memory bandwidth requirement and an observed memory bandwidth to the maximum overhead value obtained by these computations.

This memory-system characterization does not depend on the real-time application, and thus has to be done only once for a given platform. The complete table for the SABRE Lite is shown as a graph in Figure 5.

3.3 Constructing the real-time application memory profile

We construct a profile of the memory-bandwidth requirements of the real-time application in order to allow the run-time system to estimate which microbenchmark instance matches best its behavior in a given sampling interval. To construct the profile, we must consider how to collect memory bandwidth information, the potential impact of any concurrent best-effort applications on the validity of the collected information, and the granularity of information to manage at run time.

Estimating the memory bandwidth requirement of the real-time application amounts to a resource requirement estimation problem, analogous to Worst Case Execution Time (WCET) analysis. For WCET, methods have been developed that rely on either analysis of the source code or observations during testing. Due to the difficulty of understanding the precise behavior of memory controllers on particular instructions, we follow a testing-based approach. Such approaches, however, are sensitive to the choice of test data, as different inputs can cause the application to execute different sequences of instructions. Exhaustively testing the real-time application over all possible inputs is typically not practical. In the context of WCET analysis, techniques have been developed to reduce the number of tests, by considering only values that trigger paths that are likely to have the maximal resource consumption [33]. These techniques are independent of the resource considered and thus should be adaptable to the analysis of memory bandwidth. Once we have chosen the set of test inputs, we run the real-time application on each test input, with sampling enabled, and accumulate for each sample the maximal memory bandwidth requirement observed for any input.

The above analysis, however, only considers the behavior of the real-time application when it is run on the system alone, and thus does not take into account the delay incurred by concurrent memory accesses by best-effort applications. Given a previous delay of T samples, the estimated bandwidth requirements of the previous T samples can have an influence on the actual bandwidth requirement of the current sample. Our goal is to limit the delay to a percentage of the expected running time of the real-time application within a period. From this percentage and the expected running time we can calculate the number of samples T by which the real-time application can be delayed. We then update the memory bandwidth requirement collected for each sample by the larger of its observed maximum and the maximum of the memory bandwidth requirements collected for the T previous samples.

The result of the above two steps is an array with an entry for each sample within the period and the maximum memory bandwidth requirement expected at that moment in time. In principle, such an array could be used at run time directly, in combination with the microbenchmark information, to determine the maximum possible overhead based on the measured bandwidth information. However, to ensure that the behavior of the real-time application within a slice is uniform, the duration of a sample must be small, and thus the size of the resulting array is typically large. Furthermore, the information represented by the array is likely to be overprecise, and thus inaccurate, due to the inherent measurement imprecision.

To increase the granularity of the information maintained at run time, we smooth the data, merging samples with similar memory bandwidth requirements into *plateaux* with an approach inspired by algorithms for adaptive approximation by piecewise constants [13]. The algorithm merges samples, starting with those merges that generate the least approximation, to materialize the phases of the application. The algorithm stops when it has reduced the profile to n plateaux. The result of the merging process is a smaller array representing a sequence of plateaux, each having an offset, a duration, and a maximum memory bandwidth requirement.

3.4 Run-time scheduling of best-effort applications

We allow best-effort applications to run as long as they are not expected to impact the real-time application. Our approach at run time is to use the collected profile information to accumulate the maximum possible overhead for each sample. If the accumulated sum becomes greater than the threshold specified by the system designer, the best-effort applications are suspended and the associated cores run high priority idle tasks so that those cores generate no further memory accesses. The best-effort applications are later resumed when the real-time application ends its execution in the current period.

The maximum possible overhead during a sample is computed as follow: first, using the elapsed execution time of the real application in the current period, the run-time system knows the current plateau and therefore the microbenchmark instance. The current sample's memory bandwidth is then used as an index into the microbenchmark table to obtain the maximum overhead value. This approach requires the run-time system to be aware of the real-time application profile and of the maximum overhead that can be tolerated. Our prototype implementation is done within a Linux kernel module and this information is communicated from the application to the module using a callback on a parameter in the *sysfs*.

4 Evaluation

Our goals for our approach are first to ensure that the real-time application execution time is not impacted by best-effort application and second that best-effort applications run as long as they do not impact the real-time ones. In the rest of this section, we evaluate both goals using the subset of 8 applications from the MiBench suite [15] presented in Table 2, using the datasets described there. Before giving the results of our experiments, we first present the computed memory profiles for these applications. All experiments are run 100 times on our SABRE Lite system.

4.1 Constructing memory profiles

Figure 6 shows the memory profiles of the MiBench applications on the considered datasets. The figure shows the maximum measured bandwidth requirement for each sample, and the result of smoothing these values into plateaux, as described in Section 3.3. In all cases, we found that 5 plateaux were sufficient to capture the main variations in the application memory bandwidth.

4.2 Preservation of real-time properties

We consider that our approach is *effective* when the overhead on the real-time application remains under the threshold specified by the system designer, regardless of the properties of any high-bandwidth best-effort applications that run simultaneously. In our experiments, we use 5% as the threshold; 5% is commonly viewed as a lower bound on the precision of performance measurements. We run each MiBench application in parallel with one to three instances of the `add` kernel without the delay loop. Table 3 shows the maximum run time of each MiBench application when run alone, the maximum overhead when the `add` kernels run continuously (No limit), and the maximum overhead observed with our approach (5% limit). In every case, the maximum overhead observed with our approach is under 5%.

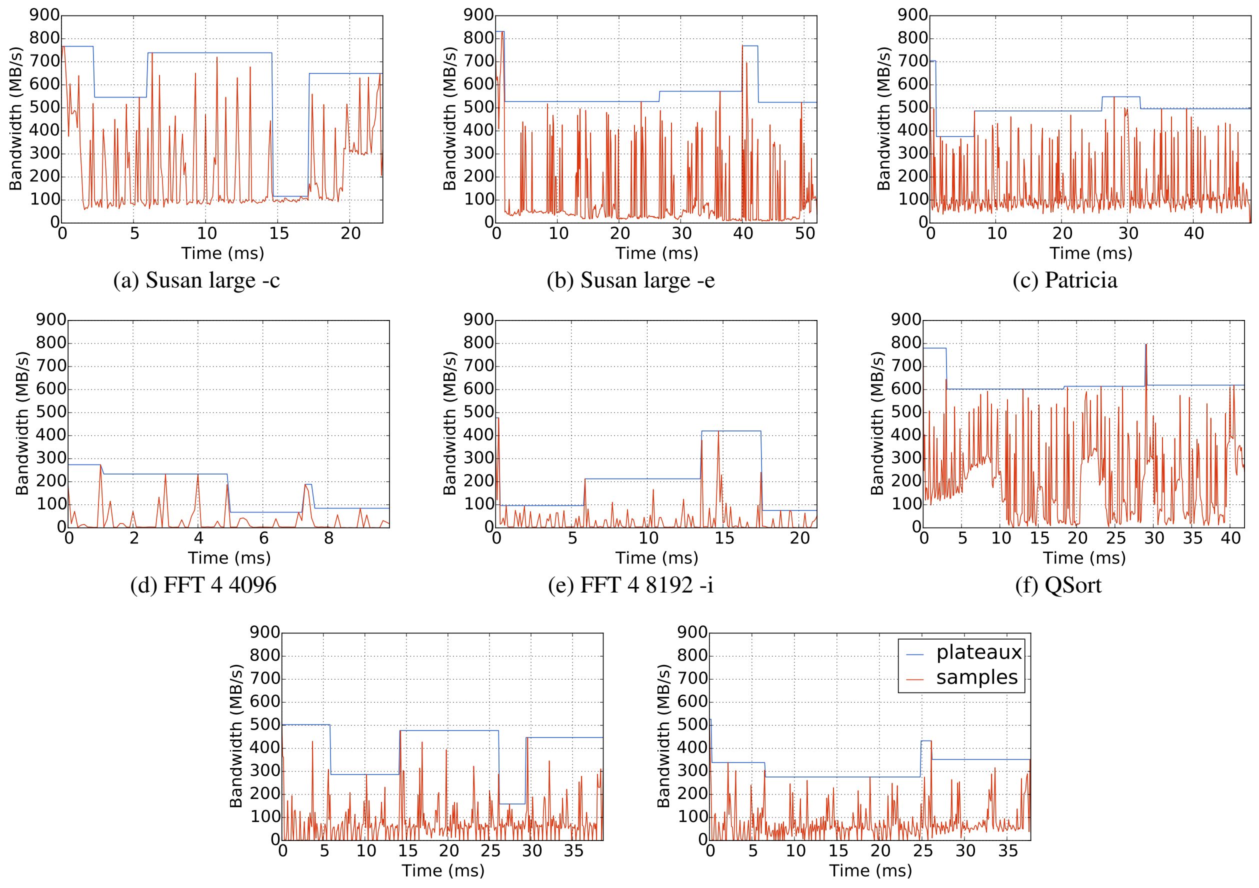
Application	Max run time alone (ms)	Overhead	
		No limit	5% limit
susan large -c	25.46	32%	3%
fft 4 8192 -i	24.34	22%	1%
qsort	51.68	20%	0%
fft 4 4096	10.74	18%	4%
rijndael encode	43.91	16%	3%
patricia	60.37	15%	1%
susan large -e	56.87	10%	2%
rijndael decode	41.52	10%	2%

Table 3: Maximum overhead on MiBench when run with up to 3 loads

4.3 Progress of best-effort applications

We consider that our approach is *beneficial* when the best-effort applications are able to run with the real-time applications concurrently. To study this property, we require a best-effort application with varying memory requirements. For this, we have implemented an application that finds the lowest-cost path between two points in a map represented as a weighted graph. Such a computation would be performed by an outsmarting traffic application in an automotive environment. To further vary the memory requirements, we design the application to intersperse graph exploration phases with computation phases. Identical copies of the best-effort application run on one to three cores. We use the MiBench applications as the real-time application. Each MiBench application is run periodically, with a period of 100ms.

In our baseline solution, the best-effort applications are suspended whenever the real-time application begins to execute, amounting to 0% of concurrent execution. Our approach allows the best-effort applications to continue as long as the overhead bound is expected to be respected. For susan small -c, running 1, 2, or 3



(a) Susan large -c (b) Susan large -e (c) Patricia
(d) FFT 4 4096 (e) FFT 4 8192 -i (f) QSort
(g) Rijndael encode (h) Rijndael decode

Figure 6: MiBench memory profiles

instances of the best-effort application without our approach leads to overheads on susan small -c of 3%, 6%, and 9% respectively, while our approach allows 1, 2 or 3 instances of the best-effort application to execute concurrently with susan small -c during 54% of susan small -c's activation, leading to a measured overhead of at most 2.75% on susan small -c. For FFT 4 8192 -i, 1 or 2 instances of the best-effort application cause no overhead on FFT 4 8192 with or without our approach. For 3 instances, without our approach, FFT 4 8192 incurs an overhead of over 8%, while our approach allows the best effort applications to run for 76% of the execution of FFT 4 8192 -i with a measured overhead of at most 2% on FFT 4 8192 -i.

5 Related work

A variety of approaches have been proposed to reduce the impact of memory contention on process execution times. These range from offline approaches, in which a Worst Case Execution Time (WCET) is computed that takes memory contention into account, to various changes to software, hardware, or a combination of both to reduce or eliminate the impact of memory contention on application execution times.

WCET approaches. Pellizzoni et al. [26] have developed a method for calculating the WCET in a multicore context, building on methods used in a single core context. Nowotsch et al. [24] have identified the problems of interaction and contention in the context of an ARINC 653 partitioned avionic environment executing on a multicore platform. Bin et al. [9, 10] have developed a methodology to compute the WCET of avionic applications sharing the same multicore system. Jean et al. [17] have studied the problem of WCET for multicore processors in the context of an embedded hypervisor in the context of avionic systems.

Our approach relies on run-time monitoring and can benefit from any advance in WCET computation.

Software approaches. Caccamo et al. [12] and Yun et al. [34] have developed mechanisms for sharing the memory bandwidth in the context of a multicore architecture with no shared cache. First, they measure the memory traffic generated by each core, using the hardware L1 miss counter. Then, they pause the cores that are generating too much traffic. We target hardware that has a shared L2 cache, on which the number of L1 cache misses does not reflect the memory bandwidth usage.

Muralidhara et al. [23] use an 8-core hardware platform connected to the main memory by several channels, each independently controlling a portion of the physical memory. The interferences between applications can thus in theory, be eliminated, if the data used by each application are associated with one or more dedicated channels. Applications that do not interfere can be grouped on the same channel. Liu et al. [19] combine "cache coloring" with the partitioning by channel in the Linux kernel in order to partition the cache and the memory.

Seo et al. [29] use hardware counters to calculate the number of retries necessary for a request generated by the last level cache to be accepted by the memory controller. This information is used to determine the memory contention level. Then, they construct a model relating the level of memory contention to the application performance which is used to schedule the tasks that consume the most memory on the same core. This study targets whole system performance improvement, while we aim first at protecting real-time applications.

Hardware approaches. Ungerer et al. [32] have designed a multicore architecture for applications having varying degrees of criticality that permits a safe computation of the WCET. Lickly et al. [18] propose a new multithreaded architecture for executing hard real-time tasks that provides precise and predictable timings.

Moscibroda et al. [22] propose a new memory controller designed to provide memory access fairness across different consumers. Shah et al. [30] present a new scheduling policy for the bus arbiter which mixes a time-division based policy that is designed to respect real-time constraints and a static priority based policy that is designed to have good performance. In the context of a chip-multiprocessing system where the number of processors is greater than or equal to the number of tasks, Schoeberl et al. [28] propose a Time Division Multiple Access memory arbiter to control access to the main memory.

All of these approaches involve hardware that does not currently exist, while our approach targets current machines.

Mixed approaches. Pellizzoni et al. [25] propose an approach in which hardware buffers are introduced that make it possible to schedule accesses to shared resources in such way as to avoid that two consumers/producers simultaneously access the same resources. Applications must be structured into phases that have particular memory-access properties. Even if such hardware were available, the approach would not be compatible with legacy best-effort applications, which are not structured in this way. Boniol et al. [11] propose an algorithm for restructuring applications automatically to fit the requirements of such a system. Finally, Rafique et al. [27] designed a "fair bandwidth sharing memory controller" able to spread memory bandwidth across all the consumers, which is coupled to a "feedback-based adaptive bandwidth sharing policy" managed by the operating system.

Our approach requires neither new hardware nor any changes to the best-effort application source code.

6 Conclusion

In this paper, we have presented an approach permitting to mix applications with different levels of criticality on a single small multicore machine while bounding the overhead that the real-time application can incur due to memory-demanding best-effort applications. Our approach relies on an off-line analysis of both the memory system and the real-time application, and a run-time system that controls the scheduling of the best-effort applications. Our approach allows the best-effort applications to run concurrently with the real-time application as long as the overhead limit on the real-time application can be guaranteed to be respected. No modifications to the best-effort applications are required.

Our approach suspends all best-effort applications as soon as the possibility of an excessive delay is detected. To further increase the amount of time in which best-effort applications are allowed to run, an alternate approach would be to suspend only the best-effort applications running on the cores having the greatest L1 cache activity. Unlike L2 cache activity, which is global to the system, measuring core-specific L1 cache activity is possible on standard processors, because the L1 cache is core specific. Another approach would be to exploit the observation that during an execution phase with low memory bandwidth requirements an application incurs no memory-induced delay, regardless of the requirements of other concurrent applications. When the current plateau of the real-time application indicates that it is in such a phase, it could be possible to restart the best-effort applications, for the phase's duration.

Finally, our approach currently accommodates only one real-time application, or multiple real-time applications without preemption. Handling multiple real-time applications with preemption would require switching real-time application profiles when a real-time application is preempted by another. We leave this to future work.

References

[1] Freescale boards. `http://www.freescale.com/webapp/sps/site/overview.jsp?code=SABRE\_HOME{}\&fsrch=1\&sr=1\&pageNum=1`.

[2] Okl4 microvisor. `http://www.ok-labs.com/products/okl4-microvisor`.

[3] PikeOS. `http://www.sysgo.com`.

[4] ARM. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, rev C.b, November 2012.

[5] ARM. *Cortex-A9 Technical Reference Manual*, rev r4p1, June 2012.

[6] ARM. *Level 2 Cache Controller L2C-310 Technical Reference Manual*, rev r3p3, June 2012.

[7] ARM. *Cortex-A9 MPCore Technical Reference Manual*, June rev r4p1, 2012.

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP* (2003), pp. 164–177.

[9] Bin, J., Girbal, S., Perez, D. G., Grasset, A., and Merigot, A. Studying co-running avionic real-time applications. In *Embedded Real Time Software and Systems (ERTS)* (Feb. 2014).

[10] Bin, J., Girbal, S., Perez, D. G., and Merigot, A. Using monitors to predict co-running safety-critical hard real-time benchmark behavior. In *International Conference on Information and Communication Technology for Embedded Systems (ICICTES)* (Jan. 2014).

[11] Boniol, F., Cassé, H., Noulard, E., and Pagetti, C. Deterministic execution model on COTS hardware. In *International Conference on Architecture of Computing Systems (ARCS)* (2012), Springer-Verlag, pp. 98–110.

[12] Caccamo, M., Pellizzoni, R., Sha, L., Yao, G., and Yun, H. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2013), pp. 55–64.

[13] DeVore, R. A. Nonlinear approximation. *ACTA NUMERICA 7* (1998), 51–150.

[14] Freescale, S. *i.MX 6Dual/6Quad Applications Processor Reference Manual*, rev 1, April 2013.

[15] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, IEEE International Workshop* (2001), pp. 3–14.

[16] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. The performance of μkernel-based systems. In *SOSP* (1997), pp. 66–77.

[17] Jean, X., Gatti, M., Faura, D., Pautet, L., and Robert, T. A software approach for managing shared resources in multicore ima systems. In *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd* (Oct. 2013), pp. 7D1–1–7D1–15.

[18] Lickly, B., Liu, I., Kim, S., Patel, H. D., Edwards, S. A., and Lee, E. A. Predictable programming on a precision timed architecture. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (2008), ACM, pp. 137–146.

[19] Liu, L., Cui, Z., Xing, M., Bao, Y., Chen, M., and Wu, C. A software memory partition approach for eliminating bank-level interference in multicore systems. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2012), pp. 367–376.

[20] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2013), pp. 45–54.

[21] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995).

[22] Moscibroda, T., and Mutlu, O. Memory performance attacks: Denial of memory service in multi-core systems. In *16th USENIX Security Symposium (SS)* (2007), pp. 18:1–18:18.

[23] Muralidhara, S. P., Subramanian, L., Mutlu, O., Kandemir, M., and Moscibroda, T. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-44)* (2011), pp. 374–385.

[24] Nowotsch, J., and Paulitsch, M. Leveraging multi-core computing architectures in avionics. In *European Dependable Computing Conference (EDCC)*, pp. 132–143.

[25] Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., and Kegley, R. A predictable execution model for COTS-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Apr. 2011), pp. 269–279.

[26] Pellizzoni, R., Schranzhofer, A., Chen, J.-J., Caccamo, M., and Thiele, L. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010* (Mar. 2010), pp. 741–746.

[27] Rafique, N., Lim, W.-T., and Thottethodi, M. Effective management of DRAM bandwidth in multicore processors. In *Parallel Architecture and Compilation Techniques (PACT)* (Sept. 2007), pp. 245–258.

[28] Schoeberl, M., and Puschner, P. P.: Is chip-multiprocessing the end of real-time scheduling. In *9th International Workshop on Worst-Case Execution Time (WCET) Analysis* (2009).

[29] Seo, D., Eom, H., and Yeom, H. Y. MLB: A memory-aware load balancing for mitigating memory contention. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)* (2014), USENIX Association.

[30] Shah, H., Raabe, A., and Knoll, A. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Design, Automation Test in Europe Conference Exhibition (DATE)* (Mar. 2011), pp. 1–4.

[31] Steinberg, U., and Kauer, B. NOVA: A microhypervisor-based secure virtualization architecture. In *EuroSys* (2010), pp. 209–222.

[32] Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quinones, E., Gerdes, M., Paolieri, M., Wolf, J., Casse, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., and Mische, J. Merasa: Multicore execution of hard real-time applications supporting analyzability. vol. 30, pp. 66–75.

[33] Williams, N., and Roger, M. Test generation strategies to measure worst-case execution time. In *Automation of Software Test, ICSE Workshop on* (2009), pp. 88–96.

[34] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 24th Euromicro Conference on* (July 2012), pp. 299–308.