HAL
archives-ouvertes.fr

Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System

Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, Gilles Muller

To cite this version:

Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, Gilles Muller. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. 28th EUROMICRO Conference on Real-Time Systems (ECRTS'16), Jul 2016, Toulouse, France. hal-01346979

HAL Id: hal-01346979

https://hal.inria.fr/hal-01346979

Submitted on 20 Jul 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System

Antoine Blin*† Cédric Courtaud† Julien Sopena† Julia Lawall† Gilles Muller†
*RENAULT s.a.s. †Sorbonne Universités, Inria, UPMC Univ Paris 06, LIP6

***Abstract*—Complex embedded systems today commonly involve a mix of real-time and best-effort applications. The recent emergence of low-cost multicore processors raises the possibility of running both kinds of applications on a single machine, with virtualization ensuring isolation. Nevertheless, memory contention 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. Our approach estimates and limits the impact of the memory contention incurred by the best-effort applications on the execution time of the real-time application. We show that our approach is compatible with the hardware counters provided by current small commodity multicore processors. Using our approach, the system designer can 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.**

I. Introduction

In many embedded system domains, such as the automotive industry, it is necessary to run applications with different levels of criticality [13]. Some applications may have 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 suggestions on how to outsmart traffic. Traditionally, multiple applications are integrated in a vehicle using a *federated architecture*: Every major function is implemented in a dedicated Electronic Control Unit (ECU) [28] that ensures fault isolation and error containment. This solution, however, multiplies the hardware cost, and, in an industry where every cent matters, is increasingly unacceptable.

Recently, efforts have been made to develop an *integrated architecture*, in which multiple functions share a single ECU. AUTOSAR [16] is a consortium of actors from the automotive industry that defines a software architecture to exploit the benefits of integrated architectures by facilitating the reuse of applications. The AUTOSAR standard targets applications that control vehicle electrical systems and that are scheduled on a real-time operating system that is compliant with the AUTOSAR OS standard. Infotainment applications, however, typically target a Unix-like operating system, and thus still require the use of a federated architecture.

Recent experimental small uniform memory access commodity multicore systems provide a potential path towards a complete low-cost integrated architecture. Systems such as the Freescale SABRE Lite [1] offer sufficient CPU power to run multiple applications on a single low-cost ECU. Using *Virtualized architectures* [8], [18], [34], multiple operating systems can be used without modification. Recent hypervisors targeting embedded systems, such as SeL4 [2] and PikeOS [3], make it possible in the context of the automotive domain to dedicate one or several cores to a class of applications, and thus provide CPU isolation.

CPU isolation, however, is not sufficient to ensure that real-time applications can meet their performance constraints. For current inexpensive small multicore systems, the memory bus is also a shared resource. Therefore, it has been observed that the memory usage of applications running on one core may impact the execution time of applications running on the other cores [19], [26], [27]. An industrial base-line solution is to run critical applications in mutual exclusion, which guarantees no interference [14]. But this approach only permits multicore parallelism for non-critical applications, and thus wastes CPU resources and leads to longer latencies for non-critical applications.

In this paper, we propose a run-time software-regulation approach that has the goal of maximizing parallelism between real-time and best-effort applications running on a single low-cost multicore ECU. Our approach uses an overhead estimation derived from offline profiling of the real-time application, when running alone and in parallel with various loads, to estimate the slow down on the real-time application caused by memory interferences. When the estimated overhead reaches a predefined threshold, our approach suspends the best-effort applications, allowing the real-time task to continue executing without interference. Suspended best-effort applications are resumed when the real-time application ends its current activation. Our solution requires only system-wide memory counters that are available on most commodity multicore platforms.

A key observation behind our approach is that the overhead incurred by the real-time application depends both on the amount of traffic generated on the various cores and on the ratio between reads and writes in this traffic. To address this issue, we propose (i) a per-application off-line analysis for characterizing the performance overhead induced by increases in memory bandwidth and by various read-write ratios, that is sensitive to the read-write ratios in the different phases of the real-time application, and (ii) a run-time system, implemented within the operating system or 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, we first manually analyze the real-time application source code so as to identify phases during which the

application does a recurring job that is likely to generate a common memory access behavior. Then, we construct a per-phase overhead table for the real-time application based on the results of running it in parallel with a large variety of loads. Finally, at run time, the run-time system periodically samples the global memory bandwidth usage. On each sample, it uses the overhead table to estimate the overhead for the current sample, given the observed global bandwidth. If the accumulated estimated overhead becomes too high, the run-time system suspends all of the best-effort applications for the remainder of the current activation of the real-time application, to ensure that the current activation incurs no further overhead.

We have prototyped our approach on a SABRE Lite four-core multicore system using Linux 3.0.35. The run-time system is implemented as a kernel module, enabling it to sample the memory bandwidth using counters available in the memory subsystem, and to suspend and resume the best-effort applications on the best-effort cores. In our experiments, one core runs the real-time application, while one or more of the other three cores run best-effort applications. We assume that an upper bound on the number of active best-effort cores is fixed by the system designer and is thus constant throughout the activations of the real-time application. To emulate real-time applications, we have chosen the MiBench embedded benchmark suite [17] because it targets embedded systems. MiBench has been used in many studies, as reflected by the more than 2700 citations to the MiBench article.[1]

Our main contributions are the following:

- We introduce a load microbenchmark for characterizing the impact of memory accesses on execution time overhead for a given multicore system. This microbenchmark is configurable in terms of the ratio of reads and writes, and in terms of the delay between sequences of read and write memory accesses.
- We characterize the memory behavior of the MiBench applications. We show that 13 of the 35 applications may suffer from more than 5% overhead due to memory contention on the SABRE Lite. For applications with high memory demands such as `qsort`, the overhead is up to 183%.
- We propose a new approach to limit the overhead induced by loads to a threshold chosen by the system designer. Our approach uses a run-time controller that periodically samples the global memory bandwidth usage and uses profiling information to estimate the incurred overhead on the real-time application. Our approach suspends the best-effort applications as soon as the estimated overhead exceeds the threshold minus the proportion of the application execution time represented by one sample.
- We evaluate our approach on the 13 most demanding instances of the MiBench benchmarks that exhibit an overhead greater than 10% without our approach and a short running time of below 50ms. Our evaluation uses an overhead threshold of 5%. Under a variety of constant loads, our approach limits the overhead on 12 of these application instances to under 5%, while for the remaining application instance, the overhead only reaches 5.10%.
- We study the amount of parallelism permitted by our approach, *i.e.*, the percentage of the activation of the real-time application during which best-effort applications are also allowed to execute. For 12 of the 13 selected applications, we observe an increase in parallelism as compared to a baseline solution that suspends the best-effort applications on each activation of the real-time application. 7 of the applications achieve at least 70% of parallelism for low-bandwidth loads, regardless of the number of active best-effort cores. For the other 6 applications, the gain in parallelism depends on the number of cores used for the best-effort applications.

The rest of this paper is organized as follows. Section II first presents the SABRE Lite and the MiBench benchmark suite. We then illustrate the problem of overhead due to high memory bandwidth on the MiBench applications. Section III presents our approach, focusing on our off-line and run-time profiling strategies. Section IV evaluates our approach on the MiBench applications. Finally, Section V presents related work, and Section VI concludes.

II. Problem characterization

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

A. Architecture of the SABRE Lite

In this paper, we target embedded systems, as used in the automotive domain, which has strong hardware cost requirements. We choose the SABRE Lite multicore system [15] (see Figure 1) since it has already been adopted by some industry leaders as an experimental platform.

The processor of the SABRE Lite is an i.MX 6, which is based on a 1.2 GHz 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 can be either shared by all the cores or partitioned in multiples of 1/16th of the cache size. The Multi Mode DRAM Controller (MMDC) manages access to one gigabyte of DDR3 RAM that can be used by all cores [15]. 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 MMDC contains hardware counters that measure global memory traffic (read/write bytes, read/write accesses, etc.) on the platform [15], but no hardware counter is provided to identify the core that is the source of a L2 miss.

On the SABRE Lite, when using DDR3 RAM, the MMDC is accessible through a single AXI channel. This AXI channel has two dedicated request queues: a 16 entry queue for read requests and a 8 entry queue for write requests. Each request queue entry holds the information to access up to one cache line. A round-robin arbitration mechanism is used to send pending read and write requests into a final reordering buffer, before the request is sent to the RAM. We will show in Figure 5 that this mechanism has a significant impact on the

[1] Google Scholar, February 2016

bandwidth that can be achieved when mixing read and write accesses.

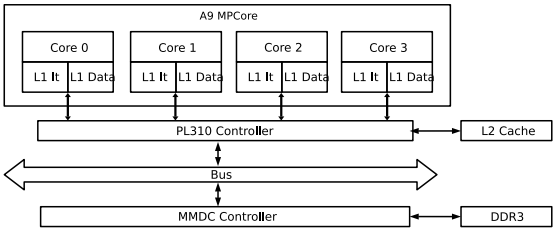


Fig. 1: Architecture of the SABRE Lite board

B. MiBench

Our experiments use the MiBench benchmark suite [17]. MiBench comprises 35 applications in 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. All of the benchmarks are provided with "large" and "small" data sets. We run the MiBench applications on a 3.0.35 Linux kernel that has been ported by Freescale to the i.MX 6 architecture.[2] All MiBench applications are compiled using GGC 4.9.1 with the option `-O2`. Data inputs are stored in an in-memory file system, to eliminate the cost of disk access. The L2 cache is not partitioned and is only used by the MiBench application running alone on core 0. Each experiment involves 150 runs, where we have discarded the first 20, to minimize variability. Table I shows the mean run time, the standard deviation, and the maximum run time. The mean run time ranges from 1 ms for `susan small -c` to 3 seconds for `crc32 large`, showing the large spectrum of application types.

C. Execution time impact of memory contention

Given the capabilities of the SABRE Lite board, one approach to reducing memory contention between classes of applications is to partition the L2 cache. Partitioning the L2 cache among the cores avoids interference at the cache level and limits contention to the memory bus accesses. Still, reducing the L2 cache size may impact performance for memory demanding applications. We first study the impact of cache partitioning on the performance of applications running alone, and then study the extent to which cache partitioning resolves the problem of memory contention between applications.

Figure 2 shows the impact of cache partitioning on the performance of the MiBench applications when run alone, as compared to the non-partitioned case. In each case, we have performed 150 runs, and discarded the first 20 results. We compare the maximum execution times, since we care about the worst case. Two configurations are studied: (i) the cache is split in half with one half associated to core 0, running the MiBench application, and the other half associated to the other three cores, (ii) the cache is split in two asymmetric

[2]https://github.com/boundarydevices/linux-imx6/tree/boundary-imx_3.0.35_4.1.0/

Application		Description	Mean runtime (ms)	Max runtime (ms)
basicmath	large	auto: math calculations	54.82 ± 0.03	54.94
	small		12.31 ± 0.01	12.33
bitcount	large	auto: bit manipulation	413.63 ± 14.50	449.63
	small		27.46 ± 1.12	30.52
qsort	large	auto: quick sort	23.44 ± 0.08	23.59
	small		18.28 ± 0.05	18.38
susan -e	large	auto: image recognition	56.54 ± 0.08	56.75
	small		2.08 ± 0.02	2.13
susan -s	large	auto: image recognition	270.97 ± 0.01	270.99
	small		17.96 ± 0.01	17.98
susan -c	large	auto: image recognition	23.80 ± 0.05	23.92
	small		1.08 ± 0.02	1.15
adpcm	large	telecom:	550.29 ± 0.08	550.49
encode	small	speech processing	30.83 ± 0.01	30.88
adpcm	large	telecom:	523.33 ± 0.07	523.52
decode	small	speech processing	26.06 ± 0.01	26.09
fft	large	telecom: FFT	120.17 ± 0.21	121.00
	small		8.40 ± 0.04	8.48
fft -i	large	telecom: inverse FFT	122.30 ± 0.17	122.98
	small		17.89 ± 0.05	18.01
crc32	large	telecom: cyclic	3068.97 ± 0.06	3069.18
	small	redundancy check	157.59 ± 0.01	157.62
patricia	large	network: tree structure	283.28 ± 2.97	289.77
	small		49.42 ± 0.06	49.58
dijkstra	large	network: shortest path	228.89 ± 0.21	229.33
	small		53.06 ± 0.03	53.14
sha	large	security: secure hash	82.20 ± 0.02	82.26
	small		7.63 ± 0.01	7.65
rijndael	large	security: block cipher	285.96 ± 0.30	286.92
encode	small		27.02 ± 0.11	27.25
rijndael	large	security: block cipher	264.83 ± 0.15	265.32
decode	small		24.89 ± 0.07	25.09

TABLE I: MiBench applications without cache partitioning

parts, 1/4 being associated to core 0, and 3/4 being shared by the other three cores. The latter setting allows the threads of multi-threaded best-effort applications to share more L2 cache data, thus potentially improving their performance. When the cache size available to the MiBench application is reduced to 1/4, there is a performance degradation of less than 5% on all applications except `qsort`, `susan small -c`, and `susan small -e`. Table I shows that `susan small -c` and `susan small -e` have the shortest durations of any of the MiBench applications, and thus are particularly sensitive to any overhead. Overall the results suggest that the MiBench applications mostly fit into a quarter of the L2 cache and are not memory intensive.

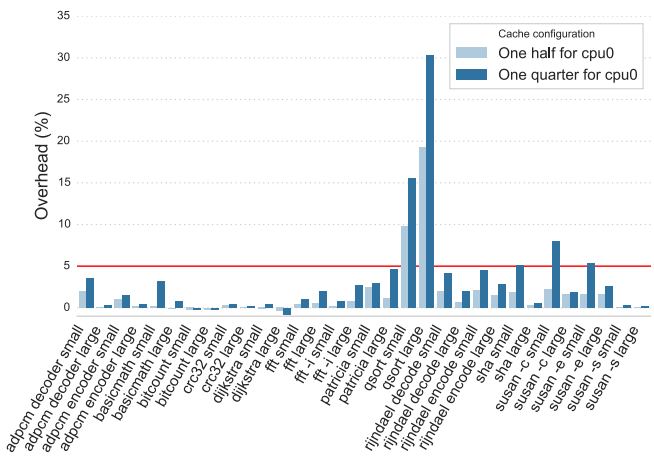


Fig. 2: Impact of partioning of MiBench performance

We then study the performance degradation that occurs when the memory bus is highly loaded. We have developed a

load program in assembly code that performs repeated memory writes, in such a way as to maximize the probability of an L2 cache miss for each write. When run alone on the SABRE Lite, the generated load is 2020 MB/s. We run the MiBench application on core 0 and one instance of the load program on each of the other three cores. All processes run under the FIFO Linux scheduling policy with maximum priority and are pinned to their core to prevent migration.

The results are shown in Figure 3. In each case, the baseline is the running time of the application when run alone and without cache partitioning. The overhead ranges up to 183%, in the case of `qsort large`. For all cases where there is an overhead, the overhead is reduced by partitioning the cache. This is because the load program evicts cache lines to force memory writes, and as all processes share the cache, the load program may remove cache lines used by the MiBench application. On the other hand, whether the application has access to half of the cache or a quarter of the cache has little impact. We conclude that cache partitioning is useful to reduce memory contention. Still, even when the cache is partitioned, there are 21 cases where a MiBench application running on a given dataset suffers from an overhead that is greater than 5%. We observe small differences on the overhead between the 1/4 configuration and the 3/4 configuration. We believe that these differences come from the cache partitioning, which can impact the behaviour of the applications.

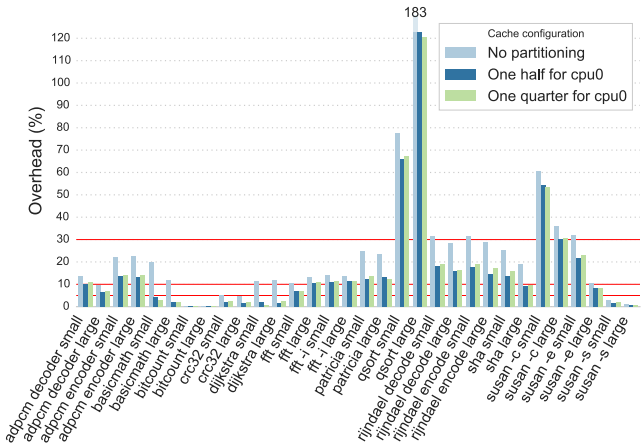


Fig. 3: Impact of load on MiBench performance depending on the partioning scheme

III. 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 of the remaining cores. The maximum number of active best-effort cores used by the system must be selected in advance. The choice could be made by the system designer or by using hardware counters to detect the number of active best-effort cores. In our work, for simplicity, we use the first approach. Our goal is to obtain as much parallelism between the best effort and real-time applications as possible, as long as the overhead that the best-effort applications introduce on the real-time application remains below a specified threshold. As is standard for real-time computing, we assume that the real-time application is known in advance, that it is periodic, and that it can be profiled to determine its worst-case execution time and worst-case resource consumption properties during each activation. On the other hand, 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 two stages. The first stage, performed offline by the system designer, begins with a manual analysis of the real-time application source code to identify *phases* in which the application has a constant memory access behavior (read-write ratio). The system designer then runs the real-time application against a wide range of constant loads, and measures the number of memory accesses and the execution time of each phase, to obtain the phase's average overall bandwidth and incurred overhead. For this analysis, we have developed a load microbenchmark that makes it possible to generate various loads, in terms of both bandwidth and the read-write access ratio. The result of this profiling stage is a table mapping bandwidths to overheads, for each phase and for each number of active best-effort cores.

In the second stage, at run time, a run-time system, integrated into the OS kernel or the hypervisor, samples the system memory bandwidth and uses the overhead table obtained from the profiling stage to accumulate a running sum that conservatively overestimates the maximum possible accumulated overhead for the current sampling period of the real-time application. If the estimated accumulated overhead is greater than the threshold specified by the system designer, minus the maximum amount of overhead that can be accumulated in a single sample, *i.e.*, the percentage of the total execution of the real-time application that is represented by the sample duration, the run-time system suspends all of the best-effort applications. Suspended applications are allowed to run again when the real-time application completes its current activation.

In the rest of this section, we describe the various analyses and mechanisms that support our approach. All further experiments are done with a partitioned cache so as to focus on contention at the level of the memory subsystem. We use the 1/4 - 3/4 L2 partitioning scheme, which provides sufficient cache space for the MiBench applications and which maximizes the space available to the best-effort applications.

A. Generating constant memory loads

In order to gather the maximum overhead on the application caused by a constant memory bandwidth generated by loads, we extend the load program used in Section II-C, which generates a worst case in terms of write accesses. We increase the range of generated memory bandwidths, by interleaving a set of write accesses with a set of read accesses, so as to induce competition between the write and read request queues. We also make it possible to add a delay between memory accesses using a wait loop.

Figure 4 shows the read and write loops of the microbenchmark. The microbenchmark performs sequential accesses to maximize the memory bandwidth. Our experimental analysis of the memory bandwidth behavior of a real-time application runs the application in parallel with one or more instances

```
stress_read_write:
    mov r11, #0
    mov r12, #0

    mov lr, r3              @ r3 has write_nb
    add lr, r4              @ r4 has read_nb
    lsl lr, #5              @ lr *= 32

outer_loop:
    mov r6, r0              @ r0 has array address
    mov r7, r1              @ r1 has array size

stress_loop:
    mov r8, r3
    mov r9, r4
    mov r10,r5              @ r5 has delay_nb

    subs r7, lr             @ branch to end if there
    ble stress_loop_end     @ is not enough room

write_loop:
    subs r8, #1
    stmgeia r6!, {r11,r12} @ write write_nb cache
    stmgeia r6!, {r11,r12} @ lines and increment the
    stmgeia r6!, {r11,r12} @ pointer stored in r6
    stmgeia r6!, {r11,r12} @ accordingly
    bgt write_loop

    mov r12, #0             @ reset r11 and r12
    mov r11, #0             @ to avoid overflows

read_loop:
    subs r9,#1
    ldmgeia r6!, {r11,r12} @ read read_nb cache lines
    ldmgeia r6!, {r11,r12} @ and increment the pointer
    ldmgeia r6!, {r11,r12} @ stored in r6 accordingly
    ldmgeia r6!, {r11,r12}
    bgt read_loop

delay_loop:
    subs r10,#1             @ increment r11 and r12
    add r11, #1             @ delay_nb times
    add r12, #1
    bgt delay_loop
    b stress_loop

stress_loop_end:
    subs r2, #1             @ r2 has the number of
    bgt outer_loop          @ stress loop iteration
```

Fig. 4: Main loops of the load microbenchmark

of this microbenchmark, each pinned to its own core. We vary the ratio of read and write accesses in the read and write loops, such that the sum is 10 (*e.g.*, 3 reads per 7 writes). A sum of 10 permits a variety of read-write ratios. To take into account both symmetric and asymmetric loads, we consider numbers of wait-loop iterations of the form $1x$ (one load), $1x, 1x$ and $1x, 2x$ (two loads), and $1x, 1x, 1x$ and $1x, 2x, 3x$ (three loads), for various values of x between 0 and 8000. We have observed that the traffic resulting from a delay of 8000 iterations has essentially no impact on the real-time application. In each run, we obtain the execution time by measuring elapsed CPU cycles, and obtain the memory bandwidth of all running instances by measuring the number of exchanged bytes using the counters of the memory controller.

To illustrate the behavior of the microbenchmark, we run it alone on core 0, for the targeted range of read-writes ratios and numbers of wait-loop iterations. The obtained bandwidths are presented in Figure 5. The highest bandwidth (2020MB/s) is obtained when only write requests are generated and there are no wait-loop iterations, *i.e.*, the behavior of the original load program. This configuration produces the highest

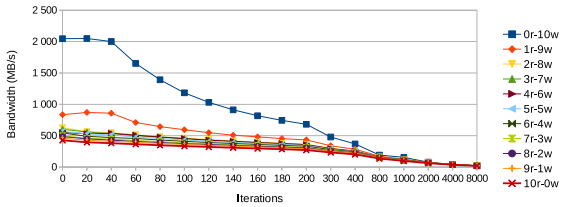


Fig. 5: Bandwidth of the microbenchmark run in isolation

bandwidth because the controller does not have to wait for write completion, in contrast to the case of reads. Mixing read and write requests furthermore introduces competition for accessing the reordering buffer, which reduces the obtained bandwidth substantially.

B. Profiling a real-time application

In order to be able to compute a conservative estimate of the overhead incurred by an application in the face of arbitrary loads, we compute offline a profile of each real-time application, reflecting the worst-case overhead that the application incurs when run in parallel with a wide range of constant loads. Our approach assumes that the real-time application traffic does not vary over a considered period of time, *i.e.*, a phase. Therefore, we first identify phases that have this property in the real-time application, using a combination of manual source code examination and memory profiling. Then, the profile for an application is constructed in two steps: data collection and overhead estimation.

In the data collection step, we first run the application in isolation a number of times, and collect the maximal observed execution time per phase p, $ExecT_p$, and the corresponding observed number of memory accesses, Acc_p. We then run the application a number of times in parallel with a range of constant loads, l, and likewise collect for each load and phase the execution time, $ExecT_{l,p}$, and the number of memory accesses, $Acc_{l,p}$, observed in each run. For each run with load l and for each phase p, we then compute the observed bandwidth as $ObsB_{l,p} = Acc_{l,p}/ExecT_{l,p}$, and the overhead as $Ovd_{l,p} = (ExecT_{l,p}/ExecT_p) - 1$. The result is a set of mappings of bandwidths $ObsB_{l,p}$ to overheads $Ovd_{l,p}$ for the given real-time application. For each mapping, we furthermore note the number of load processes, the read-write ratio and the number of wait-loop iterations used to generate the load l, and the phase p.

As the data collection step works on average bandwidths collected over entire (phase) runs, the result does not cover the complete set of memory bandwidths that can be observed during a given execution at a finer granularity. To be able to estimate the overheads incurred for arbitrary memory bandwidths, we extrapolate from the observed overhead values using least squares polynomial fitting, as implemented by the `polyfit` function of the Python `numpy` library.[3] Constructing an appropriate polynomial raises two challenges. First, least squares polynomial fitting requires choosing an appropriate degree for the polynomial, and second, it produces

[3] http://www.numpy.org/

a polynomial that is as close as possible to all of the points, while we want one that is a conservative approximation of the overhead and thus that sits just above all of the points.

To choose the degree, we take a brute force approach of trying a number of possible degrees, and determining via simulation which gives the best results. We have designed a simulator that takes as input an execution trace, consisting of periodic samples of the overall memory bandwidth of a real-time application running with some load processes, and a table mapping memory bandwidths to overheads for the given application. The simulator then estimates the total overhead on the application for the given execution trace and table. We run the simulator on execution traces for a variety of constant loads, using tables obtained from fitting polynomials with various degrees. Based on the results, we choose the degree that gives the lowest error, measured using residual sum of squares (RSS), between the estimated overhead and the actual overhead for the largest number of read-write ratios. We consider only degrees between 1 and 5, to avoid the erratic behavior that is characteristic of high degree polynomials.

Least squares polynomial fitting interpolates a polynomial that closely matches the complete set of data points. To instead construct a polynomial that tracks the greatest overheads, we interpolate a polynomial individually for each of the read-write ratios, with various delays, and then take the maximum value of any of these polynomials for each bandwidth of interest. To avoid the result being excessively influenced by polynomial values that are not near any observed data points, we include a polynomial in the maximum overhead computation only for bandwidths that exceed the maximum observed bandwidth by at most 5%. For bandwidths that are beyond this point for all read-write ratios, we use the overhead inferred for the bandwidth that is 5% beyond the overall maximum observed bandwidth. This value is used up to 3000MB/s. Beyond that value, we consider the overhead to be 0%. Indeed, we have only seen bandwidths over 3000 in the first sample of our experiments with loads; in the rest of the samples, in all executions of the MiBench applications with loads, the overall bandwidth never exceeds 2000 MB/s.

The above procedure approximates the overhead for a given observed bandwidth in the case of a constant load. To be more general, we also consider the possibility that the load changes within the sampling interval. For example, if the overall bandwidth is 400MB/s during the first quarter of a sampling interval, and is 300MB/s in the rest of the sampling interval, then the sampling process will observe an overall bandwidth of 325MB/s. To address this imprecision, we follow a *packing* strategy, that estimates the worst-case overhead that can be incurred when the load changes once within each sampling interval. For this, we consider how the overall bandwidth $ObsB$ observed within a sampling interval can be decomposed into two other bandwidths $ObsB_1$ and $ObsB_2$ and fractions of a sampling interval t_1 and t_2, where $0 \leq t_1, t_2 \leq 1$, such that $t_1 + t_2 = 1$ and $t_1 \cdot ObsB_1 + t_2 \cdot ObsB_2 = ObsB$. For $ObsB_1$ and $ObsB_2$, we consider all pairs of multiples of 20.48 MB/s, which is the granularity of the tables used by our runtime system (see Section III-C), between 0 and 3000, such that, without loss of generality, $ObsB_1 < ObsB_2$ and such that the values of t_1 and t_2 are in the required range.

For each of the pairs of possible observed bandwidths, we estimate the sample overhead as follows. Based on the polynomial analysis, each of the observed bandwidths $ObsB_1$ and $ObsB_2$ is associated with an overhead on the real-time application in the case of a constant load. To determine the effect of combining the bandwidths within a single sample, we observe that the overhead, which was calculated in terms of running times, also reflects the ratio between the amount of bandwidth, Req, required by the real-time application within a sampling period and the amount of bandwidth, $ObtB$, that the real-time application actually obtains. Req is simply the bandwidth observed when the real-time application is run alone. $ObtB$ can be computed from the overhead $OvdB$ for a given constant bandwidth $ObsB$, as $ObtB = Req/(OvdB+1)$. From the obtained bandwidth information, we then compute the overhead incurred in the context of the overall observed bandwidths $ObsB_1$ and $ObsB_2$ as $Ovd = Req/(t_1 \cdot ObtB_1 + t_2 \cdot ObtB_2)$. For the resulting overhead table, we take the maximum overhead satisfying all of the criteria.

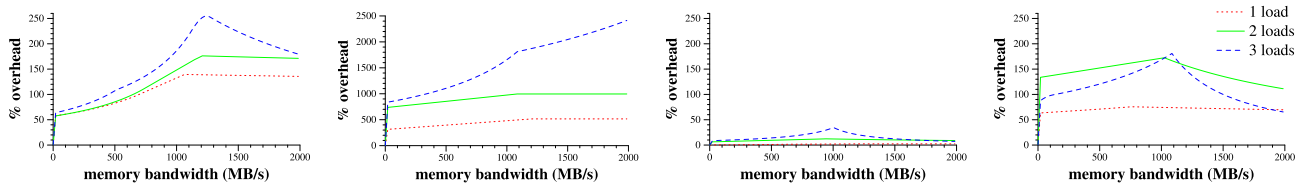
Figure 6 shows the overhead tables for the four phases of `susan small -c`. The second phase has an estimated overhead for all bandwidths up to 10 times higher than the other phases, but as shown subsequently in Figure 7i, this phase has a very short duration. In all of the phases, there is a high point around 1000 MB/s, and then the overhead drops off. The drop off represents the fact that if the application is to achieve such a high bandwidth, then it must in some way have taken over the memory bus, and is incurring delay on the best-effort tasks. Note that the average bandwidths per phase, based on which the overheads are interpolated, typically only go slightly beyond the high point, but that greater bandwidths are observed in practice at finer granularities.

C. Run-time system

The run-time system is implemented as a Linux kernel module that periodically samples the memory bandwidth. At the end of each sampling interval, the run-time system obtains the overhead associated with the bandwidth observed in the current sample and suspends all best-effort applications if the result of adding this overhead to a running sum becomes greater than the desired threshold.[4] Since we target an embedded system with predefined real-time applications, the kernel module contains all the necessary information, including the overhead tables and the maximum allowed overhead.

Sampling is triggered by a timer interrupt on one of the cores dedicated to best-effort applications. On each timer interrupt, the value of the memory subsystem counter is read and then reset to 0. The estimated overhead associated with the current sample is obtained from the appropriate overhead table according to the number of active best-effort cores. To make look up in this table efficient, we structure the table such that the required index can be obtained by a right shift of

[4]Technically, we take the threshold minus the proportion of the application execution time represented by one sample, to ensure that the worst case of no progress in the next sample will not cause the overhead on the real-time application to exceed the threshold.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Fig. 6: Overheads associated with the four phases of Susan small -c. Phase 2 has estimated overheads 10 times greater than the other phases.

the value of the memory subsystem counter, *i.e.*, the number of memory accesses in the current sample. The amount to shift is chosen as a tradeoff between the need to limit the table size, and the need to avoid rounding error. We choose a shift of 10, which has the effect of dividing the number of bytes by 1024. As motivated subsequently in Section IV-A, we use a sampling interval of 50μs. Each successive entry in the overhead table thus represents a bandwidth increment of 20.48MB/s. This approach introduces an approximation at two levels: the overhead is that of a bandwidth resulting from rounding down to the nearest multiple of 20.48 and in practice the sampling intervals are not all exactly 50μs. Nevertheless, this approach imposes little overhead on the best-effort core running the run-time system, thus maximizing the parallelism between the real-time application and the best-effort applications.

Finally, to suspend the best-effort applications when the estimated overhead exceeds the threshold, we modified the Linux kernel to add a new inter-processor interrupt (IPI). When suspension is required, the run-time system sends an IPI to the best-effort cores. Each best-effort task is preempted by the IPI handler, that then loops on a flag signaling the end of the real-time task activation. When the real-time application ends its current activation, the flag is set, the IPI handlers end and the best-effort tasks resume their executions.

At the end of each activation of the real-time application, the run-time system performs a L1 cache flush on core 0. Doing so avoids incurring cache writebacks at the beginning of the next activation, and thus ensures a constant read-write ratio in the first phase of the application, as was intended by the choice of phase boundaries. Placing the flush after the application's activation best exploits any available slack time to avoid incurring any extra load on the real-time application.

IV. Evaluation

Our goals for our approach are first to ensure that the execution time of the real-time application is not excessively impacted by the best-effort applications and second to ensure that the best-effort applications are allowed to run as long as they do not impact the real-time ones. In the rest of this section, we evaluate the efficiency of our approach on the MiBench applications and datasets that, as shown in Figure 2, can have an overhead greater than 10% without our approach and that have a running time of at most 50ms. For space reasons, we focus on the applications with short running times, since in these cases the duration of a single sample represents a high percentage of the overall run time, thus introducing the greatest risk of exceeding the overhead threshold.

Our approach has been prototyped in Linux 3.0.35, which was the latest stable version available for our platform when we started the project. The L2 cache is partitioned such that 1/4 is allocated to the real-time application and 3/4 is allocated to the best effort ones. We set the overhead threshold to 5%, as 5% is commonly viewed as a lower bound on the precision of performance measurements.

A. Overhead of run-time sampling

The sampling interval used by the run time controlling mechanism is a critical parameter for our solution. The higher the sampling frequency, the faster the system will react when there is a possibility of exceeding the acceptable overhead. However, sampling relies on interrupts which, at high frequency, risk inducing a substantial overhead on any best-effort application that runs on the core that performs the monitoring.

To evaluate the cost of sampling, we use again the MiBench applications, and measure the slowdown incurred with a 10μs or 50μs sampling period, as shown in Table II. When sampling runs on the same core as a MiBench application, in the role of a best-effort application, the overhead for a 10μs sampling period is up to 209%. Such an overhead is too penalizing for best-effort applications. With a 50μs sampling period, sampling only induces an overhead of up to 27%, which is compatible with our goal of improving parallelism between real-time and best-effort applications. On the other hand, when sampling runs on a different core from the MiBench application, now in the role of a real-time application, the overhead on the MiBench application is always below 1% and is sometimes negative, showing that sampling does not impact the performance of the real-time application.

	Real-time core		Best-effort core	
Application	10 μs	50 μs	10 μs	50 μs
adpcm -d small	0.26%	-0.50	30.59 %	3.63 %
adpcm -e small	0.06%	-0.27%	31.87 %	4.07 %
fft small	0.59%	0.69%	30.30 %	2.52%
fft -i small	0.90%	0.22%	30.51 %	2.82 %
patricia small	0.12%	0.03%	47.59 %	4.14 %
qsort large	0.44%	0.62%	34.00%	3.19 %
qsort small	-0.12%	-0.70%	32.00%	16.7 %
rijndael -d small	0.26%	0.00%	30.70 %	2.76 %
rijndael -e small	0.67%	0.27%	31.42%	3.19%
sha small	0.14%	-0.33%	29.83 %	2.83 %
susan -c large	0.15%	0.39%	31.60%	2.90%
susan -c small	-0.19%	0.85%	31.80 %	3.53 %
susan -e small	0.45%	0.62%	29.64 %	3.28 %

TABLE II: Overhead of sampling on MiBench applications

For real-time applications with short activations, however, a 50μs sampling period does represent a large portion of their execution time. For example, the maximum running time of `susan -c small` is 1.15ms, and thus each sample equals at least 4.3% of its duration, and the maximum running time of `susan -e small` is 2.13ms, and thus each sample equals at least 2.3% of its duration. Our approach stops the best-effort applications one sample before the 5% threshold may be exceeded, implying that even a moderate estimated overhead in the first sample will prevent any further parallelism in the affected activation for these applications. Still, our approach respects the desired threshold.

B. Application memory profiles

Figure 7 shows the memory profiles for the selected applications that exhibit different phases. Write accesses (blue) are shown on top of read accesses (green). Phases are typically delimited by loops in the source code. Our approach assumes that a single memory access patterns recurs throughout a phase. Still there are variations within the phases for some MiBench applications, such as `qsort`. We find that our estimated overheads are sufficient to protect the real-time application as long as the read-write ratio remains roughly constant during a phase. Note that some of the phases are very small, such as the first phase of `rijndael` and the second phase of `susan small`.

C. Efficiency for constant loads

We study the impact of using our approach when running the selected MiBench applications, while running loads exhibiting all the read-write ratios considered when creating the overhead tables. In total, for each application, there are 18 different load values, with 11 different read-write ratios and 5 configurations of loads on best-effort cores, leading to 990 experiments. Each experiment involves 30 runs, of which we drop the first 10 results.

We calculate the overhead on a MiBench application by measuring its running time at the end of the activation with the running time of the application alone with the L2 cache partitioned. Figure 8 shows the overhead distribution for each application in the form of a violin plot. The width of a violin at a particular overhead value indicates the number of runs of the application that exhibit that overhead. The maximum overhead is reached by `sha small` with an overhead of 5.10%. All other applications have an overhead under 5%. The large variations in the overheads of `susan small -c` and `susan small -e` are due to their short execution times.

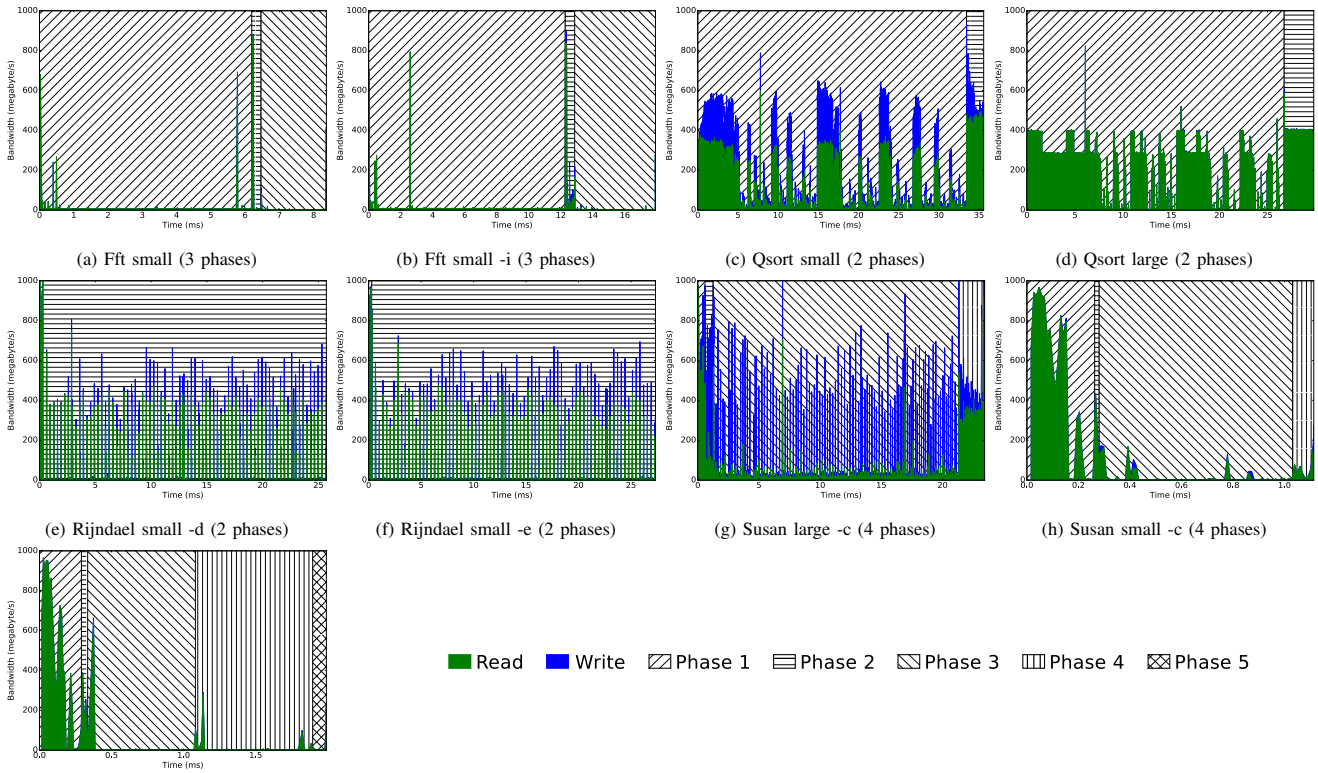
We next study the degree of parallelism we can obtain for best-effort applications, with various loads. Parallelism is measured as the percentage of time during which best-effort applications are executed in parallel with the real-time application. Figure 9 shows the worst-case degree of parallelism for the applications, for all of the studied multiples of wait-loop iterations, among the 20 considered runs in each case. The degree of parallelism achieved for the various applications differs greatly, thus showing the need for our application-specific profiles. For 7 of the 13 applications, all configurations achieve at least 70% parallelism when the loads become dominated by non memory related computations: both `adpcm`s, both `fft`s, both `rijndael`s, and `sha small`. For the remaining applications, except `qsort large` and `susan small -c`, the degree of parallelism depends highly on the number of active best-effort cores. Finally, `qsort large` and `susan small -c` start with a long memory intensive phase (see Figure 7d), during which the overhead threshold is reached.

V. 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.

a) WCET approaches: Pellizzoni et al. [30] have developed a method for calculating the WCET in a multicore context. Bin et al. [9], [10] have developed a methodology to compute the WCET of avionic applications sharing the same multicore system. Jean et al. [19] 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.

b) Software approaches: Caccamo et al. [12] and Yun et al. [36] have developed mechanisms for sharing the memory bandwidth in the context of a multi-core architecture with an accurate knowledge of the memory consumed by each core. They implemented their solution on a quad-core platform with two last level caches, each cache being shared by two cores (Cores 0 and 1 share an LLC, and Cores 2 and 3 share another LLC). To get accurate information on each core's memory consumption, they monitor the LLC miss rate and they disable core 0 and core 2 to eliminate cache sharing effects. Our solution targets hardware platforms that do not provide per-core memory bandwidth information. Kritikakou et al. [20] have developed a solution to manage contention on shared resources. They introduce a set of observation points in the binaries of the real time applications. At each observation point, the run-time system determines if best-effort tasks must be suspended. Their approach is independent of the resources involved. Consequently, their solution can be applied to any contention issue on shared resources and does not rely on the hardware to obtain measurements. They then extend their solution to schedule multiple real-time tasks [21]. Our solution differs from these approaches because it does not require any modification of the application binary. Indeed, such a modification could add an additional overhead on the real-time application. Muralidhara et al. [25] use an 8-core hardware platform connected to the main memory by several channels, each independently controlling a portion of the physical memory. They group applications that do not interfere on the same channel. Liu et al. [23] combine cache coloring with the partitioning by channel in the Linux



(a) Fft small (3 phases)

(b) Fft small -i (3 phases)

(c) Qsort small (2 phases)

(d) Qsort large (2 phases)

(e) Rijndael small -d (2 phases)

(f) Rijndael small -e (2 phases)

(g) Susan large -c (4 phases)

(h) Susan small -c (4 phases)

(i) Susan small -e (5 phases)

Fig. 7: Memory profiles of selected MiBench applications

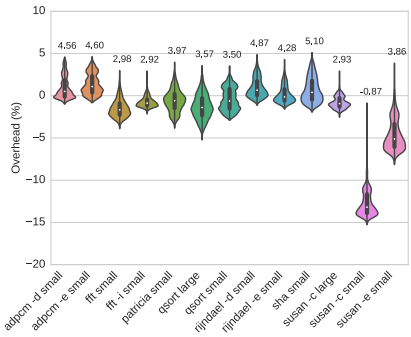


Fig. 8: Overhead for MiBench applications with constant loads

kernel in order to partition the cache and the memory. Seo et al. [32] determine the memory contention level using hardware counters to calculate the number of retries necessary for a memory request to be accepted by the memory controller. They use this information to improve, by scheduling, the whole system performance,

c) Hardware approaches: Ungerer et al. [35] have designed a multicore architecture for applications having varying degrees of criticality that permits a safe computation of the WCET. Lickly et al. [22] propose a new multithreaded architecture for executing hard real-time tasks that provides precise and predictable timings. Moscibroda et al. [24] propose a new memory controller designed to provide memory access fairness across different consumers. Shah et al. [33] present a new scheduling policy for the bus arbiter to respect real-time constraints and have good performance. All of these approaches involve hardware that does not currently exist, while our approach targets COTS machines.

d) Mixed approaches: Pellizzoni et al. [29] introduce hardware buffers that make it possible to schedule accesses to shared resources in such a way as to prevent two consumers/-producers from simultaneously accessing the same resources. Applications must be structured into phases that have particular memory-access properties and are thus able to take advantage of the resource guarantees provided by the scheduler. Boniol et al. [11] propose an algorithm for restructuring applications automatically to fit the requirements of such a system. Finally, Rafique et al. [31] designed a fair bandwidth sharing memory controller, 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.

VI. Conclusion and Future Work

In this paper, we have presented an approach permitting to mix real-time and best-effort applications on a single small COTS 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 the real-time application, and a run-time system that controls the scheduling of the best-effort applications. No modifications to the best-effort applications are required. 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. We have investigated the feasibility of the approach on MiBench applications, and found the limits both in terms of sampling and phase precision. We have studied the behavior

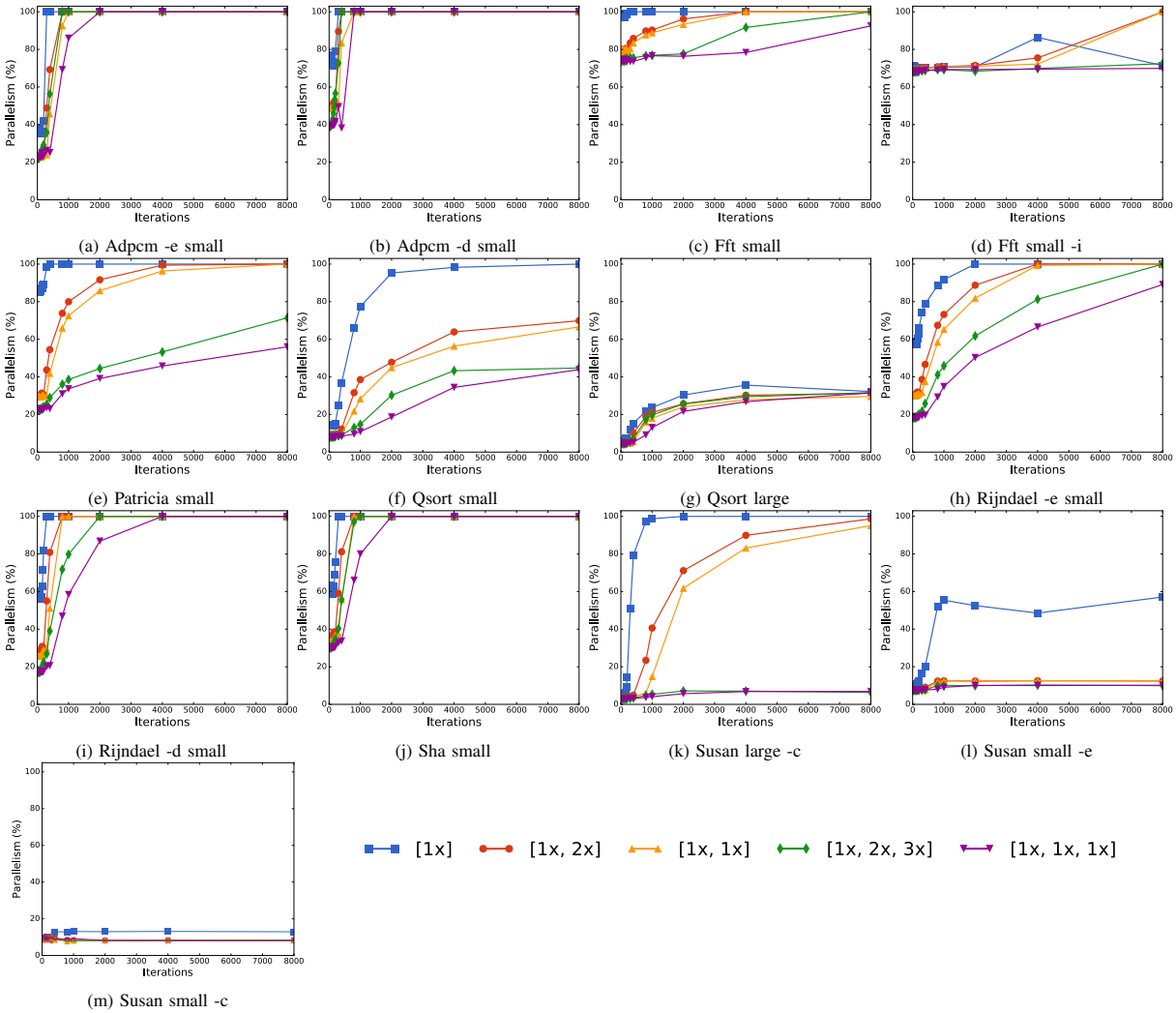


Fig. 9: Parallelism for MiBench applications with constant loads. The different curves represent different multiples of the wait-loop iterations between sequences of accesses.

of the 13 MiBench applications and datasets that incur an overhead of over 10% without our mechanism and are of short duration. Of these, 12 always incur an overhead of less than 5% with our mechanism, regardless of the load, and one incurs an overhead of 5.10%. Furthermore, 7 achieve 70% of parallelism for low-bandwidth loads, regardless of the number of active best-effort cores.

Currently, 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, alternate approaches could be considered that reduce the demand of the best effort applications incrementally. One approach would be to slow down the clock speed of the best effort cores, when the hardware permits this operation (the SABRE Lite hardware does not). Another approach would be to suspend only the best-effort processes running on the core 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. A third approach would be to exploit the different bandwidth requirements of the different phases of the real-time application. As phases with low bandwidth requirements incur little delay, regardless of the overall memory traffic, it could be possible to restart the best-effort applications when the real-time application enters such a phase. All of these approaches would require degrading the execution of the best-effort applications well before reaching the overhead threshold, to ensure that this threshold continues to be respected.

A current limitation of our work is that it involves manual analysis of the source code to determine the phases. In the future, we plan to automate this process by combining automated source code analysis to identify repetitive patterns in the source code and filtering of the memory profiles to identify memory phases. A second limitation is that the approach does not adapt to cases where the memory demands varies with the input data. To address this issue, we could compute memory profiles for a variety of inputs, and then merge the worst case for each observed bandwidth and load into a single table.

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

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

[10] J. Bin, S. Girbal, D. G. Perez, and A. Merigot. 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] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti. Deterministic execution model on COTS hardware. In *International Conference on Architecture of Computing Systems (ARCS)*, pages 98–110. Springer-Verlag, 2012.

[12] M. Caccamo, R. Pellizzoni, L. Sha, G. Yao, and H. Yun. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, pages 55–64, 2013.

[13] C. Ficek, N. Feiertag, K. Richter, and M. Jersak. Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems. In *Embedded Real Time Software and Systems (ERTS)*, Feb. 2012.

[14] S. Fisher. Certifying applications in a multi-core environment: The worlds first multi-core certification to sil 4. *SYSGO AG*, 2014.

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

[16] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, 2009.

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

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

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

[20] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 119–128. IEEE, 2014.

[21] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 139. ACM, 2014.

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

[23] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *21st PACT*, pages 367–376, 2012.

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

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

[26] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, pages 132–143, May 2012.

[27] J. Nowotsch and M. Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 151–160, New York, NY, USA, 2013. ACM.

[28] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956, 2009.

[29] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *RTAS*, pages 269–279, Apr. 2011.

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

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

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

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

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

[35] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, Sept. 2010.

[36] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th ECRTS*, pages 299–308, July 2012.