On the Influence of Shared Memory Contention in Real-time Multicore Applications

Giovani Gracioli
Hardware Software Integration Lab (LISHA)
Center for Mobility Engineering (CEM)
Federal University of Santa Catarina (UFSC)
Joinville, Santa Catarina, Brazil
Email: giovani@lisha.ufsc.br

Antônio Augusto Fröhlich
Hardware Software Integration Lab (LISHA)
Computer Science Department (INE)
Federal University of Santa Catarina (UFSC)
Florianópolis, Santa Catarina, Brazil
Email: guto@lisha.ufsc.br

***Abstract*—The continuous evolution of processor technology has allowed the utilization of multicore architectures in the embedded system domain. A major part of embedded systems, however, are inherently real-time (soft and hard) and the use of multicores in this domain is not straightforward due to their unpredictability in bounding worst-case execution scenarios. One of the main factors for unpredictability is the coherence through memory hierarchy. This paper characterizes the influence of contention for shared data memory in the context of embedded real-time applications. By using a benchmark, we have measured the impact of excessive shared memory invalidations on five processors with three different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory organizations (UMA and ccNUMA). Results have shown that the execution time of an application is affected by the contention for shared memory (up to 3.8 times slower). We also provide an analysis on Hardware Performance Counters (HPCs) and propose to use them in order to monitor and detect excessive memory invalidations at run-time.**

I. Introduction

Several embedded real-time applications are implemented in a dedicated hardware logic (i.e., Application-Specific Integrated Circuits – *ASICs*) to obtain maximum performance and fulfill all the application's requirements (processing, real-time deadlines, etc). For instance, digital signal processing algorithms and baseband processing in wireless communication, should process a big amount of data under real-time conditions. Nevertheless, as they are usually implemented in a dedicated hardware, these applications present restrictions in terms of developing support (e.g., bug fixes, updating, and maintainability).

The continuous evolution of processor technology, however, has enabled multicore (e.g. *Symmetric Multiprocessing - SMP*) architectures to be also used in the embedded real-time system domain [1]. In this context, an application is implemented on top of a Real-Time Operating System (RTOS), composed of several real-time cooperating threads (threads that share data).

In this scenario, due to multicore processor organization, some important characteristics must be considered, specifically, the memory hierarchy [2], [3]. The memory hierarchy holds an important role, because it affects the estimation of the Worst-Case Execution Time (WCET), which is extremely important in the sense of guaranteeing that all threads will meet their deadlines through the design phase evaluation (schedulability analysis) [4], [5], [6].

Several works have been proposed to deal with memory organization in multicore architectures and provide real-time guarantees [1], [7], [8], but they only consider scenarios where threads are independent, that is, there is not data sharing. In situations where threads share data in a cooperating fashion, a partitioning/locking mechanism does not avoid the contention for shared data. For instance, consider a scenario composed of "n" threads sharing data, running on "m" different cores in a pipeline order (thread 2 after thread 1, thread 3 after thread 2, and so on). Thread 1 executes and writes in the shared data location. When the thread 2 accesses the shared data, it gets an invalid access and must ask (snoop request) for the most recent copy of the data or recover it from a higher memory level. The task of performing a snoop request is done automatically by the memory controller hardware, which increases the threads' execution time, even without their knowledge. The time to complete a snoop request is considerably slow (comparable to access the off-chip RAM) [9], which can lead to an unexpected increase of the thread's execution time and deadline losses.

In this paper, we bring the problem of contention for shared memory to the embedded real-time system domain by measuring its influence on five different processors with three different cache-coherence protocols (MESI, MOESI, and MESIF)[1] and two memory organizations (UMA and ccNUMA). We use a benchmark composed of two versions of the same application (sequential and parallel). If the sequential version is schedulable (proved by a schedulability analysis), the parallel version should be schedulable as well (it executes the same code but in parallel). We demonstrate in our experiments that due to the current multicore memory organization, the parallel version has its execution time affected (up to 3.8 times slower), which can lead to deadline losses. In order to monitor and detect when data sharing occurs, we also provide an analysis of Hardware Performance Counters (HPCs) in one multicore processor. HPCs are special registers available in the most modern microprocessors through the hardware Performance Monitoring Unit (PMU). They offer support to counting or sampling several micro-architectural events at run-time [13] and can be used to capture hardware events that reflect the software behavior.

[1]Further information about cache-coherence protocols can be found in [10], [11], [12].

In summary, in this paper, we make the following contributions:

(I) We motivate the problem by measuring the influence of contention for shared memory in the context of hard real-time applications; **(II)** We evaluate the problem on five different multicore processors, with three different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory organizations (UMA and ccNUMA) by using a benchmark composed of a sequential and parallel versions of the same application; **(III)** We evaluate HPCs in one of the five processors. HPCs together with the OS scheduler and memory partitioning are good alternatives to decrease the contention for shared data memory and provide predictability and performance gains to real-time applications. We present an analysis of hardware events that can be used to this purpose.

The remainder of this paper is organized as follows. Section II evaluates and discusses the problem. Section III analyzes some hardware events. Section IV provides an overview of related works, and Section V concludes the paper.

II. Problem Evaluation and Discussion

The problem we are addressing in this paper raises from the memory hierarchies present in the today's SMP architectures and their memory coherence protocols. When a core writes into a data that other cores have cached, the cache-coherence protocol invalidates all copies, causing an implicit delay in the application's execution time. At the same way, when a core reads a shared data that was just written by another core, the cache-coherence protocol does not return the data until it finds the cache that has the data, annotate that cache line to indicate that there is shared data, and recover the data to the reading core. These operations are performed automatically by the hardware and take hundreds of cycles (about the same time as accessing the off-chip RAM), increasing the application's execution time [9]. Two kinds of scaling problem occur due to shared memory contention [9]: access serialization to the same cache line done by the cache coherence protocol and saturation into the inter-core interconnection, preventing parallel processing gains. Reducing the effects of cache line contention can significantly improve the application's overall performance and avoid deadline misses

In order to evaluate the influence of contention for shared data memory in the execution time of an application, we have designed a benchmark to generate memory invalidatons composed of two versions of a pipeline application and a best-case application for comparing purposes (Figure 1):

(I) **Sequential**: in this version, two threads are executed in a sequential order. There are no memory conflicts (Figure 1(a)). The objective of this version is to simulate an algorithm that does not have shared memory invalidations.

(II) **Parallel**: two threads run at the same time and share data (Figure 1(b)). The objective of this version is to evaluate the performance of the previous version when it is implemented in a multicore architecture. Both threads (1 and 2) from sequential and parallel versions have the same operations and memory accesses (see Figure 2). Common sense dictates that this version should run about two times faster than the sequential one. Consequently, if the sequential version is

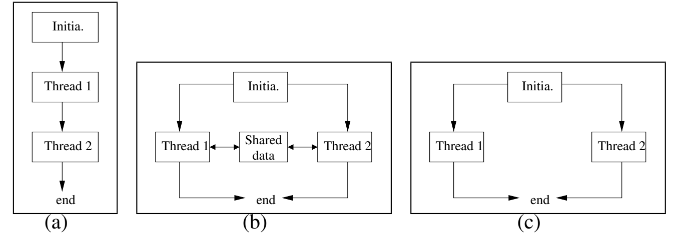


Figure 1. Benchmark applications: (a) sequential (b) parallel (c) best-case.

schedulable (proved by a schedulability analysis), the parallel version should be schedulable as well. We do not use any kind of synchronization (i.e., semaphores, mutexes, or condition variables) to ensure the data consistency, because we are only interested in measuring the shared data contention overhead.

(III) **Best-case application**: two threads run at the same time, but do not share data (Figure 1(c)). This application should run about 2 times faster than the sequential one on a multicore processor. The objective is to have a best-case scenario comparable to the sequential and parallel versions.

All three applications have two two-dimensional arrays of varying size ROWS x COLS and a loop of 10000 iterations, in which math operations are executed. The shared data in the parallel version is accessed by reading and writing in both arrays and in the same positions, thus we are sure that both threads are accessing the corresponding cache line. Figure 2 shows part of the source code from one of the functions implemented in the sequential and parallel versions. At initialization phase, the arrays are started with zero and the threads are created (parallel and best-case applications). The benchmark was implemented as a Linux application, in C++ (GNU g++ 4 compiler without using optimization parameters), and using the pthread library version 2.11.12 for the parallel and best-case versions. Each thread in these versions was assigned to a different core by using the *pthread_setaffinity_np* function. We ran each application version 100 times and then we extracted the sampled worst-case execution time[2]. The time was measured by the Linux *time* tool. The kernel version used in all test was the 2.6.32. The arrays' size was set to 64x64 (32 KB), 128x128 (128 KB), 256x256 (512 KB), 512x512 (2 MB), 1024x1024 (8 MB), and 2048x2048 (32 MB), considering an integer as 4 bytes. Table I shows the configuration of 3 processors used during the evaluations.

Figure 3 presents the WCET (in logarithm scale) for each application version on the Intel Core 2 Quad Q9550 processor. As expected, the best-case application was about 2 times faster than the sequential one. However, we note that, independently of the arrays' size, the parallel version was always slower than the sequential version (up to 1.31 time). The relative standard deviation for the sequential, parallel, and best-case applications was 0.78%, 0.71%, and 0.36% of the total execution time, respectively. In addition, we repeated the evaluation using an optimized version of the Linux kernel [9]. Basically, the Linux was modified to avoid locks and atomic instructions by reengineering data structures and unnecessary sharing. All applications presented similar performance, and the parallel

[2]From now on, whenever we refer to the WCET we are actually referring to the sampled WCET.

```
1   register unsigned int sum0;
2   register unsigned int sum1;
3
4   for(unsigned int i = 0;  i  <= REP; i++) {
5       for(unsigned int j = 0;  j  < ROWS; j++) {
6               sum0 = 0;
7               sum1 = 0;
8           for(unsigned int k = 0; k  < COLS; k++) {
9               sum0 += array0[j][k];
10              sum1 += array1[j][k];
11          }
12          for(unsigned int k = 0; k  < COLS; k++) {
13              array0[j ][0]  = sum0;
14              array1[j ][0]  = sum1;
15          }
16      }
17  }
```

Figure 2. Part of the source code of sequential and parallel applications.

Table I. AMD OPTERON 280, INTEL I5-650, AND INTEL Q9550 CONFIGURATIONS.

Feature/ Processor	Opteron 280	Intel i5-650	Intel Q9550
Frequency	2.4 GHz	3.2 GHz	2.83 GHz
Physical dies	2	1	1
Cores per die	2	2	4
SMT	-	2	-
Bus Speed	HyperTransport 1.0 GHz	QPI 1.3 GHz	FSB 1.3 GHz
L1 private data cache size	128 KB 2-way associative	32 KB	64 KB
L2 cache size	1 MB 16-way associative	256 KB per core (private)	12 MB
L3 shared	-	4 MB	-
Memory architecture	ccNUMA	ccNUMA	UMA
Coherence protocol	MOESI	MESIF	MESI

version was always slower than the sequential one. This performance degradation is caused especially by data sharing (lines 9, 10, 13, and 14 of Figure 2), which causes excessive invalidations in the same cache line due to the MESI cache-coherence protocol and bus snooping. Moreover, considering the embedded real-time domain, this performance degradation may imply in deadline losses.

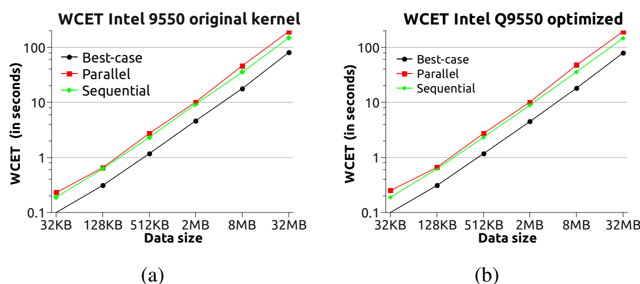


Figure 3. Benchmark evaluation on Intel dual-core Q9550 processor: (a) Original Linux kernel (b) Optimized Linux kernel.

We also ran the three applications in other two SMP processor, Intel Xeon 5030 and a PowerPC-based dual-core on the Cell architecture (MESI/UMA)[3]. We obtained even worse execution times: for the PowerPC processor the parallel version was up to 2.62 times slower than the sequential one, while for the Intel Xeon 5030 the parallel was up to 3.87 times slower when the data size was 32 MB. For the Intel Xeon processor, the performance degradation increased due to bus saturation (FSB) caused by a greater frequency of data transfer between the L2 cache and the main memory and also by the cache-coherence protocol that writes the data back to memory.

Our next evaluation was carried out in the AMD Opteron 280 processor (MOESI/ccNUMA - see Table I). We evaluated two execution scenarios:

(I) **Scenario 1**: threads running on cores in the same die (CPUS 0 and 2). (II) **Scenario 2**: Linux Completely Fair Scheduler (CFS) responsible for allocating a thread to a core. CFS supports the creation of groups of tasks, round-robin, FIFO, and real-time scheduling policies. Jones presents a complete description about the CFS implementation [14].

The objective of this experiment was to analyze the relation between the physical core location and the shared memory coherence by measuring the applications WCET. Figure 4(a) shows the measured WCET for scenario 1. The parallel version was always slower than the sequential one (up to 2.82 times using arrays' size of 32 KB). We can also note a decreasing in the WCET difference as the arrays' size increases. The relative standard deviation in this scenario for the sequential, parallel, and best-case was 2.86%, 9.88%, and 4.29%. Figure 4(b) represents the measured WCET for scenario 2. The applications performance was similar to the scenario 1. The relative standard deviation for the sequential, parallel, and best-case was 4.68%, 3.91%, and 4.99%. We also ran the threads on cores located in different dies, and the sequential application was slightly faster than the parallel one for data size of 8 and 32 MB. As expected from a ccNUMA processor, there is a variation in the WCET when executing applications in cores physically located on different dies.

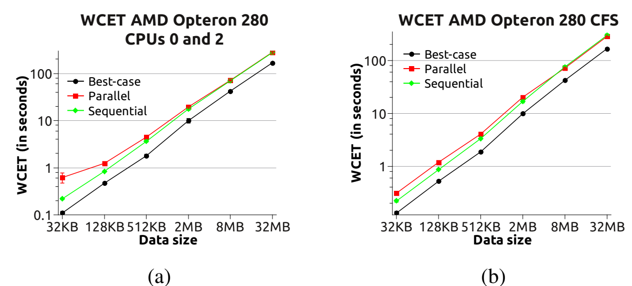


Figure 4. Benchmark evaluation on AMD Opteron 280 processor assigning threads to specific cores: (a) CPUS 0 and 2 (b) Linux CFS scheduler.

Our next evaluation was carried out in the Intel i5-650 processor (MESIF/ccNUMA[4] - see Table I - without using hyperthreding). We ran each application in two different scenarios:

(I) **Scenario 1**: threads running on two different cores

[3]Note for the reviewers: Due to space constraints, we suppress the graphs of these experiments.

[4]Remember here that the non-uniform memory in the Intel i5 refers to cache instead of the main memory as in the computer architecture point of view.

(CPUS 0 and 1). (II) **Scenario 2**: Linux CFS responsible for allocating a thread to a core.

Figure 5 shows the WCET of each application in the two described scenarios. We can observe that the difference between the parallel and best-case WCET was almost the same in the two scenarios. In general, the parallel version was faster than the sequential one. Only in scenario 1 with data size of 32 MB, the parallel one was slower than the sequential. The relative standard deviation of sequential, parallel, and best-case applications in scenario 1 was 0.95%, 3.30%, and 1.53%, and in scenario 2 was 1.35%, 3.07%, and 2.92% of the total execution time. The performance increasing can be explained by the processor organization, composed of Intel's QuickPath Interconnect (QPI) and MESIF protocol. Compared to the FSB, QPI provides higher bandwidth and lower latency for NUMA-based architectures. Each processor has an integrated memory controller and features a point-to-point link (all processors are connected), allowing parallel data transfer and shortest snoop request completion [12].

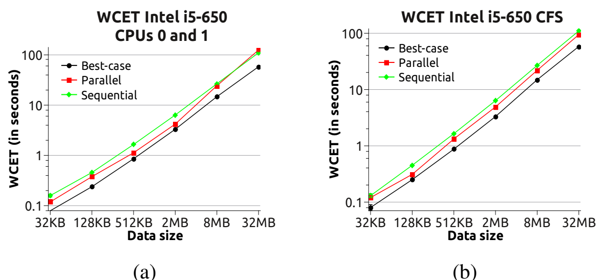


Figure 5. Benchmark evaluation on Intel i5-650 processor assigning threads to specific cores: (a) CPUS 0 and 1 (b) Linux CFS scheduler.

Discussion. During our evaluations we observed a set of interesting facts regarding the evaluated problem:

SMP architectures: We executed our benchmark on five different processors that implement MESI, MESIF, and MOESI, as cache-coherence protocol, and two different memory organizations (ccNUMA and UMA). The five processors have proved that the impact of memory coherence is not negligible and should be mainly considered for real-time and processing intensive applications. The ccNUMA processors have suffered less impact considering the contention for shared data due to their bus, cache-coherence protocol, and memory organizations. However, because their different memory access times and unpredictability, ccNUMA architectures are not the most adequate for real-time applications. A execution time degradation up to 3.8 times for the parallel application compared to the sequential one was obtained using the Intel Xeon 5030 processor, which can certainly violate real-time guarantees if not correctly handled.

Memory partitioning: In order to reduce cache line invalidations and the interference among threads that do not share resources, methods such as memory partitioning [2] could be used. However, in a cooperating real-time application, such as those found in digital signal processing area, where threads share data, memory partitioning does not solve the problem, because threads will access the same data location on the memory hierarchy. Memory partitioning can be used together with other techniques, such as scheduling and cache locking, in order to decrease the contention for shared resources between several applications and application's threads.

Operating systems: We used the Linux operating system. In general, OSs do not have any support for handling the contention for shared data. Moreover, the state-of-art real-time multicore scheduling algorithms do not consider the problem. Scheduling is a good alternative to solve the problem, because it is totally transparent to applications, there is no need to change APIs nor libraries, and can be easily integrated in virtually any RTOS. HPCs can be used to provide to the OS scheduler information about data sharing. The next section provides an analysis of some hardware events that can be used to this purpose.

III. HPCs Analysis

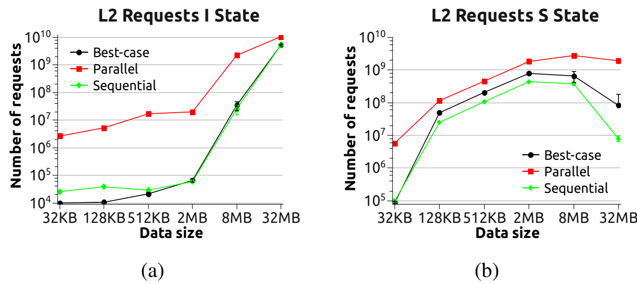
HPCs are a good alternative to monitor shared memory invalidations and provide to the OS a correct view of the application behavior. The current related work neither take into consideration the problem nor provide an analysis of available HPCs. Hence, our objective is to analyze the main hardware events that could be used to monitor shared memory invalidations at run-time. Observing the events available in the Intel Core 2 Q9550 processor, we identified that the following events are interesting to our purpose, because they provide information about cache lines and bus snooping [15]:

(I) **L2 cache requests:** we count all completed L2 cache requests. This event can count occurrences of accesses to cache lines at different MESI states. Since data sharing and invalidations are generated when lines are in S or I states, we monitor all L2 cache requests for these two states; (II) **L1 data cache snooped by other core:** we count the number of times the L1 data cache is snooped for a cache line that is needed by the other core in the same processor. The cache line is either missing in the data cache of the other core or is available for reading only and the other core wishes to write the cache line. We monitor all snoops for cache lines in S and I states; (III) **External snoops:** we count the snoop responses to bus transactions. We monitor all snoop responses to bus transactions that found a cache line in the modified state (HITM).

We use the *perf* Linux tool to read the HPCs and the *libpfm4* to list all available HPCs and use them as input to *perf*. We ran each application (Figure 1) and each event configuration for 100 times and extracted the average value. All tests were executed in the Intel Q9550 processor using Linux 2.6.32.

Figure 6 shows the measured values for the L2 cache request events (in logarithm scale). In Figure 6(a), we can note an exponential increase in the events for the sequential and best-case application after 2 MB. When the data size is 32 MB, all the three application have similar behavior, because the data size is greater than the shared L2 cache size. Consequently, there is more data being replaced from/to the cache to/from the main memory and thus more invalid cache lines. Hence, this event is not a good alternative to monitor shared memory invalidation for data sizes greater than 2 MB.

Figure 6(b) shows the L2 cache requests for lines in the shared state. The difference among the three applications was

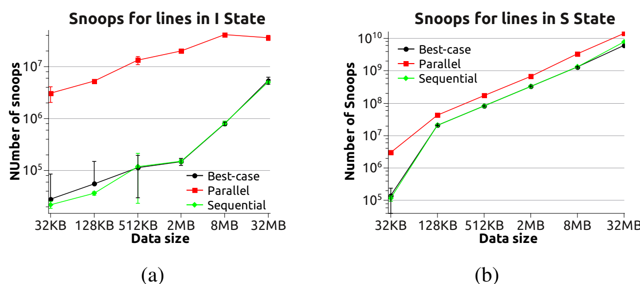


(a) (b)

Figure 6. Benchmark HPCs evaluation on Intel Q9550 processor: L2 requests for (a) I state and (b) S state.

not too significant. The explanation for shared data in the sequential and parallel applications is the natural implementation of a multicore kernel, where mutual exclusion of shared data structures is guaranteed by shared spin locks. When the data size is greater than the half size of the L2 cache, we observe a reduction in the cache lines in S state and an increasing of cache lines in I state. As both threads in the parallel version are always writing into the shared data, this event has shown not to be a good alternative to measure memory invalidations.

The next evaluation measured the number of times the L1 data cache is snooped by another core. Figure 7 shows the obtained values of snoops for cache lines in I state (Figure 7(a)) and S state (Figure 7(b)). There is a greater difference in the number of snoops between the parallel and the other two applications considering snoops for cache lines in I state, as also observed in Figure 6. The parallel version has obtained up to 2 order of magnitude more snoops than the sequential and best-case applications. As the data size increase, there is more false sharing occurring, more data being replaced in the cache, and consequently more snoops. False sharing occurs when threads running on different cores access data addresses that are in the same cache line. As conclusion for this event, the number of snoops for cache lines in I state is a good option for monitoring shared data invalidations.



(a) (b)

Figure 7. Benchmark HPCs evaluation on Intel Q9550 processor: snoop requests for I and S states.

Finally, the next experiment evaluates the number of snoop responses to bus transactions. Bus transactions are generated due to the cache coherence. For example, when a processor writes into a shared data (cache line in the S state), a bus transaction is generated to invalidate other data copies. The writing operation only ends when the snoop response for the bus transaction arrives. Figure 8 shows the number of snoop responses to bus transactions that reach a cache line in the M state, that is, a core has written to a data and another core wants to read or write the same data.

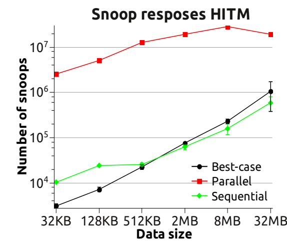


Figure 8. Benchmark HPCs evaluation on Intel Q9550 processor: external snoops HITM.

We can note that the curves in Figure 8 are similar to the curves in Figure 7(a). This was expected, because both threads have the same writing and reading operations. Figure 7(a) has shown the number of times the L1 data cache of a core is snooped and Figure 8 shows the number of snoop responses of this same core for a snoop request from the other core. Thus, both events are complementary and good options to monitor shared memory invalidations.

For instance, the scheduler could monitor the number of external snoops HITM and snoop requests for I state during a quantum in all cores. If the read value is greater than a threshold (5000 for example – sequential and best-case applications have obtained up to 100 requests in a quantum of 10 ms) in two or more cores, the scheduler can detect memory invalidation activities in those threads running on different cores, and take a decision, as to schedule or not a thread or to stop it for a while in order to decrease the overhead associated to the memory coherency.

Improving the PMU support. Initially, hardware designers have added PMU capabilities into processors to collect information for their own use [16]. However, PMUs features have become useful for other performance measurements, such as energy management and scheduling decision. In consequence, the hardware designers are now adding more functionalities to PMUs, which can certainly help the software developer even more. Features like data address sampling, monitoring address space intervals, processing cycles spent in specific events, and interrupt generation when a pre-defined value is reached could be very useful for detecting memory coherency activities. We believe that the analysis made by this paper can help hardware designers to improve PMU features in multicore processors and their use in real-time applications.

IV. Related Work

Shared cache partitioning is the most common method used to address contention and provide real-time guarantees to multicore applications. Partitioning is used to isolate applications workloads that interfere each other and thus increasing predictability [17], [2]. Another important research topic related to memory hierarchy in multicore architectures is the timing and delay analyses. A framework for estimating the worst-case response time of tasks sharing an instruction cache was developed by Suhendra et al. [18]. However, this work assumes that data memory references (i.e., data cache) do not interfere in the tasks' execution time. We have shown in this paper that the data memory hierarchy poses an important influence in the application's execution time. *Schedule-Sensitive* and *Synthetic*

are two methods to measure cache-related preemption and migration delays (CPMD) [19]. The evaluation shows that the CPMD in a system under load is only predictable for working set sizes that do not trash the L2 cache [19].

Considering HPCs as an alternative to easily detect sharing pattern among threads and help scheduling decisions, Bellosa and Steckermeier were the first to suggest using HPCs to dynamically co-locate threads onto the same processor [20]. Tam et al. use HPCs to monitor the addresses of cache lines that are invalidated due to cache-coherence activities [16]. West et al. [21] propose an online technique based on a statistical model to estimate per-thread cache occupancies online through the use of HPCs. However, data sharing is not considered by the authors. Another work to address shared resource contention via scheduling was proposed by Zhuravlev [3]. The paper identifies the main problems that can cause contention in shared multicore processors (e.g., memory controller contention, memory bus contention, prefetching hardware, and cache space contention). The authors propose two scheduling algorithms (*Distributed Intensity* - DI, and *Distributed Intensity Online* - DIO). DI uses a threads' memory pattern classification as input, and distributes threads across caches such that the miss rates are distributed as evenly as possible. DIO uses the same idea, but it reads the cache misses online, through HPCs. Calandrino and Anderson have proposed a cache-aware scheduling algorithm [7]. The algorithm uses HPCs to estimate the working set of each thread and to schedule them in order to avoid cache thrashing and provide real-time guarantees for soft real-time applications. However, to correctly estimate the working set, the threads must not share data and and the data size of the running threads must be less than the cache size. FP_{CA} is a cache-aware scheduling algorithm that divides the shared cache space into partitions [22]. Tasks are scheduled in a way that at any time, any two running tasks' cache spaces (e.g., a set of partitions) are non-overlapped. A task can execute only if it gets an idle core and enough cache partitions. In general, all the above related work do not consider a multicore system where threads share data. We demonstrated through our benchmark evaluation that the contention for shared memory data can influence the application's execution time and lead to performance degradation and deadline losses.

V. Conclusion

This paper evaluated the influence of contention for shared data memory in the context of real-time multicore applications. We have designed a benchmark in order to measure how the application's execution time is affected by the memory coherency hardware mechanism (e.g., snooping). The benchmark, composed of two versions of an application (sequential and parallel), was evaluated in 5 different processors with 3 different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory architectures (UMA and ccNUMA). The results have shown that real-time applications mainly on top of UMA processors must consider the coherence between the memory hierarchy. A execution time degradation up to 3.87 times in the parallel version was obtained only because the contention for shared data memory. As a step forward to mitigate the problem, we analyze some hardware events that could be used by the OS scheduler at run-time to obtain a precise view of the data sharing activities between cores. As future work, we plan to integrate the studied HPCs into a real-time multicore scheduler.

References

[1] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *RTSS '06*, pages 101–110. IEEE, 2006.

[2] S.P. Muralidhara, M. Kandemir, and P. Raghavan. Intra-application cache partitioning. In *IPDPS 10'*, pages 1–12, 2010.

[3] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10*, pages 129–142, 2010.

[4] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE '05*, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08*, pages 300–303, New York, NY, USA, 2008. ACM.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7:36:1–36:53, May 2008.

[7] John M. Calandrino and James H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *EUROMICRO'09*, pages 194–204, Washington, DC, USA, 2009. IEEE.

[8] R.I. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. techreport YCS-2009-443, University of York, 2009.

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI 2010*.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th Ed., September 2006.

[11] AMD. Amd64 architecture programmers manual volume 2: System programming, June 2010. Publication # 24593. revision: 3.17.

[12] Intel. An introduction to the intel quickpath interconnect, January 2009. Document Number: 320412-001US.

[13] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[14] M. Tim Jones. Inside the linux 2.6 completely fair scheduler, December 2009 [Accessed: 02 Mar. 2011].

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253668-037US. January 2011.

[16] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Ope. Sys. Rev.*, 41:47–58, March 2007.

[17] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO '06*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS '09*, pages 57–67, Washington, DC, USA, 2009.

[19] Andrea B. and B. B. Brandenburg J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *OSPERT '10*, pages 33–44, Brussels, Jul 2010.

[20] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Par. Dist. Comp.*, 37:113–121, August 1996.

[21] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Sys. Rev.*, 44:19–29, December 2010.

[22] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT '09*, pages 245–254, New York, NY, USA, 2009. ACM.