The Real Influence of Shared Memory Contention on Real-time Multicore Applications

Giovani Gracioli*, Julio Sincero†, Wolfgang Schröder-Preikschat†, Antônio Augusto Fröhlich*

*Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
{giovani,guto}@lisha.ufsc.br

†Department of Computer Science 4
Friedrich-Alexander University Erlangen-Nuremberg
Erlangen, Germany
{julio.sincero,wosch}@informatik.uni-erlangen.de

***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 real influence of contention for shared data memory in the context of embedded real-time applications. By using a worst-case benchmark, we have measured the real impact of excessive shared memory invalidations on five processors with three different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory organization (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). Towards a solution, we propose an architecture composed by several OS techniques, such as memory partitioning and scheduling. We also provide an analysis on Hardware Performance Counters (HPC) and propose to use them in order to monitor and detect excessive memory invalidations at run-time.**

***Index Terms*—Contention for shared data memory, Real-time scheduling, Multicore processors**

I. Introduction

Several embedded real-time applications are implemented in a dedicated hardware logic (i.e. *Application-Specific Integrated Circuit - ASIC*) to obtain maximum performance and fulfill all 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, maintainability).

The continuous evolution of processor technology, however, together with its decreasingly cost, has enabled multicore (e.g. *Symmetric Multiprocessing - SMP*) architectures to be also used in the embedded real-time system domain [1]. Thus, the same applications can be ported to software, with similar performance and more support flexibility to their developers. In this context, an application is implemented on top of a Real-Time Operating System (RTOS), composed by 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].

Several works have been proposed to deal with memory organization in multicore architectures and provide real-time guarantees [6], [7], [8], [1], but they only consider scenarios where threads are independent, that is, there is not data sharing. In this case, the influence among threads (contention for cache space and interference in the cache lines) can be solved by some memory partitioning/locking mechanism provided by a special hardware [5] or implemented into the (real-time) scheduler [9], [3]. Partitioning/Locking techniques avoid overlapping of shared cache spaces and reduce the contention for the shared resource, increasing the application's throughput.

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 by "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), as demonstrated in Figure 1. 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 lower memory level (previous cache level or main memory). The task of performing a snoop request, in the current SMP processors, 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) [10], which can lead to an unexpected increase of the thread's execution time and, in case of a real-time application, to deadline losses.

In this paper, we bring the problem of contention for shared memory to the embedded real-time system domain by measuring its real influence on five different modern processors with

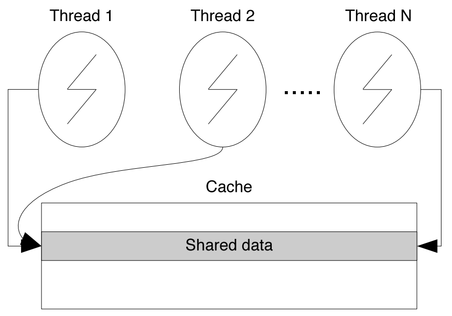


Figure 1. An example of pipeline application composed by threads sharing data.

three different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory organization (UMA and ccNUMA). We use a benchmark composed by two versions of the same application – sequential and parallel – for comparison purpose. 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, obviously it should be faster). 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. Towards a solution, we envision an environment where the operating system (OS) real-time scheduler, together with a memory partitioning and Hardware Performance Counters (HPCs) support, could detect excessive memory invalidations among threads and take a decision (scheduling) in order to avoid deadline losses. Scheduling is a good solution because it does not require special hardware support and is easily integrated in practically any RTOS [3]. Moreover, the scheduler is completely transparent to applications, that is, there is no need to change the application source code neither OS APIs nor libraries.

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

- We motivate the problem by measuring the real influence of contention for shared memory in the context of real-time applications where deadlines must be always met.
- We evaluate the problem on five different multicore processors, with three different cache-coherence protocols (MESI, MOESI, and MESIF) and two memory organization (UMA and ccNUMA) by using a benchmark composed by a sequential and parallel version of the same application.
- Towards the problem solution we propose an architecture composed by OS techniques, such as scheduling, HPCs, and memory partitioning, in order to provide predictability and real-time guarantees. Moreover, we evaluate HPCs in one of the five processors. HPCs can be used to monitor and detect when two or more threads are sharing data. 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.

The remainder of this paper is organized as follows. Section II describes in details the problem that we are addressing and summarizes the background needed to follow the rest of the paper. The benchmark evaluation is presented in Section III. Section IV discusses the results. Section V provides an overview of previous work on memory hierarchy in multicore architectures and memory-aware and real-time multicore scheduling. Finally, Section VI concludes the paper.

II. Problem Description and Background

A. The Problem

In SMP systems, each processor fetches its own instructions and operates its own data, characterizing a Multiple Instruction Multiple Data (MIMD) system. They feature a unique shared Uniform Memory Access (UMA) or a cache-coherent Non-Uniform Memory Access (ccNUMA), easing data sharing. Two memory organization widely used in today's SMP processors are presented in Figure 2. In Figure 2(a), "n" cores have a private level 1 cache, share a larger level 2 cache and the main memory. An example of processor that uses this architecture is the Intel Core 2 Quad Q9550. Figure 2(b) shows another memory hierarchy. There are two physical dies, where each one is composed by two cores. Each core has a private L1 cache, each die shares a L2 cache, and a L3 cache and main memory are shared by all cores. Intel Xeon L7455 with 24 cores and 6 dies is an example of this architecture. Note that there are variations from these two architectures (e.g. *Symmetric Multithreading - SMT*, L2 shared cache for all dies, etc), but the same principle of sharing the lower levels of memory still applies.

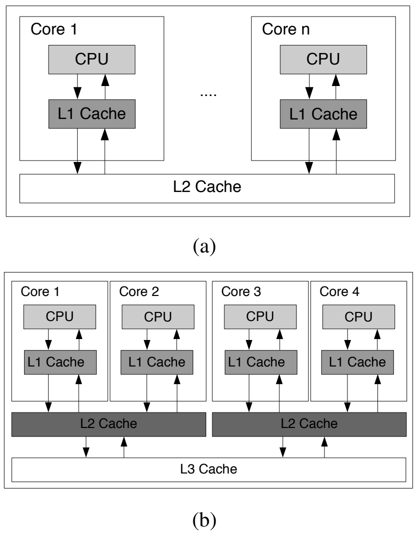


Figure 2. (a) An architecture with shared level-2 cache. (b) An architecture with a cluster sharing level-2 and all cores sharing the level-3.

The problem we are addressing in this paper raises from these memory hierarchies present in the today's SMP architectures and their memory coherence protocols. Each core has its own data and uses its private data cache for speeding up the processing. However, when cores share data, each copy of the data is placed in the core's private cache and the cache-coherence protocol is responsible for keeping the consistency between each copy (through bus snooping).

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 [10]. Two kinds of scaling problem occur due to shared memory contention [10]: access serialization to the same cache line done by the cache coherence protocol, which prevents parallel speedup, and saturation into the inter-core interconnection, also preventing parallel processing gains.

The contention for shared memory causes an increase in the application's throughput and deadline losses. One can argue that there would be processing speedup by just turning the cache off and using the main memory directly. Nevertheless, it is a misconception that worst-case execution times with caches are equal to ones without caches [11]. Moreover, it is common to find a considerable inter-thread interaction in multithreaded application. For example, some applications from NAS parallel and SPEC OMG benchmark suites have up to 20% of inter-thread interaction, and up to 60% of this interaction is affected by cache line contention [2]. Reducing the effects of cache line contention can significantly improve the application's overall performance and avoid deadline misses.

B. Cache-coherence Protocols

The MESI protocol is the most common cache-coherence protocol which supports write-back cache [12]. There are four possible states that a cache line can be marked:

- **Modified (M)**: the cache line is present only in the current cache and its data is dirty. The value must be written to the memory before a reading. The write-back changes the state to exclusive.
- **Exclusive (E)**: the cache line is present only in the current cache and it is clean. The state can be changed to the Shared state when a read request from another core arrives or to Modified state when a write operation is performed.
- **Shared (S)**: indicates that the cache line is shared by other caches and its state is clean.
- **Invalid (I)**: a cache line in this state does not hold a valid copy of data. The valid data can be in the main memory or in another processor cache.

The MOESI protocol adds a fifth state to the MESI protocol [13]. The **Owned (O)** state represents data that is both modified and shared. The S state no longer implies that the cache line is clean. Therefore, there is no need to write modified data back to the memory before sharing it. Only one processor can hold a cache line in the O state – the same cache line in other processors must be in the S state. The processor which holds a cache line in the O state is the only one to respond to a snoop request. When a snoop request arrives, the cache line switches to the O state, and the new duplicate copy is made in the S state. As a result, any cache line in the O state must be written back to memory before it can be evicted.

The MESIF protocol includes a **Forward (F)** state that is used to respond to request for a copy of a cache line [14]. The newly created copy is placed in the F state and the cache line previously in the F state is put in the S state or in the I state. Thus, there is only one copy in the F state and the remaining copies are in the S state. MESIF and MOESI protocols are usually used in ccNUMA architectures, such as Intel Nehalem and AMD Opteron respectively, while MESI is widely used in UMA architectures, such as Intel dual-core.

C. Hardware Performance Counters

HPCs are special registers available in the most modern microprocessors through the hardware Performance Monitoring Unit (PMU). HPCs offer support to counting or sampling several micro-architectural events, such as cache misses and instructions counting, in real-time [15]. However, they are difficult to use due to the limited hardware resources (for example Intel Nehalem supports event counting with seven event counters and AMD Opteron provides four HPCs to measure hardware events) and complex interface (e.g. low-level and specific to micro-architecture implementation) [16].

Nevertheless, it is possible to use multiplexing techniques in order to overcome the limitation in the number of HPCs [17], [15] or specific libraries that make the use of HPCs easier [18], yet adding a low overhead to the application. HPCs can be used together with OS techniques, such as scheduling and memory management, to monitor and identify performance bottlenecks in order to perform dynamic optimizations [19]. In multicore systems, for instance, it is possible to count the numbers of snoop requests, last-level cache misses, and evicted cache lines. In the next section we provide an analysis of the described problem by measuring the real influence of shared memory contention through a benchmark.

III. Problem Evaluation

In order to evaluate the influence of contention for shared data memory in the execution time of an application, we have designed a worst-case benchmark composed by two versions of a pipeline application and a best-case application for comparing purposes (Figure 3):

- **Sequential**: in this version, two functions are executed in a sequential order. There are no memory conflicts (Figure 3(a)). The objective of this version is to simulate an algorithm implemented in hardware, that is, the algorithm does not face the shared memory invalidation problem.
- **Parallel**: two threads run at the same time and share data (Figure 3(b)). The objective of this version is to evaluate the performance of the previous version when it is implemented in a multicore architecture. Both functions (1 and

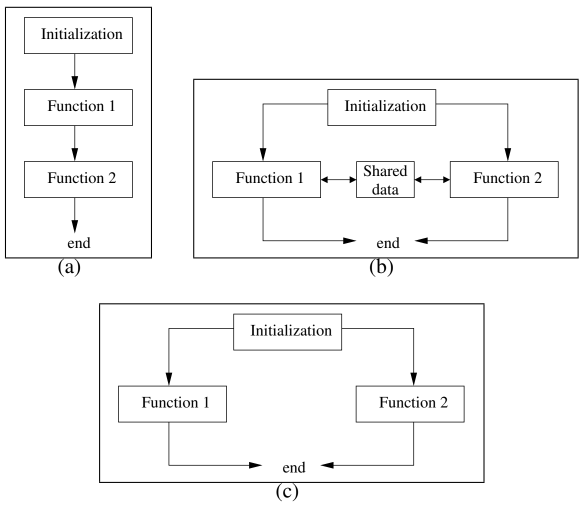


Figure 3. Benchmark applications: (a) sequential (b) parallel (c) best-case.

2) from sequential and parallel versions have the same operations and memory accesses. Common sense dictates that this version should run about two times faster than the sequential one. Consequently, if the sequential version is 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.

- **Best-case application**: two threads run at the same time, but do not share data (Figure 3(c)). Obviously, this application should run about 2 times faster than the sequential one in 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 repetitions, 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. 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++ compiler), and using the pthread library for the parallel and best-case versions. We ran each version in five different processors (Table I), varying the arrays' size.

Figure 4 presents the WCET (in logarithm scale) for each application version on the Intel Core 2 Quad Q9550 processor. We ran each application for 10 times and then extracted the WCET. The values were measured by the Linux *time* tool. We note that, independently of the arrays' size, the parallel version was always slower than the sequential version (up to 1.31 time). As expected, the best-case application was about 2 times faster than the sequential one. In addition, we repeated the evaluation using an optimized version of the Linux kernel [10]. Basically, the Linux was modified to avoid locks and atomic instructions by reengineering data structures and unnecessary sharing. Figure 4(b) shows the measured WCET. All applications presented similar performance, and the parallel version was always slower than the sequential one. This performance degradation is caused by the shared memory invalidations (e.g. bus snooping and MESI cache-coherence protocol), which implies in deadline misses considering the embedded real-time domain.

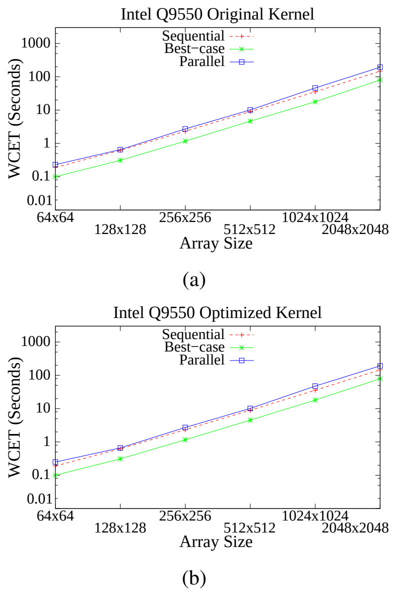


Figure 4. Benchmark evaluation on Intel dual-core Q9550 processor: (a) Original Linux kernel (b) Optimized Linux kernel.

We also ran the three applications in other two SMP processor, Intel Xeon 5030 and a PowerPC-based dual-core on the Cell architecture. Both use MESI as cache-coherence protocol and are UMA architectures. Figure 5 shows the benchmark performance on the two processors. 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. We also assigned threads to different cores in relation with their physical location (threads in the same physical die and threads in different physical dies[2]) and observed that there is almost no variation in the WCET for these SMP/UMA processors.

Our next evaluation was carried out in the AMD Opteron 280 processor (see Table I). This processor implements MOESI as cache-coherence protocol and differ from the previous processors because it is a ccNUMA architecture. We evaluated three execution scenarios:

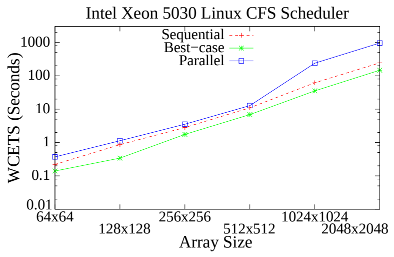
- **Scenario 1**: threads running in cores in different dies (CPUS 0 and 1).

[1]Prices of new processors at www.amazon.com April 2011.

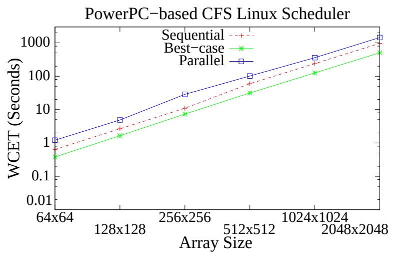
[2]By using the *proc filesystem* (/proc/cpuinfo) is possible to gather this information.

Feature / Processor	**Opteron 280**	**Intel i5-650**	**Intel Q9550**	**Xeon 5030**	**PowerPC**
Frequency	2.4 Ghz	3.2 Ghz	2.83 Ghz	2.66 Ghz	3.2 Ghz
Instruction set	64 bits	64 bits	64 bits	64 bits	64 bits
Physical dies	2	1	1	2	1
Cores per die	2	2	4	2	2
SMT	-	2	-	2	-
Bus Speed	HyperTransport 1.0 Ghz	QPI 1.3 Ghz	FSB 1.3 Ghz	FSB 667 Mhz	1.6 Ghz
L1 private data cache size	128 KB 2-way associative	32 KB	64 KB	16 KB	32 KB
L2 cache size	1 MB 16-way associative	256 KB per core (private)	12 MB	4 MB 8-way associative	512 KB
L3 shared	-	4 MB	-	-	-
Cache alignment	64 bytes	64 bytes	64 bytes	128 bytes	64 bytes
Memory architecture	ccNUMA	ccNUMA	UMA	UMA	UMA
Coherence protocol	MOESI	MESIF	MESI	MESI	MESI
Linux version	2.6.32	2.6.32	2.6.32	2.6.32	2.6.32
Price[1]	$265.50	$210.99	$289.99	$349.95	Not found

Table I
AMD Opteron 280, Intel i5-650, Intel Q9550, Intel Xeon 5030, and PowerPC-based on Cell processors configurations.



(a)



(b)

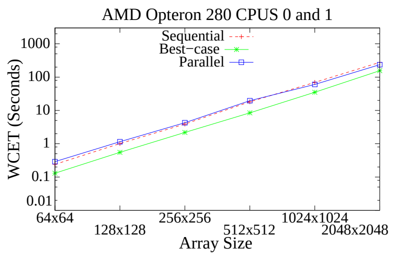
Figure 5. Benchmark evaluation on (a) Intel Xeon 5030 and (b) PowerPC-based processors.

- **Scenario 2**: threads running in cores in the same die (CPUS 0 and 2).
- **Scenario 3**: Linux Completely Fair Scheduler (CFS) responsible for allocating a thread to a core. CFS uses a time-ordered red-black tree to build a "timeline" of future task execution. All runnable tasks are sorted by a run-time key, which represents the expected CPU time a task should have. The scheduler gives the CPU to the leftmost task in the tree (the task that has gained the smallest amount of processing time). The executed tasks are put into the tree more to the right, and thus giving the chance for every task to become the leftmost task within a deterministic amount of time (the tree is self-balanced, which means that any path in the tree will be at most twice as long as any other). 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 [20].

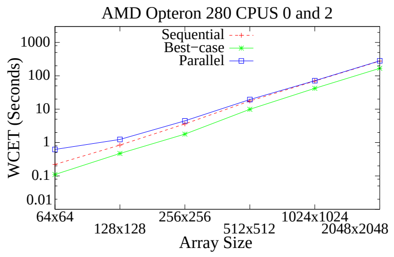
The objective of this test was to analyze the relation between the physical core location and the shared memory coherence by measuring the applications WCET. We assigned each thread in a specific core by using the *pthread_setaffinity_np* function. Figure 6(a) shows the WCET for scenario 1. For arrays' size of 1024x1024 and 2048x2048 the sequential application was slightly faster than the parallel one. For all other sizes, the parallel one was slower than the sequential. In Figure 6(b) we present the measured WCET for scenario 2. In this scenario, the parallel version was always slower than the sequential one (up to 2.82 times using arrays' size of 64x64). We can also note a decreasing in the WCET as the arrays' size increases. Finally, Figure 6(c) represents the measured WCET without assigning threads to any specific core, the CFS is responsible for assigning threads to cores. In this scenario, the applications performance was similar to the previous one. We can conclude that the WCET is related to the arrays' size, the bigger arrays' size the faster the execution time. Additionally, as expected from a ccNUMA architecture, there is a variation in the WCET when executing applications in cores physically located in different dies.

Our next evaluation was carried out in the Intel i5-650 processor. This processor implements MESIF as cache-coherence protocol and it is also a ccNUMA architecture. We ran each application in three different scenarios:

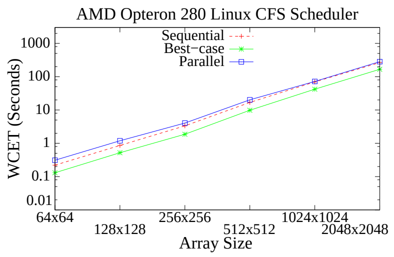
- **Scenario 1**: threads running in two different cores (CPUS 0 and 1).
- **Scenario 2**: threads running in the same core using *Symmetric Multithreading* (SMT) technology (CPUS 0 and 2).
- **Scenario 3**: Linux CFS responsible for allocating a thread



(a)

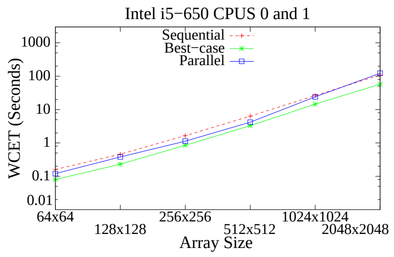


(b)

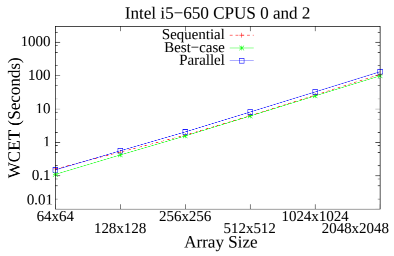


(c)

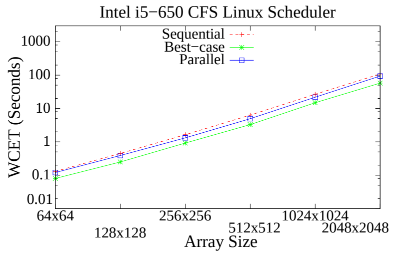
Figure 6. Benchmark evaluation on AMD Opteron 280 processor assigning threads to specific cores: (a) CPUS 0 and 1 (b) CPUS 0 and 2 (c) Linux CFS scheduler.



(a)



(b)



(c)

Figure 7. Benchmark evaluation on Intel i5-650 processor assigning threads to specific cores: (a) CPUS 0 and 1 (b) CPUS 0 and 2 (c) Linux CFS scheduler.

to a core.

Figure 7 shows the WCET of each application in the three described scenarios. We can observe that the difference between the parallel and best-case WCET was basically the same in all scenarios. In scenario 2, the SMT obviously degrades performance of the parallel and best-case applications. In general, the parallel version was faster than the sequential one, only in scenario 1 with arrays' size of 2048x2048 it was slower than the sequential. This can be explained by the processor organization, composed by Intel's QuickPath Interconnect (QPI) and MESIF cache-coherence protocol. Compared to the front-side bus (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 [14].

IV. Discussion and Future Work

During our evaluations we observed a set of interesting facts regarding the described problem:

SMP architectures. We executed our benchmark on five different processors (Intel Core 2 Quad Q9550, PowerPC-based on Cell, Intel Xeon 5030, Intel i5-650, and AMD Opteron 280), that implement MESI, MESIF, and MOESI, as cache-coherence protocol, and have 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. We will concentrate our research in UMA systems that traditionally use MESI as memory coherency because they presented a worse degradation due to contention for shared memory. 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.

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 because is totally transparent to applications, there is no need to change APIs nor libraries, and can be easily integrated in RTOS.

Memory partitioning/locking. In order to reduce cache line invalidations and the interference among threads or other applications that do not share resources, methods such as memory partitioning and locking [5], [2] could be used or easily solved by a special hardware support. However, in a cooperating real-time application, such as those found in digital signal processing area, where threads share data, a memory partitioning/locking do not solve the problem because threads will access the same data location on the memory hierarchy. Memory partitioning/locking can be used together with other techniques, such as scheduling, in order to decrease the contention for shared resources between several applications (inter-application contention) and application's threads (intra-application contention).

Solution Statement. Towards a solution for the problem, we envision a set of OS techniques designed together to provide predictability and real-time guarantees for embedded real-time applications. The proposed architecture is depicted in Figure 8 and it is a step forward to mitigate the effects of contention for shared memory. It is composed by a two-level OS cache partitioning to minimize the contention for cache space between different applications (inter-application partitioning) and between threads of the same application (intra-application partitioning) [2] and a shared memory-aware scheduler responsible for detecting and minimizing the influence of memory coherence from threads that share data (e.g. access serialization to the same cache line and saturation in the inter-core interconnection), while still compromising with real-time guarantees. Moreover, improved parallel programming techniques, such as *sloppy counters* [10], can also be incorporated into OS synchronization primitives.

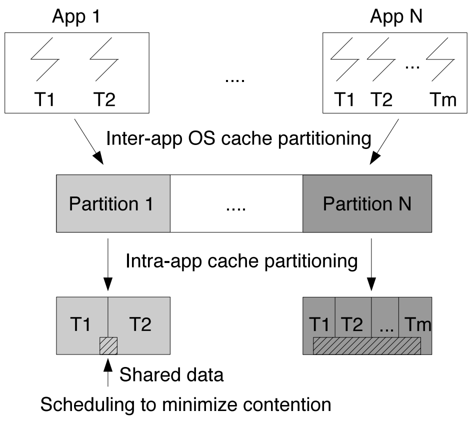

App 1
T1
T2
....
App N
T1
T2
...
Tm
Inter-app OS cache partitioning
Partition 1
....
Partition N
Intra-app cache partitioning
T1
T2
T1
T2
...
Tm
Shared data
Scheduling to minimize contention

Figure 8. Proposed solution architecture.

As embedded real-time systems usually have their requirements known at design time, it is possible to scan the source code and extract information about threads, including data sharing, with a specific tool [21], and use these information at run-time to help the scheduler decision. In addition, to monitor and detect patterns of threads memory accesses, and dynamically change the cache partitions size and decide when to take a scheduling decision related to threads that share data, the OS needs to gather information at run-time. HPCs are a good alternative for this purpose, since they are common in current multicore processors and the overhead for reading them is relatively low [19].

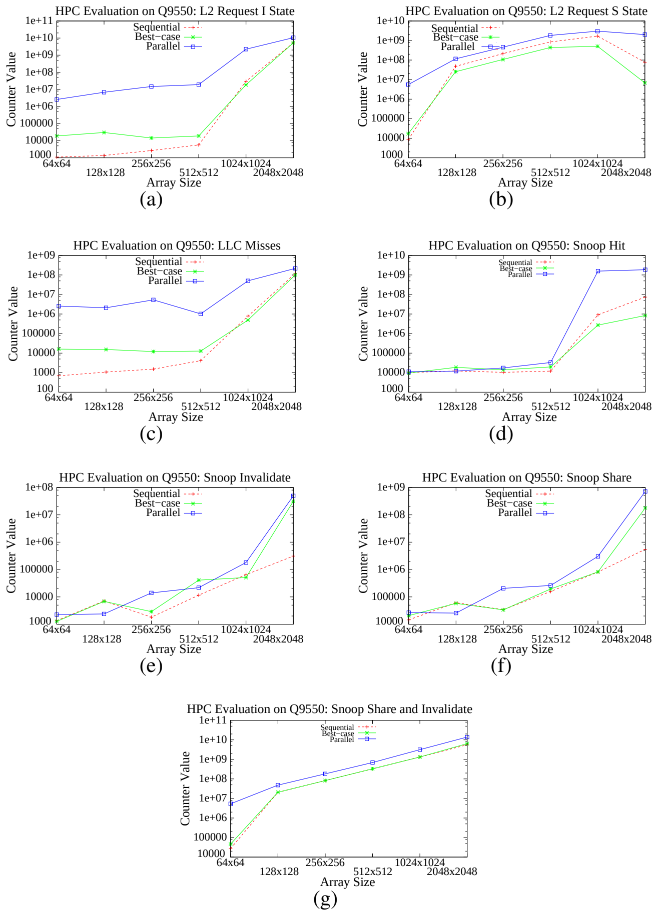


Figure 9. Benchmark HPCs evaluation on Intel Q9550 processor.

In this context, we present a preliminary analysis of HPCs running our benchmark in the Intel Q9550 processor. 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 for 10 times. Figure 9 shows the average value for each HPC: L2-cache request for I state, L2-cache request for S state, Last-Level Cache (LLC) misses, Snoop request hit, Snoop request Invalidate, Snoop request Share, and Snoop request Share and Invalidate. All HPCs are dependent of the application arrays' size. Snoop Hit, Snoop Invalidate, and Snoop Share have presented similar behavior for small arrays' size. As the arrays become bigger, the Snoop Hit has shown more variation between the parallel and the other two applications. Snoop Share and Invalidate has shown a constant behavior for the parallel version. L2-cache requests for I cache line state is a good alternative when the shared data is not too big. L2-cache requests for I and LLC misses had practically the same behavior. HPCs can be used to monitor the shared data addresses and mount a threads' sharing pattern with low overhead. An example is the *shMap*, a vector of 8-bit saturating counters [19]. Each counter in the vector corresponds to a region in the virtual address space and its value is the representation of the sharing patterns detected inside that region. In summary, advanced HPCs capabilities allows the OS to have a precise view from applications' behavior and make smarter decisions at a fine granularity (e.g. cache line) [19]. However, as we demonstrated in our benchmark evaluation, the performance degradation does not depend on the shared data size. It is more important to detect and monitor the correct shared memory address than an entire region. As future work, we intend to implement and evaluate a mechanism to detect and monitor shared data using HPCs.

V. Related Work

In the next subsections we summarize the main publications related to memory hierarchy in multicore systems and memory-aware and real-time multicore scheduling.

A. Memory Hierarchy

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. Several hardware- [5] and software-based [22], [23], [2] cache partitioning have been proposed in the last years. The software-based approach has the advantage to be completely transparent to applications and there is no need to special hardware support. The most common software-based technique is the *page coloring*. This technique explores the translation from virtual to physical memory addresses presented in the virtual memory systems, in such a way that addresses from different applications are always mapped to predefined cache regions [11], [22]. Differently from the previous works, which focused on multi-application workloads, Chen analyzed different policies for managing shared caches for multi-threaded applications [24]. The work has shown that the shared-cache miss rate can be reduced by allocating a certain amount of space for shared data. Muralidhara proposed a more sophisticated dynamic software-based partitioning technique that partitions the shared cache space among threads of a given application [2]. At the end of each 15 ms interval, the dynamic cache partitioning scheme uses HPCs information, such as cache hit/misses, cycle and instruction counts for each thread, in order to allocate different cache space based on individual thread performance. The objective is to speed up the critical path, that is, the thread that has the slowest performance and, consequently, improve the overall performance for the application. The results have shown a performance gain of up to 23% over a statically partitioned cache [2].

Cache partitioning can also be used together with some special hardware support, such as *cache locking* [5]. Cache locking prevents data in the cache from being replaced at run-time, easing the schedulability analysis and obtaining predictability for real-time application [5]. However, in the most used current processors, there is no hardware support for cache locking.

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. [25]. 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. Moreover, the system model used by the authors is simple, it does not consider preemptions and data exchange among tasks are only done by message passing. *Schedule-Sensitive* and *Synthetic* are two methods to measure cache-related preemption and migration delays (CPMD) [26]. In the first method, delays are recorded online by executing tests and collecting the measured data. The drawback of this method is the impossibility of controlling when a preemption or migration happens, recovering many invalid data [27]. The second method tries to overcome this problem by explicitly controlling preemptions and migration of a task, and thus measuring the delays. 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 [26].

B. Memory-aware and Real-time Multicore Scheduling

OS support for multicore applications, such as memory management, scheduling, and synchronization primitives, has received special attention in the last few years. For example, Corey [28] and Barrelfish [29] are two OS specifically designed to take full advantage of multicore processors. In addition, improvements on standard parallel programming techniques, such as proposed by *sloppy counters* [10], help on reducing the scalability problems found in the today's OS. Multicore scheduling algorithms focused on NUMA architectures are also an important research topic. Heterogeneity-Aware Signature-Supported (HASS) [30] and AMPS [31] are two examples of this kind of scheduling algorithm. HASS uses architectural signatures to match the application to a "best" core. These signatures are generated off-line and are embedded into the system's binary code. Initially, the algorithm does not support real-time applications. AMP is composed by three components: (i) asymmetric-aware load balancing responsible for ensuring that each core has a proportional load depending on its computing power; (ii) faster-core scheduling ensures that threads will run on faster cores whenever they are underutilized; and (iii) NUMA-aware migration that dynamically prevents thread migration in cores by using a thread resident set that contains the current pages in memory used by the thread. This resident set is used to estimate the migration overhead of a thread.

Several scheduling algorithms have been proposed in order to provide real-time guarantee for multicore applications. Brandenburg and Anderson discuss how the implementation of the ready queue, quantum-driven x event-driven scheduling, and interrupt handling strategies affect a global real-time scheduler [32]. The results indicate that implementation issues can impact schedulability as much as scheduling-theoretic tradeoffs. Moreover, in their case study, the best performance was achieved by using a fine-grained heap, event-driven scheduling, and dedicated interrupt handling. An empirical comparison of Global-EDF (G-EDF), Partitioned-EDF (P-EDF), and Clustered-EDF (C-EDF) on a 24-core Intel platform, assuming run-time overhead (e.g. release, context switch, and schedule times) and cache-related delay, has concluded that P-EDF outperforms the other evaluated algorithms in hard real-time scenarios [27]. Moreover, the same study suggests the use of "less global" approaches (P-EDF and C-EDF-L2, which cluster at the cache level 2) in contrast of "more global" approaches (G-EDF and C-EDF-L3) in hard real-time applications. A real-time scheduling for multiprocessors is proposed by Cho [1]. The authors proposed a new abstraction about task execution behavior on multiprocessors, named the time and local remaining execution-time plane (T-L plane). The entire scheduling over time is the repetition of T-L planes in various sizes. Consequently, a single T-L plane scheduling implies on feasible scheduling over all times. Another scheduling algorithm is presented by Srinivasan and Anderson [33]. They show that the PD2 Pfair Algorithm is an optimal rate-based scheduling algorithm, since it correctly schedules any feasible intra-sporadic task system on M processors.

In the context of memory-aware real-time multicore scheduling algorithms, Calandrino and Anderson have proposed a cache-aware scheduling algorithm [8], [7]. In their approach, the scheduling process is divided in two phases: (i) all tasks that may induce significant memory-to-L2 traffic are combined into groups off-line; and (ii) at run-time, a scheduling policy that reduces concurrency within groups is used. The paper introduces the concept of *megatask*, which represents a task group and is treated as a single schedulable entity. All cores are symmetric, share a chip-wide L2 cache, and each core supports one hardware thread. Nevertheless, the algorithm fails in measuring the real influence of cache thrashing in an environment with threads sharing memory. FP_{CA} is a cache-aware scheduling algorithm that divides the shared cache space into partitions [9]. 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. The authors proposed two schedulability tests, one based on a linear problem (LP) and another one as an over-approximation of the LP test. Tasks are not preemptive and the algorithm is blocking, i.e. it does not schedule lower priority ready jobs to execute in advance of higher priority even though there are enough available resources. FP_{CA} is executed whenever a job finishes or a new job arrives.

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 [34]. Tam, Azimi, and Stumm use HPCs to monitor the addresses of cache lines that are invalidated due to cache-coherence activities and to construct a summary data structure for each thread, which contains the addresses of each thread that are fetching from the cache [35]. Based on the information from this data structure, the scheduler mounts a cluster composed by a set of threads, and allocates a cluster to a specific core. However, this approach is not feasible for UMA processors, since there is no performance gains in allocating a thread to a specific core. West et al. [36] 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. DIO performed better than the Linux CFS both in terms of average performance as well as execution time stability from different executions [3].

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.

VI. Conclusion

This paper evaluated the real influence of contention for shared data memory in the context of real-time multicore applications. We have designed a worst-case 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 by 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.

Our proposal towards a solution is a combination of OS techniques (e.g. inter- and intra-application memory partitioning, well-designed synchronization primitives, and a shared-aware real-time scheduler) in order to alleviate the effects of shared memory coherency (i.e. access serialization to the same cache line done by the cache coherence protocol and saturation into the inter-core interconnection). As future work, we will investigate techniques to be incorporated into the OS real-time scheduling and evaluate our proposed solution.

VII. Acknowledgments

This work was partially supported by the Coordination for Improvement of Higher Level Personnel (CAPES) grant, projects RH-TVD 006/2008 and 240/2008, and by the German Research Council (DFG) under grant no. SCHR 603/7-1.

References

[1] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 101–110.

[2] S. Muralidhara, M. Kandemir, and P. Raghavan, "Intra-application cache partitioning," in *IPDPS 10': Proceedings of the 25th IEEE International Symposium on Parallel Distributed Processing*, 2010, pp. 1–12.

[3] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 129–142.

[4] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 600–605.

[5] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*. New York, NY, USA: ACM, 2008, pp. 300–303.

[6] H. Leontyev and J. H. Anderson, "A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees," in *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 191–200.

[7] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 299–308.

[8] J. H. Anderson, J. M. Calandrino, and U. C. Devi, "Real-time scheduling on multicore platforms," in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 179–190.

[9] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 245–254.

[10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*.

[11] J. Liedtke, H. Haertig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 213–224.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Fourth Edition, September 2006.

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

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

[15] B. Sprunt, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, Jul/Aug 2002.

[16] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 101–110.

[17] J. May, "Mpx: Software for multiplexing hardware performance counters in multithreaded programs," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium.*, Apr. 2001, p. 8 pp.

[18] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 289.2–.

[19] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Operating System Review*, vol. 43, pp. 56–65, April 2009.

[20] M. T. Jones, "Inside the linux 2.6 completely fair scheduler," December 2009 [Accessed: 02 Mar. 2011]. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[21] R. L. Cancian, M. R. Stemmer, A. Schulter, and A. A. M. Fröhlich, "A tool for supporting and automating the development of component-based embedded systems," *Journal of Object Technology*, vol. 6, no. 9, pp. 399–416, Oct. 2007, tOOLS EUROPE 2007 — Objects, Models, Components, Patterns.

[22] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 455–468.

[23] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 89–102.

[24] Y. Chen, W. Li, C. Kim, and Z. Tang, "Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy," in *IPDPS 09': Proceedings of the 24th IEEE International Symposium on Parallel Distributed Processing.*, may 2009, pp. 1–11.

[25] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *RTSS '09: Proceedings of the 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 57–67.

[26] A. Bastoni and B. B. B. J. H. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Brussels, Belgum, Jul 2010, pp. 33–44.

[27] B. B. Brandenburg and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers," in *RTSS '10: Proceedings of the 31st IEEE Real-Time Systems Symposium*. San Diego, CA, USA: IEEE Computer Society, 2010, pp. 14–24.

[28] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. hua Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *OSDI 08': Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, 2008, pp. 43–57.

[29] A. Schupbach, P. Simon, A. Baumann, and T. Roscoe, "Embracing diversity in the Barrelfish manycore operating system," in *MMCS 08': Proceedings of the Workshop on Managed Many-Core Systems*. ACM, June 2008.

[30] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *SIGOPS Operating System Review*, vol. 43, no. 2, pp. 66–75, 2009.

[31] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–11.

[32] B. B. Brandenburg and J. H. Anderson, "On the implementation of global real-time schedulers," in *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 214–224.

[33] A. Srinivasan and J. H. Anderson, "Optimal rate-based scheduling on multiprocessors," *Journal of Computer and System Science*, vol. 72, no. 6, pp. 1094–1117, 2006.

[34] F. Bellosa and M. Steckermeier, "The performance implications of locality information usage in shared-memory multiprocessors," *Journal of Parallel Distributed Computing*, vol. 37, pp. 113–121, August 1996.

[35] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," *SIGOPS Operating System Review*, vol. 41, pp. 47–58, March 2007.

[36] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online cache modeling for commodity multicore processors," *SIGOPS Operating System Review*, vol. 44, pp. 19–29, December 2010.