A Dual-Priority Real-Time Multiprocessor System on FPGA for Automotive Applications

Antonino Tumeo[1] Marco Branca[1] Lorenzo Camerini[1] Marco Ceriani[1] Matteo Monchiero[2]
Gianluca Palermo[1] Fabrizio Ferrandi[1] Donatella Sciuto[1]

[1]Politecnico di Milano - DEI
Via Ponzio 34/5
20133 Milano, Italy

[2]HP Labs
1501 Page Mill Rd.
Palo Alto 94304 CA, USA

Abstract

This paper presents the implementation of a dual-priority scheduling algorithm for real-time embedded systems on a shared memory multiprocessor on FPGA. The dual-priority microkernel is supported by a multiprocessor interrupt controller to trigger periodic and aperiodic thread activation and manage context switching. We show how the dual-priority algorithm performs on a real system prototype compared to the theoretical performance simulations with a typical standard workload of automotive applications, underlining where the differences are. [1]

1 Introduction

Multiprocessor Systems-on-Chip (MPSoCs), composed of several processing elements and on-chip memories, have become the standard for implementing embedded systems. Thanks to the presence of multiple processing units, these systems potentially allow a better management of periodic workloads and can react faster to external, aperiodic events.

Nevertheless, the adoption of these powerful architectures in real time systems opens several problems concerning scheduling strategies [16], since it is well known that optimal scheduling for multiprocessor systems is a NP-Hard problem [11]. Furthermore, on real systems, it is not trivial to coordinate and correctly distribute tasks on the different processors, even more when events are triggered by interrupts from external peripherals. This is, for example, the case for automotive applications, in which several periodic tasks to check the status of sensors and other mechanisms run in parallel with tasks triggered by external events like security warnings. Efficient hardware solutions must support a low overhead real time scheduler to allow the design of a usable system.

Real time scheduling algorithms for multiprocessors are usually divided in two classes: local and global schedulers. Local schedulers rely on a pre-partitioning of the tasks on the different processors, and then tries to schedule them in the best way possible. Global schedulers, instead, try to allocate all the tasks on all the processors. Among all the real time scheduling algorithms proposed in the recent years for multiprocessor systems, an interesting solution is the *Multiprocessor Dual Priority Scheduling algorithm* [8]. Thanks to its hybrid local-global nature, it guarantees periodic task deadlines (local) but allows to serve aperiodic requests with very good average response time (global), which is crucial for reactive systems. Since it is based on the offline computation of the worst case response time for periodic tasks, it has low memory usage and low computational overhead, resulting thus suitable for efficient implementation on small embedded systems.

Field Programmable Gate Arrays (FPGAs) are emerging as an interesting design alternative for system prototyping and implementation for critical applications when the production volume is low. Lately, several multiprocessor solutions on FPGAs have appeared, but no one has been targeted to real time applications. In this paper, we present a real time, shared memory multiprocessor architecture on FPGA which supports a dual priority scheduling microkernel thanks to the implementation of a dedicated multiprocessor interrupt controller. The main goal of this work is to propose a realistic multiprocessor architecture with support for a light real time dual priority operating system layer, to evaluate it with a realistic workload (the MiBench automotive benchmark set), and to compare its results to the theoretical performance obtained with the simulation of the scheduling algorithm, observing the aspects that in an actual architecture can impact the performance.

The paper proceeds as follows. Section 2 presents some related works in the field of multiprocessor real time systems. Section 3 presents our multiprocessor architecture with details on the multiprocessor interrupt controller. Section 4 presents the kernel with dual priority scheduling and finally Section 5 discusses the experimental evaluation.

2 Related Work

Recently, a large number of works on multiprocessor real time systems have appeared, both from the algorithmic and the architectural point of views. As multiprocessors became the standard for embedded systems, the attention has been focused on proposing scheduling solutions that allow good

[1]Research partially funded by the European Community's Sixth Framework Programme, hArtes project.

response times with sporadic and aperiodic tasks. Many commercial real time operating systems still rely on single processor architectures for manageability purposes or adopt simple priority-based preemptive scheduling in multiprocessor solutions [1–3].

Achieving good aperiodic response time is a distinctive element for real world systems targeted to automation and automotive applications, in which it is not only necessary to guarantee hard deadlines with periodic tasks, but efficient reaction to external events is required. Multiprocessors appear as an interesting solution since thanks to the presence of multiple processing elements, workloads can be distributed on different processors and thus aperiodic and periodic tasks, with hard or soft deadlines can advance in parallel. A common approach in multiprocessor systems is to partition periodic tasks among processors statically and then use a well-known uniprocessor scheduling algorithm as a local scheduler [10, 14, 15]. Aperiodic tasks, however, are allowed to migrate to any processor [7]. Another approach focuses instead on trying to find the best allocation on all the processors available in the system. These type of multiprocessor global schedulers, however, do not deal with aperiodic tasks [4, 5, 12].

However, all these works attack the problem from a theoretical point of view. Our purpose is different. We choose to implement an existing algorithm for real time task scheduling and then check its performance on a realistic architecture rather than by simulation. We developed several hardware devices to support reactive behavior and allow inter processor communication and implemented on top of it a microkernel supporting the Multiprocessor Dual Priority algorithm [8], which thanks to its hybrid local-global nature is suitable to manage aperiodic tasks with good response time. With this system, we aim at measuring and identifying which factors can influence the performance w.r.t. the efficiency of the algorithm.

3 Architecture

This section presents the architecture of our real time multiprocessor system on FPGA. We will first describe the basic architecture and then give some details on the multiprocessor interrupt controller design that is used to distribute activation signals for the scheduler and trigger the execution of aperiodic tasks.

3.1 Basic architecture

The target architecture has been realized with the Xilinx Embedded Developer Kit (EDK) 8.2 and synthesized with Xilinx ISE version 8.2 on a Virtex-II PRO XC2VP30 Speed Grade -7 FPGA targeting a frequency of 50 MHz. This architecture, shown in Figure 1, is composed of several MicroBlaze processors connected to a shared On-chip Peripheral Bus (OPB). Each processor has access to a local memory for private data (stack and heap of the executing thread), implemented through Block RAMs (BRAMs), and to an external shared Double Data Rate (DDR) RAM memory for shared instructions and data. Instruction cache is implemented for each processor, bringing down access latency from 12 to 1 clock cycle in case of hit. Local memories, which have the same latency of caches, are instead used for data. A shared BRAM is connected to the OPB for boot purposes. The system adopts an ad-hoc coprocessor (Synchronization Engine) that provides hardware support for lock and barrier synchronization primitives and a CrossBar module that allows inter-processor communication for small data passing without using the shared bus. The system features a multiprocessor interrupt controller specifically designed for this architecture. This controller distributes to all the processors in the system interrupt signals of various nature. It forwards the signal triggered by the system timer, that determines the scheduling period and starts the scheduling cycle, to an available processor. It connects to the peripherals so they can launch interrupts signal to any processor and trigger the start of aperiodic tasks for subsequent elaboration. Peripherals can be interfaces to sensors and data acquisition systems, like for example Controller Area Networks (CANs) interfaces, widely used in automotive applications. They are connected to the shared OPB and seen by all the processors, while their interrupt lines are connected to the multiprocessor interrupt controller which in turn forwards them to the MicroBlazes.

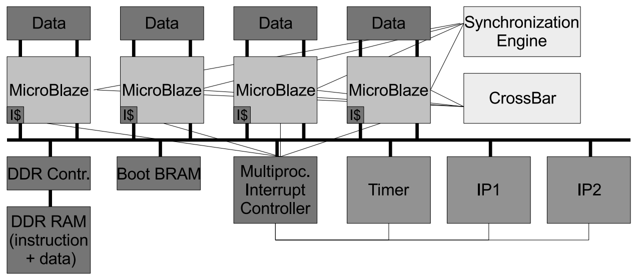


Figure 1. The architecture of our real-time system prototype.

3.2 Multiprocessor Interrupt Controller

The multiprocessor interrupt controller [17] is an innovative design that allows the support of several features in a multiprocessor architecture designed with the Xilinx toolchain. The MicroBlaze soft-core has a very simple interrupt management scheme. It only has a single interrupt input, so if multiple peripherals that generates interrupts are implemented in the system, an interrupt controller is required. However, when multiple processors are used, the standard interrupt controller integrated in the Xilinx Embedded Developer Kit is ineffective, since it only permits to propagate multiple interrupts to a single processor. Our multiprocessor interrupt controller design, instead, allows to connect them to multiple processors introducing several useful features:

it *distributes* the interrupts coming from the peripherals to free processors in the system, allowing the parallel execution of several interrupt service routines;

it allows a peripheral to be *booked* by a specific processor, meaning that only a specific processor will receive and handle the interrupt coming from a specific peripheral;

it supports *interrupts multicasting and broadcasting* to propagate a single interrupt signal to more than one processor;

it supports *inter-processor interrupts* to allow any processor to stop the execution of another processor.

Interrupt distribution allows to exploit the parallelism of a multiprocessor system to manage interrupt services routines: concurrent interrupt handlers can be launched when many interrupts are generated at the same time. This effectively permits a more reactive system to contemporary external events. Our design adopts a fixed priority scheme with timeout. When an interrupt signal is propagated to a processor, it has a predefined deadline to acknowledge its management. If the processor is already handling an interrupt, it cannot answer since interrupt reception is disabled. So, when the timeout fires, the interrupt signal to that processor is disabled and the interrupt is propagated to the subsequent processor in the priority list. Booking can be very useful when dynamic thread allocation is used. In fact, if a processor offloads a function to an intellectual property core, we may want that the same processor that started the computation manage the read-back of the results. Thus, with booking the interrupt that signals the end of the IP core work is propagated only to a designated processor. Inter-processor interrupts allow processors to communicate. They can be useful for example for synchronization or for starting a context switch with thread migration from a processor to another. Broadcast and multicast are important to propagate the same interrupt signal to more than one processing elements, like a global timer that triggers the scheduling on all the processors. These features are exposed to the microkernel with very simple primitives that are seamlessly integrated in the Xilinx toolchain. The interrupt controller is connected to the OPB and, when the processors access its registers for configuration and acknowledgments, mutual exclusion is used. Thus, controller management is sequential, but the execution of the interrupt handlers is parallel.

4 Real time support

This section briefly illustrates how the chosen scheduling algorithm, Multiprocessor Dual Priority (MPDP), works and describes the implementation details of the real time microkernel and how it interacts with the target architecture.

4.1 Multiprocessor Dual Priority Algorithm

The scheduling approach adopted in our architecture is based on the Multiprocessor Dual Priority (MPDP) algorithm [8]. This solution adapts the dual priority model for uniprocessor systems to multiprocessor architectures with shared memory.

The dual priority model [9] for single processor systems provides an efficient method to responsively schedule aperiodic tasks, while retaining the offline guarantees for crucial periodic tasks obtained with fixed priority. This model splits priorities into three bands, Upper, Middle and Lower. The periodic tasks are considered as hard tasks and are assigned two priorities, one from the Upper and one from the Lower band. Aperiodic tasks instead are considered as soft tasks and have a priority in the middle band. A periodic task is released with a priority in the Lower Band, so it may be

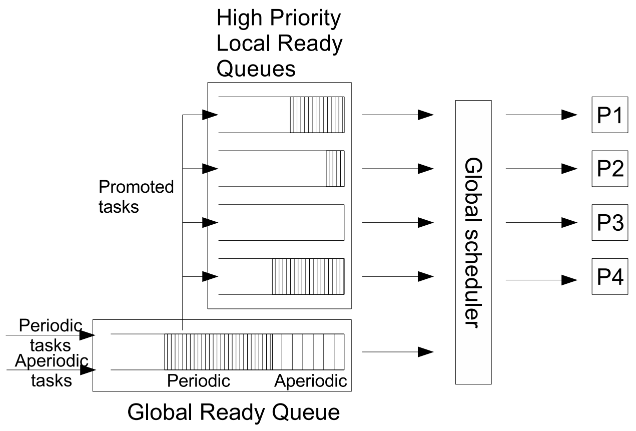


Figure 2. The conceptual organization of the MultiProcessor Dual Priority scheduler. There is a global ready queue for low priority periodic and aperiodic tasks and a local ready queue for high priority task.

preempted by other hard tasks with higher priorities in the same band, by aperiodic tasks in the middle band or by any periodic task in the upper priority band. However, at a fixed time, called promotion time, its priority is promoted to the upper band. From this moment, it could be preempted only by other periodic tasks promoted in the upper band with an higher priority. While the hard periodic task are in the lower priority bands, the aperiodic soft tasks proceed. If a periodic task is promoted, it overcomes any executing aperiodic tasks. The aperiodic task gets preempted and resumes its execution only when all the promoted higher priority periodic tasks have ended. Promotions guarantee that hard periodic tasks satisfy their deadline. Thus, if a feasible scheduling of the periodic tasks at their higher band priority exists, the dual priority algorithm allows to execute not predictable aperiodic tasks with reasonable answer times while maximizing the utilization of the system during the free timeslices. A priori guarantees for periodic tasks are obtained through offline computation of worst case responses, using the analysis derived from fixed priority scheduling [6]. The key is to determine the correct promotion time U_i for each task i. Promotion time is $0 \leq U_i \leq D_i$, where D_i is the deadline for task i and 0 is its release time. It is computed starting from the recurrence relation:

$$W_i^{m+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{W_i^m}{T_j} \rceil C_j$$

which is also used as a schedulability test. W_i is the length of a priority level busy period for task i, starting at time 0 and culminating in the completion of the task. T represents the period and C the worst case execution time. In the worst case, task i is unable to execute any computations at its lower band priority, but when promoted to the upper band, it will be subjected to interference only from tasks with higher priority in this band. (i.e. all the tasks with priority $hp(i)$). Promotion time is thus $U_i = D_i - W_i$. Iteration starts with $w_i^0 = 0$ and ends when $w_i^{m+1} = w_i^m$ or $w_i^{m+1} > D_i - U_i$, in which case task i is unschedulable.

This model, can be easily adapted to multiprocessor systems. The approach, proposed in [8] and summarized in Figure 2, is a hybrid between local and global scheduling. Hard periodic tasks before promotion can be executed on any processors (global scheduling), while after promotion they are executed on a predefined processing element (local scheduling). Initially, periodic tasks are statically distributed among the processors. The uniprocessor formula is used to compute worst case response times of periodic tasks on a single processor. During runtime, when a task arrives, being it periodic (hard) or aperiodic (soft), it is queued in a Global Ready Queue (GRQ). At the beginning, aperiodic tasks have higher priority than periodic tasks and are queued in FIFO order. Periodic tasks are sorted according to their fixed low priority. The global scheduler selects the first N tasks from this queue to execute on the processors. There are N High Priority Local Ready Queues (HPLRQ), used to queue promoted periodic tasks for each processor. When a periodic task is promoted, it is moved from the GRQ to the respective HPLRQ. If there are any tasks in this queue, a processor is not allowed to execute tasks from the GRQ. A promotion implies a change in priority and can cause a preemption, while the task migrates to its original processor. This scheme permits to advance the periodic work, guaranteeing at the same time that the system can satisfy aperiodic demands without missing the hard periodic deadlines. Thus it seems suitable for implementation on a real time systems which must perform periodic critical tasks while reacting at the same time to external events with good response times.

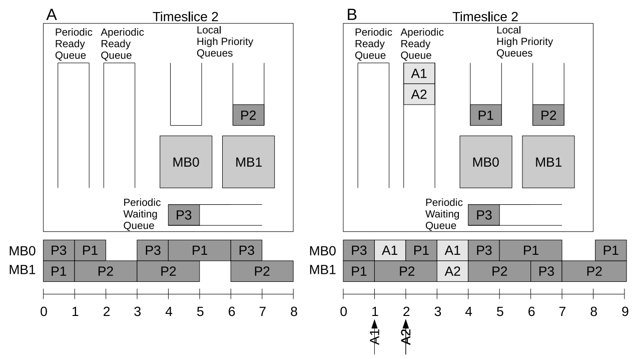
In [8] this algorithm is presented only from a theoretical point of view, with an efficiency analysis of the algorithm that assumes negligible system overheads. However, task migration and context switching have a cost. Furthermore, on a real shared memory multiprocessor system there are physical overheads determined by contentions on the shared resources which dynamically changes with the workload. Our objective is to propose a real multiprocessor system implemented on FPGA that adopts this algorithm, testing it with a real workload for automotive application and showing the impact of all these aspects.

4.2 Implementation details

Our multiprocessor dual priority microkernel implementation exploits the underlying memory model of the target architecture. In fact, the private data of the task are allocated in the fast, local memory of the processors, and are moved to shared memory and again in local memory when context changes and task switching occur. Scheduling and promotion phases are triggered by a system timer interrupt. The multiprocessor interrupt controller assigns this interrupt to a processor which is currently free from handling other interrupts. The scheduling phase is performed by a single processor. The others can continue their work while the task allocation is performed. Our implementation slightly differs from the original MPDP algorithm proposal since we use two different queues for periodic tasks in low priority (Periodic Ready Queue) and aperiodic tasks (Aperiodic Ready Queue), which make the global scheduling easier and faster. Furthermore, effectively being on a closed system, we need to park periodic tasks while they have completed their execution and are waiting for the next release. We thus use a Waiting Periodic Queue, in which periodic tasks are inserted ordered by proximity to release time. In each scheduling cycle, periodic tasks that have reached their release time enter from this queue the Periodic Ready Queue for the actual scheduling on the multiprocessor. Then, periodic tasks in the Periodic Ready Queue are checked for promotion. If a promotion happens, the periodic task is moved to the High Priority Local Queue of its target processor, in a position determined by its high priority value. At this point, task assignment can be performed according to the MPDP model. Processors with tasks in their High Priority Local Queue get the highest in priority from their specific queue. If any, aperiodic tasks (which have middle band priorities) are assigned to other processors according to their arrival order (oldest tasks are scheduled first). Finally, if there are still free processors, non promoted periodic tasks (which have lower band priorities) are allocated. After the complete task allocation is performed, the scheduling processor triggers inter-processor interrupts to all the processors that have received new tasks, starting a context change. If a task is allocated on the same processor it was currently running on, the processor is not interrupted and can continue its work. If the task allocated to the scheduling processor itself before the scheduling cycle has been changed, the scheduling routine exits launching the context change. Note that, before promotion, periodic tasks can be assigned to any processors, while after promotion they can execute only on the predefined (at design time) processors. Thus, it could be possible that two processors switch each other their tasks. If a processor completes execution of its current task, it will not wait until the next scheduling cycle but it will automatically check if there is an available task to run, following the priority rules.

Tasks contexts are constituted by the register file of the MicroBlaze processor and the stack. During context switching, the contexts are saved in shared memory, stored in a vector that contains a location for each task runnable in the system. The context switch primitive, when executed, loads the register file into the processor and the stack into the local memory. The implementation of the context switching mechanisms take care of the exiting from the interrupt handling state and the stack relocation in local memories. Aperiodic tasks are triggered by interrupts from peripherals or external devices. When these interrupts occur, the interrupt controller distributes them to processors which are not handling other interrupts. Thus, if a processor is executing the scheduling cycle, or it is executing a context switch, it will not be burdened by the aperiodic task release. Furthermore, multiple aperiodic task releases can be managed at the same time.

Figure 3 shows an example of scheduling on a dual processor architecture with three periodic and two aperiodic tasks. The table reports the basic information of the tasks required for the scheduling. Priorities can be 0 and 1 for periodic tasks in low priority mode and 3 and 4 in high priority. Aperiodic tasks are thus positioned with priority 2. Schedule A shows that without aperiodic tasks, we have an available slot in timeslice 2 on MicroBlaze 0. However, we can see that to guarantee completion before timeslice 3, task P2 has been promoted to high priority. Schedule B adds the two aperiodic tasks, which arrive at the beginning of timeslices 1 and 2. Part of task A1 is executed as soon as it ar-



Task	Proc.	Length	Period	Deadline	Low Pri.	High Pri.	Promot.
P1	0	2	4	4	0	4	2
P2	1	2	3	3	0	3	1
P3	0	1	3	3	1	3	2
A1	-	2					
A2	-	1					

Figure 3. A sample schedule with three periodic and two aperiodic tasks on a dual MicroBlaze architecture. The status of the queues without and with aperiodic workload is shown respectively in A and B.

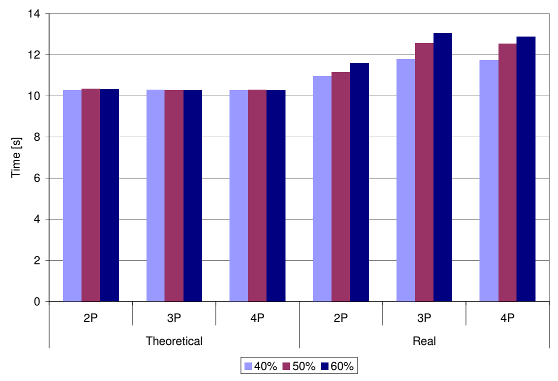


Figure 4. Response time in seconds of an aperiodic task on our system with different periodic utilization and different number of processors.

rives, since P1 in timeslice 1 is in low priority. However, at timeslice 2, P1 gets promoted to its high priority, A1 is interrupted and P1 completed. A2 arrives at timeslice 2 and it is inserted in the queue after A1. So it waits for the completion of the higher priority promoted periodic tasks and the allocation of the remaining part of A1 before starting.

5 Evaluation

To test our dual priority real-time multiprocessor system we adapted the automotive set of the MiBench benchmark suite [13]. MiBench is a collection of commercially representative embedded benchmarks. Among them, we decided to port the automotive set as it is more representative for a real time and reactive systems. In these systems, there are not only sets of critical periodic tasks that must meet their deadline (e.g. to diagnose sensors), but also many aperiodic tasks (like the ones triggered by security systems) that must be served in the best way possible. In this benchmark set there are basically four groups of applications: *basicmath*, which performs simple mathematical calculations not supported by dedicated hardware and can be used to calculate road speed or other vector values (three programs: square roots, first derivative, angle conversion), *bitcount*, which tests bit manipulation abilities of the processors and is linked to sensor activity checking (five different counters), *qsort*, which executes sorting of vectors, useful to organize data and priorities, and finally *susan*, which is an image recognition package that can recognize corners or edges and can smooth an image, useful for quality assurance video systems or car navigation systems. We used a mix of datasets (small and large) for the different benchmarks launched on the system. The small datasets represents the minimum workload for a useful embedded system, the large datasets provides a real world application. We selected the workloads in order to obtain a variable periodic utilization, and measured the response time of an aperiodic task triggered by an interrupt. The benchmarks have been executed varying the number of processors of the target platform. We run a total of 19 tasks on the system, 18 periodic and 1 aperiodic. The aperiodic task is the susan benchmark with the large dataset. This choice is justified by the fact that some of the services performed by susan can be connected to car navigation systems and are triggered by aperiodic interrupts that, for example, can signal the arrival of the image to analyse from the cameras. All the other applications are executed as periodic benchmarks running in parallel on the system with different datasets (small and large). Periodic utilization is determined varying the periods of the applications in accordance to their critical deadline. The worst case response times of the tasks have been determined taking in account an overhead for the context switching and considering the most complex datasets. Promotion time and schedulability have been calculated using the recurrent formula through an in-house tool that takes in input worst case execution times, period and deadlines of the tasks and produces the task tables with processor assignments and all the required information for both our target architecture and the simulator.

Figure 4 shows the average response time of the selected aperiodic task on architectures from 2 to 4 processors, with a periodic utilization of the systems from 40% to 60%. As MPDP adopts a best effort approach to serve aperiodic tasks, utilization near half the capacity of the system is an appropriate balanced choice between exploitation of the system and the responsiveness to external events. The theoretical data for 2, 3, 4 processors architectures are calculated with a simulator that adopts the same approach of the scheduling kernel of the target architecture, considering a small overhead (2%) for context switching and contentions. Scheduling phase is triggered each 0.1 seconds by the system timer. The aperiodic task, on a single processor architecture, should execute in 10.26 seconds with the given dataset at 50 MHz. The data show that, with these utilization, the system is not fully loaded and the algorithm should execute the aperiodic task with very limited response times, almost near the execution time, with all the architectures, with the only overheads of context switching when moving the task on free processors (10.32 seconds in the worst case). However, the results on the prototype show that this assumption is not true. It is easy to see that the real 2 processors architecture is respectively 7%, 8% and 12% slower in response times than the simulated architecture with 40%, 50% and 60% periodic utilization respectively. Aperiodic response times should grow with periodic utilization when the system is heavily loaded: it is clear that context switching and contention, in particular on the shared memory and bus, are important constraints for our architecture. The trend is confirmed when analysing the 3 processors solutions: here the prototype is 15%, 22% and 27% slower than the expected response times. With more processors contention is higher, and the system loses responsiveness. In particular, task switching, with movements of contexts and stacks for many applications from and to shared memory, generates consistent traffic, even with a clever implementation of the algorithm that limits switching only when necessary. Nevertheless, with 4 processors, our prototype gives almost the same results obtained with 3 MicroBlazes, even slightly better. This is a clear indication that the bus and memory access patterns have stabilized, and with these utilization the behavior is almost the same. Note that when using 4 processors, a system utilization of 50% means that the workload is double w.r.t. a system with 2 processors at 50%. This means that, even if the aperiodic response time is worse, the system is anyway doing much more periodic work. On 4 processors, with a 60% workload, our architecture can reach a response time of 12.88 seconds, 25% worse than the optimal response time obtained in simulation, definitely a good result considering the characteristics of the MicroBlaze soft core, the number of tasks running simultaneously on the system and their periods.

6 Conclusions

In this paper we presented a shared memory multiprocessor architecture on FPGA featuring real time support. The system implements the Multiprocessor Dual Priority algorithm that allows management of periodic hard real-time tasks but can serve aperiodic request, supporting reactive applications such those in the automotive and industrial control environments. The thin real time operating system layer is built on top of a multiprocessor interrupt controller that allows distribution of external interrupts to all the processors in the system, inter-processor communication with multicasting and broadcasting and peripherals booking. This hardware device permits to trigger aperiodic tasks while at the same time distributing the scheduling and task assignment phases to the processors. We tested our system with the MiBench standard benchmark set for automotive applications, comparing its results with the response times of aperiodic tasks obtained with pure simulation of the algorithm. We found that, on a realistic architecture, aspects like context switching overheads and contention on the shared memory and bus are a constraining factor that designers should carefully consider when developing a real time system, with response times that can result up to 25% worse than the expected values. This approach could be adopted to evaluate and validate other scheduling algorithms for multiprocessor embedded systems.

References

[1] Mentor Nucleus RTOS. Available at http://www.mentor.com.

[2] Real-Time Operating System for Multiprocessor Systems - RTEMS. Available at http://www.rtems.com/.

[3] WindRiver VxWorks real time operating system. Available at http://www.windriver.com/vxworks/.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. Technical Report UNC-CS TR01016, University of North Carolina at Chapel Hill, 2001.

[5] B. Andersson and J. JONSSON. Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, pages 337–346, 2000.

[6] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[7] J. M. Banús, A. Arenas, and J. Labarta. An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 809–815, 2002.

[8] J. M. Banùs, A. Arenas, and J. Labarta. Dual priority algorithm to schedule real-time tasks in a shared memory multiprocessor. In *International Parallel and Distributed Processing Symposium*, 2003.

[9] R. Davis and A. Wellings. Dual priority scheduling. In *16th IEEE Real-Time Systems Symposium*, pages 100–109, Pisa, 1995.

[10] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 152–161, 1995.

[11] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. In *Tutorial: hard real-time systems*, pages 205–219, Los Alamitos, CA, USA, 1989. IEEE Computer Society Press.

[12] J. Goossens, S. Funk, and S. Baruah. Edf scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations. In *8th International Conference on Real-Time Computing Systems and Applications*, pages 321–330, 2002.

[13] 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 *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[14] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *Transactions on Computers*, 38(8):1110–1123, Aug. 1989.

[15] S. Saez, J. Vila, and A. Crespo. Soft aperiodic task scheduling on hard real-time multiprocessorsystems. In *6th International Conference on Real-Time Computing Systems and Applications*, pages 424–427, Hong Kong, China, 1999.

[16] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, June 1995.

[17] A. Tumeo, M. Branca, L. Camerini, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. An interrupt controller for FPGA-based multiprocessors. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 82–87, Samos, Greece, 2007.