Customizable RTOS to support communication infrastructures and to improve design space exploration in MPSoCs

Alexandra Aguiar alexandra.aguiar@pucrs.br
Sergio Johann sergio.filho@acad.pucrs.br
Felipe Magalhaes felipe.magalhaes@acad.pucrs.br
Fabiano Hessel fabiano.hessel@pucrs.br
Faculty of Informatics - PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil

***Abstract*—Multiprocessed System-on-Chip (MPSoCs) have become a recurrent implementation alternative to modern embedded systems and, lately, have counted on resources previously available only on general purpose machines. In this context, it is possible to highlight that many techniques formerly adopted in general-purpose computers have been studied and adapted to the embedded reality. Thus, embedded communication infrastructures such as buses and networks-on-Chip (NoCs) are based on general-purpose solutions and are widely accepted for embedded systems. Also, embedded systems make use of Operating Systems (OS), as they provide standard interfaces to access hardware resources, including the communication facilities. However, although the underlying communication infrastructure can differ in order to improve a given metric, such as performance, power or area, it is desirable that the software layer remains the same, especially in terms of the application's and OS's code improving the overall software quality. Still, certain OS parameters can directly influence on the overall system performance. This paper presents a highly configurable Real Time OS (RTOS) that implements a communication protocol to provide a transparent communication interface for both bus- and NoC-based MPSoCs' applications.**

I. Introduction

Embedded systems have presented, increasingly, a rising number of features leading to a significant growth in the design complexity of applications. Also, systems have usually had their implementation based in multiple processing elements integrated on the same die, running at a lower clock frequency, due to common energy consumption constraints [1].

Another design challenge of using Multiprocessors System-on-Chip (MPSoC) concerns the way that communications among the internal components are performed. Bus-based systems have been widely used but may suffer from performance penalties whenever the system overly grows in terms of amount of elements [2]. A common way out of this problem is to use NoC-based systems, where higher communication throughput can be achieved.

The communication infrastructure decision can vary according to certain characteristics of the entire platform (hardware and software). In this case, it is desirable that the application remains the same, avoiding code rewriting and design rework, improving the overall software quality. Still, since communication-based decisions impact directly on the final system's behaviour, it is important to have both OS and framework support to explore different decisions in order to enable a wide design space exploration.

Therefore, this paper presents an RTOS that is highly configurable, enabling the OS to be tailored to enhance the overall system performance. Still, we've implemented a communication protocol that provides a transparent communication layer in bus- and NoC-based MPSoCs. We use a framework where new applications can be added and the entire platform can be customized, in terms of OS parameters and communication model. In this case, the application's software can be used in two different communication infrastructures in a straightforward fashion. We evaluate our platform measuring the OS performance and its memory footprint results as well as the overhead of the communication protocol. This paper is an extension of the poster presented in [**?**] where only the communication model was evaluated.

The remainder of this paper is organized as follows. Section II shows some other RTOSs with MPSoC support. Section III depicts our framework design flow for design space exploration. In section IV, details regarding the communication protocol can be found. Following, we have the evaluation and discussions presented in Section V as Section VI concludes the paper with final remarks and future work.

II. Related Work

Many studies discuss the support to specific features of MPSoCs since there is a high dependency on the underlying hardware in such platforms. HeMPS is a homogeneous NoC-based MPSoC platform [3] and uses a MIPS-like processor (Plasma [4]), a local memory (RAM), a DMA controller and a Network Interface (NI) based in the HERMES NoC. It has a preemptive microkernel that provides basic communication primitives used to implement message passing communication. Similar to the HeMPS platform , there is the Open-Scale [5] proposal, where instead of a Plasma processor, a Secretblaze is employed. Still, the Open-Scale platform provides more services in the software level with its RTOS.

We present Table I where we compare existing features in the presented studies against our proposal (HellfireOS, in the table). We also include in this comparison the Plasma RTOS, which is a tiny preemptive RTOS from which the Open-Scale RTOS was extended [5].

978-1-4799-2409-7/13$31.00

TABLE I
OPERATING SYSTEM FEATURES

Feature	Plasma RTOS	Open-Scale	HeMPS	HellfireOS
Supported architectures	MS Windows hosted / Plasma	Secret-Blaze	Plasma	Plasma, MIPS R4K, ARM7, x86
Dynamic Loader	No, single object	Yes, GOT management (code)	Yes	Yes, GOT management (code), stack rellocation (data)
Kernel	Preemptive	Preemptive	Preemptive	Preemptive, cooperative
Scheduling	Priority round robin	Round robin with task credits	Round robin	2-level scheduling (rate monotonic and round robin)
Priority inversion avoidance	No	No	No	Yes, priority inheritance
Memory allocation	Dynamic	Dynamic	Static	Dynamic
Semaphores, mutexes, queues	Supported	Supported	Not supported	Supported
Task communication	Synchronization (intra-core)	Synchonization (intra-core), message passing (extra-core)	Message passing (extra-core)	Syncronization, mailboxes (intra-core) and transparent message-passing (intra and extra-core)
Task migration support	No	Yes, distributed management	Yes, centralized menagement	Yes, distributed management
Additional libraries	Math library	Math library	None	HFLibC (libc and extended FP math)

III. HELLFIRE FRAMEWORK AND DESIGN SPACE EXPLORATION

One of the biggest challenges in embedded systems' design is, in the early stages of their development, to decide among the many design possibilities available. Usually, there are multiple metrics of interest that must be considered, such as timing, resource usage, energy usage and cost as well as multiple design parameters, for instance, the number and type of processing cores, size and organization of memories, interconnect options, scheduling policies, among others.

Still, it is very difficult to stablish a relationship between design choices and metrics of interest, since typical systems' aspects such as concurrency, dynamic application behaviour, and resource sharing need all to be considered at the same time. Besides, existing modelling approaches and tools usually require the system to be described in different models and languages throughout its design, leading to a model continuity problem.

We use the Hellfire Framework (HellfireFW), firstly introduced in [6], as an alternative approach to this problem. HellfireFW allows a complete deployment and test of multiprocessed critical and non-critical embedded applications, defining the HW/SW architecture to be employed by the designer. The design flow implemented by the HellfireFW is presented in Figure 1. HellfireFW assumes the HellfireOS (HFOS) [6] as the system's kernel and is detailed throughout this section.

We assume the entry point of the design to be an application, initially manually divided into a set of tasks, described in C language. Each task, besides the code implementation itself and its data, must be represented, at least, by the following n-uple $\tau_i =< id_i, uid_i, p_i, e_i, d_i, lc_i >$where, id_i concerns its local identity, uid_i its unique identity, p_i its period; e_i its execution time (since we assume a real-time system, this usually concerns the task's Worst-Case Execution Time - WCET); d_i its deadline; lc_i is the data volume generated by message traffic.

The p_i, e_i and d_i parameters are controlled by a given real-time scheduling algorithm and must be described in an abstract time unit, known in HellfireOS as *ticks*[1]. Some of these scheduling policies are described in [7], [8], [9], [10], [11], [12].

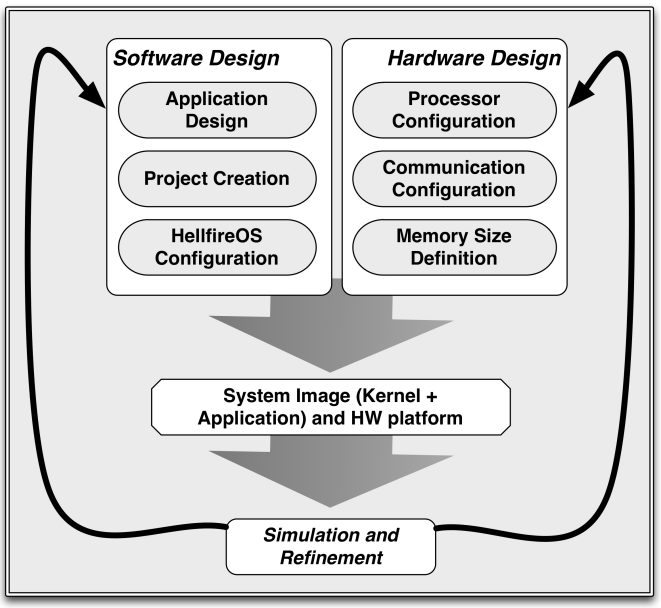

Fig. 1. Hellfire Framework Design Flow

After designing the application, the HellfireFW project must be created. This is the step where the initial HW/SW platform configuration is defined. The C application is executed on the top of the HellfireOS stack. HellfireOS is a micro-kernel based Real-time Operating System - RTOS, highly configurable and easily portable. To ease the OS port to other architectures, HellfireOS uses a modular structure as depicted in Figure 2.

All hardware specific functions and definitions are implemented on the Hardware Abstraction Layer (HAL), which is unique for a specific hardware platform solution, simplifying the port of the kernel onto different platforms. The micro-kernel itself is implemented on top of this layer. Features like standard C functions and the kernel Application Programming Interface (API) are implemented on top of the micro-kernel. Communication and migration drivers, memory management and mutual exclusion facilities are implemented on top of the kernel API and the user application is the highest level in the HellfireOS development stack.

After following these steps, an MPSoC platform configuration is performed, with a given number of processors, a personalized instance of HellfireOS on each processor and a static task mapping[2]. The user must then trigger the simulation of the system, which runs for a given time window and generates several graphical results for the designer to analyse.

[1] *Tick* is the minimum scheduling unit. All real-time parameters are represented in ticks and the tick itself can be associated to a given and known time measure, such as 10ms, for instance.

[2] Initial mapping defined at design time. At runtime tasks may be reallocated to different processors.

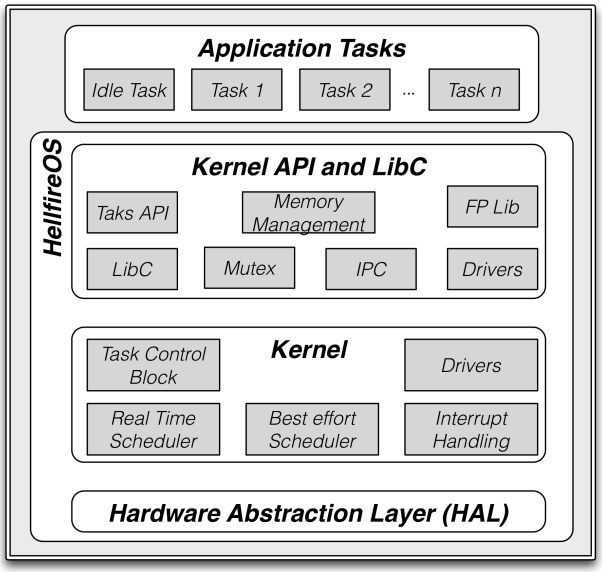

Fig. 2. HellfireOS Structure Stack

If the results are satisfactory, the SW part of the platform can be transported to a prototype, such as an FPGA, in a straightforward fashion. Otherwise, the designer can change the HW/SW settings and rerun the simulation as long as refinements are needed.

IV. Communication model for bus- and NoC- based MPSoCs

This section presents the communication model we adopted and its implementation targeting communicating tasks in bus- and NoC-based MPSoCs based on the HellfireFW.

A. Hardware Aspects

We assume an MPSoC implemented with either bus or NoC communication. Following, there are some details regarding how they were implemented.

Bus-based MPSoC. We implemented a bus where a bus arbiter with a rotative algorithm was used. In this case, we have IP units that are connected to the bus through wrappers. The central bus control is performed by the bus-arbiter, which receives access requirements from the wrappers. We adopted a centralized arbiter as it simplifies the choice of which processor has the right of using the bus, thus, decreasing implicit overheads. Wrappers are used to interconnect the IP to the communication channel and to control data access into the bus.

Network-on-Chip - NoC. For our NoC implementation we use the HERMES NoC project [13], which uses a mesh topology and it is composed by routers, buffers and controllers (switch control). Still, the internal queue scheduling uses the Priority Round Robin algorithm. The packet routing algorithm used is XY routing [14].

B. Software Aspects

Communication related issues. We use OS primitives to perform the message exchange, such as *HF_Send()*, *HF_Receive()* and *HF_UniqueIDSend()*. It is a developer's responsibility to allocate proper buffers at the application level and to specify the unique task identification correctly. Figure 3 shows an example of a task code that uses send and receive primitives. This code does not need any kind of change for both bus- and NoC-based MPSoCs as the API is completely transparent.

```
void task(void){
    uint16_t cpu, size, i;
    uint8_t task;
    uint8_t buf[200];

    for(;;){
        HF_Receive(&cpu, &task, buf, &size);
        printf("\n%s", buf);
        HF_Send(HF_Core(4), 3, "three", 150);
    }
}
```

Fig. 3. Communication system calls in task implementation

These primitives are implemented following the producer/consumer reference model and each task has a circular reception queue, with configurable size to hold incoming packets. During the receiving process, whenever the receiving queue is empty, the task is either blocked (in case of a blocking primitive) or kept in the primitive until a *timeout* occurs. Likewise, during the sending process, the task can be blocked whenever network contentions become a reality. Still, if the receiving task has no extra room in its receiving queue, packets are then simply discarded. We had to adopt the discard strategy since there is only a single hardware queue per processor and, usually, more than one task per processor. Therefore, if a given task is not treating its received packets, only its packets are discarded aiming not to compromise other tasks' reception process.

Additionally, specific tasks, such as to put packets into reception queues, to block or to release tasks in a given moment, are all managed by the drivers of the operating system. Figure 4 presents the steps needed to perform a message exchange between two processors in bus- and NoC-based environments.

For both cases, the sending primitive encapsulates the message into packets, fulfilling the task's sending queue (1). The packet is then removed from this queue and it is then copied to the hardware outgoing queue (2). The network interface in then notified as the packet is sent through the network (3). After the packet reaches its destination processor by getting into the incoming queue, an interrupt is sent to the operating system, which removes the packet from the queue and decodes it, before forwarding it to the target task's queue (4). Finally, the message can be used at the application level (5).

Our communication primitives were implemented in two different levels of abstraction. High level primitives are exposed in the OS's API and are responsible for encapsulating and dividing messages into data packets, dealing with details

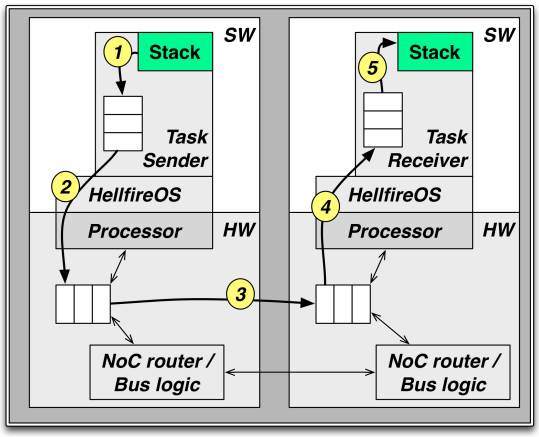

Fig. 4. Communication among application tasks, hardware and software queues

such as padding and sequencing. Internally, system's drivers are responsible for transferring packets between source and target. They work with fixed-sized packets and signal the communication network interface while sending data to hardware queues. Likewise, these drivers are signalled while receiving data, as they take the data off the hardware receiving queues and copy them to the circular packet queues of each task.

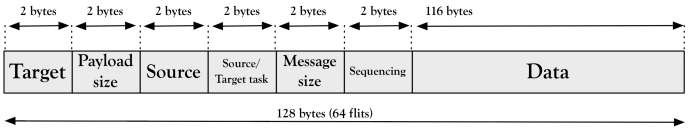

Fig. 5. Packet's structure

We present the packet's structure in Figure 5. Packets start with a header where the target's address is put and size of the payload[3] is placed. After the header, there is an identification of the source node, another identification concerning the source task and a last one about the destiny or target task. Still, information as the message size (which can be larger than the packet size[4]), a sequence number and payload data that will be put in the task's software queue, compose the remainder of the packet. Thus, packet data which are placed after the payload are managed by the OS's communication driver as its content is not relevant to the communication mean anyway.

This packet's structure causes a communication overhead of about 9.37% (in a 64-flit[5] packet, with 16 bits each) due to header's information. The packet's size can be altered but the hardware queues size and OS's configuration must follow that change (it is not transparent nor automatic). We adopted this packet size after performing several tests analysing the trade-off among queues size, packet's processing time and padding overhead. Figure 6 presents the peak performance of a data transfer at the application level for different hardware queues' sizes. From these tests we've assumed an ideal situation, where a unique message of each size is sent and real-time tasks, for sending and receiving, use all processing capacity of two neighbour processors to perform the communication.

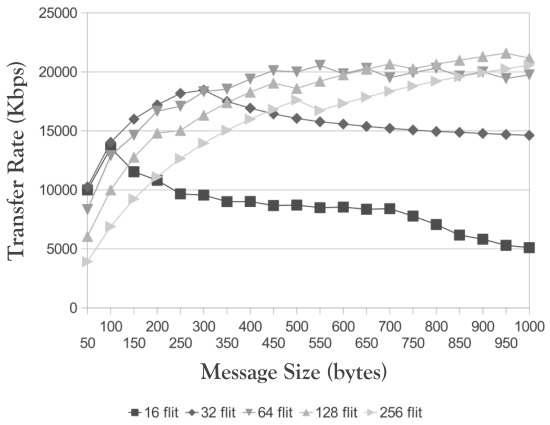

Fig. 6. Hardware queue's size, peak performance for two nodes message exchange

V. Evaluation and Discussions

A. Implementation and Validation Methodology

We used an architecture simulator based on [15] to perform our experiments. This simulator allows the use of several processing nodes [6], formed by MIPS-like processors. The simulator implements the MPSoCs presented in Section IV-A, and it has a very close precision to the hardware prototype.

On all experiments, the task stack size is 2KB, the scheduling policy is Rate Monotonic, the operating core frequency for all nodes is 25MHz (unless otherwise specified) and the time between interrupts (*tick* time) is 10.48ms. MPSoC size is variable and each core has a 512KB shared data and instruction memory. Packet size is 64 flits on all tests and the receiving software queue has 16 slots for each core. Operating system and application code were size optimized and compiled with GCC 4.6.0 port to the MIPS architecture.

B. Experimental Results

1) Operating System Performance: Table II presents the latency of some system calls and HellfireOS events. The latency of task management syscalls (HF_AddTask(), HF_AddPeriodicTask(), HF_BlockTask(), HF_ResumeTask() and HF_KillTask()) is constant, independent of the number of tasks and task set parameters. Other system calls such as HF_Calloc(), HF_Malloc(), HF_Realloc() and HF_Fork() depend on the amount of memory specified and task stack size. Message passing primitives have a performance closely related to task parameters and message size, being this fact detailed on Section V-B3.

[3]Useful load of the data packet.

[4]Packets have a fixed size.

[5]We adopt a *flit* as the minimum transfer unit for the protocol in both bus- and NoC-based networks

[6]The number of nodes is parametrizable. Currently, up to 256 nodes in a mesh-based NoC can be simulated.

TABLE II
OVERHEAD FOR COMMON SYSTEM CALLS AND EVENTS

Syscall / Event	Cycles	Time
Context switch (10 tasks)	1563	62us
Network ISR (64 *flit* packet)	1079	43us
HF_Send() (512 byte message)	29440	1177us
HF_Receive() (512 byte message)	23078	923us
HF_AddPeriodicTask()	2842	114us
HF_BlockTask()	88	<4us
HF_ResumeTask()	84	<4us
HF_Fork()	49107	1964us
HF_KillTask()	3087	123us
HF_ChangeTaskParameters()	122	5us
HF_CurrentTaskId()	9	<1us
HF_Malloc() (4kB)	754	30us
HF_Free() (4kB)	727	29us
HF_Calloc() (4kB)	21302	852us
HF_Realloc() (4kB to 8kB)	10760	430us

The architecture used in this work uses a shared code and data memory, so system calls such as HF_Calloc() are costly. Also, no MMU was used so the *copy-on-write* technique can not be used on the HF_Fork() system call. As an alternative, all the task's data must be replicated, as the parent task can not neither have its data modified nor be blocked until its child finishes.

Table III presents two benchmarks of the ParMiBench Suite [16]. These benchmarks were ported to HellfireOS, and as the overhead is being measured here, tasks were configured as best effort. For the SHA benchmark, the hash of four messages of 2KB each is calculated, using a single core (threaded benchmark) and 5 cores (one master and four slaves) in a 3x2 mesh MPSoC. In the multicore version, one processor distributes the messages to slave processors that calculate the message hash and send it back to the master. The Bitcount benchmark is arranged in a similar way, but 512 byte message is divided into four pieces and only 128 bytes are sent for each slave. The process is repeated for four different bit counting algorithms from the benchmark.

TABLE III
BENCHMARKS PERFORMANCE

	Multithreaded, single core	Multithreaded, 5 cores
SHA	1310978	542817
Bitcount	1421037	1037653

Although message passing impacts on system overhead, the SHA algorithm executed 2.41 times faster on the distributed version. The execution time drops from 52.44ms to 21.71ms, and 8 messages are exchanged between the master processor and slaves. On the single core version messages are directly exchanged between threads using shared memory. The second benchmark has a greater communication overhead (due to very small messages) and doesn't scale well. The distributed version executed 1.37 times faster than the single processor version, and 32 messages are exchanged between the master processor and slaves. The execution time drops from 56.84ms to 41.51ms. It is possible to observe that when large messages are sent among cores, the message passing overhead can be amortized.

2) Memory Footprint: Different configuration parameters impact on both functionality and kernel memory footprint, as shown in Table IV[7]. The kernel can be configured for several parameters, including hardware-related aspects, such as the presence of a multiply/divide unit.

TABLE IV
HELLFIREOS KERNEL SIZE

	Kernel Size (with HW mul/div)	Kernel Size (w/o HW mul/div)
Kernel A (no task reports, no FP math, no migration, no management)	21.06KB	22.52KB
Kernel B (no FP math, no migration, no management)	21.31KB	23.74KB
Kernel C (no migration, no management)	33.77KB	35.39KB
Kernel D (no task migration management)	40.15KB	42.16KB
Kernel E (full kernel)	44.44KB	46.61KB

Kernel data is not taken into account, as it is allocated dynamically from the memory pool at runtime. Also, the application code size is not taken into account either.

3) Communication: In order to evaluate the message-passing performance at the application level a simple yet highly communicating application was implemented, varying the message size from 50 bytes to 1000 bytes. Half the processors contain senders, and the other half, receivers. All tasks were configured to execute each 104ms (period) during 82ms (capacity) through the period yielding an 80% processor utilization. Along with the application task, other operating system tasks execute at the same time on the same core. The same application was used in several configurations, where it was only scaled for the desired number of cores (the same code was recompiled without modifications).

Figure 7 presents the results observed for several MPSoC configurations. As it can be seen, bus-based communications for this application start to decrease its performance due to network congestions when a higher number of processors communicates. The last simulated case was a 256 core, 16x16 mesh MPSoC and the NoC could keep the high throughput, avoiding congestions. A 256 core bus-based MPSoC suffers from high penalty for this application, highlighting the superior performance of the NoC approach for communication-intensive applications. In cases where performance is closely related to communication efficiency (such as in streaming applications), the proposed model offer a scalable and manageable way of describing distributed applications.

[7]On all experiments, code was optimized for size on the compiler.

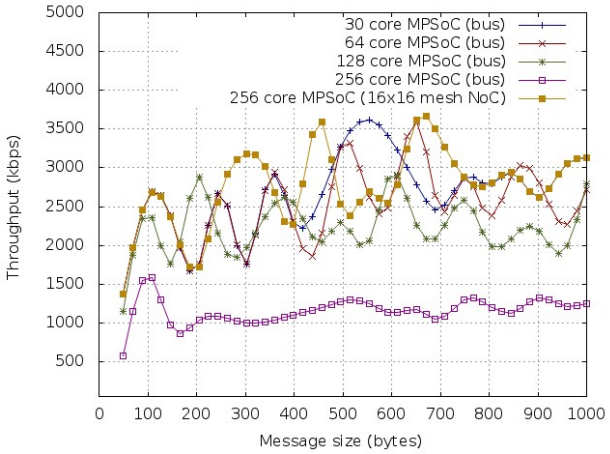

Fig. 7. Throughput for bus and NoC-based MPSoCs

4) Discussion: The proposed design flow in Section III and the implementation strategies of HellfireOS, presented in Section IV-B can be combined, enabling a very powerful design space exploration tool. Designers do not need to change neither the application nor the OS's code to test their systems under various different scenarios.

The scenarios to be explored concern: (i) different application strategies in multiprocessed systems (using the same communication API, only changing the application's logic itself); (ii) different OS configuration (heap size, tasks' memory use, scheduling options, etc); (iii) different processor options (architecture, type, frequency); (iv) different memory sizes, and; (v) different communication strategies (bus or NoC) and, for each, its own specific parameters.

Each execution on such a rich simulation environment allows the designer to evaluate the best possible scenario for the multiprocessed system. Besides, if there is an equivalent hardware platform described, for example, in VHDL, it is possible to use the application and OS binaries in a straightforward fashion, preventing the occurrence of the model continuity problem.

VI. Conclusion

Current multiprocessed embedded systems can be difficult to manage as the number of processing nodes and application complexity increase. Still, many applications demand predictable response to real-time events, so new approaches must be used on the software level to improve performance and programmability. This paper presents an efficient and configurable RTOS with support to bus- and Noc-based MPSoCs, following the Hellfire Framework design flow for better design space exploration. We showed a comparison between our approach and other related works and evaluated the OS efficient communication support. Future work include the extension of the proposed task model to heterogeneous architectures and the use of automatic partitioning and mapping tools.

References

[1] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4167779

[2] T. Le and M. Khalid, "NoC prototyping on FPGAs: A case study using an image processing benchmark," jun. 2009, pp. 441–445.

[3] E. Carara, R. de Oliveira, N. Calazans, and F. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pp. 1345–1348, 2009.

[4] S. Rhoads, "Plasma - most MIPS I(TM) opcodes," Accessed in http://opencores.org/, 2011.

[5] R. Busseuil, L. Barthe, G. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, and L. Torres, "Open-scale: A scalable, open-source noc-based mpsoc for design space exploration," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 30 2011-dec. 2 2011, pp. 357 –362.

[6] "Omitted for blind review."

[7] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[8] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behaviour," *IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.

[9] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, 1982.

[10] W. H. Hesselink and R. M. Tol, "Formal feasibility conditions for earliest deadline first scheduling," Tech. Rep., 1994.

[11] M. Andrews, "Probabilistic end-to-end delay bounds for earliest deadline first scheduling," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, 2000, pp. 603–612.

[12] U. Schwiegelshohn and R. Yahyapour, "Analysis of first-come-first-serve parallel job scheduling," in *In Proceedings of the 9th SIAM Symposium on Discrete Algorithms*, 1998, pp. 629–638.

[13] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *Integr. VLSI J.*, vol. 38, no. 1, pp. 69–93, 2004.

[14] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[15] "Omitted for blind review."

[16] S. Iqbal, Y. Liang, and H. Grahn, "Parmibench - an open-source benchmark for embedded multiprocessor systems," *Computer Architecture Letters*, vol. 9, no. 2, pp. 45 –48, feb. 2010.