Hellfire: A Design Framework for Critical Embedded Systems' Applications

Alexandra Aguiar, Sérgio J. Filho,
Felipe G. Magalhães, Thiago D. Casagrande, Fabiano Hessel

Faculty of Informatics – PUCRS

Hellfire framework (HellfireFW) presents a design flow for the design of MPSoC based critical embedded systems. Health-care electronics, security equipment and space aircraft are examples of such systems that, besides presenting typical embedded system's constraints, bring new design challenges as their restrictions are even tighter in terms of area, power consumption and high-performance in distributed computing involving real-time processing requirement. In this paper, we present the Hellfire framework, which offers an integrated tool-flow in which design space exploration (DSE), OS customization and static and dynamic application mapping are highly automated. The designer can develop embedded sequential and parallel applications while evaluating how design decisions impact in overall system behavior, in terms of static and dynamic task mapping, performance, deadline miss ratio, communication traffic and energy consumption. Results show that: i) our solution is suitable for hard real-time critical embedded systems, in terms of real-time scheduling and OS overhead; ii) an accurate analysis of critical embedded applications in terms of deadline miss ratio can be done using HellfireFW; iii) designer can better decide which architecture is more suitable for the application; iv) different HW/SW solutions by configuring both the RTOS and the HW platform can be simulated.

Laboratory:
GSE

Publication:
ISQED 2010

Web:
http://www.inct-sec.org/papers/

Funding:
FAPESP, CNPq, CAPES,
Ministério da Ciência e Tencologia

Hellfire: A Design Framework for Critical Embedded Systems' Applications

Alexandra Aguiar, Sérgio J. Filho, Felipe G. Magalhães, Thiago D. Casagrande, Fabiano Hessel

Faculty of Informatics – PUCRS – Av. Ipiranga 6681, Porto Alegre, Brazil
E-mail: alexandra.aguiar@pucrs.br, fsergio.johann, felipe.magalhaes, thiago.casagrandeg@acad.pucrs.br, fabiano.hessel@pucrs.br

Abstract — Hellfire framework (HellfireFW) presents a design flow for the design of MPSoC based critical embedded systems. Health-care electronics, security equipment and space aircraft are examples of such systems that, besides presenting typical embedded system's constraints, bring new design challenges as their restrictions are even tighter in terms of area, power consumption and high-performance in distributed computing involving real-time processing requirement. In this paper, we present the Hellfire framework, which offers an integrated tool-flow in which design space exploration (DSE), OS customization and static and dynamic application mapping are highly automated. The designer can develop embedded sequential and parallel applications while evaluating how design decisions impact in overall system behavior, in terms of static and dynamic task mapping, performance, deadline miss ratio, communication traffic and energy consumption. Results show that: i) our solution is suitable for hard real-time critical embedded systems, in terms of real-time scheduling and OS overhead; ii) an accurate analysis of critical embedded applications in terms of deadline miss ratio can be done using HellfireFW; iii) designer can better decide which architecture is more suitable for the application; iv) different HW/SW solutions by configuring both the RTOS and the HW platform can be simulated.

Keywords — MPSoC, Embedded Systems Design, HW/SW Co-design

Hellfire: A Design Framework for Critical Embedded Systems' Applications

Alexandra Aguiar, Sérgio J. Filho, Felipe G. Magalhães, Thiago D. Casagrande, Fabiano Hessel
Faculty of Informatics – PUCRS – Av. Ipiranga 6681, Porto Alegre, Brazil
E-mail: alexandra.aguiar@pucrs.br, {sergio.johann, felipe.magalhaes, thiago.casagrande}@acad.pucrs.br, fabiano.hessel@pucrs.br

Abstract— Hellfire framework (HellfireFW) presents a design flow for the design of MPSoC based critical embedded systems. Health-care electronics, security equipment and space aircraft are examples of such systems that, besides presenting typical embedded system's constraints, bring new design challenges as their restrictions are even tighter in terms of area, power consumption and high-performance in distributed computing involving real-time processing requirement. In this paper, we present the Hellfire framework, which offers an integrated tool-flow in which design space exploration (DSE), OS customization and static and dynamic application mapping are highly automated. The designer can develop embedded sequential and parallel applications while evaluating how design decisions impact in overall system behavior, in terms of static and dynamic task mapping, performance, deadline miss ratio, communication traffic and energy consumption. Results show that: *i)* our solution is suitable for hard real-time critical embedded systems, in terms of real-time scheduling and OS overhead; *ii)* an accurate analysis of critical embedded applications in terms of deadline miss ratio can be done using HellfireFW; *iii)* designer can better decide which architecture is more suitable for the application; *iv)* different HW/SW solutions by configuring both the RTOS and the HW platform can be simulated.

Keywords— MPSoC, Embedded Systems Design, HW/SW Co-design

I. Introduction

Embedded systems have increasingly become part of people's lives being present in day-by-day life issues, like healthcare electronics, automotive industry and entertainment devices. Nevertheless, they are also present in advanced matters, within the robots field, space applications and manned and unmanned aircraft, where a given failure may cause serious damage to the environment, be the major cause of millions of human lives' losses, cause the loss or severe damage to expensive equipment or be the cause of large, nonrecoverable financial losses. These systems are known as *Critical Embedded Systems* and their tasks' execution often imposes timing restrictions.

Multiprocessors System-on-Chip (MPSoC) have become each more a viable choice to implement critical and non-critical embedded systems [1]. This trend combined with the growing complexity of *real-time* (RT) constrained embedded systems, has a major impact in the entire design and implementation of critical embedded systems.

Commonly, the design of timing constrained MPSoCs with short energy consumption budgets and strict time-to-market prevents the software deployment and test on the target hardware architecture since this usually is a difficult and time consuming approach. Ergo, it is needed alternative methodologies where the final hardware prototype has equivalence of the code with system's hardware and software simulator and emulator.

In this paper, we present the Hellfire Framework (HellfireFW), an environment for the design of critical embedded systems' applications. HellfireFW follows its own design flow where an MPSoC emulator is employed and over which a specific RTOS solution - the HellfireOS - is executed. The designer can validate the system in several aspects, such as overall performance, deadline miss ratio, energy consumption and dynamic task mapping (through task migration). Details regarding to the migration scheme are not part of this paper's scope whereas a more detailed description of the MPSoC emulator is presented in [2]. Still, HellfireFW improves the overall system quality, since it helps the designer to perform design space exploration through the customization and simulation of different HW/SW scenarios.

The main contributions of Hellfire framework include:

- simple deployment and test of critical embedded software;
- design space exploration in terms of HW and SW configurations;
- an integrated tool where it is possible to configure RTOS and processor's features and to analyze the system behavior based in automated results generated by debug facilities of the system.

The remainder of the paper is organized as it follows. The next section shows some related work. Section **3** presents the framework that includes several components needed to create a complete critical embedded MPSoC followed by Section **4**, where the results taken are shown. Finally, Section **5** concludes the paper and presents some future work.

II. Related Work

Co-design is an ideal way to explore the design space and to create a suitable platform architecture [3]. It usually starts at system level, in which the final system division into hardware and software have not been completed yet. After the initial steps, the functional verification can be made through co-simulation mechanisms which help in both system's validation and refinement.

Usual critical embedded systems' constraints, such as a tight time-to-market and the high complexity of current designs, lead to simulators in different abstraction levels, from Instruction Set Simulators (ISS) to system-level descriptions, usually implemented in SystemC or SpecC. In these simulators, usually both hardware and software models are executed in order to analyze the system behavior at an earlier development stage.

In this context, several works [4], [5], [6] and [7] show some effort in modeling the RTOS at a higher level of abstraction. The main issue with this approach is that when the design is refined,

although the RTOS model can be translated automatically into software services, usually a widely adopted RTOS is preferred, which prevents the results taken during the modeling to be accurate when the real hardware implementation is finished.

[8] present the *code equivalence* problem, that is, if the code executed by the virtual system is different from the target hardware code, the system's behavior can be different from what was simulated. Still wrong design choices can be made. Moreover, when using high level simulation approaches, usually the system timing behavior is either not taken into account or is not completely accurate [9]. Simulators where timing of the target architecture is not accurately replicated are not allowed to precisely model critical and parallel embedded systems, as MPSoCs.

These researches are valid since they help the designer to refine the HW/SW partition. Unfortunately, this validation is limited for the code equivalence problem and for restrictions of the assessment of the system timing properties. As opposed to some works shown previously, we use ISSs in our study. Although it represents a lower simulation speed, it allows the use of the same application code both in the simulation level and in the target architecture, thus dealing with the code equivalence problem.

Besides the MPSoC ISS simulator, another main structure of the proposed framework is the RTOS, since depending on the chosen option the system can be more or less suitable for critical embedded systems. Therefore, we researched several RTOS's and compared them with our own solution, the HellfireOS.

The MicroC/OS-II [10] is a highly portable, real-time and multitasking kernel and it can manage up to 255 tasks and provides several services, such as semaphores and mutual exclusion semaphores. Although this OS has many ports to different architectures, it lacks of multi-processor support.

The AMX [11] real-time operating system has been recognized as an excellent real-time operating system, which meets the critical needs of the most challenging real-time applications. Although it supports nested interrupts with priority ordering and message passing with configurable message length it does not have a hardware test facility, neither inter-processor communications.

CMX [12] is a real-time multi-tasking operating system for microprocessors, microcomputers and DSPs and it has several features such as, task, event and message management, semaphores, among others. Target architecture specific coding makes this operating system less portable that the other solutions.

The Integrity [13] operating system is targeted to high-end embedded products which support hardware memory protection. This OS is mainly used in avionic systems and task isolation from kernel space provides a stable system execution. Its object-oriented design allows strict access control and verification of security and integrity of data, communications, and individual components.

The MQX [14] real-time operating system was specifically developed for deeply embedded applications. It provides real-time performance within a tiny memory footprint and was designed to be easily configured to balance code size with performance requirements. Alternative settings can be selected, benchmarked and iterated to optimize cost and performance.

HellfireOS, developed to compose the Hellfire Framework described in this paper, besides offering basic OS features, supports *multi-processors* along with dynamic task mapping through *task migration*. Task migration can be either *explicit*, the designer is responsible for inserting *migration points* in the application code, or *implicit*, an automated mechanism which monitors system load sends migration orders to the processors. Usually, the main benefits achieved by task migration are: *i)* fair balance of system's workload; *ii)* a decrease in energy consumption and; *iii)* support of fault tolerant techniques. Moreover, HellfireOS is *application oriented*: only the modules needed for a given application are included in the final binary image, reducing its total size. Still, HellfireOS is open source, making it cost attractive compared to commercial solutions.

HellfireOS is a portable micro-kernel based implementation [15], [16], [17] composed of several modules. The kernel can be configured in several different features, such as the maximum number of tasks on a processor and the task stack size, among others. The idea is to allow the designer to optimize the final kernel image size according to software application and system constraints. Thus, we allow the designer to let the system more suitable to run on low memory constrained architectures, like typical critical embedded systems.

A comparison among these OS's is summarized in Table I, where some crucial features that must be supported in critical embedded systems' MPSoCs are highlighted.

TABLE I
COMPARISON AMONG RESEARCHED OS'S

	MP Support	Task Migration	License
MicroC / OSII	No	No	Commercial
AMX	No	No	Commercial
CMX	No	No	Commercial
Integrity	Yes	No	Commercial
MQX	No	No	Commercial
HellfireOS	Yes	Explicit and Implicit	Free

III. HELLFIRE FRAMEWORK

The Hellfire Framework (HellfireFW) allows a complete deployment and test of parallel critical embedded applications, defining the HW/SW architecture to be employed by the designer. The HellfireFW design flow is presented in Figure 1.

Application Design. The entry point of the design flow is an application implemented in C language, manually divided into a set of tasks. A task τ_i is defined as a n-uple $(id_i, r_i, WCET_i, D_i, P_i)$ and the parameters stand for identification, release time, worst case execution time, deadline and period of task τ_i, respectively.

Regarding to tasks' implementation, HellfireOS (described in Section **2**) supports applications composed by hard and soft real-time periodic tasks and *best effort* tasks and the parameters defined by our task model must be included during the insertion of a task in the system, by using a specific HellfireOS system

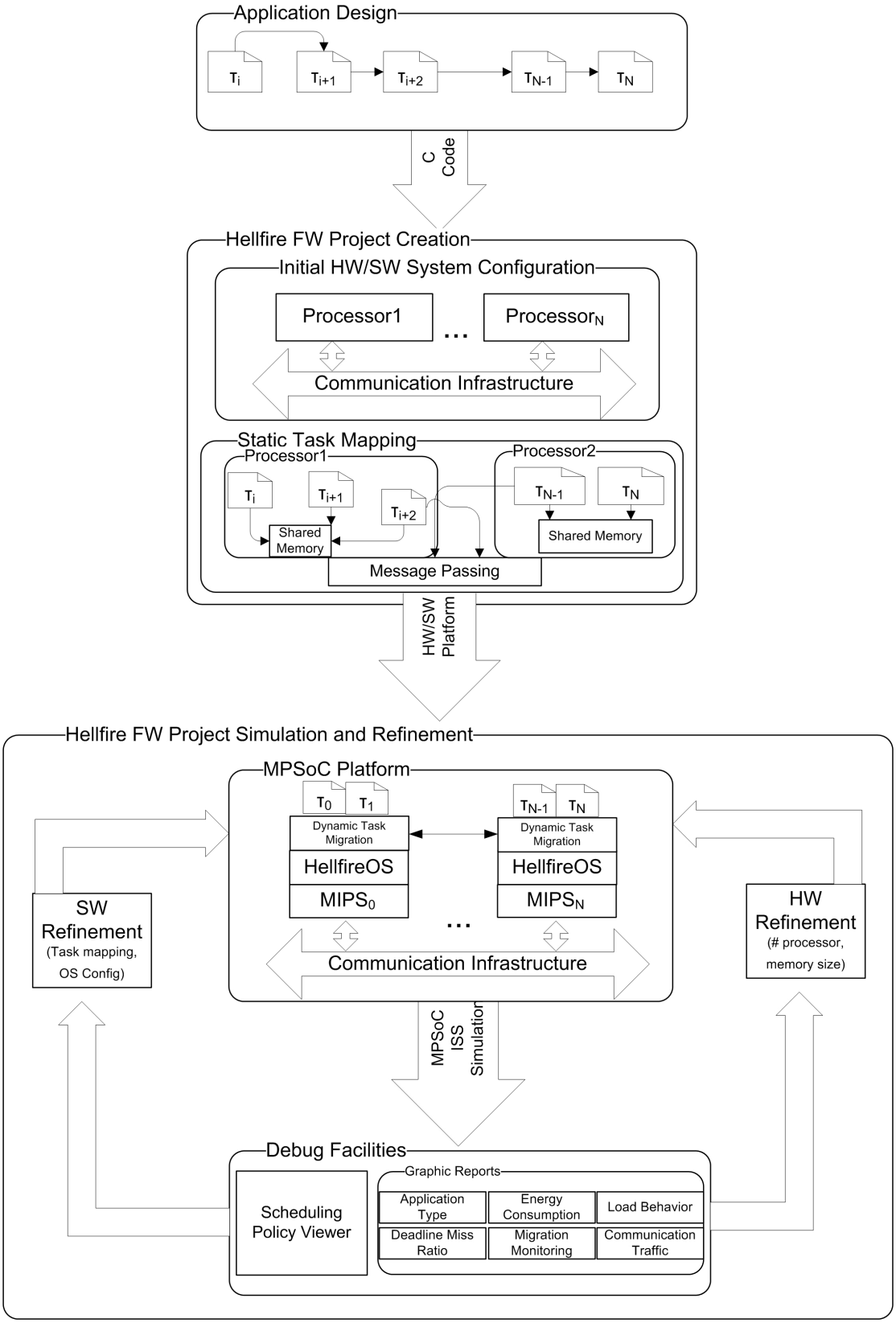

Fig. 1. Critical Embedded System's Design Flow with Hellfire Framework

call. The task behavior is defined as a block of C code. Figure 2 shows an example of a task in HellfireOS.

The tasks can communicate through either shared memory (in the same processor) or message passing (in different processors). The output expected in this phase is a C code with the tasks that represent the application.

```
6   void task(void){
7       while (1) {
8           /*...
9           Task behavior description
10          ...*/
11      }
12  }
13
14  void ApplicationMain(void){
15      OS_AddPeriodicTask(task, "Task", 0, 5, 8, 8);
16      //parameters: C function with task behavior
17      //and Task Model: idi, ri, WCETi, Di, Pi
18  }
```

Fig. 2. Task Implementation Example in HellfireOS

HellfireFW Project Creation. After the application design, the designer must create a project in HellfireFW. In this step, the initial HW/SW platform is defined, along with features as number of processors, memory size and HellfireOS specific configurations. Also, the static task mapping is done based on designer's experience as tasks are distributed among the processors. Possible configurations for each processor can be seen in Figure 3 and include: memory settings, general definitions, core frequency, scheduling policy (which allows the insertion of new scheduling policies developed by the designer) and application source file selection.

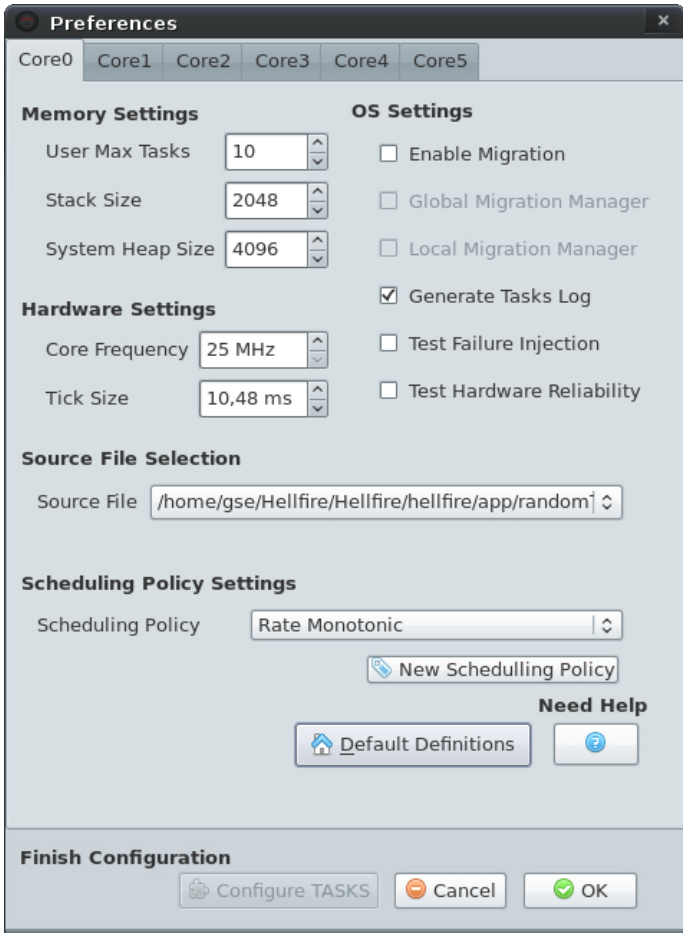

Fig. 3. HellfireFW Configuration Window Screenshot

In order to ease the OS port to other architectures, we developed HellfireOS as a modular OS, which structure is depicted in Figure 4. All hardware specific functions and definitions are implemented on the 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 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.

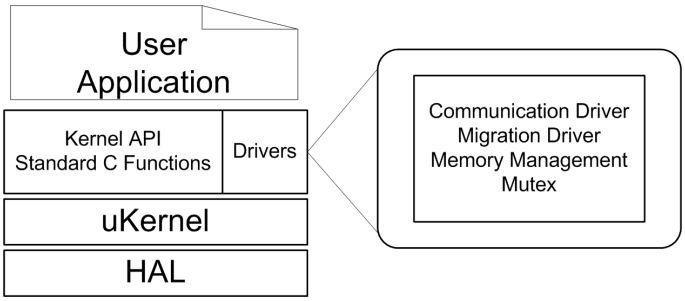

Fig. 4. HellfireOS Structure Stack

As the output of this step, is expected a MPSoC platform with a given number of processors, a personalized instance of HellfireOS on each processor and a static task mapping. The user must then trigger the simulation of the system.

HellfireFW Simulation and Refinement[1]**.** When the designer triggers a simulation, the MPSoC ISS Simulator is activated. Although HellfireFW is not dependent on a single hardware architecture, for validation purposes, a MIPS-like [18] ISS, with support to message passing, was implemented. As stated before, while the ISS is slower when compared to higher level simulators it is free from the *code equivalence* problem.

After simulation is completed, several debug facilities can help the designer to refine and adjust the HW/SW platform. These facilities allow the following analysis to be done:

- **Scheduling policy viewer.** Helps the designer to graphically evaluate whether the chosen scheduling policy is appropriate for the application. Figure 5 shows a screenshot of this viewer. In this example, the *Rate Monotonic* scheduling policy was used in a single processor *p0* where three periodic tasks following the HellfireOS task model must be executed: $(\tau_1, 0, 3, 20, 20)$, $(\tau_2, 0, 2, 5, 5)$, $(\tau_3, 0, 2, 10, 10)$. The scheduling window shows the tasks throughout the system execution time. Control buttons are present below the scheduling window and allow the designer to follow the tasks' scheduling during the simulation time. The bottom window, named *Processor Status*, shows each processor behavior in terms of *load* (in percentual), *memory use* (in Kilobytes) and *energy consumption* (in microJoules), which are generated values from the MPSoC ISS Simulator.
- **Graphic reports.** Another debug facility presented by HellfireFW, is the automatic generation of several graphic reports,

[1]Refinement in the HellfireFW are configuration changes performed in the HW/SW platform.

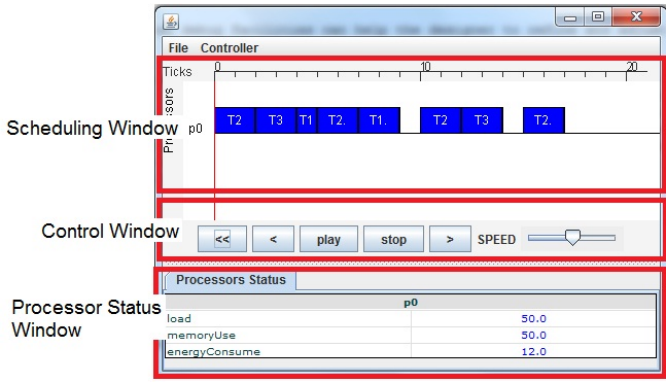

Fig. 5. Scheduling policy viewer

which can be selected by the framework menu, as depicted in Figure 6. The graphics available are:

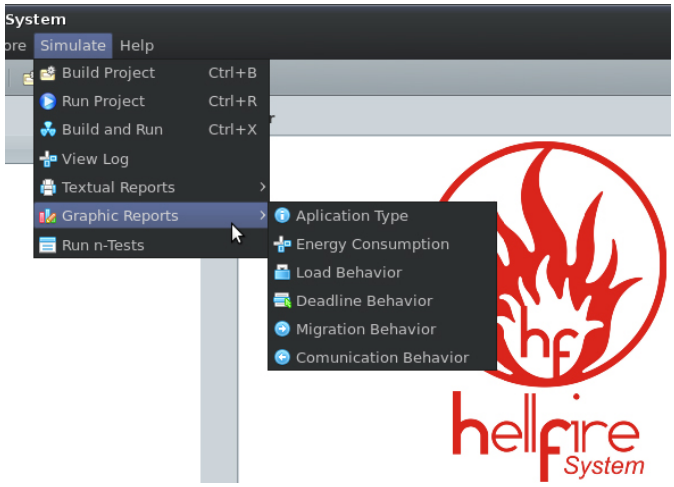

Fig. 6. Graphic reports available in HellfireFW

1. *application type*, where a histogram shows the timing characteristics of each processor's task set. In this graphic, the amount of hard real-time, soft real-time and best effort tasks per processor can be seen;
2. *energy consumption*, where the final energy consumption estimative of each processor in unified in a single graphic, for comparison purposes;
3. *load behavior*, where the load behavior of each processor throughout the system's life time is shown. This feature is used, mainly, to decide whether the static mapping is satisfactory or not;
4. *deadline miss ratio*, where the deadline miss ratio per processor throughout the system's life time is presented. This analysis helps the designer to decide about the scheduling policy and task mapping, since overloaded processors may miss real-time deadlines;
5. *migration monitoring*, where the migrations occured during the system's life time are shown. It allows the designer to evaluate whether using the migration mechanism (explicit or implicit) is effective or not;
6. *communication traffic*, where the inter-task communication traffic regarding to message passing exchange, during the system's life time, is shown. Through the analysis of this graphic, task mapping problems can be found, since intensive communication tasks may be desired to be mapped in the same processor;

The HellfireFW design flow allows the HW/SW refinement, after a careful analysis of system behavior is performed and it can be done in two main different aspects:

Software refinement. Specific features of HellfireOS can be changed in order to better adapt it to the implemented application. A very important feature to be evaluated is the tick[2] time period. The timing interrupts, simulated by our ISS, are used to provide system ticks to the OS. The tick time value must be well balanced, since very long time slices let the system less responsive (losing the real-time notion), whereas very short slices cause excessive OS context switches, increasing the system's overhead.

HellfireOS uses a timing register which is 32-bit wide to measure the tick time value as well as an activation bit of this register to trigger the timing interrupt. Depending on both processor clock frequency and the chosen activation bit, the system period (in *ms*) is either shorter or wider.

The tick period (in *s*) is calculated following the formula:

$$period = \frac{2^{act}}{freq}$$

where: act is the activation bit used by the timing register and $freq$ is the processor clock frequency in Hz.

In HellfireFW it is possible to configure the processor clock frequency and the activation bit used by the timing register. This provides a wide exploration of the tick time value, allowing the designer to find the most suitable granularity of HellfireOS tick time to the system application. Columns 1, 2 and 3 of Table II shows the tick time values (in *ms*) measured varying the activation bit of the timing register from 15 to 21. Although this is a 32-bit register, we highlight a range where we can find strict tick time values, suitable for hard real-time tasks. Still, we measured the number of system ticks per second according to processor clock frequency and timing register activation bit, in order to better analyze the OS overhead, as discussed later in this paper. Results are shown considering columns 1, 2 and 4 of Table II.

Moreover, the dynamic task mapping (task migration) is provided by HellfireOS and it must be evaluated, since the designer can decide whether the migration achieved the desired objectives or not. Task migration has several purposes, such as fault tolerance support, workload balancing, performance improvement, energy consumption reduction among others. This mechanism is characterized by copying a task from a source processor to a destination one. Then, source task's instance is killed and the task's execution in continued on the target processor from the same point where the preemption occurred.

HellfireOS implements *partial migration*. In this case, only task *data* (context) is migrated from one processor to another. Although it may seem to be a restrictive implementation, considering the overhead introduced by other approaches like relocatable code, partial migration presents a significant performance boost over a complete one [19]. The migration process is handled by a dedicated driver so after receiving a migration

[2]HellfireOS basic real-time execution unit (timeslice).

TABLE II

TICK TIME VALUES (IN *ms*) AND TICKS PER SECOND VARYING PROCESSOR CLOCK FREQUENCIES AND TIMING REGISTER ACTIVATION BIT

Processor clock frequency	Activation bit	Tick time (ms)	# of ticks per second
25 MHz	15	1.31	763.36
	16	2.62	381.68
	17	5.24	190.84
	18	10.48	95.42
	19	20.97	47.69
	20	41.94	23.84
	21	83.88	11.92
33 MHz	15	0.99	1010.10
	16	1.98	505.05
	17	3.97	251.89
	18	7.94	125.94
	19	15.88	62.97
	20	31.77	31.48
	21	63.55	15.74
50 MHz	15	0.65	1538.46
	16	1.31	763.36
	17	2.62	381.68
	18	5.24	190.84
	19	10.48	95.42
	20	20.97	47.69
	21	41.94	23.84
66 MHz	15	0.49	2040.82
	16	0.99	1010.10
	17	1.98	505.05
	18	3.97	251.89
	19	7.94	125.94
	20	15.88	62.97
	21	31.77	31.48
100 MHz	15	0.32	3125.00
	16	0.65	1538.46
	17	1.31	763.36
	18	2.62	381.68
	19	5.24	190.84
	20	10.48	95.42
	21	20.97	47.69

message, the kernel can quickly perform the requested migration.

Additionally, HellfireOS supports two different approaches of task migration: *explicit* and *implicit*. Explicit migration occurs when the designer inserts specific migration points in the application code whereas implicit one is defined by a migration policy. In this case, migration managers are used to monitor the system load status and to order migrations to be performed, aiming to balance the system workload.

Finally, even if migrations are not used, the designer can redo the static task mapping in this step to optimize the application performance.

Hardware refinement. Regard to number of processors present in the system. Still, features as processor frequency or memory size can also be changed. Additionally, communication behavior can be defined in terms of energy consumption or overall overhead.

After the designer applies the desired changes, a new HW/SW platform is ready to be easily simulated and verified.

IV. RESULTS

Results were taken aiming to show the advantages of using the HellfireFW, its possible configurations and how it helps the designer to better decide the most suitable HW/SW platform for a given application.

Initially, we analyze the impact of the tick time value adopted by HellfireOS in the maximum number of hard real-time tasks. This tick time depends on a timing interrupt, as summarized in Table II and results are shown in Figure 7, where a graphic depicts the relation of the tick time size and the maximum amount of hard real-time tasks that can be held by HellfireOS.

When using short tick time sizes, the amount of hard real-

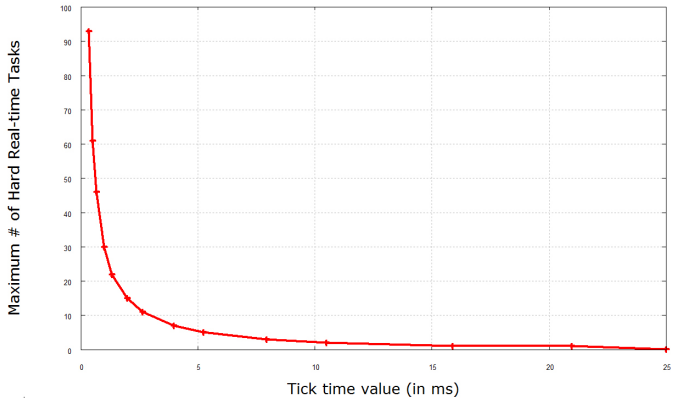

Fig. 7. Maximum Number of Concurrent Hard Real-time tasks as a function of tick time size

time tasks is increased due to the high responsiveness degree reached by HellfireOS. When the tick time size is wider, HellfireOS starts loosing real-time notion of the system and eventually hard real-time tasks will no longer be available. As we are targetting critical embedded systems, we tend to use the shortest tick time values aiming to increase the HellfireOS responsiveness and allow the use of more concurrent hard real-time tasks.

In this context, we analyzed the HellfireOS overhead introduced by context switches since the shorter the tick time value is, the higher is the number of needed OS context switches. The overhead of HellfireOS varies according to the activation bit used by the timing register. Since a context switch in HellfireOS takes, in average, 1500 cycles, after calculating the total amount of system ticks per second, presented in Table II, it is possible to define the proportional OS overhead by multiplying the number of system ticks that occur in a second times the average OS overhead (in cycles). This result must be divided by the processor clock frequency, in Hz. Figure 8 depicts the graphic with OS overhead (in percentage, compared to total execution time of the system) in function of the timing register activation bit employed.

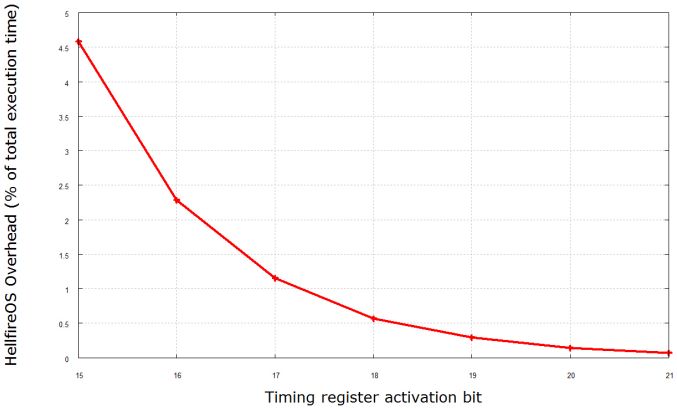

Fig. 8. HellfireOS context switch overhead as a function of timing register activation bit

It is important to notice the relation of Figures 7 and 8. In the former, shortest tick sizes mean that more hard real-time tasks can be held by HellfireOS. In the latter, shortest tick sizes (reached by less significant bits of activation, in different processor clock frequencies) mean higher OS overhead. Although this is an expected behavior, we highlight that even in the worst cases (where the shortest tick time available is used), the OS overhead is lower than 5% of total execution time. This is a very low price to pay when compared to the possibilities of integrating more than 80 hard real-time tasks in a single processor.

Additionally, to exploit the possibilities of personalization offered by HellfireFW, we developed three different task set scenarios as case study. These scenarios, with 100 tasks in total, are divided in:

1. *Task set 1*, composed by 80 hard real-time tasks and 20 best effort tasks;
2. *Task set 2*, composed by 50 hard real-time tasks and 50 best effort tasks, and;
3. *Task set 3*, composed by 20 hard real-time tasks and 80 best effort tasks.

We tested these three scenarios in all possible processor clock frequencies (25 MHz, 33 MHz, 50 MHz, 66 MHz and 100 MHz) maintaining the same timing register activation bit. In this case, we used the 17th bit to activate the timing register and the tick time values (in *ms)* were previously presented in Table II. We analyzed the three task scenarios regarding to deadline miss ratio. Results are presented in Figure 9. In this case, the designer

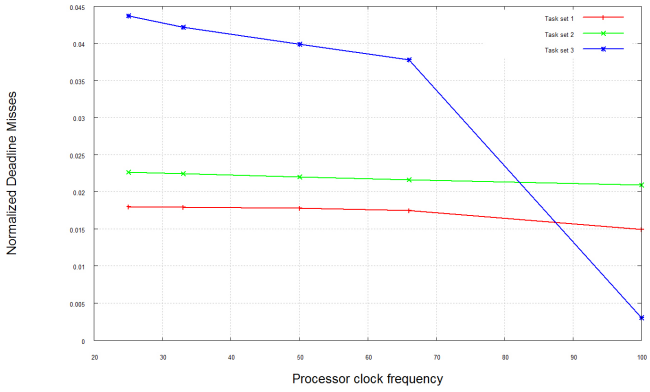

Fig. 9. Deadline miss behavior of different task set scenarios with a fixed activation bit and varying processor clock frequencies

can easily choose the most suitable processor frequency of several task sets, in terms of deadline miss ratio. Still, the designer can observe the impact of choosing either lower or higher frequencies in the deadline miss ratio.

Besides, we tested how HellfireFW helps the designer to choose the most suitable tick time value, by changing the timing register activation bit. We used the same three task set scenarios previously presented and varied the timing register activation bit from 15 to 21 to processor clock frequencies of 25 MHz and 100 MHz. We analyzed the three task scenarios regarding to deadline miss ratio and results for 25 MHz processor are presented in Figure 10. In this graphic, all task sets have more deadline misses when the period increases, since this reduces the maximum amount of hard real-time tasks of the system. Although the third task set is the one where less hard real-time tasks are found, it also presents a growing deadline miss ratio, especially in the first change of the activation bit. This occurs because in the first tick time value, almost all hard real-time tasks executed

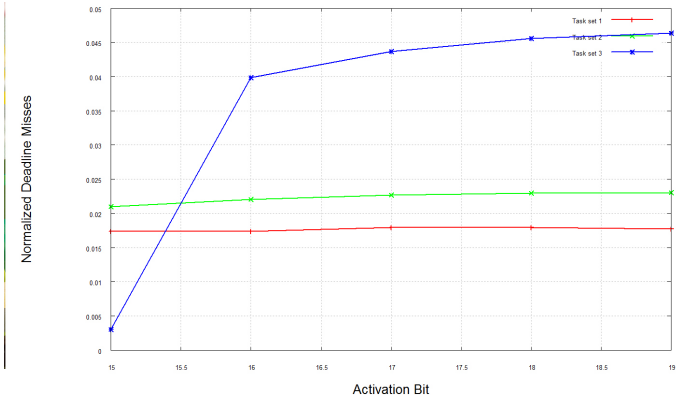

Fig. 10. Deadline miss behavior of different task set scenarios varying timing register activation bit running at 25MHz.

without missing deadlines and when the tick time value was increased, the responsiveness of the system decreased causing the severe deadline miss ratio increase observed in the graphic.

Similar behavior can be observed when running these tests on a 100 MHz processor. Figure 11 depicts the deadline miss ratio behavior and it can be observed that deadline miss ratio also increases as the tick time value increases.

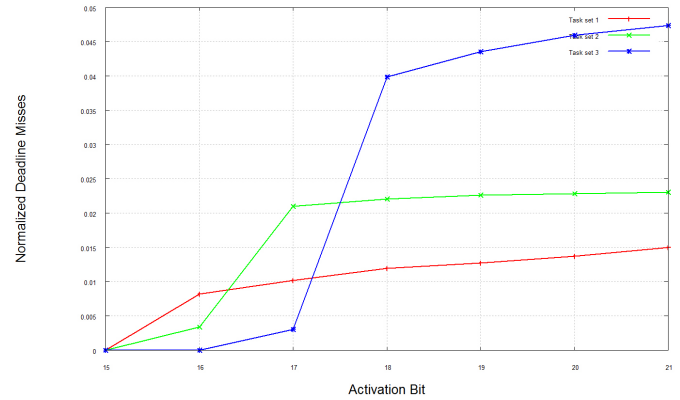

Fig. 11. Deadline miss behavior of different task set scenarios varying timing register activation bit running at 100MHz.

The deadline misses occurred because we tested how the system behaves when exceeding its maximum load. Although the tick time value dictates the maximum amount of hard real-time tasks without missing deadlines, the designer can test and analyze the possible configurations that may affect the system performance in a single integrated tool.

V. Concluding Remarks and Future Work

Critical embedded systems have tight computational requirements that can be achieved with MPSoCs solutions. New challenges arise, such as the need for helping tools to the designer, that allow the development of applications that respect a series of critical constraints.

In this paper we presented a HW/SW framework to develop, validate and test critical embedded systems' applications. Our solution provides an easy way to develop and validate critical embedded software in a given platform and also to perform HW/SW design space exploration allowing the designer to combine different SW mapping possibilities onto several HW platform architectures, as well as several OS configurations to analyze different aspects of the system. In results we show the limitations of our simulator and its advantages regarding the behavior of critical embedded systems.

Future work includes measuring the impact of different memory and communication strategies and how the scheduling policy impacts in the deadline miss ratio. Still a detailed communication overhead behavior analysis is desired.

Acknowledgment

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9.

References

[1] Ahmed Jerraya, Hannu Tenhunen, and Wayne Wolf, "Multiprocessor systems-on-chips," *Computer*, vol. 38, no. Issue 7, pp. 36– 40, July 2005.

[2] Sergio Johann Filho, Alexandra Aguiar, César Augusto Missio Marcon, and Fabiano Passuelo Hessel, "High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping," in *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, Washington, DC, USA, 2008, pp. 27–33, IEEE Computer Society.

[3] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

[4] A. Gerstlauer, Haobo Yu, and D.D. Gajski, "Rtos modeling for system level design," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 130–135.

[5] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic rtos model for real-time systems simulation with systemc," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, Feb. 2004, vol. 3, pp. 82–87 Vol.3.

[6] H. Posadas, J. Adamez, P. Sanchez, E. Villar, and F. Blasco, "Posix modeling in systemc," in *Design Automation, 2006. Asia and South Pacific Conference on*, Jan. 2006, pp. 6 pp.–.

[7] A. Shaout, K. Mattar, and A. Elkateeb, "An ideal api for rtos modeling at the system abstraction level," in *Mechatronics and Its Applications, 2008. ISMA 2008. 5th International Symposium on*, May 2008, pp. 1–6.

[8] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in soc design," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, 2002, p. 620, IEEE Computer Society.

[9] Gunar Schirner and Rainer Dömer, "Introducing preemptive scheduling in abstract rtos models using result oriented modeling," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, New York, NY, USA, 2008, pp. 122–127, ACM.

[10] Micrium, "Microc/os-ii kernel overview," http://www.micrium.com/, Accessed, September 2009., 2009.

[11] KADAK Products Ltd., "Amx real time operating system.," http://www.kadak.com/, Accessed, September 2009., 2009.

[12] Inc. CMX Systems, "Cmx-rtx real-time multi-tasking operating system," http://www.cmx.com/rtx.htm, Accessed, September 2009., 2009.

[13] Inc. Green Hills Software, "Integrity real-time operating system," http://www.ghs.com/, Accessed, September 2009., 2009.

[14] ARC International, "Arc mqx rtos," http://www.arc.com/, Accessed, September 2009., 2009.

[15] Andrew S. Tanenbaum, *Modern Operating Systems 2nd Edition*, Prentice Hall, 2001.

[16] Andrew S. Tanenbaum and A. Woodhull, *Operating Systems Design and Implementation*, Prentice Hall, 1997.

[17] William Stallings, *Operating Systems Design Principles 5th Edition*, Prentice Hall, 2004.

[18] Open Cores, "Plasma most mips i(tm) opcodes," http://www.opencores.org.uk/projects.cgi/web/mips/, Accessed, September 2009, 2007.

[19] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, pp. 241–299, 2000.