Framework for Real-time Embedded Systems based on COntRacts

FP6/2005/IST/5-034026

Deliverable: D-EP7v2

Multiprocessor execution platforms

Authors: Paolo Gai

Responsible: Paolo Gai

Purpose & Scope

This document describes the design of the implementation of a subset of the FRESCOR contract model on a reconfigurable multiprocessor hardware platform. The hardware platform chosen is the Altera FPGAs, hosting Multicore Nios II designs. The goal of this activity is to provide a prototype implementation which can be used as a proof of concept for showing how a complex API like the one in FRESCOR can be implemented on a minimal RTOS implementing the OSEK API, using a partitioning scheme to address the multicore complexity of the FPGA design.

License and Distribution

Destinator	Function/role	For Action	For Information
Frescor partners		X	X
European Commission		X	X

Revision history

File: **D-EP7v2_1_6**

Release/Status	Date	Author/Reviewer	Comments
1.0	25/11/2007	Evidence, PJ	First version
1.1	28/11/2007	UY, Rob	Typos
1.2	30/11/2007	Evidence, PJ	Added comment following Michael's feedback
1.3	01/12/2007	UY, Attila	Typos
1.4	10/01/2008	UC, Ignacio	Typos
			Version 1.4 submitted as D-EP7v1
1.5	25/03/2009	Evidence, PJ	Updates for v2
1.6	10/05/2009	Evidence, PJ	Integrated Rob Davis (UY) comments, added timing performance information

FRESCOR project (FP6/2005/IST/5-034026) is funded in part by the European Union’s Sixth Framework Programme. This work reflects only the author’s views; the EU is not liable for any use that may be made of the information contained herein.

Contents

1 Introduction

The goal of this task is to provide support for the contract model defined in WP2:AC on reconfigurable multiprocessor platforms, and specifically for the Nios II processor on ALTERA FPGA's board. In particular, the output of this task is a RTOS kernel that supports a subset of the contract model for real-time applications. Given the novelty of this task, only a subset of the contract model (the core part for temporal protection and shared resources) will be supported by the implementation. This task will also validate the results by providing a set of demo programs able to use all the support contract model features and dynamically reconfigure themselves. An additional support for Microchip dsPIC is also provided to ease the exploitation effort.

The document structure is as follows: Section 2 highlights the modifications with respect to the first version of D-EP7. Section 3 provides a set of background information about the architecture of the kernel used in the task, which is ERIKA Enterprise. ERIKA Enterprise is basically an OSEK/VDX kernel, with support for partitioning on heterogeneous multicores like the Altera Nios II soft core. Then, this document gives an overview of the Altera Nios II architecture providing an idea of the multicore support provided by Nios II. Section 4 describes the design done by Evidence to implement a static version of the FRESCOR API. Section 5 describes the support for multicores implemented in the ERIKA Enterprise kernel.. Section 6 shows some typical footprints of an ERIKA Enterprise application on Nios II. Section 7 and 8 describes describes the debugging under Lauterbach Trace32 and the coverage tests. Section 9 describes the support for Microchip dsPIC. Section 10 highlights the fulfillment of the evaluation criteria of the FRESCOR Evaluation report. The final sessions are Summary, Appendix and Bibliography.

The work of task EP7 has been divided in two periods:

- during the first period, we designed the implementation of the multicore support by selecting a subset of the API. Moreover, the EDF scheduler implementation prototype has been implemented (Section 3);
- during the second period, we implemented the multicore support on the Nios II platform; each core moreover has support for the IRIS scheduling algorithm [1] to provide a minimal implementation of a resource reservation scheme running on each core. Finally, support for Microchip dsPIC has been added to ease the dissemination of the results.

2 Modification w.r.t. the first version of this document

Section 3 has no modifications (except for the version of the Altera software used, that has been upgraded to 8.1). Section 4 has been updated with the work performed in the second period. The other Sections have been inserted in the second period.
The implementation of the functions `frsh_contract_get_resource_and_label` and `frsh_resource_get_vres_from_label` has been dropped, because their function is somehow naturally supported in the OSEK/VDX environment, where literal labels are already defined for each resource and VRES in the OIL file definition.

3 Background

3.1 OSEK/VDX

The purpose of this subsection is to give the reader a simple idea of the main OSEK concepts which are useful to understand the implementation we have done in this task on the ERIKA Enterprise kernel. It does not aim to be a comprehensive description of the standard, but just a highlight of the main differences between the POSIX API and the OSEK API.
The OSEK/VDX (http://www.osek-vdx.org) standard is a standard API for an open-ended architecture for distributed control units in vehicles. The name OSEK stands for “Offene Systeme und deren Schnittstellen für die Elektronik im Kraft-fahrzeug” (Open systems and the corresponding interfaces for automotive electronics), where VDX stands for “Vehicle Distributed eXecutive”. The standard is the result of the merge of two projects made in

France and Germany aiming at the standardization of the API for the operating system part on vehicles.
The philosophy which drove the specification of the OSEK/VDX API is related to the design of a standard interface which can be suited for the implementation on small microcontrollers. For that reason, the API allows scalability through conformance classes. In this case, conformance classes are like the POSIX Profiles defined in standard 1003.13, with the difference that whereas the POSIX profiles are basically subsets of the bigger POSIX API, in the case of OSEK the design has started from minimal requirements which has then be enriched with more features like error checking, blocking primitives, and priority queues handling.
The main difference between a POSIX system on which the FRESCOR API is based and an OSEK system is the fact that OSEK systems are inherently *static*. Static in the sense they do not support dynamic allocation of memory, they do not support dynamic creation of tasks, and there is in general no flexibility in the specification of the constraints like priorities, support for preemption, and so on. The choice of being static is important to allow implementation of the API on very small microcontrollers. Typical implementation of OSEK, in fact, has a footprint from 1 to 4 Kb of flash memory, which makes them suitable for the implementation on low-range 8 bit microcontrollers.
Being static for an OSEK system means basically that all the options typically covered in POSIX as parameters of the API functions must be implemented off line. To support that feature, The OSEK specification specifies a configuration language named OIL.
The OIL language basically defines the object which will be present in the final application (how many tasks, how many mutexes, ...), with the specification of most of the needed parameters for each object (for example, the OIL file contains the specification of the task priority. Once written, the OIL file has to be parsed by a configuration program, which generates the RTOS configuration.

The following example shows the layout of an OIL file used in ERIKA Enterprise. The OIL file is divided in two parts. The first part, included below, is an implementation definition, which defines the objects which can be instantiated. Basically, you can think at the implementation definition as a declaration of an operating system object type:

```
OIL_VERSION = "2.4";

IMPLEMENTATION my_osek_kernel {
[...]
    TASK {
        BOOLEAN [
            TRUE { APPMODE_TYPE APPMODE[]; },
            FALSE
        ] AUTOSTART;
        UINT32 PRIORITY;
        UINT32 ACTIVATION = 1;
        ENUM [NON, FULL] SCHEDULE;
        EVENT_TYPE EVENT[];
        RESOURCE_TYPE RESOURCE[];

        /* my_osek_kernel specific values */
        ENUM [
            SHARED,
            PRIVATE { UINT32 SIZE; }
        ] STACK;
    };
[...]
};
```

This part is typically provided by the operating system vendor, and does not have to be written by the user. The user then specifies only the second part, which instantiates the objects defined in the first part. The following example defines a task Task1 with its parameters:

```
CPU my_application {
```

```
    TASK Task1 {
        PRIORITY = 2;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        AUTOSTART = TRUE;
        STACK = SHARED;
    };
};
```

Once the OIL file is written, it has to be parsed by a OIL compiler. In the case of ERIKA Enterprise, Evidence provides an OIL compiler which is integrated in the RT-Druid design environment as an Eclipse Plugin. The result, in the case of ERIKA Enterprise, Is a set of .c, .h files and a set of scripts which are then used to compile the application. If we consider a single OIL file, for example, the generated code looks like the following (we just reported a subset of the data structure generated, and we suppose the OIL file only defines a single task):

eecfg.h

```
[...]
#define EE_MAX_TASK 1
#define Task1 0
[...]
```

eecfg.c

```
[...]
const EE_ADDR EE_hal_thread_body[EE_MAX_TASK] = {
      (EE_ADDR)FuncTask1
};

const EE_TYPEPRIO EE_th_ready_prio[EE_MAX_TASK] = {
      0x2
};

EE_TID EE_th_next[EE_MAX_TASK] = {
      EE_NIL
};
```

As you can see, the eecfg.h contains the declaration of the identifiers of the objects, which are then used to address a set of arrays which stores the kernel data structures like priorities, queues, and so on. Notably, all the parameters specified in the OIL file which cannot be changed at runtime are declared as `const`, meaning they will be allocated in the microcontroller flash (and not in RAM). This feature is really important in small microcontrollers, where only a few K bytes of RAM are available and where typically everything is known and static since the beginning of the application.

Then, when the application runs, the OSEK/VDX primitives will be called using the identifiers generated in the eecfg.h file, like in the following example:

```
ActivateTask(Task1);
```

As it can be noted, this approach is completely different from the one followed in the POSIX API, where each Operating System object has to be dynamically created by passing attributes stored into data structures in RAM.

Notably, also the FRESCOR API is based on the same approach as POSIX, and as part of this task we will convert part of the FRESCOR API to be implemented as an OIL specification in a static way, to enable the implementation of a low-footprint version of the FRESCOR contracts.

Another notable difference in the design of the OSEK API is the possibility not to have blocking primitives in the API. This feature, available for the so called Basic tasks, together with the usage of Fixed Priority Scheduling with Immediate Priority Ceiling allows RTOS implementations which supports stack sharing between different tasks. In

particular, ERIKA Enterprise supports two modes of operations: Mono Stack (only one stack for each CPU) and Multi Stack (which support a set of different stacks which can be shared by group of tasks). Please note that Stack Sharing techniques cannot be applied to a standard POSIX implementation because POSIX always provide support for blocking primitives (like file I/O, synchronization, ...).

3.2 The Altera Nios II softcore processor

The Altera Nios II soft-core processor is basically a 32 bit RISC processor which is provided by Altera for the usage in the Altera FPGAs (the current description is based on version 6.0 of the Altera Nios II EDS; During the writing of this document we are also doing the porting of our tools to the newest version of the software, version 7.2 and then 8.1, which completely changed the Nios II build system).

To provide a simple and efficient way to design a system-on-a-programmable-chip (SOPC), Altera provides a set of tools which covers the complete design of the hardware and software needed to design the FPGA configuration.

The first tool which is provided by Altera is Quartus II. Quartus II is used to take a set of hardware blocks specified in Verilog or VHDL, and generate the FPGA byte code which has to be loaded in the FPGA part.

One of the blocks available is then a SOPCBuilder block. SOPCBuilder is a wizard which can be used to design a complete system-on-a-programmable-chip. The idea is that the tool can be used to plug together different blocks (see Figure 1).

In particular, the user can specify the list of blocks composing a system, connecting them using a bus and finally specifying interrupt priorities and address spaces.

Once generated, the block is ready to be included in a Quartus design to implement more complex features.

At the system level, what happens is that SOPCBuilder instantiates an “Avalon Switch Fabric”, which is basically a crossbar switch connecting all the master and slaves in the system. Arbitration logic is also automatically generated, as well ad address decoding hardware. Figure 2 shows a block diagram where the gray part contains arbitration logic generated for a particular configuration.

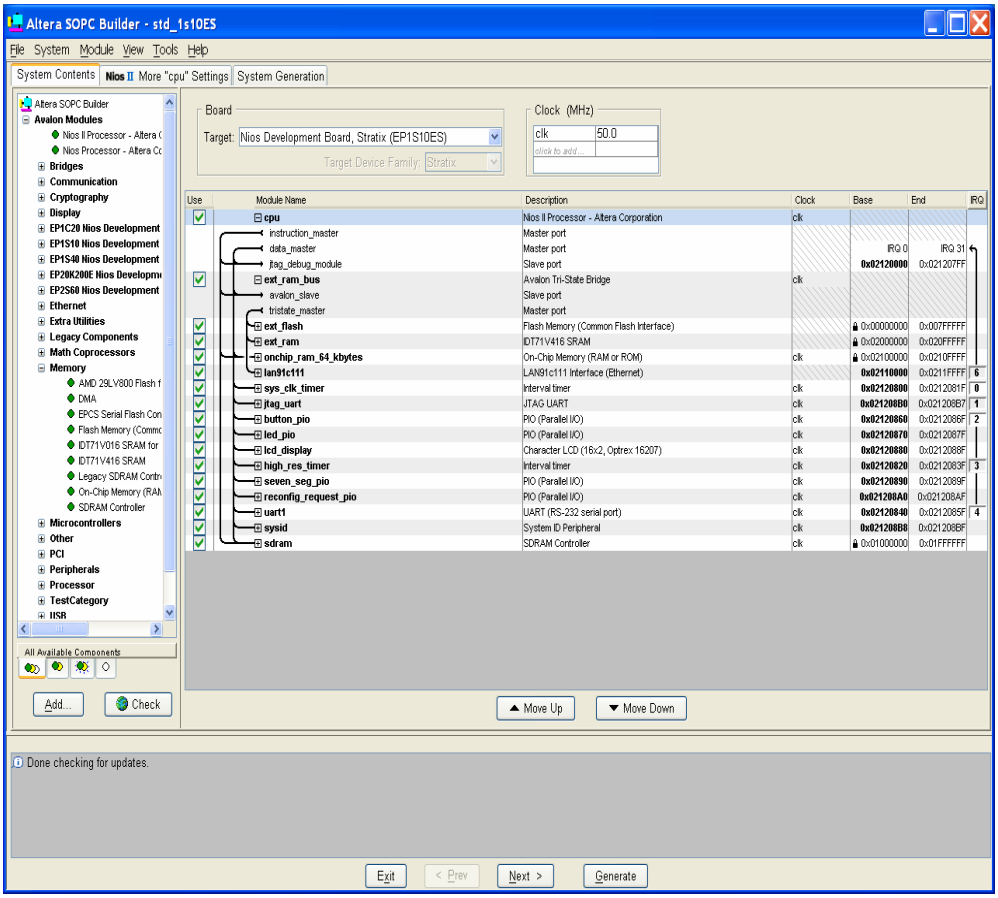


Figure 1: the SOPCBuilder wizard

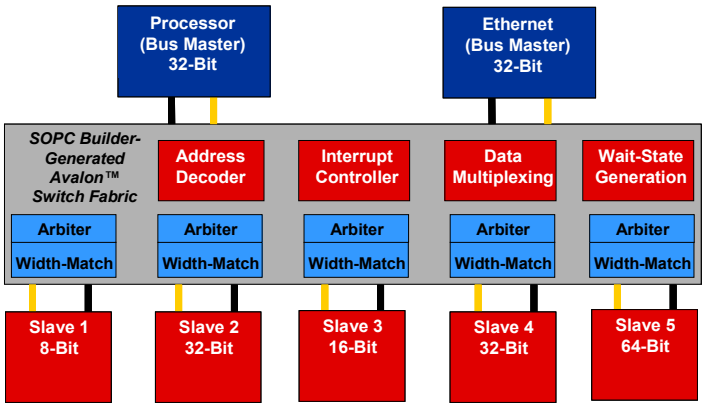


Figure 2: SOPCBuilder generates the Avalon Switch Fabric to connect different peripherals in the system

As a result, a typical system designed using SOPCBuilder can contain one or more Nios II CPUs, as well as different peripherals. The system then is composed easily and automatically configured to produce an embedded design based on FPGA.

On the CPU side, SOPCBuilder allows the possibility of inserting in the list the Nios II soft core. Nios II is basically a VHDL block implementing a 32 bit RISC CPU with Harvard architecture. The CPU has a customizable instruction set, as well as the possibility to have caches, and three different implementations giving a trade off between computational power and number of logic elements used inside the FPGA.

Multiple Nios II blocks can be hosted on the same SOPCBuilder blocks, forming in that way a multicore system on a FPGA.

The support for multicore computing provided by Altera is the following:
- There is no support for cache coherency between two Nios II processors. Caches are separate, and concurrent access to multicore shared data structures have to explicitly disable the cache.
- Altera provide a way to implement an atomic operation between two Nios II CPUs by using the Altera Mutex peripheral. Basically, locking and unlocking can be done thanks to the fact that write operations on 32 bit registers are sequentialized by the Avalon switch Fabric.
- Interprocessor interrupts can be implemented easily by using appropriately connected GPIO interfaces. In particular, an interprocessor interrupt can be implemented in hardware by using a shared GPIO peripheral which outputs a bit for each CPU in the system. Then, each CPU has an input pin which generates interrupts on edges.
- Peripheral drivers provided by Altera cannot be shared between different CPUs. Basically, each peripheral has to be statically allocated to a specific CPU.
- Nios II can share the same Ram and Flash memory, as well as have separate private memory regions.

4 Static version of the FRESCOR API

In this section we will describe the modifications we will do on the FRESCOR API to adapt it to a static approach like the one proposed by OSEK/VDX. The goal for this implementation will be an implementation of a reduced subset of the FRESCOR API which can fit in a 10Kb ROM footprint. As it can be seen in Section 6, we succeeded in maintaining such a low footprint when compiling the system with the gcc option -Os.

The content of this section is the following:
- we first describe the implementation of the EDF scheduling algorithm on top of OSEK/VDX (this implementation has been performed in the first period of FRESCOR);
- then, we described the choice we have done for implementing blocking primitives;
- after that, we describe the selection of the FRESCOR API implemented in the second period.

4.1 Efficient EDF implementation

One of the main constraints in the choice of the scheduling algorithm for a static implementation of the FRESCOR API has been the availability of an efficient EDF implementation, which was needed in order to provide an efficient IRIS implementation as shown in Section 4.5.

For that reason, on the first part of this task we concentrated on the implementation of an efficient EDF scheduler which could be integrated in an OSEK System. There are two issues in implementing an EDF scheduler on an OSEK system. The first one is that there is no explicit support for time in the OSEK system. In fact, the timing support we require is a lightweight internal timing reference, whereas OSEK exports an alarm interface leaving the counter objects implementation defined. We resolved the issue by adding an internal timing support which can be used in the implementation of the kernel primitives.

The second problem in implementing EDF on a small microcontroller is that time handling on small microcontrollers is in general difficult, because to be efficiently implemented EDF needs a timing reference which is both with a high resolution (to handle short and fast activities with deadlines in the order of tens/hundreds of microseconds) and with a long lifetime (because EDF scheduled tasks are based on absolute deadlines).

These timing properties has been resolved in the POSIX systems by using a `struct timespec` data structure, containing a 32 bit integer representing nanoseconds (providing this way high resolution), and a 32 bit integer representing seconds (giving a lifetime of around 136 years, more than enough for practical applications). This approach in general cannot be implemented on small microcontrollers, where handling a 64 bit data structure imposes a significant overhead over the scheduler implementation.

To avoid the limitations of the usage of the `struct timespec` on a microcontroller environment we proposed the usage of a circular timer approach (see Figure 3). The main idea is that the system does not store an absolute timing reference, but only a timing reference which is relative to the current time, where the current time is implemented by using a free running timer. In this way, it is possible to handle absolute deadlines with a spread that must always be between t-T/2 and t+T/2, where t is the current time and T is the lifetime of the timer implementing the timing reference.

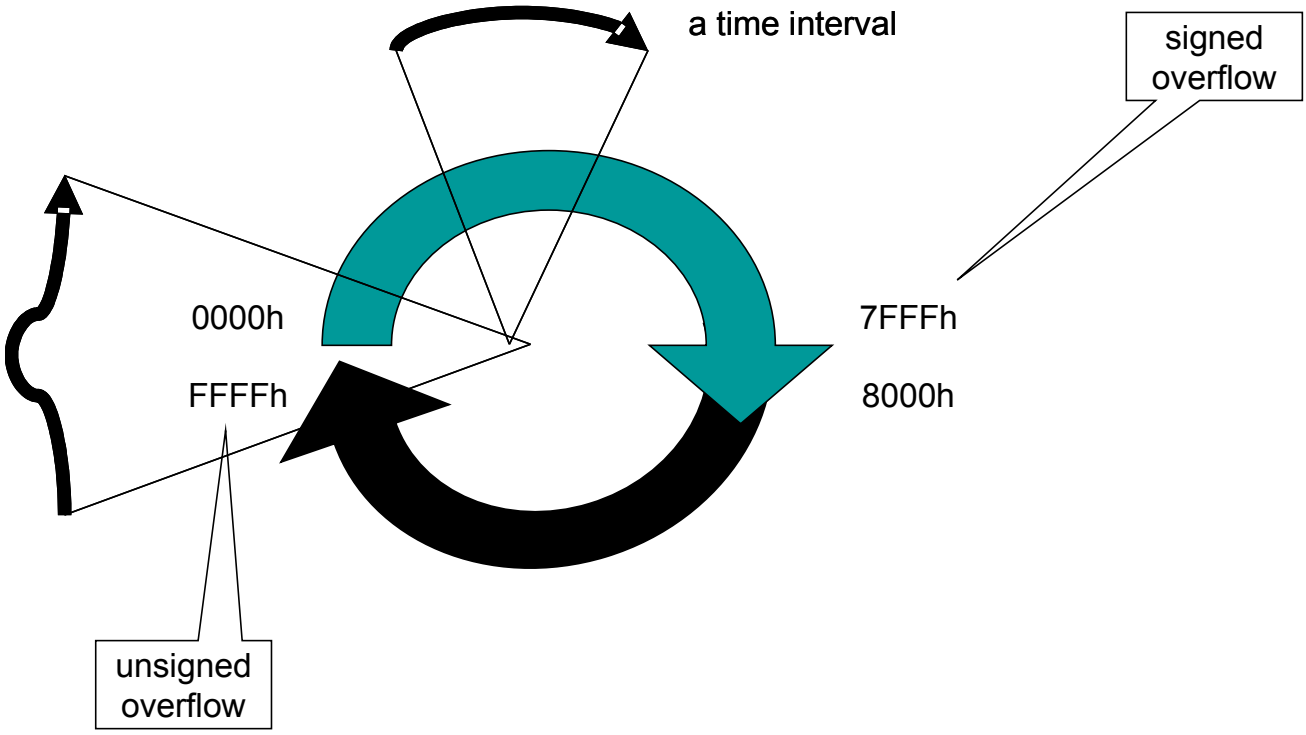


Figure 3: Circular timer implementation in ERIKA Enterprise

At that point, before going on the Nios II soft core, we decided to implement the feature on the FLEX boards produced by Evidence. In fact, most of the source code to be implemented is portable C code, and the FLEX environment was already available with the latest version of the ERIKA Enterprise software at the time of the implementation. The FLEX boards hosts a Microchip dsPIC microcontroller which support a 32 bit hardware timer (the same size of the timers used in the Nios II soft cores). Considering a timer clock running at 2 MHz we obtained a lifetime of 1073 seconds, more than enough for many practical applications.

Then, we proceeded in implementing a new conformance class, which we called “EDF”. The EDF conformance class retains the same static approach of the OSEK/VDX standard, with a slightly simplified API (basically, the API does not include the TerminateTask primitive). Maintaining the OSEK/VDX interface, we then modified the RT-Druid configurator by adding the EDF conformance class. In the OIL File, the user have to specify that he wants to use the EDF conformance class, specifying the timer tick implemented by the hardware as in the following line:

```
KERNEL_TYPE = EDF { TICK_TIME = "25 ns ";};
```

Then, the user have to specify for each task a relative deadline, which RT-Druid will convert in the number of ticks using the tick duration specified in the line above. To specify the relative deadline, the user has to specify the following line:

```
TASK myTask1 {
      REL_DEADLINE = "10 ms ";
};
```

The rest of the API in the system remained the same, demonstrating the possibility of having an EDF scheduler as an alternative scheduling algorithm.
The result of this first step has been published in version 1.4.1 of the ERIKA Enterprise Basic GPL for Microchip dsPIC, available for download from the Evidence web page, http://www.evidence.eu.com/content/view/129/250/.

4.2 Blocking primitives

Another problem we noted in the implementation of the subset of the FRESCOR API is the fact that the FRESCOR API subsumes the existence of blocking primitives in the API. Also, resource reservation schemes and in general temporal protection implementation imposes an interleaving between different task instances, forbidding in this way the implementation of a stack sharing mechanism.

For that reason, we had to choose to allocate a separate stack to each task handled under the FRESCOR API, meaning that in the OIL file, when compiling an ERIKA Application using the FRESCOR scheduler, we will have to use the multi stack configuration.

The usage of a multistack configuration implies additional overhead which can be estimated in a larger code size (typically in the order of 300-400 bytes ROM), a slower execution (typically around 1 microsecond more at 50MHz), and a greater RAM usage (each stack must be enlarged by 40 bytes in the case of Nios II). Moreover, using the multistack option is more difficult to share the stack space. The amount of RAM cannot be evaluated in general, and largely depends on the specific application.

As it will be seen in Section 7, the multistack configuration is also exported to the debugger kernel awareness module using the ORTI file. Using ORTI, the debugger will be able to provide an estimation of the percentage of usage of the stack.

4.3 FRESCOR API made static

Finally, we needed to address the fact that whereas the FRESCOR API allows the dynamic creation of entities as well as implementing renegotiation, the OSEK/VDX interface is mainly static, with all the basic features implemented and configured off line with a specification in the OIL File.
For that reason, we decided to implement a trade off between the dynamic FRESCOR API and the OSEK approach. Basically, we did the following design choices:

- Vres are defined statically in the OIL file. This does not seem to be a great restriction since small microcontrollers will be a closed environment without new application entering in the system at runtime;
- Threads will be bound to Vres at runtime. The option will allow the possibility to move threads to different Vres, allowing the implementation of various resource reservation schemes. A first binding will always be provided in the OIL file;
- Temporal isolation, resource scheduling and reclaiming will be done, obviously, at runtime thanks to the usage of the IRIS [1] scheduling algorithm;
- The OIL file will be extended to contain a specification for the Vres, in a way to be able to perform an off line negotiation limiting the amount of code needed at runtime on the final target. The spare capacity will be divided among the available Vres at runtime, thanks to the reclaiming features of IRIS.

Considering these choices, we decided to implement the following subset of the FRESCOR API:

- Basic Services:
  - frsh_init
  - frsh_strerror

- Contract Creation and Initialization:

  - frsh_contract_get_basic_params
  - frsh_contract_get_timing_reqs

- Negotiate contract functions:
  - frsh_thread_bind
  - frsh_thread_unbind
  - frsh_thread_get_vres_id
  - frsh_vres_get_contract

- Synchronization objects:
  - frsh_syncobj_signal
  - frsh_syncobj_wait
  - frsh_syncobj_wait_with_timeout
  - frsh_timed_wait

- Obtaining information from the scheduler:
  - frsh_config_is_admission_test_enabled
  - frsh_vres_get_remaining_budget
  - frsh_vres_get_usage
  - frsh_vres_get_budget_and_period

The following features of the FRESCOR API has not been implemented, because in general they give too much overhead on small microcontrollers or are too far away from OSEK /VDX approach to system design:
- No support for shared objects
  - Monitoring of the execution of critical sections is heavyweight
- No hierarchical scheduling
- No distribution (tasks are statically partitioned on multicore Nios II designs)
- No feedback control
  - EE targets embedded systems with very low performance, where feedback control can be heavyweight
- No memory management
  - Typical applications on microcontrollers do not have a memory allocator
- No energy management
  - The Nios II soft core is not capable of DVS
  - It is impossible to turn on/off the cores at run-time

As a result, we believe we reached a good compromise in the choice of the features which can be included in a minimal version of the FRESCOR API, and we are confident that the subset can be implemented with a reduced footprint similar to current OSEK implementations.

4.4 OIL specification for single cores running the FRESCOR API

The ERIKA Enterprise operating system supports the OIL language thanks to an OIL compiler and code generator implemented in the Evidence RT-Druid tool.
To provide a proper support for the implementation of multicore systems based on the FRESCOR framework, we implemented a specific version of the OIL compiler and code generator inside the RT-Druid tool.
This section, in particular, describes the enhancements needed in the system to support the IRIS scheduling algorithm and the FRESCOR API on single cores. Section 5.1 contains additional details of the OIL implementation for multicores.
The resulting OIL specification is provided as a separate file to this document. In the following paragraphs we will describe in detail the enhancements done specifically for the FRESCOR project, together with some examples.

4.4.1 Specification of hardware timers

To support the EDF scheduler and the IRIS scheduler, the system designer needs to specify the timers that needs to be used to raise the timing interrupts. The following is a snippet of the OIL implementation describing the

specification:

```
OIL_VERSION = "2.4";
IMPLEMENTATION ee {
  OS {
    ENUM [
      NIOSII {
        STRING TIMER_FREERUNNING;
        ENUM [
          SINGLE {
            STRING TIMER_IRQ;
          },
          MULTIPLE {
            STRING TIMER_IRQ_BUDGET;
            STRING TIMER_IRQ_RECHARGE;
            STRING TIMER_IRQ_DLCHECK;
            STRING TIMER_IRQ_SEM;
          }
        ] FRSH_TIMERS;
      },
    ] CPU_DATA[];
    ...
  };
  ...
};
```

As it can be seen, the OIL file always require a free running timer used as timing reference. Then, for raising the interrupts, we provide two ways of configuring the timers: one that requires only one hardware timer (for FPGA designs with limited hardware resources, as well as for Microchip dsPIC), and another one for designs that can afford different timers for each interrupt source.
In the case of a single timer, the system will multiplex the various asynchronous events by properly programming a single hardware resource. In the case of multiple timers, there will be one hardware timer for each one of these events:

- Budget exhaustion;
- IRIS recharging events;
- IRIS deadline check;
- FRESCOR synchronization object timeouts.

An example of OIL file snippet complaining to the specification described before is the following

```
CPU test_application {
  OS EE {
    CPU_DATA = NIOSII {
      TIMER_FREERUNNING = "HIGH_RES_TIMER_0";
      FRSH_TIMERS = MULTIPLE {
        TIMER_IRQ_BUDGET = "TIMER_CAPACITY_0";
        TIMER_IRQ_RECHARGE = "TIMER_RECHARGING_0";
        TIMER_IRQ_DLCHECK = "TIMER_DLCHECK_0";
        TIMER_IRQ_SEM = "TIMER_SEM_0";
      };
    };
    ...
  };
  ...
};
```

Please note that the specification of the timers have to track exactly the same names used in the Altera SOPCBuilder specification.

As a result, the code generator generates a set of #defines which will be used to configure the timers in a proper way.

4.4.2 Specification of scheduler and contracts configuration

These settings control the specification of the scheduler used in the particular application. Two choices are possible: EDF or FRSH.

The EDF scheduler is the scheduler which has been implemented in the first part of the project. The main option of this setting is the specification of the TICK_TIME, which maps the hardware timer granularity. Basically, deadlines are specified using a time representation (in s, ms, us). Then, the code generator converts the deadline specification in hardware ticks. For example, a relative deadline of "1ms", with a TICK_TIME of "50ns" will produce a value of 20000 that will be stored in the internal kernel data structures. Then, each task has the possibility to specify a REL_DEADLINE, which is the relative deadline that is used to schedule the task.

The FRSH scheduler, on the other hand, implements the FRESCOR API. As it can be seen, the available options include the specification of the contracts, as well as the possibility to select whether the synchronization objects should be used or not (to save flash memory and hardware resources).

Please note that when no time units are specified for deadlines, budgets and periods, it is intended that the specification is in hardware ticks.

About the contract specifications, the required attributes include the budget and period of the contract, as well as its name. Moreover, a CPU_ID parameter is provided for multicore designs (see Section 5).

Although the system does not handle dynamic reconfiguration of the contract set, the system does in any case provide some kind of flexibility. In particular, the amount of bandwidth reserved for the contracts can be greater than one, this to allow dynamic binding of tasks to contract in the case of mode changes. When parsing the contract information, RT-Druid does a static acceptance test to guarantee that the first allocation of tasks to contracts has a CPU utilization less than 1. It is then responsibility of the programmer to do a proper reallocation of contracts to tasks at runtime.

The following is the description of the OIL implementation:

```
OIL_VERSION = "2.4";
IMPLEMENTATION ee {
  OS {
    ENUM [
      EDF {
        STRING TICK_TIME;
        BOOLEAN NESTED_IRQ;
      },
      FRSH {
        ENUM [
          CONTRACT {
            STRING NAME;
            UINT32 BUDGET;
            UINT32 PERIOD;
            STRING CPU_ID;
          }
        ] CONTRACTS[];
        BOOLEAN USE_SEM;
        STRING TICK_TIME;
      }
      ...
    ] KERNEL_TYPE;
    ...
  };
```

```
  TASK {
    STRING REL_DEADLINE;
    STRING CONTRACT;
    ...
  };
  ...
};
```

An example of a FRSH specification in a real application is the following:

```
CPU test_application {
  OS EE {
    KERNEL_TYPE = FRSH {
      TICK_TIME = "20ns";
      USE_SEM = TRUE;
      CONTRACTS = CONTRACT {
        NAME = "c1";
        BUDGET = "20ms";
        PERIOD = "100ms";
      };
      ...
    };
  };

  TASK Task1 {
    CONTRACT = "c1";
    ...
  };
};
```

As it can be noted, the first binding of contracts to tasks is done statically in the task specification. Then, at runtime, the binding can be changed using the FRSH primitives for bind and unbind.

4.5 IRIS implementation details

This section briefly describes the data structures and the implementation detail of the IRIS scheduler implemented into the FRSH module in the ERIKA Enterprise source code.

To implement the FRSH API, a new kernel specification has been added in the `/pkg/kernel/frsh` directory of the ERIKA Enterprise tree. All relevant FRSH include files have been stored into the `/pkg/kernel/frsh/frsh_include` . Only the subset of FRSH functionality used in the ERIKA Enterprise implementation has been defined. Synchronization objects has been implemented in a separate directory under /pkg/kernel .

The API structure is somehow similar to the one of the EDF kernel. That is, as in EDF, the TerminateTask primitive is not supported.

The main difference in the implementation of the FRSH kernel compared to the other scheduling algorithms implemented in ERIKA is the fact that, due to its complexity, most of the basic functionalities has been separated in helper functions contained into the file pkg/kernel/frsh/src/ee_cap.c . This helped simplifying the testing phase, and removed duplication of the source code. The overhead for this approach is very limited and it is around 10 assembler instructions for each call, that is more than balanced with the fact that at the end the kernel uses a few Kb less of source code, improving cache performances and footprint figures.

About the kernel data structures used in the system, we partitioned the data structures in various groups, to make these data structure maps the concepts which are present in the OIL file.

In particular, the following structures have been defined:

- a *contract* data structure, storing budget and periods as specified in the OIL file. The data structure is declared as const, to allow for a smaller RAM footprint.
- A *VRes* data structure, which stores the details of the Virtual Resource which has been created together with the contract. The Vres data structure contains the available budget, the absolute deadline, the status of the VRES, and the task linked to it (if any).
- A *Task* data structure, which stores the priorities, status, locked counter, number of activations of a task. It also stores a timedout flag and a link to the Vres attached to it.

About the Vres data structure, it must be noted that all the Vres are statically instantiated by the OIL compiler at system generation. The Vres statuses are somehow different from the theoretical specification done in the original paper [1]. In particular, a FREEZED status has been added to solve the following scenario that may appear due to the fact that we are using a circular timing reference (as explained in Section 1.1):

- A task stops or ends its execution. The Vres is put into the INACTIVE state, with positive budget and deadline in the future.
- As time passes the Vres deadline goes in the past. Activating the task linked to the Vres provokes a correct reassignment of the Vres parameters following the IRIS algorithm.
- As time passes, it may happen that the absolute deadline of the Vres, due to the circular timing implementation, wraps around and become a deadline “in the future”. At this point, activating the task generates a wrong behavior.

To solve this, a periodic interrupt has to be programmed to move all the Vres in INACTIVE state and deadline in the past to the FREEZED state. Activating a Vres in FREEZED state provokes a new assignment of the Vres parameters. The periodic interrupt is the Deadline Check interrupt implemented into /pkg/kernel/frsh/src/ee_dlcheck.c .

About resource usages, the algorithm used is the SRP algorithm. A preemption level is assigned to a task, and the system ceiling is raised when resources are locked. When resources are locked, timing isolation of tasks is disabled, that is no preemption is implemented if a task ends its budget while owning a resource. This approach has also been analyzed in the literature as “overrun and payback” mechanism by [3] and [4] w.r.t. a task continuing to execute when its budget runs out but it is holding a resource. In that case, the task budget becomes negative and the time consumed in excess will be borrowed from the next task instances (in pathological cases a task may be forced not to schedule for a few instances due to that, as shown in Section 7). The rationale behind this choice is that it is too costly on a microcontroller to implement some control over the time a task owns a resource. Being the application environment somehow controlled, there is hope that the designer will not abuse of the timing spent in a critical resource. Please note that also binding and unbinding is disabled when a task owns a resource. This has some drawbacks on the task data structure and on the Vres links as explained in the following paragraph.

About the task data structure, a special note has to be done on the Vres links. In particular, the Vres link is used by the kernel to store the Vres which is linked to the task. Vice versa, the Vres has a link to the task which is linked to it. These two links are modified by the bind/unbind functions. In fact, a VRes must be able to store the fact that a task has been unbound, or that a task has been bound to it. On the other hand, the same applies for the Task. Also please note that the task data structure has an additional pointer to Vres, named `vres_deferred`, which is used to map the fact that a new Vres binding is available for the task, but that binding has been deferred due to the fact that the task was owning a resource.

About the implementation of budget exhaustion, it has to be noted that a special interrupt source has been dedicated to that functionality. In particular, a hardware timer is programmed to fire at the time where the budget exceeds, preempting the running thread. The budget exhaustion interrupt source code is empty, because the budget handling is implemented in any case in the end-of-interrupt functions. Upon a budget exhaustion, the Vres linked to the task is put into the recharging queue, and a recharging interrupt is set. The recharging queue will be then emptied when the recharging IRQ fires, or when there are no task to schedule.

About the statuses of tasks and Vres, please note that there is need to have a separate status for Vres and tasks. The Vres status control the possibility for a task to execute, whereas a task state controls the status of the activity performed by the task. That is, a task may be ready to execute but its Vres may be in recharging mode, blocking the execution of a task. This separation of the statuses is also visible in the structure of the source code. In fact, the handling of the Vres structure and the queuing of Vres in the Vres queues is mainly done in

`/pkg/kernel/frsh/src/ee_cfg.c`, where the task handling and its queuing in the task queues is done in the files implementing the primitives.

The implementation of the synchronization objects is very similar to the implementation of a binary semaphore with timeout. The synchronization object timeout is then implemented by using a dedicated interrupt. As a difference with the FRESCOR implementation, we decided not to void the budget upon a synchronization block. This because the IRIS algorithm is able to automatically reclaim the unused budget without the need of further intervention. Also, the budget not used could be reassigned to other tasks using the bind primitive.

Finally, deadline miss and budget overruns has not been implemented for synchronization objects. The rationale for that is that the IRIS algorithm somehow decouples the budget/deadline issues that are task parameters, with the budget and period which are used for the Vres. Implementing deadline and budget overrun on tasks would have inserted an overhead which not negligible, which has been avoided in this implementation.

5 Multicore systems and FRESCOR

It is clear that an implementation for an RTOS a multicore system based on Nios II has to use in the best way the architecture limitations imposed by a Nios II system.
In particular, the implementation done for ERIKA Enterprise on the Nios II soft core takes advantage of the Nios II limitations providing the following features:

- Static partitioning of tasks to processors. The rationale of this choice is that Nios II does not support cache coherency, and for that reason it is not convenient to implement migration between the different CPUs. Moreover, peripherals are private to processors, making impossible to reach a peripherals from a CPU which does not have access to it.
- Support for queuing spin locks by providing an enhanced implementation that uses the Altera Mutex peripheral provided by Altera.
- Support for partitioning source code by using the OIL file. (see later).
- Support for automatic interprocessor interrupt support, which is used to notify actions from one CPU to another. The support for the interprocessor interrupt is hidden inside the OSEK API, meaning that the user continues to write OSEK-compatible source code, which will automatically use interprocessor interrupts when needed. As an example, a call to a primitive like `ActivateTask(Task1);` will be resolved locally if the task has been statically allocated to the CPU where the primitive has been executed, or will be resolved in an interprocessor interrupt if the task is allocated to a remote CPU. On the remote CPU, on the other hand, the reception of an interprocessor interrupt will cause the execution of a task activation which will be of course resolved locally (see the figures in Section 7).
- Support for a startup barrier, allowing all the CPUs to synchronize at startup before starting sending interprocessor interrupts. This feature has been implemented thanks to the possibility to specify an initial locker CPU for the Altera Mutex peripheral used to implement atomic operations.
- Support for automatic Cache disabling. ERIKA Enterprise supports a unique method for implementing cache disabling by using the bit 31 cache disabling feature provided by Nios II. This feature allows to write source code which resolves in accessing data structures with cache disabling or not depending on the allocation of tasks to CPUs. In particular, if a shared resource is used only locally to a CPU, no cache disabling is needed taking advantage of the Nios II caches, whereas if a data structure is used concurrently by tasks allocated to different CPUs, then all the data accesses to the structure will automatically use cache disabling without changing the source code.
- Support for automatic usage of spin-locks in shared resource handling. In a single stack system the only way to handle mutual exclusion on critical sections between tasks allocated to different CPUs is to use spin locks. ERIKA Enterprise provides an integration of different features which basically uses queuing spin locks only when the resource has been declared as used by tasks allocated on more than one CPU.
- Support for Lauterbach multicore debugging. To ease the debugging of complex multicore systems, ERIKA Enterprise provide an automatic generation of the scripts needed to compile and load a multicore application on a Lauterbach debugger which has been bought in the context of the FRESCOR Project.
- Support for the ORTI language to enhance the interpretation of the status of the system (see Section 7).

5.1 OIL support for multicores, and implementation details

The OIL Language described in the previous sections has been enhanced to provide support for Nios II multicore designs.

In particular, each Task has the possibility to specify the CPU to which it is allocated. Moreover, as noted before in Section 4.4.2, the contract parameters specified into the FRSH Section contains the specification of the CPU_ID to allow the partitioning of contracts to different CPUs.

In this way, the definition of a OIL file for a multicore Nios II system will contain all the information that are needed for system generation.

The behavior of the OIL code generator in the case of the FRSH kernel and a multicore Nios II setting is the following:

- Specific configuration files are generated for each CPU;
- Tasks and contracts are instantiated in the CPU they are linked to;
- A task allocated to a CPU cannot be linked to a Contract which is allocated to a different CPU. RT-Druid in this case generates an error:
- If ORTI generation is selected, each CPU will have an ORTI file that describes the status of the tasks and contracts linked to the specific CPUs.

The multicore support implementation needed a complete restructuring of the remote notification feature which were present in the previous versions of the kernel.

The remote notification feature is a part of the ERIKA Enterprise kernel which is responsible to communicate a set of events to other CPUs. As an example, activating a task on another CPU is implemented as a call to the `rn_send` function inside the implementation of the `ActivateTask` primitive. The `rn_send` function is then responsible to annotate in the common data structures about the event, and to inform the remote CPU of the event, by using an appropriate interprocessor IRQ and an IRQ handler. From the user point of view, nothing changes in the API called, because the remote tasks are visible on a CPU as task numbers with a special “remote notification flag”.

The implementation modification done to support the FRSH framework has been mainly related to the support of the FRSH API functions on the remote notification framework, with the support of the existing functions plus the addiction of remote bind/unbind operations.

We did not add synchronization objects to the remote notification framework, because a generic implementation of the synchronization object allowing queuing of task coming from different CPUs has an overhead in terms of timing and footprint not negligible, and for that reason we decided to limit its features only to each single core in the multicore system.

5.2 Building a multicore Nios II example

This section briefly explains the process we followed in building a 4 CPU multicore Nios II system that we used for the implementation of the FRSH API on multicores. Figure 4 shows a collage of a set of screenshots showing the system which has been used for the tests. The system is also available in QAR format as an appendix to this document.

The starting point of the multicore system has been a single core example available in VHDL from Altera.

Using SOPCBuilder, we designed a system composed by 4 Nios II cores with trace support. The trace pins are then exposed for each CPU on the main VHDL block produced by SOPCBuilder. A custom VHDL block provided by Lauterbach is responsible to connect these wires and to multiplex them over the Mictor connector available on the Stratix IIS60 board. The custom block is needed for two reasons: the first one is to add a delay of half a clock cycle to enable the Lauterbach debugger to better sample the data coming from the hardware traces; the second one is to enable the block to selectively trace one of the four traces produced by the CPUs. In this way, it is possible, from the Lauterbach interface, to debug the four cores using the JTAG interface, as well as selectively trace one of the

cores at a time.

The SOPCBuilder system also had a set of timers for each CPU to implement the free running timer, the system timer used for alarms, and the the specific timer used for FRSH, such as the budget, recharging, deadline check, and synchronization objects timers.

The memory interface is a single SRAM chip which is partitioned in four sections, one for each CPU. During the development of the kernel, we discovered a bug in the Altera Avalon specification which showed up as a permanent lock of the memory bus, provoking the freeze of the complete system. To solve this, an Altera support FAE informed us that the design required the addition of an additional memory bridge named in Figure 4 as `pipeline_bridge_0`. Due to the continuous conflicts on the SRAM access, and due to the presence of the additional bridge, the execution timings of this multicore hardware cannot be taken as a reference for execution time measurements, and that is the reason why Section 6 does not cover the execution timings of the various primitives but only their footprint.

Finally, Figure 4 on the bottom shows the hardware support needed for the multicore interrupts, in particular for the interprocessor interrupts (named `ipic_output`, and `ipic_input_0` to `ipic_input_4`), and for implementing the spin locks and and in general the atomic operations needed by the operating system (named `mutex`).

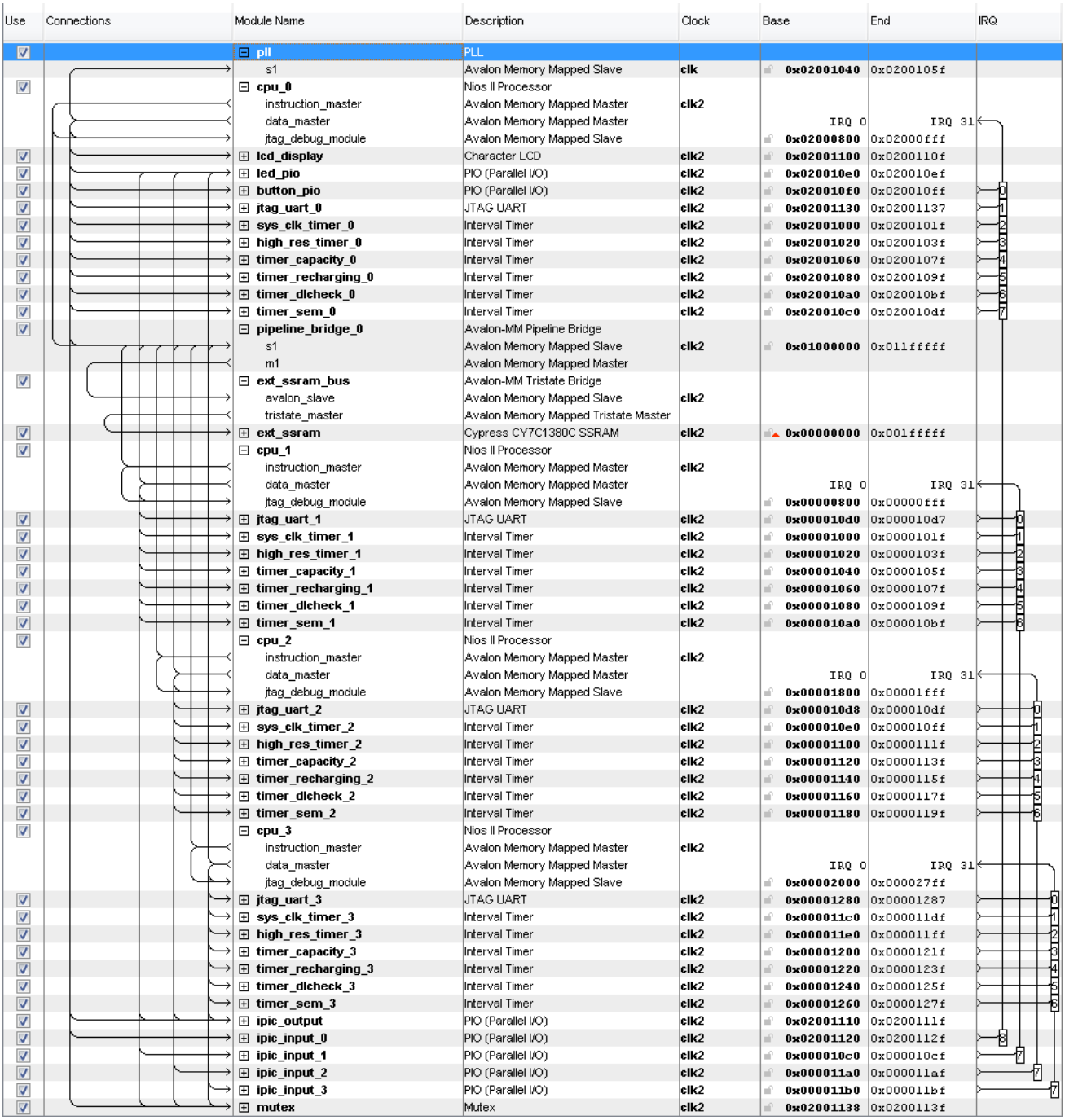


Figure 4: the Nios II 4CPU design in SOPCBuilder 8.1

6 Typical footprint and timing performance

This session includes an evaluation of the typical footprint and of the typical performance we can expect from an implementation of the FRSH API over ERIKA Enterprise. Please remember that at the beginning of the project, the target implementation footprint was around 10Kb ROM space for an implementation of a subset of the FRSH API on top of an Altera Nios II Softcore.

In the first phase of the project, we started with an initial implementation aimed at implementing the main features of the FRSH API without too much attention to the ROM footprint. As a result, the first implementation had an

initial size on Nios II of around 16Kb ROM. After that first result, we started a process of integration of the various functionalities, trying to limit redundancies in the source code as much as possible.

The results obtained in the implementation are shown in the following two tables. The first table shows the ROM and RAM Footprint with different compiler optimizations, whereas the second table shows execution times for the most common primitives.

Feature	**-O0**	**-Os**	**-O3**	**dsPIC, -O2**
Architecture specific				
ERIKA Nios II Exception handler	76	76	76	
timer handling	904	540	600	882
General functions, IRQ and thread handling				
End budget IRQ	24	4	4	3
Recharging IRQ	396	224	256	210
deadline check	260	128	128	96
End IRQ	404	252	260	162
End task	464	296	296	195
Budget and queuing	5336	2864	3232	2955
Primitives				
CancelAlarm	564	200	180	153
SetAbsAlarm	180	112	112	69
SetRelAlarm	132	64	64	48
CounterTick	1412	592	828	612
frsh_thread_bind	508	304	316	327
frsh_thread_unbind	396	244	244	249
Bind utility functions	384	168	172	162
frsh_vres_get_contract	196	72	76	63
frsh_thread_get_vres_id	240	92	92	96
frsh_init	84	60	60	81
frsh_syncobj_signal	412	276	332	324
frsh_timed_wait	700	452	452	438
frsh_syncobj_wait	656	452	452	384
frsh_syncobj_wait_with_timeout	928	608	608	555
Synchronization object utilities	1320	776	868	939
GetTime	56	36	36	15
GetResource and ReleaseResource	672	396	400	315
Schedule	136	96	96	87
ActivateTask	304	216	260	198
StartOS	24	4	4	
Single timer interrupt multiplexer				939

Total	17168	9604	10504	10557
RAM and ROM				
ROM	164	164	164	188
RAM	620	620	620	528

The table above shows the footprint of the various functionalities implemented in the system. The information has been taken from a project map generated during the compilation process of a single CPU core application composed of 5 tasks, 2 resources, 1 counter, 3 alarms, 8 contracts, and 2 synchronization objects.

As it can be seen, the footprint of the current implementation heavily depends on the compiler optimizations, but in both cases it is suitable for an implementation in medium size microcontrollers. Moreover, the proposed implementation, with compiler optimizations enabled, basically meets the initial goal of having a footprint near the 10K range.

The table also shows a fourth column, which includes dsPIC results (see Section 10) with the -O2 option. As it can be seen, the dsPIC code is generally smaller than the correspondent Nios II code (please remember that dsPIC has less registers 10) than Nios II (which has 32), and an assembler instruction size of 3 bytes (the Nios II has 4 byte instructions). Also, the dsPIC has the bottleneck of having the timer multiplexer which is used for architectures with a single timer (and not 4 timers dedicated to the various needs of the FRSH implementation, which are budget exhaustion, recharging, deadline check and synchronization object timeout).

The following table reports some typical timings for the most common primitives. The measurements have been performed on the application cited in the previous table, compiled with the -O3 compiler optimizations. Timings have been extracted from the traces by using Trace32.

Feature	**Time (microseconds)**
ActivateTask, no preemption	9.3
ActivateTask, preemption	13.4
An IRQ happens for an Altera timer. This includes counting the Altera alarm tick and discovering that there are no actions to perform. This includes entry and exit from the interrupt handler.	.14.7
An IRQ happens for VRES Budget exhaustion, the VRES of the running task is put in RECHARGING state, another running task is selected for execution, then a context switch is done on the task with the earlies deadline. This includes entry and exit from the interrupt handler.	17.8
An IRQ happens for VRES Budget exhaustion, the VRES of the running task is put in RECHARGING state. Since all VRES are in RECHARGING, 3 VRES are extracted from the recharging queue, and then a context switch is done on the task with the earliest deadline. This includes entry and exit from the interrupt handler.	24.4
An IRQ happens for the Deadline Check on 4 tasks.	20.79
An IRQ happens due to a recharging event of three tasks. The three tasks wakes up and are put in the ready queue.	30.1
A Bind operation is performed on the running task, over a VRES that has not the earliest deadline in the system. As a result, the running task is put in the ready queue and another task is scheduled.	15.8
An Unbind operation is performed on the running task. As a result, another task is scheduled.	9.5
GetResource (the primitive is non-blocking due to the fact that the system is using the SRP	1.13

algorithm)	
ReleaseResource, without preemption	6.6
ReleaseResource, with preemption	12.69

Please note that these timing information has not the intent to be a complete assessment of the execution timings of the proposed implementation, mainly because the execution time depends a lot on the specific Nios II version and memory hierarchy chosen. For this reason, we limited ourselves to a single core system.

As an additional comment, we can see that the performances obtained are worse than a minimalist fixed priority scheduler, due to the timer reprogramming and additional bookkeeping needed to implement the resource reservation, binding, and deadline checks. However, the numbers show that **the performance is still reasonable**. In particular, the interrupt handling routines are at the end only slightly more complex than an unmodified Altera timer interrupt (in any case they are always less than double the timings), whereas the other functionalities (task activations, resources) are not heavily affected by the additional code.

For that reason, we believe that the additional overhead needed to run resource reservations on small systems is balanced with the additional features implemented.

7 Debugging using Lauterbach Trace32 and Kernel awareness through OSEK ORTI

Debugging an embedded system which implements temporal isolation may cause additional problems when compared to traditional debugging of embedded systems.

In fact, to properly debug a system we need either some software probes, which inevitably perturbates the timing of the execution of the system (just think at the commonly used printf commands put in the middle of the source code), or some hardware devices which are able to either record the execution without stopping the CPU, or that are able to stop the CPU and stop at the same time the timers used to track the execution time.

With Altera Nios II, debugging the system by inserting printf in the source code is not a viable option because the printf is a heavyweight function. On the other hand, the available JTAG interface is not useful alone because it does not stop the hardware timers used to track the execution time of the tasks. This basically means that it is somehow impossible to stop the CPU and do a step-by-step debugging of the system without permanently perturbating the software execution.

For that reason, we decided to use a hardware debugger which is able to track the execution of the source code without perturbating the execution of the system. The hardware device we decided to use is the Lauterbach PowerTrace, which can be connected to the Nios II hardware using a proper tracing extension (see Figure 5). The FPGA board is the equipped with a special version of the Nios II softcore which is able to export information about the timings of execution of the instructions run by the CPU (in practice, the system is capable to trace branches, data read and data writes, which are enough to reconstruct the execution timings).

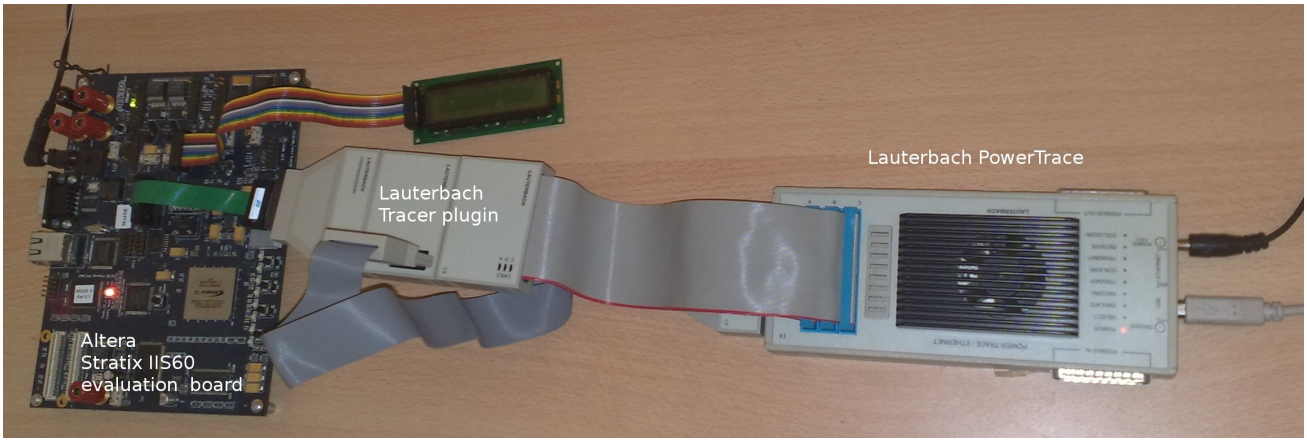


Figure 5: from left to right: the Altera Stratix IIS60 board, the Lauterbach tracer module, and the Lauterbach PowerTrace.

As a result, debugging a system such as Nios II basically consists in running the application on the target device, while recording in parallel the execution trace, to finally analyzing it offline.

The following Figures show in detail the kind of debugging details which can be obtained using the Lauterbach tools. These screenshots also show the functionalities implemented in the FRESCOR API and their effect on the scheduling done by the operating system. For the purpose of the screenshots, the timings chosen for the periods and budgets have been really small, in the order of tens of microseconds for the budget, to show on a single window the effects of the scheduling decisions. Real applications will have much lower frequency in the scheduling decisions, with negligible impact of the scheduling overhead on the system performance.

Figure 6 shows the effect of a contract change. In particular, the system is running four tasks each one with a contract of 20000 cycles over 100000 cycles (each cycle being 20ns, equivalent to a budget of 400 us over a period of 2 ms). The tasks have an indeterminate workload (basically a forever loop). The system is continuously interrupting the tasks at the end of their budget (see the execution at the line EE_nios2_IRQ_budget). Each task at that point goes into the recharging queue. At the end of the fourth task, the system recognize that all tasks have been put into the recharging queue. At that point, the system wakes up from the recharging queue a set of three tasks. The fourth task will be woken up by the recharging IRQ which appears slightly after (for the same reason the recharging IRQ will stop to fire when only three tasks will be present in the schedule, on the right side of the Figure). In the middle of the Figure, the size of the bar for Task1 increases. The reason is that a bind primitive has been called on Task1, assigning it a contract with budget of 40000 and a period of 800000. for that reason, Task1 executes now larger slices but eight times less frequent.

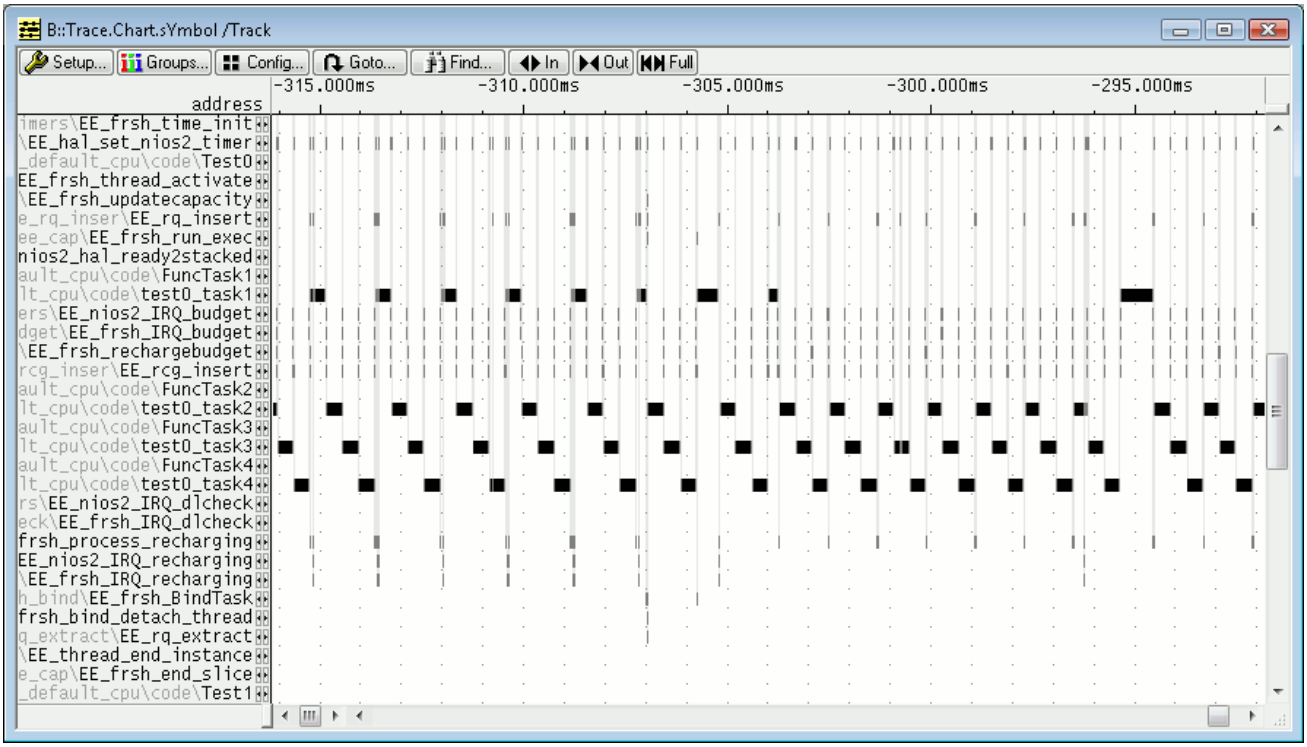


Figure 6: Four tasks with a contract change.

Figure 7 shows another scenario. In the Figure, basically all the vertical lines are related to budget exhaustion interrupts. As it can be seen, on the left of the picture Task1 has a long slice without interrupts in the middle. In that case, Task1 is executing a critical section which in turns disables the budget checking of the kernel. At the end of the critical section, when the ReleaseResource is called, Task2 is able to resume execution, gaining a set of slices to "recover" for the time stolen by Task1. Of course the example shown in the picture somehow exaggerates the phenomenon. In real scenarios, the critical sections should not be more than a few microseconds, with budgets in the order of milliseconds, provoking only slight corrections in the timeslices and not a suppression of tasks for a long period of time.

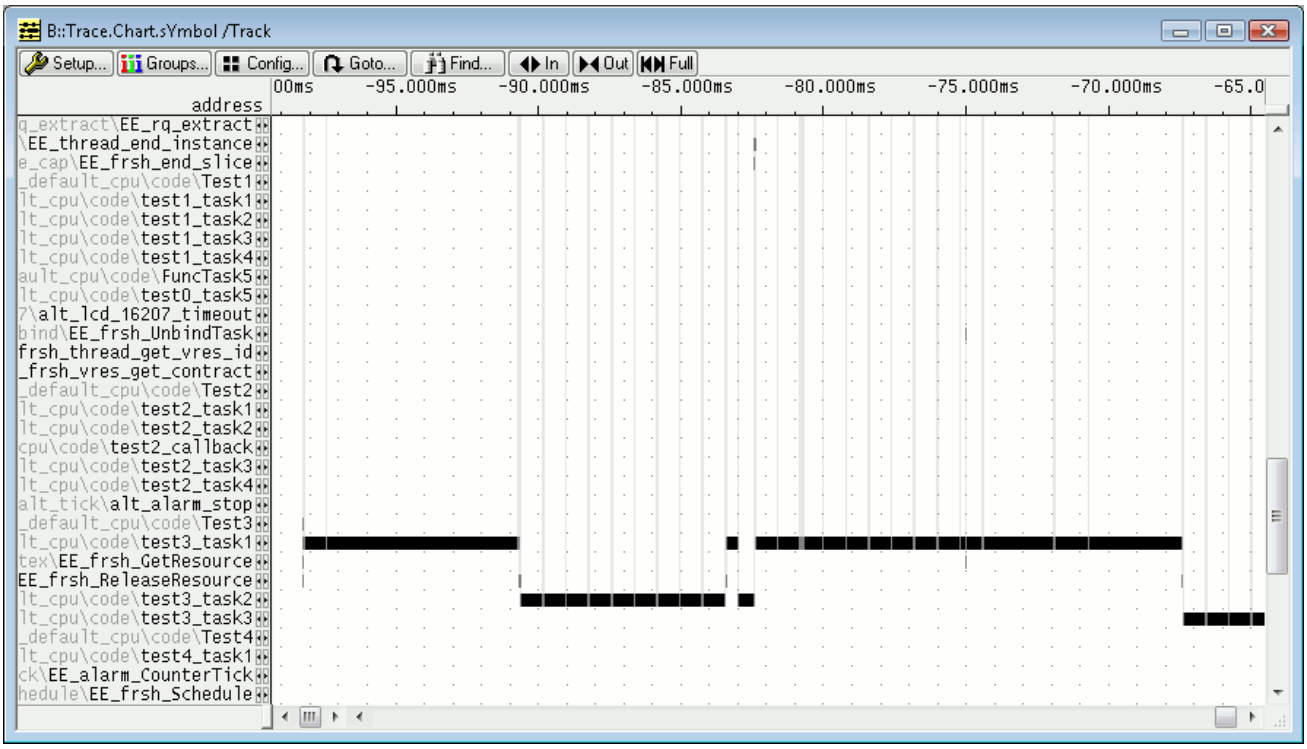


Figure 7: The effect of locking a resource for too much time.

The following Figures 8 and 9 shows the debugging environments when run on the Nios II multicore system described in Figure 4. When launching the debugger after compiling a multicore application, four windows appears, one for each CPU. Each CPU can perform step-by-step operations, breakpoints, and selective tracing on each core. In the typical configuration, CPU0 will also provoke a coordinated run, meaning that the debug commands executed by the first CPU will be forwarded to the other CPUs.

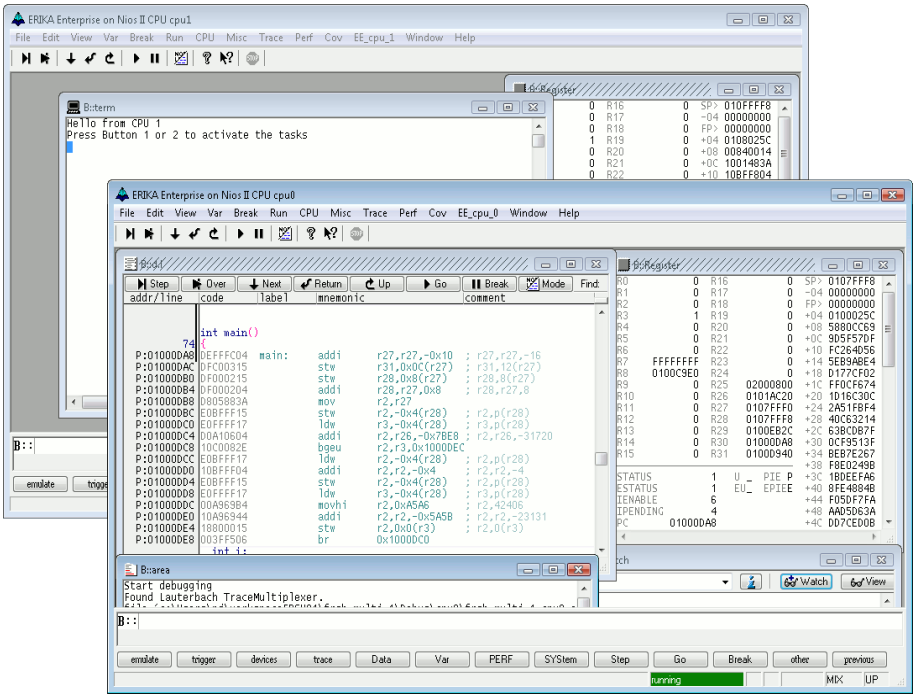


Figure 8: CPU0 and CPU1 debugging windows on a 4 cores system

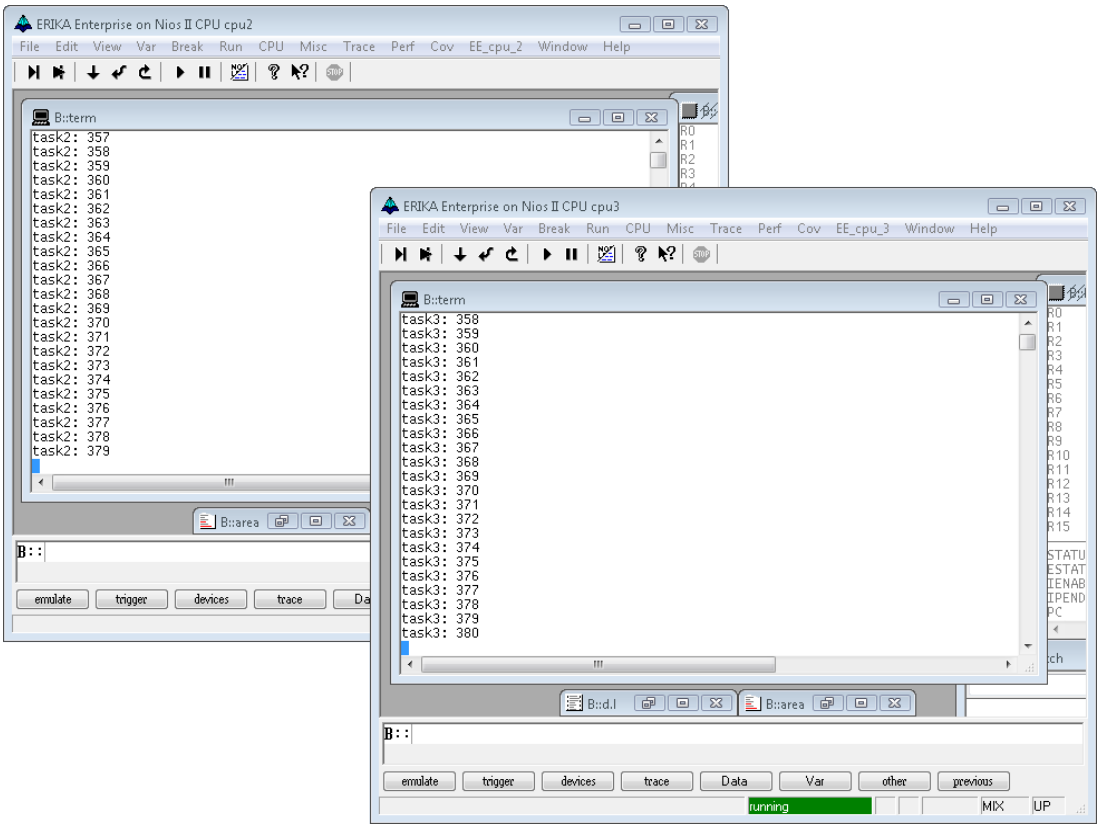


Figure 9: CPU2 and CPU3 debugging windows on a 4 cores system

The following Figures shows the behavior of a typical execution on a multicore. In this case, CPU0 runs a periodic alarm which activates a task allocated to another CPU. Figure 10 shows, from left to right, an IRQ handler start (in this case it is the Altera alarm IRQ), the execution of two `EE_frsh_thread_activate` calls that are routed to other CPUs through the `EE_rn_send` function, and finally the end of the interrupt. Figure 11 shows the result on the target CPU: an interrupt is periodically raised, activating a task that runs for some time. In particular, the figure shows the execution of two periods of the alarm. Figure 12 shows the detail of the interprocessor interrupt handler. From left to right, it can be seen the interprocessor interrupt (`EE_nios2_IIRQ_handler`), the execution of the thread activation (`EE_frsh_thread_activate`), the end of the interrupt and the start of mytask2 (`Funcmytask2`).

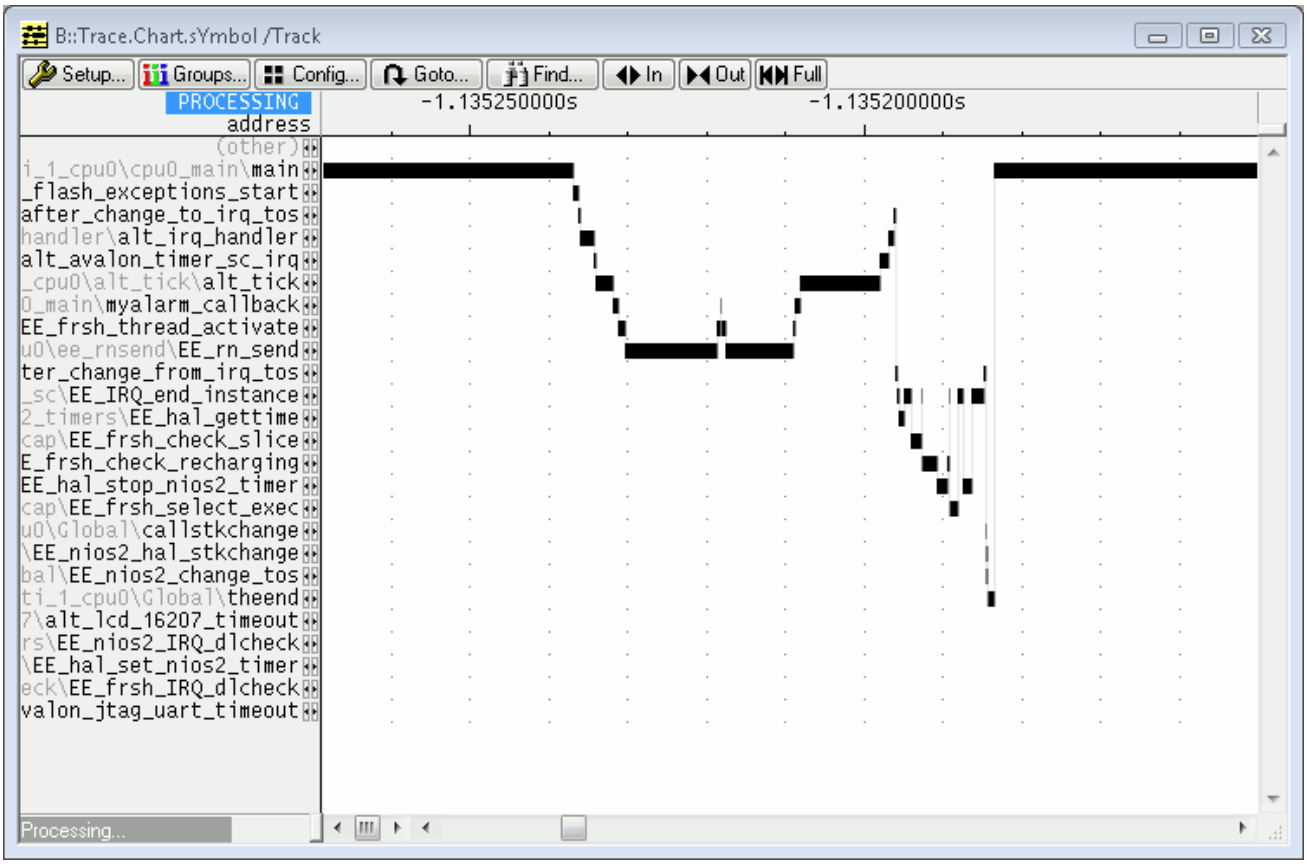


Figure 10: Activating a task allocated to another CPU

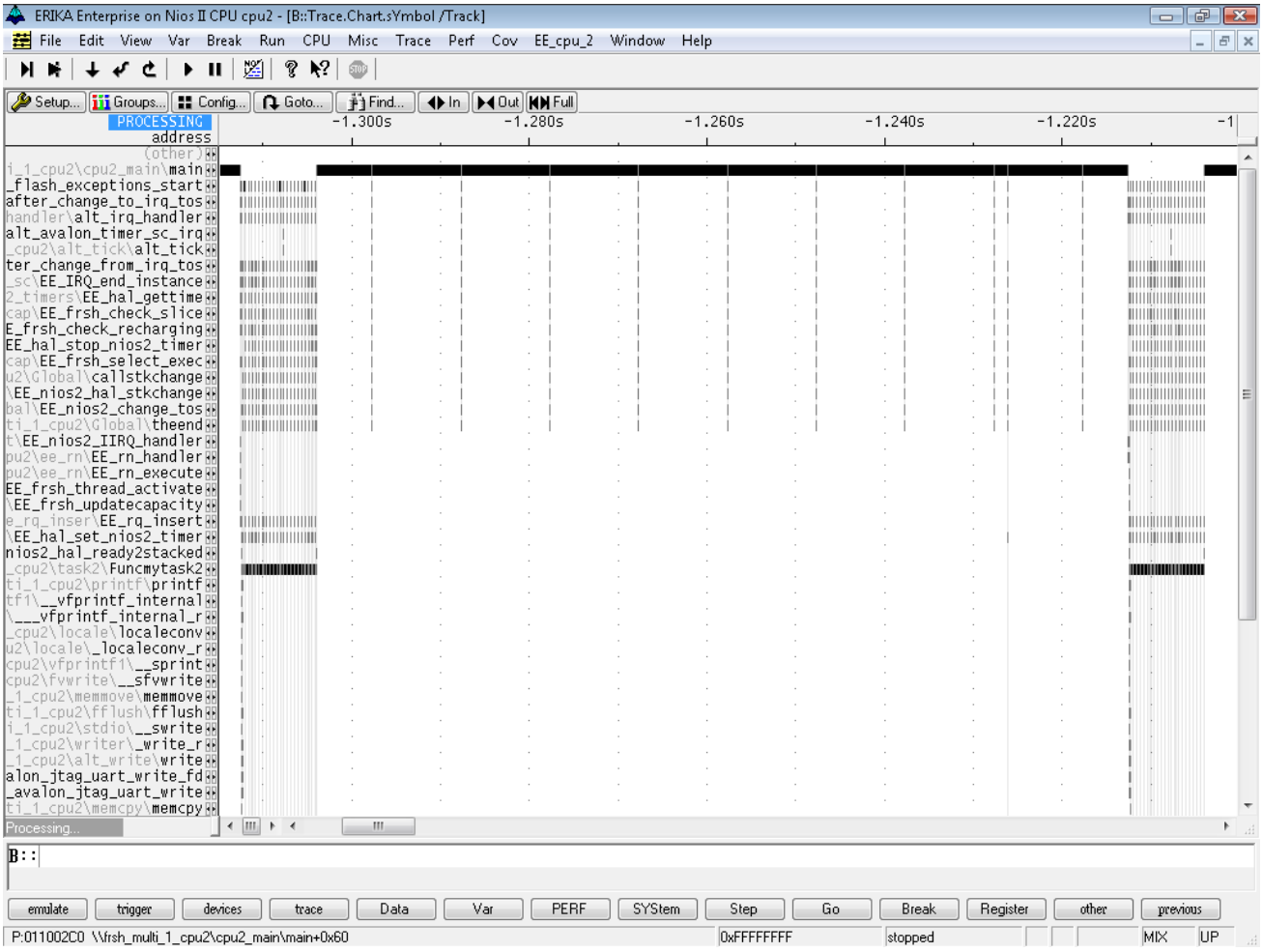


Figure 11: The remote task activation arrives as an interprocessor interrupt which activates mytask2

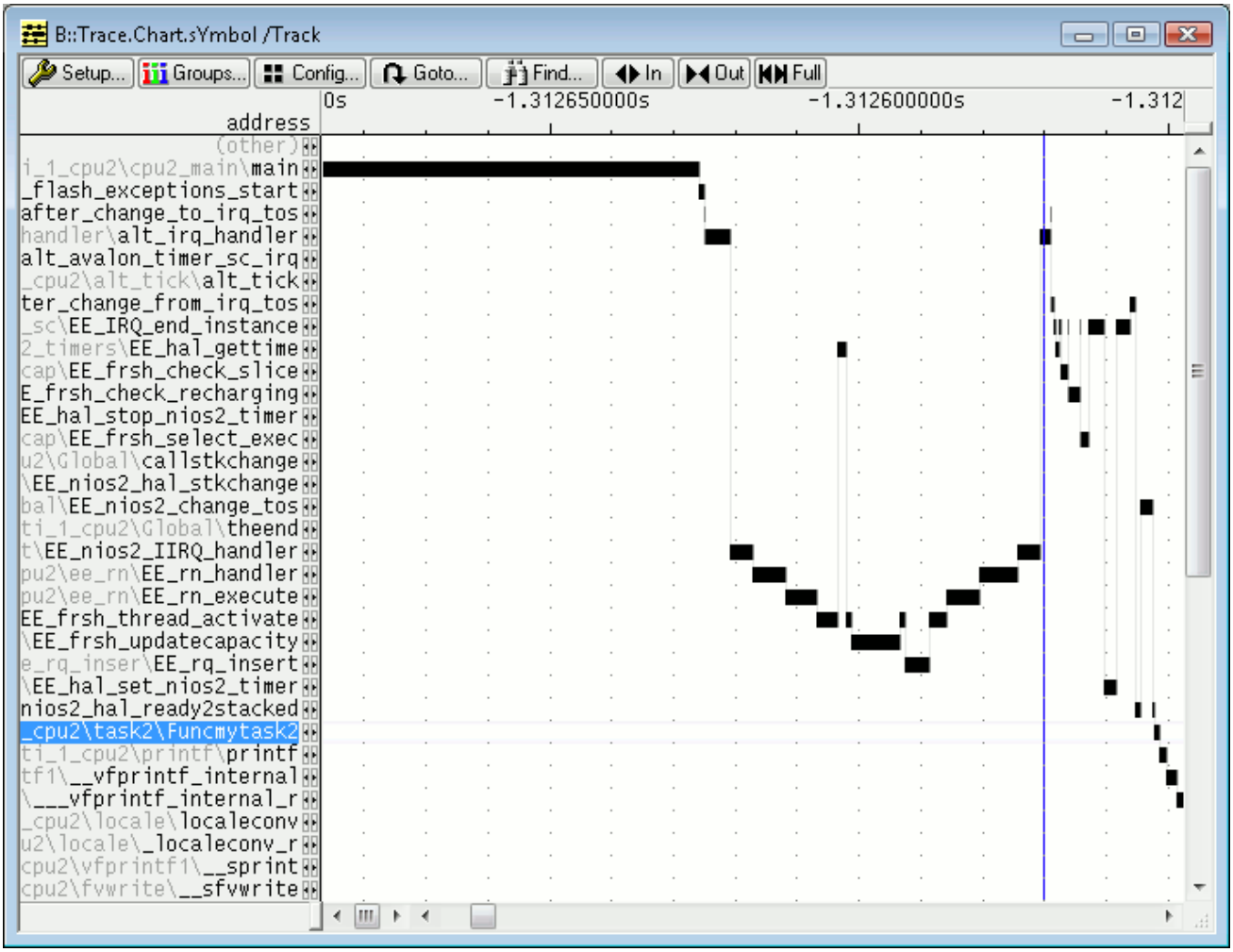


Figure 12: A detailed view of an interprocessor interrupt.

To better support the user in the analysis of the application, we also decided to implement a kernel awareness module by using the ORTI language defined by the OSEK/VDX consortium.

The ORTI language is basically a text file divided in two sections named IMPLEMENTATION and INFORMATION. The IMPLEMENTATION section is basically similar to a type description. It defines the type of an object, and the values of each field. The following is an example of the declaration of a VRES entity, which declares the types of each field, as well as an interpretation for the numerical values it may have at runtime:

```
VRES {
    ENUM "unsigned int" [
        "FREEZED    " = 0,
        "INACTIVE   " = 1,
        "ACTIVE     " = 2,
        "RECHARGING" = 4
    ] STATE, "VRES status";

    ENUM "int"[
        "NO_TASK" = "-1",
        "Task1" = 0,
        "Task2" = 1,
        "Task3" = 2,
        "Task4" = 3,
        "Task5" = 4
    ] TASK, "Task";
```

```
        ENUM "int"[
            "NO_VRES" = "-1",
            "c1" = 0,
            "c2" = 1,
            "c3" = 2,
            "c4" = 3,
            "c5" = 4,
            "c6" = 5,
            "c7" = 6,
            "clong" = 7
        ] NEXT, "Next";
        CTYPE "int" BUDGETMAX, "CT BudgetMax";
        CTYPE "int" PERIOD,    "CT Period";
        CTYPE "int" BUDGET,    "VRES Budget";
        CTYPE "int" ABSDLINE,  "VRES AbsDline";
        CTYPE "int" DLINE,     "VRES Current Dline";
    }, "Contracts and Virtual Resources";
```

Then, the INFORMATION section basically describes an expression that evaluates the memory locations where each field is stored. In this way, the debugger can make an inspection at runtime to present the information in a human readable form. The following is the INFORMATION section related to the VRES item described above:

```
VRES c1 {
    BUDGETMAX = "EE_ct[0].budget";
    PERIOD    = "EE_ct[0].period";
    BUDGET    = "EE_vres[0].budget_avail";
    ABSDLINE  = "EE_vres[0].absdline";
    DLINE     = "EE_vres[0].absdline - EE_last_time";
    STATE     = "EE_vres[0].status";
    TASK      = "EE_vres[0].task";
    NEXT      = "EE_vres[0].next";
};
```

The ORTI file is generated by RT-Druid as part of the compilation process. Then, Lauterbach is able to interpret the ORTI file, providing information which are really useful to understand the system status at a given point in time. Figure 13 shows, from top to bottom, a global view of the system, the status of each Task, the status of each Virtual Resource, and the status of the system stacks. The system stack evaluation is automatically done by the debugger comparing the stack space with a default value which has been set in the memory at boot time.

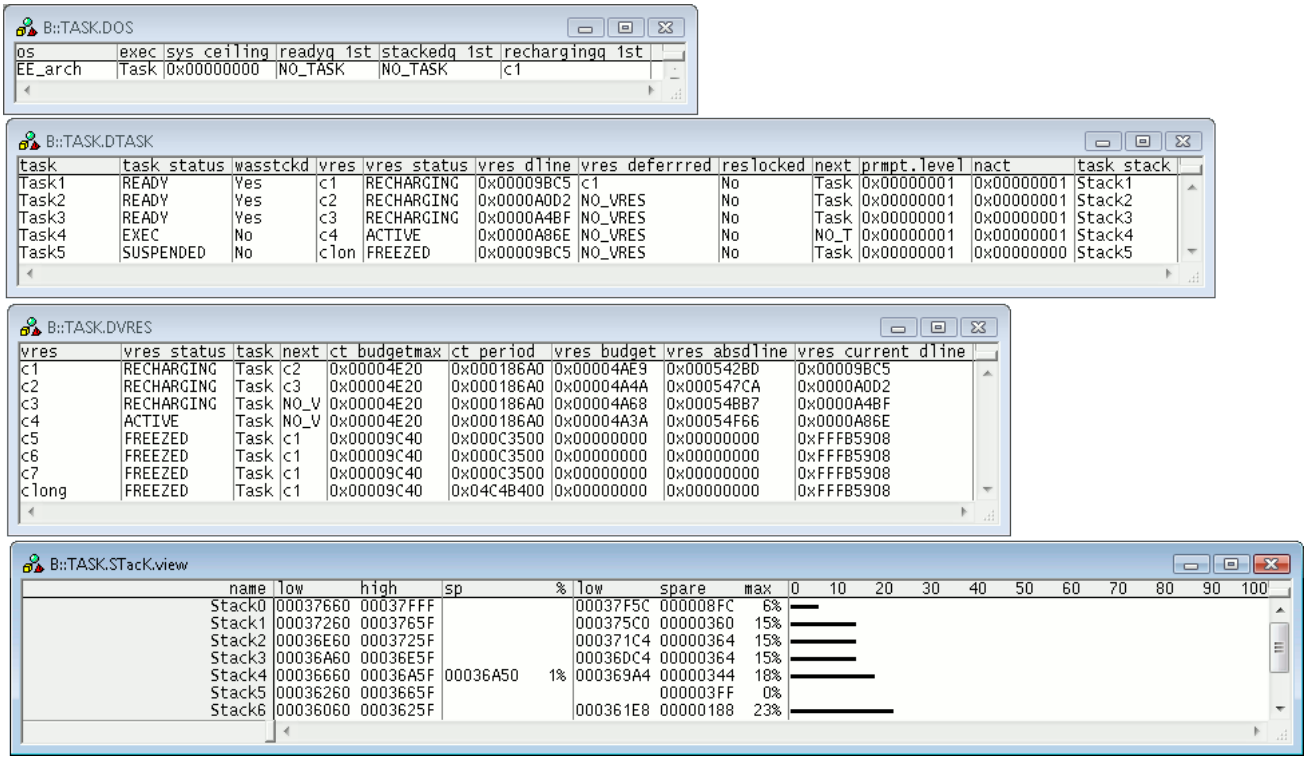


Figure 13: ORTI windows produced by Lauterbach Trace32.

Finally, the Lauterbach Trace32 is able to do other statistics, based on the fact that the ORTI file can specify the "magic" task number, which is the location where the OS stores the running task. For example, Figure 14 shows the context switches as recorded in the execution traces.

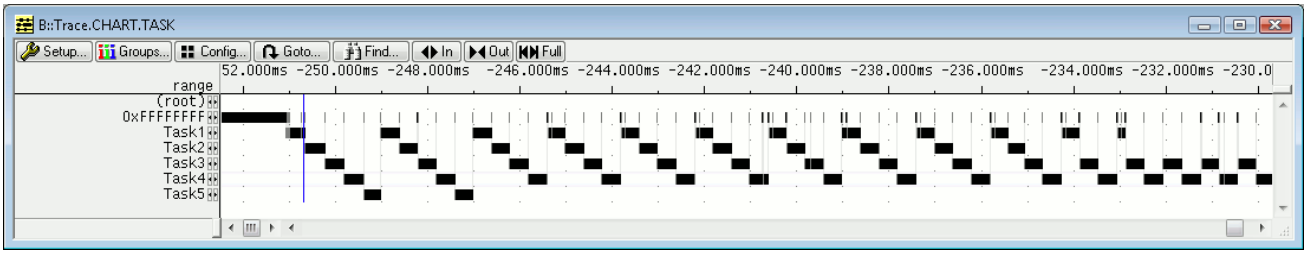
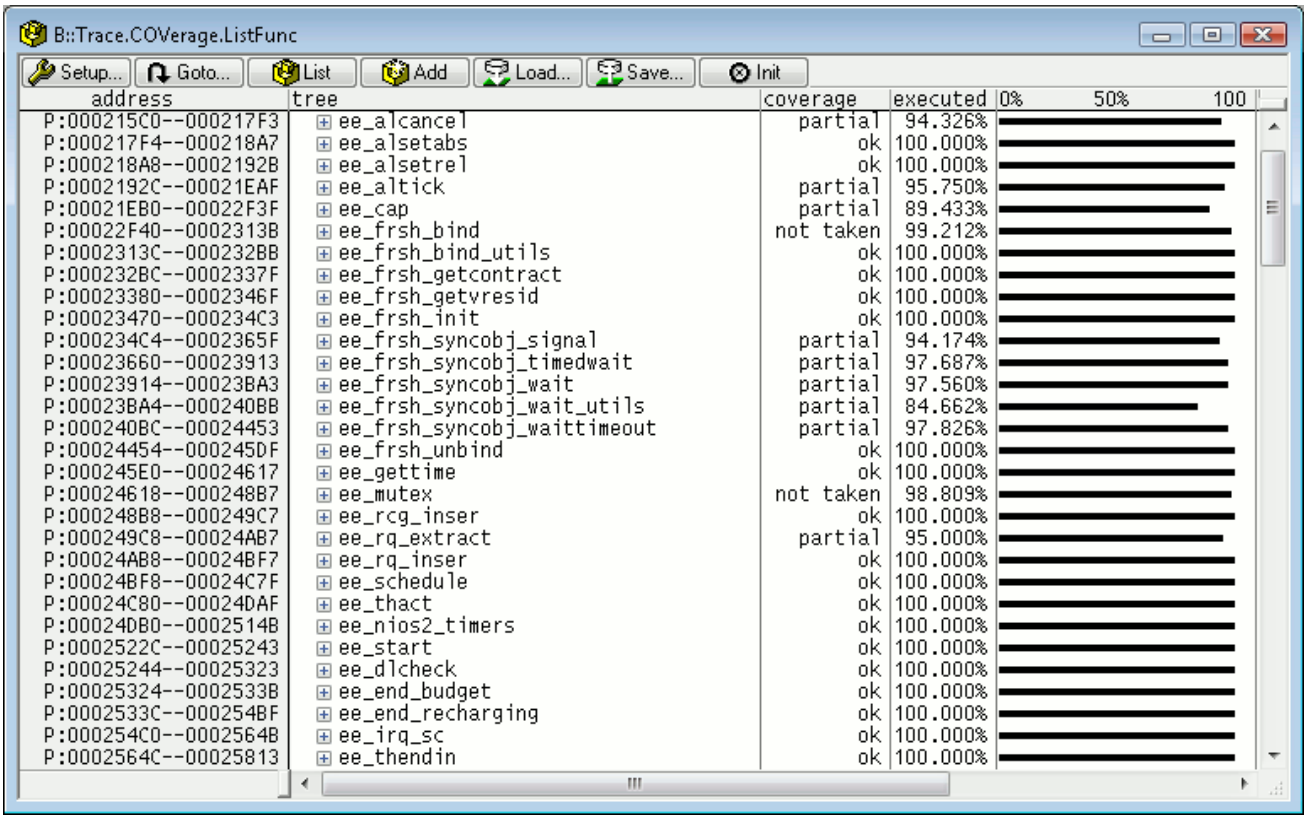


Figure 14: Task execution and context switches obtained by inspecting the recorded data trace.

8

9 Coverage Tests

To proper debug the system, we developed a set of application scenarios that calls the various primitives existing in the systems in almost all possible ways. As a result, we obtained a good coverage of the source code, as shown in Figure 15. Basically most of the system primitives implemented show a 100% coverage. The cases which has not been covered directly by the tests has then been carefully analyzed separately.



address	tree	coverage	executed
P:000215C0--000217F3	ee_alcancel	partial	94.326%
P:000217F4--000218A7	ee_alsetabs	ok	100.000%
P:000218A8--0002192B	ee_alsetrel	ok	100.000%
P:0002192C--00021EAF	ee_altick	partial	95.750%
P:00021EB0--00022F3F	ee_cap	partial	89.433%
P:00022F40--0002313B	ee_frsh_bind	not taken	99.212%
P:0002313C--000232BB	ee_frsh_bind_utils	ok	100.000%
P:000232BC--0002337F	ee_frsh_getcontract	ok	100.000%
P:00023380--0002346F	ee_frsh_getvresid	ok	100.000%
P:00023470--000234C3	ee_frsh_init	ok	100.000%
P:000234C4--0002365F	ee_frsh_syncobj_signal	partial	94.174%
P:00023660--00023913	ee_frsh_syncobj_timedwait	partial	97.687%
P:00023914--00023BA3	ee_frsh_syncobj_wait	partial	97.560%
P:00023BA4--000240BB	ee_frsh_syncobj_wait_utils	partial	84.662%
P:000240BC--00024453	ee_frsh_syncobj_waittimeout	partial	97.826%
P:00024454--000245DF	ee_frsh_unbind	ok	100.000%
P:000245E0--00024617	ee_gettime	ok	100.000%
P:00024618--000248B7	ee_mutex	not taken	98.809%
P:000248B8--000249C7	ee_rcg_inser	ok	100.000%
P:000249C8--00024AB7	ee_rq_extract	partial	95.000%
P:00024AB8--00024BF7	ee_rq_inser	ok	100.000%
P:00024BF8--00024C7F	ee_schedule	ok	100.000%
P:00024C80--00024DAF	ee_thact	ok	100.000%
P:00024DB0--0002514B	ee_nios2_timers	ok	100.000%
P:0002522C--00025243	ee_start	ok	100.000%
P:00025244--00025323	ee_dlcheck	ok	100.000%
P:00025324--0002533B	ee_end_budget	ok	100.000%
P:0002533C--000254BF	ee_end_recharging	ok	100.000%
P:000254C0--0002564B	ee_irq_sc	ok	100.000%
P:0002564C--00025813	ee_thendin	ok	100.000%

Figure 15: Coverage tests showing a really good coverage of the source code.

10 Support for Microchip dsPIC

The support for single core Microchip dsPIC has been planned during the project to ease the exploitation of the results of this task.

The strategy of making the system available for small single core microcontrollers in fact allows Evidence to reach a broader audience, basically leveraging on the community of users of the Evidence FLEX boards. A result of this effort, for example, is the paper [2] which has been developed on top of the results from this task.

The Microchip dsPIC will be included in the package of ERIKA Enterprise 1.5.0 which will be released end of May 2009. The port supports all the features described for single cores in this deliverable. Since the dsPIC device used in the implementation currently has a limited number of hardware timers, the port also included a “timer multiplexer” that makes possible to handle the timings for the various FRSH asynchronous events (budget exhaustion, deadline check, recharging interrupts, and synchronization objects timeouts) using a single 32 bit interrupt source.

11 Project Evaluation

This document shows the fulfillment of the following FRESCOR Requirements:

Ref 7.3 A: A test application will be shown running on an Altera FPGA hosting 2 or 4 Nios II processors. The system will run a subset of the FRESCOR API on the ERIKA Enterprise RTOS.
- *This document showed the support for a 4 core Nios II system*

Ref 7.5 A: A Prototype implementation on Altera Nios II and Microchip dsPIC will be developed to show the implementation of the framework on these platforms
- *This document shows the prototype for Nios II. dsPIC support will be released as part of ERIKA Enterprise 1.5.0*

Ref 8.2 A: A working prototype of a subset of the FRESCOR framework on the ERIKA RTOS (OSEK/VDX compatible)
- *ERIKA Enterprise, the OIL and ORTI support can be considered extensions to the OSEK/VDX specifications*

Ref 8.4 A: Execution support.
- *We showed an application with a total footprint of around 10k plus the application libraries and startup code. A minimal application including printf will be in the range of 30-40 Kb*

12 Summary

The first phase of this task has been devoted mainly to the specification of the features and of the subset of the FRESCOR API which will be implemented on the Nios II soft core. A first evaluation of the implementation of the EDF scheduler used as base for the IRIS scheduling algorithm has been performed. The EDF scheduler has been released as part of the ERIKA Enterprise Basic GPL version 1.4.1. Moreover, we started the porting of the ERIKA Enterprise kernel to the newest versions of the Altera tools, because the latest versions of the tools has completely changed the software build procedure requiring a redesign of the kernel build system.

In the second phase of this task we implemented the support for the IRIS scheduling algorithm, we enhanced the RT-Druid OIL compiler to support contract, we added the support for multicore systems based on Nios II FPGAs and single core systems on Microchip dsPIC, and we implemented a kernel awareness using the ORTI support provided by the Lauterbach debugger. Moreover, we performed a set of tests to show the correctness of the implementation.

13 Attachments to this document

- IRIS.PDF - The original IRIS paper in PDF [1].
- ee_appl.oil – The specification file for the OIL Language in Evidence RT-Druid.
- *cpu.qar - QAR files for single core and multicore (4 cores) designs used for making this deliverables.
- Zip file of sample applications with compilation results.
- Zipfile of the current ERIKA Enterprise kernel code, which will be released as part of ERIKA Enterprise 1.5.0 in May 2009.
- Zipfile of the map file used to compute the ROM/RAM footprint.

14 Bibliography

[1] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, Alfons Crespo, "IRIS: a new reclaiming algorithm for server-based real-time systems", Real-Time Application Symposium (RTAS 04), Toronto (Canada), May 2004
[2] Antonio Camacho, Pau Martì, and Manel Velasco, from DCS, ESAII, UPC (Barcelona, Spain), Enrico Bini (Scuola Superiore S. Anna, Italy), Implementation of self-triggered controllers, RTAS 2009 (available at http://www.evidence.eu.com/content/view/324/313/)
[3] T.M. Ghazalie, T.P. Baker. "Aperiodic Servers in a Deadline Scheduling Environment" Real-Time Systems. 9(1) July 1995.
[4] R.I. Davis, A. Burns. "Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems". In proceedings IEEE Real-Time Systems Symposium. pp. 257-268. Rio de Janeiro, Brazil. December 2006.