D3.5 Integrated Single Core Toolchain Prototype
Version 1.0

Document Information

Contract Number	249100
Project Website	www.proartis-project.eu
Contractual Deadline	m27
Dissemination Level	Restricted*/Public[1]
Nature	Prototype
Lead Authors	Jaume Abella
Contributors	All members of all institutions (BSC, RAPITA, UNIPD, INRIA, AFS)
Reviewers	Liliana Cucu
Keywords	Single Core, Toolchain

Notices:

The research leading to these results has received funding from the European Community's Seventh Framework Programme ([FP7/2007-2013] under grant agreement nº 249100.
©2010 PROARTIS Consortium Partners. All rights reserved.

[1]All Deliverables marked RE*/PU will be publically available within 6 months of their delivery to the EC

Change Log

Version	Description of change
v1.0	Initial Draft released to the European Commission

Contents

1 Introduction **5**

2 Toolchain Status Description **9**

2.1 Hardware Architecture Simulator (WP1) 9
2.1.1 Multiple-level Caches 10
2.1.2 Cache Placement and Replacement 10
2.1.3 Cache Inclusivity 10
2.1.4 Cache Allocation and Write Policies 11
2.1.5 Translation Look-aside Buffers (TLBs) 12
2.1.6 Cache and TLB Latencies 12
2.1.7 Processor Pipeline 12
2.1.8 Cache Reset Capabilities 12
2.1.9 Jitter Injection 13
2.1.10 Trace Generation and Statistics 13
2.1.11 RTOS and Application Support 13
2.2 Compiler and Run-time Libraries for Randomised Memory (WP1) . 13
2.2.1 Implementation 14
2.3 RTOS and Programming Paradigms (WP2) 15
2.3.1 POK modifications 15
2.3.2 POK services 16
2.4 Analysis Tools (WP3) 20
2.4.1 SPTA Tool and Profile Manipulation Library 21
2.4.2 R-Tool for Measurement-Based Probabilistic Timing Analysis 21
2.4.3 Hybrid Probabilistic Timing Analysis Tool 22
2.4.4 Probabilistic Response Time Analysis 23

3 PROARTIS Toolchain Installation and Usage **25**

3.1 Hardware Architecture Simulator 25
3.1.1 Installation 25
3.1.2 Usage 25
3.1.3 Software Provided 27
3.2 Compiler and Run-time Libraries for Randomised Memory 27
3.2.1 Installation 27
3.2.2 Usage 28
3.2.3 Software Provided 29
3.3 RTOS and Programming Paradigms 29
3.3.1 Installation 29
3.3.2 Usage 30

3.3.3 Software Provided 31
3.4 Analysis Tools 31
3.4.1 SPTA Tool and Profile Manipulation Library 32
3.4.2 R-Tool for Measurement-Based Probabilistic Timing Analysis 33
3.4.3 Hybrid Probabilistic Timing Analysis Tool 35
3.4.4 Probabilistic Response Time Analysis 35
Acronyms and Abbreviations 37

1

Introduction

In order to validate our findings a set of coordinated and complimentary prototypes has been developed starting from the baseline toolchain developed by month 6 (phase 1 of PROARTIS) and summarised in deliverable D3.2. This set of tools model the hardware on top of which software runs, compiler and runtime features helping randomisation, real-time operating system supporting the execution of applications and analysis tools to quantify the impact of the PROARTIS approach in terms of WCET (worst-case execution time) and average execution time.
The PROARTIS stack by months 6 and 27 is depicted in Figure 1.1. Ada support, which was a *best effort* contribution, could not be included at this stage. However, full support for C/C++ programs has been achieved as planned in the DoW. Programs are compiled to be run on top of the hardware, which is modelled by means of a hardware performance simulator (*proartis_sim*). Those programs may be time-randomised by means of compiler/runtime support. Programs are managed by the operating system lying between the program and hardware layers. Finally, statistics and traces obtained are analysed to obtain timing information.

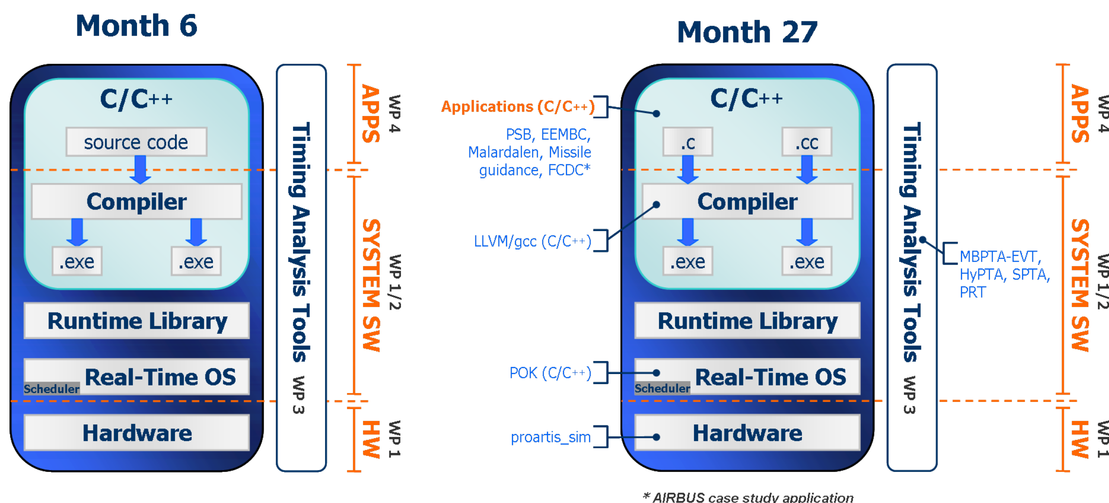

Figure 1.1: PROARTIS stack by month 6 (left) and by month 27 (right)

The set of tools implementing the stack and the way they interact is as follows (see Figure 1.2):

- Applications are compiled to run on top of the whole infrastructure. Eventually, software randomisation techniques (see deliverable D1.2) can be used. Those

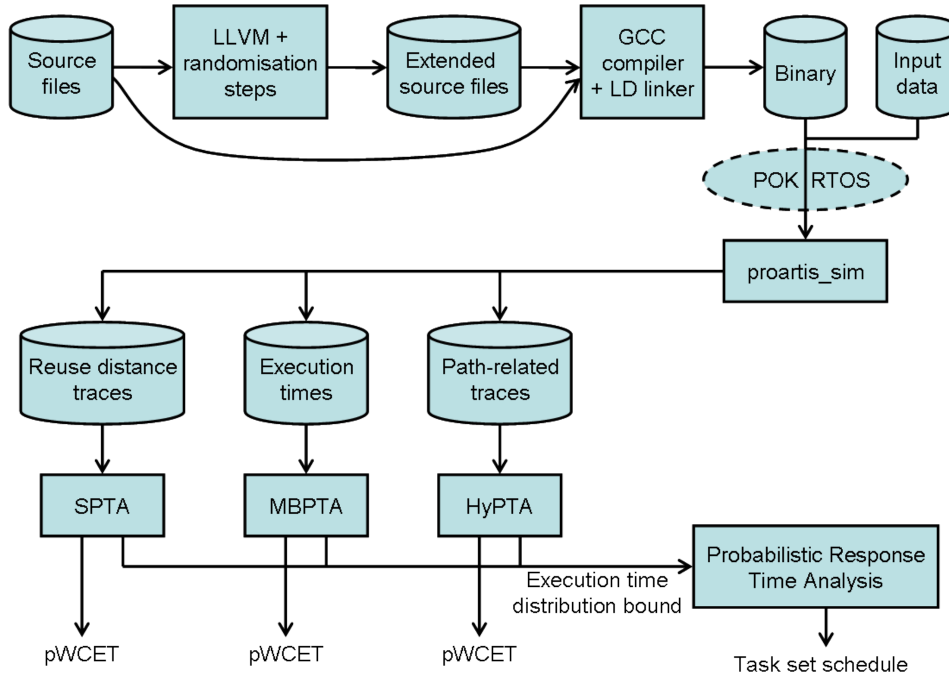

Figure 1.2: Data flow of the PROARTIS toolchain

techniques randomise the code layout and stack location of the application at runtime, but they are implemented as a compilation step in LLVM [1] so that the runtime library is attached to the application binary.

- Application binaries may be run on top of the operating system (POK [2]), which implements the composable system software services, as described in deliverable D2.2.

- Applications (and the operating system) run on top of the hardware layer (*proartis_sim* tool), which implements the hardware features required for PTA (probabilistic timing analysis) techniques as described in deliverable D1.2. As a result, the execution of applications produces statistics such as execution time (cycle count), hits/misses in each cache memory, etc., as well as different sets of traces required for the analysis techniques described in deliverable D3.4.

- In particular, MBPTA (measurement-based probabilistic timing analysis) uses execution times of a number of runs of multiple relevant inputs of the application to perform end-to-end runs and obtain a distribution bounding from above the execution time distribution of the application. Such a distribution is used to derive pWCET estimations.

- SPTA (static PTA) uses traces describing reuse distances across memory accesses to derive a highly accurate bound of the execution time distribution of the application. However, SPTA is only viable for single-path programs and for very simple hardware architectures, as opposed to MBPTA, which does not pose any constraint on the number of paths of the program or the complexity of the architecture (as long as it adheres to the properties described in deliverable D1.2). SPTA is useful, however to understand the implications of some hardware designs and to further validate MBPTA and HyPTA techniques.

- HyPTA is particularly useful to deal with programs for which relevant inputs for end-to-end runs cannot be provided by the user who, instead, can provide relevant inputs for individual parts of the application. HyPTA combines MBPTA results at the level of those parts and information about paths provided by the *proartis_sim* to build a safe distribution of the execution time of the application as described in deliverable D3.4.

- Finally, a probabilistic response time (PRT) analysis tool uses execution time distributions such as those provided by SPTA, MBPTA and HyPTA tools to schedule task sets.

This deliverable describes the tools implemented to satisfy the requirements of each of the layers in the stack and their status. Then, notes about their installation and usage are provided.

2

Toolchain Status Description

The toolchain consists of several tools interacting coordinately. In particular the toolchain includes a simulator modelling the processor at hardware level, a compiler/run-time layer performing software randomisation at the user level, a real-time operating system (RTOS) layer providing system-level services to the user applications, and a set of tools to analyse the behaviour of the software under analysis in terms of timing. Next, we describe the features of all those components.

2.1 Hardware Architecture Simulator (WP1)

The hardware performance simulator (*proartis_sim*) implements the hardware support for PTA as described in deliverable D1.2. It has been extended during this phase as described in tasks T1.2 and T3.5. In particular, the enhancements are listed next and described in coming sections:

- **Multiple-level caches**. The simulator allows modelling cache hierarchies with two levels of cache. The first level includes data and instruction caches. The second level can be configured to model separate or unified second level caches.
- **Cache placement and replacement**. Cache models include several placement and replacement policies. A PRNG (pseudo-random number generator) has been implemented to provide the required support for random placement and replacement.
- **Cache inclusivity**. Second level caches can be inclusive or non-inclusive.
- **Cache allocation and write policies**. Caches can be either write-through non-write allocate, or write-back write allocate.
- **Translation look-aside buffers (TLBs)**. TLBs have been incorporated. They are single level.
- **Cache and TLB latencies**. All caches and TLBs have parametrisable hit and miss latencies.
- **Processor pipeline**. The processor models a 4-stage pipelined processor.
- **Cache reset capabilities**. Cache contents can be reset on demand either through particular parameters or special instructions.

- **Jitter injection**. Effects of interfering software can be mimicked by means of particular parameters introducing cache accesses causing evictions.

- **Trace generation and statistics**. The simulator can generate traces required for either static probabilistic timing analysis (SPTA) and for RapiTime timing analysis.

- **RTOS and application support**. The emulator has been significantly extended with some features required by the RTOS and the case study applications. This part required huge effort.

Next we describe those features in more detail. How to use them is described later in section 3.

2.1.1 Multiple-level Caches

Current CRTES hardly use cache memories due to the complexity of their timing analysis. The PROARTIS probabilistic approach, instead, does not pose any constraint on the number and arrangement of those caches as long as they implement random placement and replacement (see deliverable D1.2). This is a key achievement and huge advance with respect state-of-the-art. Therefore, multiple-level caches, which are the common case for non-hard real-time systems, have been implemented in *proartis_sim*. In particular, cache hierarchy can be easily configured to allow different configurations as shown in Figure 2.1. Those configurations include single-level cache hierarchies (data, instructions or both) and multiple-level hierarchies by including second level caches (data, instructions, both or unified). As shown, there is high flexibility to configure the cache hierarchy.
Few constraints are imposed in the modelling of those cache hierarchies. Those constraints are as follows: (i) the cache line size of second level (L2) caches must be equal or larger than for first level (L1) caches, and (ii) the size of the L2 cache must be equal or larger than for L1 caches. Note that such constraints emanate from the implementation of cache hierarchies in any system, so they are not introduced by the PROARTIS technology. In fact, to the best of our knowledge there is no processor implementing multiple-level cache hierarchies not fulfiling those properties.

2.1.2 Cache Placement and Replacement

Proartis_sim implements several placement and replacement policies. For our purposes we have implemented random placement, random replacement with eviction on miss and random replacement with eviction on access. For comparison purposes we have implemented modulo placement and LRU replacement. Any combination of placement and replacement policies is allowed. Randomisation uses the multiply-with-carry PRNG [6] that has been implemented together with cache modules.

2.1.3 Cache Inclusivity

Inclusive caches are those such that any content in a particular lower level cache must be also present in the current cache. For instance, if a L2 cache is inclusive all contents of L1 caches whose L2 cache is this one must also be stored in such

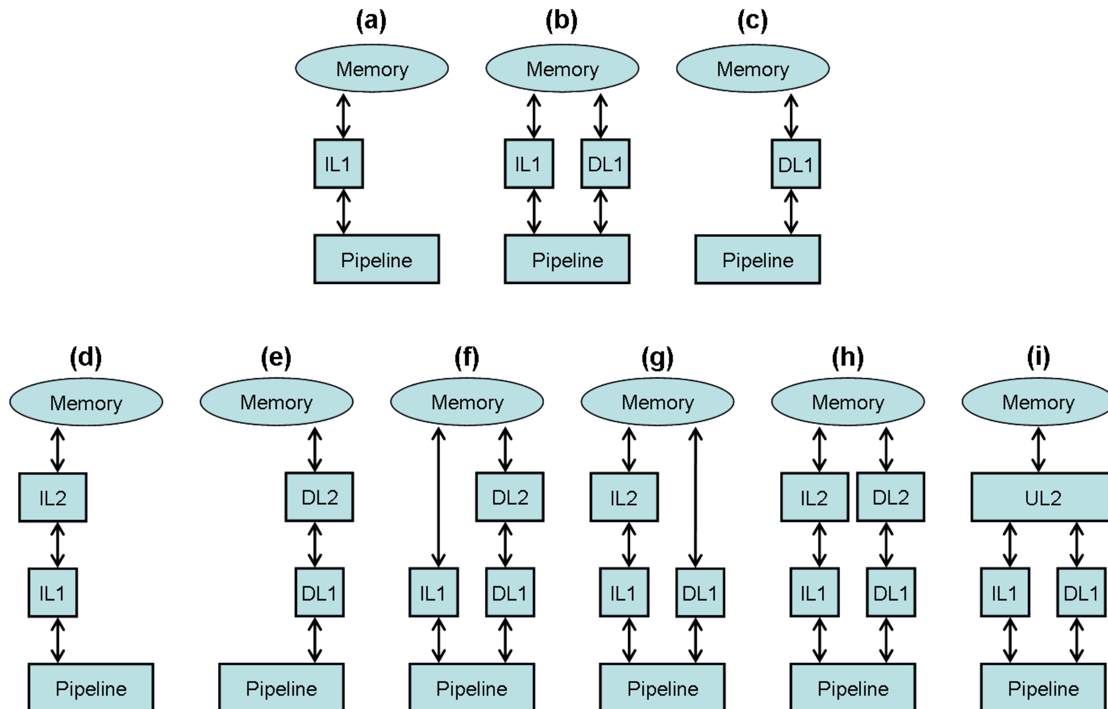

Figure 2.1: Single-level and two-level cache configurations allowed. IL1 and DL1 stand for first level instruction and data cache respectively. IL2, DL2 and UL2 stand for instruction, data and unified second level cache respectively.

L2 cache. Recalling Figure 2.1 (d), if IL2 is inclusive all contents in IL1 must be also stored in IL2. Whenever an eviction is performed in IL2, such line must be also evicted from IL1 (if present in IL1). Inclusive caches are typically used for data to ensure that write operations are performed correctly and easily in write-back caches. L1 data (DL1) write-back caches keep dirty data when written and, eventually, those lines are evicted. If their L2 cache is inclusive, those cache lines can be sent to L2 without any further check because they are present in L2 and therefore, cascade evictions and stalls due to a L2 miss will not occur.
Analogously, non-inclusive caches have been also implemented. Opposedly to inclusive caches, non-inclusive caches do not impose any constraint about inclusivity, so any particular L1 line may or may not be in L2. These designs are particularly useful for instruction caches since instructions are not modified and therefore, lines can be simply removed on a replacement without any further constraint.
Finally, exclusive caches enforce L1 and L2 contents intersection to be empty. Although this could be implemented in theory, to the best of our knowledge no processor implements such a policy. Thus, this policy has not been implemented in *proartis_sim*.

2.1.4 Cache Allocation and Write Policies

There are two common cache allocation policies on a miss: write allocate and non-write allocate. The former fetches cache lines on a write miss whereas the latter does not. Similarly, there are two write policies: write-through and write-back. The former updates the current cache level on a write operation and forwards the write operation to the upper cache level so that there is a copy of that cache line some-

where else and it can be simply removed if needed. The latter updates the current cache level only in a hit, so that the current copy of the data is the only updated copy. Typically, write-through caches are non-write allocate, whereas write-back caches are write allocate. Thus, we have implemented these two combinations.

2.1.5 Translation Look-aside Buffers (TLBs)

TLBs are typically used to translate virtual addresses into physical addresses. We have extended *proartis_sim* with data (DTLB) and instruction TLBs (ITLB). Those TLBs can be configured analogously to single-level data and instruction caches (see Figure 2.1 (a)-(c)). They are accessed in parallel with data and instruction caches. If they are properly sized, cache and TLB accesses can be served in parallel as long as there is a TLB hit. If this is not the case, the TLB miss must be served before processing the cache access because the translated address is required to access the L2 cache in case of a L1 miss. Our implementation operates this way and allows any combination of L1, L2 and TLB hierarchy among those available for each structure.

2.1.6 Cache and TLB Latencies

All caches and TLBs have independent parameters to configure their hit and miss latencies. Note that miss latencies for the last-level caches work as memory latency.

2.1.7 Processor Pipeline

The processor implements a 4-stage pipeline fulfilling the PROARTIS requirements and thus, avoids timing anomalies by construction. In the first stage (fetch) instruction cache and TLB are accessed. In the second stage instructions are decoded (fixed 1-cycle latency). In the third stage (execute) instructions are executed experiencing a fixed latency except data cache accesses, whose latency depends on whether they hit or miss in cache and TLB. The processor allows stores not to stall the pipeline even if they miss in cache by mimicking a store buffer. Finally, the fourth stage (writeback) has a fixed latency (1-cycle) to update the processor state when instructions retire. Buffers are set up across stages to mitigate the impact of stalls in execution time.

2.1.8 Cache Reset Capabilities

As part of the experiments, cache may be reset at will to emulate the case where caches are empty. For that purpose we have enabled two different means to perform such cache reset: (i) a particular parameter that resets caches when a particular program counter is reached and (ii) a specific instruction from the ISA (instruction set architecture) that does not modify the architectural state but resets cache contents. This feature has been particularly useful to study the effects of the initial conditions in the execution of the program under study.

2.1.9 Jitter Injection

Disturbing software is software that evicts some useful cache contents for a particular program. For instance, on library call the code of the call can evict some cache contents useful for the caller. In order to easily study the effect of disturbing software in cache we have enabled a feature that allows performing cache evictions as needed without requiring implementing any disturbing software, thus facilitating the generation of results.

2.1.10 Trace Generation and Statistics

Different trace formats are required for different purposes such as generating the theoretical execution time distribution (SPTA) and processing data with RapiTime. *Proartis_sim* allows generating those different traces. Furthermore, *proartis_sim* produces statistics with the number of execution cycles and detailed information for each cache and TLB including hits, misses, read hits, read misses, write hits, write misses, evictions requests received, evictions performed and uncacheable requests processed.

2.1.11 RTOS and Application Support

Several significant enhancements have been performed in *proartis_sim* emulator to support the case study and the RTOS. The main ones are as follows:

- A Memory Management Unit (MMU) together with the registers required have been implemented in the emulator to enable the RTOS to perform page management for different applications.

- Extra control registers have been incorporated into the emulator to support the RTOS and AFS applications.

- A particular instruction has been implemented to allow the application or the RTOS to reset cache state.

- In order to simplify the way data for the case study are collected, a new register has been created in the emulator to allow the application or the RTOS retrieve a random value produced by a PRNG.

2.2 Compiler and Run-time Libraries for Randomised Memory (WP1)

As part of the contributions towards tasks T1.2 and T1.3, we have developed a compiler and run-time system that randomises the location of functions and stack frames into memory. The new software component, called *Stabilizer*, generates a new binary for the program under study containing the functionalities described in deliverable D1.2. As stated before, support is provided for C/C++ programs, but not for Ada programs. Ada support was intended as a *best effort* feature, but effort had to be devoted to the C/C++ main track due to the complexity and cost of integrating all components in the toolchain.

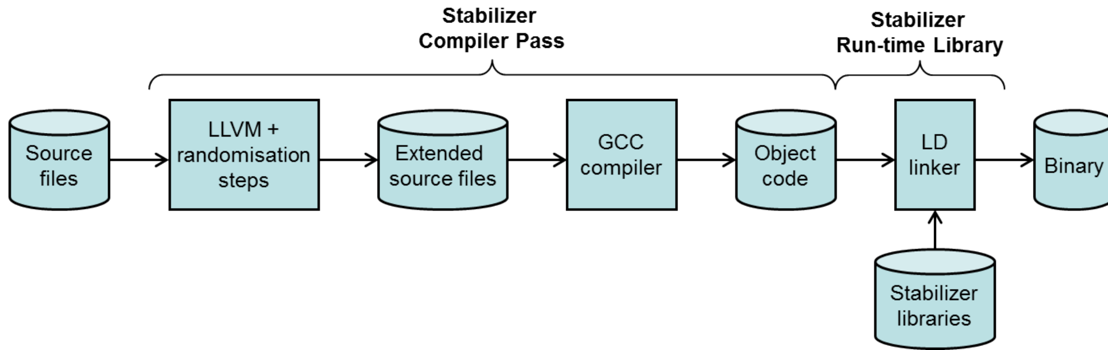

Figure 2.2: Steps for software randomisation

Figure 2.2 shows all the compiler and linker steps required to generate the new binary. In a first step, source files are transformed by the Stabilizer compiler pass (implemented on top of LLVM compiler [1]), which introduces the code required to perform function and stack frame randomisation. In second step, the resultant object files are then linked together with the Stabilizer run-time library, which provides support to random memory allocation. The Stabilizer run-time library is based on DieHard memory allocator [5] that uses heap randomisation to prevent memory errors.
Next we describe the implementation issues we have encountered to implement and port Stabilizer to the PROARTIS tool-chain. How to use them is described later in section 3.

2.2.1 Implementation

Stabilizer compiler and run-time system has been developed to be executed into real x86 and PPC systems and within the PROARTIS tool-chain. The first version of Stabilizer was available on x86 and PowerPC real machines due to its dependency on DieHard. In order to guarantee the correctness of DieHard integration into Stabilizer, we decided to implement first Stabilizer on a real machine. Unfortunately, the porting to the PROARTIS tool-chain was more complex than we initially planned due to two major reasons:

- Our simulation environment for examining the viability of the method is an embedded system without any operating system. This means that the executable needs to be *cross-compiled* in another machine and linked *statically* with all the dependent libraries. Moreover, all the calls of the application to the operating system (system calls) need to be implemented by the firmware which is linked together with the application.

  Stabilizer requires significant support from the operating system due to DieHard memory allocator. The most important one is the ***mmap()*** system call, which attaches a portion of a memory space in the virtual space of the application. In *proartis_sim*'s environment, this has required the implementation of a memory allocator, thus replacing the simple one available before, which allocates memory similarly to a stack. The new mmap() implementation calls internally from the system call the old memory allocator. Additionally, there is a need for accessing system's random number generator (/dev/urandom) from Stabilizer. In order to

reduce the overhead of such a system call, the simulator has been enhanced to provide access to such random number generator by reading a special hardware register, which returns a random value.

- The LLVM support for PowerPC targets Mac systems (not surprisingly, since LLVM is the basis of Apple's software development toolchain). For this reason, the PowerPC assembly code produced by the LLVM's code generator contains directives not understood by the GNU assembler. Therefore they have been replaced by their equivalent ones for Linux PowerPC so that they can be properly compiled.

2.3 RTOS and Programming Paradigms (WP2)

The RTOS we decided to adopt in the PROARTIS toolchain is POK [2], a minimalist kernel for real-time embedded systems that provides partitioning functionalities. The decision of adopting that RTOS is based on the considerations that it is an Open Source operating system (the code has been released under the BSD license) and it provides a basic implementation of some ARINC653 APIs (application programming interfaces). POK is, therefore, the research vehicle for implementing the RTOS features described in deliverable D2.2.

2.3.1 POK modifications

Before running POK on *proartis_sim* it has been necessary to fix some bugs that prevented the correct execution of POK on the PowerPC platform. In particular, we have found and fixed a stack overflow error; this error was caused by a wrong thread stack management, which was growing at each system call. We have also fixed other minor bugs, including one in the Round Robin scheduler procedure.
Others modifications has been applied both to properly run POK on *proartis_sim* and to meet the AFS requirements that originate from the analysis of the applications that will be used in the experiments. The modifications performed are the following:

- **Implementation of the floating point registers** (and modification of the context switch procedure accordingly);

- **Provided support for partition larger than 1Mb** as required by AFS. The memory layout of the POK code image (specified in the kernel linker script) has been reversed so that partitions are placed below the kernel;

- **Disable the interrupts during the syscalls execution;** the interrupts are enable during the idle thread execution only;

- **Reimplementation of the printf function** so that it can be executed on *proartis_sim*;

- In the ISI and DSI interrupt handler, **replacement of the infinite loop** (executed in case of error) **with the new-implemented** *exit* **function**: this has been mainly realised for debug purposes;

- **Modification of the base virtual address of every partition** in order to leave empty the first page of every partition (this eases the identification of the null pointer). The partition linker script has been modified accordingly;
- **Tuning of the Decrementer frequency**;
- **Timing service initialised after the POK boot** (as it is useless to program the first tick to elapse prior to complete the POK boot).

Moreover, we have also implemented the Fixed-priority scheduler (not provided by the original POK) as required by AFS.

2.3.2 POK services

In its original version, POK provides a set of services that can be invoked by the application code, and that mainly implement the management of:

- **Threads** (e.g. creation, start and stop);
- **Errors**;
- **Inter and intra partition ports communications**;
- **Events and Semaphores**.

Since POK aims to be compliant with the ARINC653 standard, it also makes available a minimal set of ARINC653 APIs that provides a small number of basic services. However, analysing the original POK implementation of these services (whether present) we realised that most of them are not fully ARINC653 compatible (furthermore some APIs are merely stubbed out) and needed to be implemented or heavily modified. We then selected the APIs that are required to run the FCDC application provided by AFS, and following the AFS guidelines, (re)implement them in order to make them compliant with the ARINC653 specifications.
The ARINC653 APIs required by the AFS for running the FCDC application are listed as follows:

- **Process management** APIs:
  - PERIODIC_WAIT;
  - GET_TIME;
  - CREATE_PROCESS;
  - STOP;
  - START.
- **Partition management** APIs:
  - GET_PARTITION_STATUS;
  - GET_PARTITION_START_CONDITION;
  - SET_PARTITION_MODE.
- **Sampling ports management** APIs:
  - CREATE_SAMPLING_PORT;

  - WRITE_SAMPLING_MESSAGE;
  - READ_SAMPLING_MESSAGE;
  - GET_SAMPLING_PORT_ID.

- **Queuing ports management** APIs:
  - CREATE_QUEUING_PORT;
  - SEND_QUEUING_MESSAGE;
  - RECEIVE_QUEUING_MESSAGE;
  - GET_QUEUING_PORT_ID.

- **Error management** APIs:
  - CREATE_ERROR_HANDLER;
  - GET_ERROR_STATUS;
  - RAISE_APPLICATION_ERROR.

The previous services can be invoked by the application code. The application code of every partition is made up of a main thread (that is thread in charge of creating the other threads of the partition and start them), and of "normal" partition threads. The main thread executes only once (at the beginning of the first slot assigned to the partition). The "normal" partition threads can be aperiodic (execute only once) or periodic (as its name suggests, a periodic thread executes at regular intervals). The creation of the partition(s) and of the main thread(s) is performed by the POK kernel during the boot according to the configuration directives specified by the application developer (see later). In this document thread and process are used as synonyms.
In the following we provide a brief description of every ARINC653 service enumerated above.

PERIODIC_WAIT

The PERIODIC_WAIT service request suspends execution of the requesting periodic process until the next release point in the processor time line that corresponds to the period of the process. It is usually invoked in the main procedure of a "normal" process as last instruction of a while true loop.

```
procedure PERIODIC_WAIT
          (RETURN_CODE : out RETURN_CODE_TYPE)
```

GET_TIME

The service GET_TIME requests the current value of the system clock. The system clock is the value of a clock common to all partitions in the core module.

```
procedure GET_TIME
          (SYSTEM_TIME : out SYSTEM_TIME_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

CREATE_PROCESS

The CREATE_PROCESS service request creates a process and returns an identifier that denotes the created process. This services is typically invoked by the main thread of each partition. For each partition, as many processes as the preallocated memory space will support can be created (as explained later, the number of process for each partition must be statically defined). The creation of processes (e.g., names used, number of processes) for one partition has no impact on the creation of processes for other partitions. The entry point of every process is set to the main procedure that the process should execute (and that typically is written by the application developer).

```
type PROCESS_ATTRIBUTE_TYPE is record
     NAME : PROCESS_NAME_TYPE;
     ENTRY_POINT : SYSTEM_ADDRESS_TYPE;
     STACK_SIZE : STACK_SIZE_TYPE;
     BASE_PRIORITY : PRIORITY_TYPE;
     PERIOD : SYSTEM_TIME_TYPE;
     TIME_CAPACITY : SYSTEM_TIME_TYPE;
     DEADLINE : DEADLINE_TYPE;
end record;

procedure CREATE_PROCESS
          (ATTRIBUTES : in PROCESS_ATTRIBUTE_TYPE;
          PROCESS_ID : out PROCESS_ID_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

STOP

The STOP service request makes a process ineligible for processor resources until another process issues the START service request. This service allows the current process to stop the execution of any process except itself.

```
procedure STOP
          (PROCESS_ID : in PROCESS_ID_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

START

The START service request initialises all attributes of a process to their default values. It is typically invoked by the main thread after the processes creation. If the partition is in the NORMAL mode, the process' deadline expiration time and next release point are calculated.

```
procedure START
          (PROCESS_ID : in PROCESS_ID_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

GET_PARTITION_STATUS

The GET_PARTITION_STATUS service request is used to obtain the status of the current partition.

```
procedure GET_PARTITION_STATUS
(PARTITION_STATUS : out PARTITION_STATUS_TYPE;
RETURN_CODE : out RETURN_CODE_TYPE)
```

GET_PARTITION_START_CONDITION

The GET_PARTITION_START_CONDITION service request is used to obtain the reason for starting the current partition.

```
procedure GET_PARTITION_START_CONDITION
          (START_CONDITION : out START_CONDITION_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

SET_PARTITION_MODE

The SET_PARTITION_MODE service request is used to set the operating mode of the current partition to normal after the application portion of the initialisation of the partition is complete (in this case this service is typically invoked by the partition main thread). The service is also used for setting the partition back to idle (partition shutdown), and to cold start or warm start (partition restart), when a fault is detected and processed.

```
procedure SET_PARTITION_MODE
          (OPERATING_MODE : in OPERATING_MODE_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

Sampling ports management

The sampling ports are mechanisms that allow the inter partition communications. Every sampling port has a direction (or a type) that determines which operation the port support; source (or output) ports can be used to write messages, while destination (or input) ports are used to read messages. The number and the name of the ports used must be statically specified as well as the logical link between one source port to one or more destinations port. According to the AFS recommendations, in the current implementation for the input ports it is also required to set their initial content.

CREATE_SAMPLING_PORT

The CREATE_SAMPLING_PORT service request is used to create a sampling port. An identifier is assigned by the O/S and returned to the calling process. For a source port, the refresh period is meaningless.

```
procedure CREATE_SAMPLING_PORT
          (SAMPLING_PORT_NAME : in SAMPLING_PORT_NAME_TYPE;
          MAX_MESSAGE_SIZE : in MESSAGE_SIZE_TYPE;
```

```
          PORT_DIRECTION : in PORT_DIRECTION_TYPE;
          REFRESH_PERIOD : in SYSTEM_TIME_TYPE;
          SAMPLING_PORT_ID : out SAMPLING_PORT_ID_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

WRITE_SAMPLING_MESSAGE

The WRITE_SAMPLING_MESSAGE service request is used to write a message in the specified output sampling port. The message overwrites the previous one.

```
procedure WRITE_SAMPLING_MESSAGE
          (SAMPLING_PORT_ID : in SAMPLING_PORT_ID_TYPE;
          MESSAGE_ADDR : in MESSAGE_ADDR_TYPE;
          LENGTH : in MESSAGE_SIZE_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

READ_SAMPLING_MESSAGE

The READ_SAMPLING_MESSAGE service request is used to read a message from the specified input sampling port.

```
procedure READ_SAMPLING_MESSAGE
          (SAMPLING_PORT_ID : in SAMPLING_PORT_ID_TYPE;
          MESSAGE_ADDR : in MESSAGE_ADDR_TYPE;
          -- the message address is passed IN,
          -- although the respective message is passed OUT
          LENGTH : out MESSAGE_SIZE_TYPE;
          VALIDITY : out VALIDITY_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

GET_SAMPLING_PORT_ID

The GET_SAMPLING_PORT_ID service request returns the sampling port identifier that corresponds to a sampling port name.

```
procedure GET_SAMPLING_PORT_ID
          (SAMPLING_PORT_NAME : in SAMPLING_PORT_NAME_TYPE;
          SAMPLING_PORT_ID : out SAMPLING_PORT_ID_TYPE;
          RETURN_CODE : out RETURN_CODE_TYPE)
```

QUEUING PORTS and **ERROR management services** can be invoked by the application code but currently are stubbed.

2.4 Analysis Tools (WP3)

Several tools implement the features described in deliverable D3.4 in order to perform timing analysis of the program under study. Those tools are as follows:

- SPTA tool and profile manipulation library.

- R-tool for measurement-based probabilistic timing analysis (MBPTA).

- Hybrid probabilistic timing analysis (HyPTA) tool.

- Probabilistic response time analysis.

Next we describe those tools in detail. How to use them is described later in section 3.

2.4.1 SPTA Tool and Profile Manipulation Library

This tool, which has been implemented in Java, provides a set of functionalities related to the generation of the SPTA probability distribution function and profiles for both SPTA and MBPTA. In particular, this tool provides the following:

- The Static Probabilistic Timing Analysis tool (SPTA trace) computes the probability distribution function (PDF) of the execution time of a program based on a SPTA trace produced by *proartis_sim*. This tool allows customising the cache configuration used as well as other aspects of the ISA used such as instruction latencies.

- The PDF for a set of execution times measured (MBTA trace) with *proartis_sim* can be also generated.

- PDFs for SPTA and MBTA traces can be plotted in the form of inverse cumulative distribution functions (ICDFs) and the cutoff execution time value for a particular exceedance probability can be obtained for any such ICDF.

- Optimised libraries to perform fast convolutions when processing SPTA traces have been generated. Such libraries perform quick accurate convolutions. Moreover, some features have been added to further speedup those convolutions by sacrificing accuracy in the leftmost part of the ICDF, which corresponds to probabilities not relevant for the WCET calculation.

- Finally, this tool also allows computing the ICDF from above a set of ICDFs. This way, given a set of ICDFs, the tool provides the tightest ICDF whose cutoff value for any exceedance probability is equal or higher than that for any of the input ICDFs.

2.4.2 R-Tool for Measurement-Based Probabilistic Timing Analysis

A set of scripts running on top of the R-tool [3] perform the EVT (extreme value theory) projection required for probabilistic WCET estimation. In particular those scripts provide: (i) the minimum number of runs required for the program under analysis given a trace of execution times, (ii) the Gumbel distribution bounding from above the execution time PDF (based on the trace of execution times and the minimum number of runs and (iii) the execution time matching a given exceedance probability for the Gumbel ICDF (a.k.a. the probabilistic WCET).

Together with that information, which is the most relevant for the user, R scripts also provide the indexes obtained for the continuous ranked probability score (CRPS) method to determine the minimum number of runs, as well as the parameters for the Gumbel distribution (ξ, μ and σ).

2.4.3 Hybrid Probabilistic Timing Analysis Tool

The hybrid probabilistic timing analysis tool implements HyPTA as it is presented in deliverable D3.4. This tool is a combination of five tools (the components described below). The steps of combining the five components are described after the components presentation.

Components

The hybrid probabilistic timing analysis tool has 5 main components

- RapiTime: execution time measurement and analysis tool
- ProartiSim: modified simulator to exhibit statistically predictable behaviour
- Rvd2wceteq: WCET equation extraction
- RVS PROARTIS plugin: viewer and WCET equation recomputation
- EVT analysis: tail projection tool

Instrumentation

The instrumentation approach chosen for use with the PROARTIS simulator was assembly-label instrumentation. This allows extremely low overheads (no additional instructions are added) and means that the "instrumented" binary can be deployed with no performance degradation.

The specific assembly routine used was:

#define RPT_Ipoint(I) asm volatile("rdtsc" :"=A" (tsc2)); printf("\nipoint: %d (%d)\n", I, tsc2-tsc1); asm volatile("rdtsc" :"=A" (tsc1));

Data Collection

The simulator produces traces in several forms. For the purposes of this tool integration, we need "Type 2" traces, which correspond to RapiTime traces. The simulator performs the address to ipoint mapping for you, so it must be provided with the list of addresses which correspond to the assembly labels which have been inserted by the instrumentation, and the ipoint number they correspond to. We used a small shell script to generate this file from the "nm" symbol dump of the binary.

nm 1|$grepRVS_Ipoint|sed's$: ([0−9a−f]∗)t(.∗) : \1\2 :′>1.rpt-ipoint ./remap_ipoints 1.$rpt - ipoint|sort$ >1.remap

Trace processing

The trace is then processed by the timeparser component of RapiTime. It is possible for more than one ipoint to map to a single address, which leads to the simulator only reporting the first ipoint number. A mapping process must be performed when the .rvd file is generated using xstutils in order to allow the parsing to succeed:

xstutils -o $(PROG).rvd $^-r $(ROOT) –scope-filter scope.flt –address-map test_harness.elf.remap.txt -v

Timeparser will store the execution time frequency distribution for each block (ipoint transition) in the .rvd file, allowing the rest of the tools to obtain the necessary data.

Application of EVT
This component is not implemented since it uses the MBPTA tool to proceed. Its utilisation is explained in deliverable D3.4.
Reporting
The combined profile generated by the tree composition is stored in the .rvd database. This is then presented to the user on a special "PROARTIS" tab at the top-level of the report.
This allows the PROARTIS estimate to be compared with the actual measured execution times, and with a conventional RapiTime WCET estimate.

2.4.4 Probabilistic Response Time Analysis

All three tools for probabilistic timing analysis have an intensive use of convolutions. These operations have a complexity linear dependent of the number of values of each random variable that is convoluted to another random variable. To decrease this complexity we have proposed in deliverable D3.4 two re-sampling techniques decreasing the number of values without decreasing the safeness of the analysis, but keeping the associated pessimism as low as possible. To measure these two properties, i.e. safeness and pessimism, we developed a tool calculating the response time analysis as measure. This tool has the following features:

- Fixed priority scheduling. Schedules tasks based on their priorities.
- Deadline miss probability calculation. Provides the probability of missing the deadline for the set of tasks scheduled.
- Resampling support (inherited from SPTA tool) to speedup convolutions. Arbitrary precision arithmetic is used to compute probabilities.
- Taskset generator. Tasksets can be generated to test the tool. Several parameters such as the execution time values per task, number of tasks, utilisation factor, etc. can be configured.

3

PROARTIS Toolchain Installation and Usage

This section describes how to install and use the toolchain. Whenever this information does not change with respect to deliverables D3.2 and D3.3 provided in the past, it is not replicated to avoid redundancies.

3.1 Hardware Architecture Simulator

A large number of features have been incorporated into the simulator without changing its interface and installation instructions with respect to months 6 and 12. Thus, details about its installation and use can be found in section 6 of deliverable D5.3 and deliverable D3.3. The simulator source code can be found in the directory *proartis_sim/*.

3.1.1 Installation

The zip file provided must replace the *soclib* folder in the basic SoCLib [4] installation. Further installation details can be found in D5.3.

3.1.2 Usage

Invocation of the simulator can be also found in D5.3. However, the list of parameters has been increased significantly to control the new features added into the *proartis_sim*. These parameters correspond to the features described in Section 2.1. Parameters are as follows:

Program to be run on top of *proartis_sim*.

```
-executable=soft/eembc/cacheb01
```

Whether L2 caches are used. *L2unified* must be set to *yes* if both data and instructions L2 caches are required and they must be a unified cache.

```
# L2 caches or not: yes, no
-L2forinstructions=yes
-L2fordata=yes
-L2unified=yes
```

Number of ways, sets and cache line size (in 4-byte words) for L1 and L2 data and instructions caches. If L2 caches are unified, L2 data cache configuration is used.

-icache_ways=8
-icache_sets=16
-icache_words_per_cacheline=4
-dcache_ways=8
-dcache_sets=16
-dcache_words_per_cacheline=4
-icacheL2_ways=8
-icacheL2_sets=128
-icacheL2_words_per_cacheline=4
-dcacheL2_ways=8
-dcacheL2_sets=128
-dcacheL2_words_per_cacheline=4

Replacement policy for caches: *Perfect* if all accesses must hit.
Replacement policies: perfect,lru,random_eviction_onmiss,random_eviction_onaccess
-icache_replacement=random_eviction_onmiss
-icacheL2_replacement=random_eviction_onmiss
-dcache_replacement=random_eviction_onmiss
-dcacheL2_replacement=random_eviction_onmiss

Placement policy for caches.
Placement policies: modulo,random_onrun
-icache_placement=random_onrun
-icacheL2_placement=random_onrun
-dcache_placement=random_onrun
-dcacheL2_placement=random_onrun

Whether data and instruction caches are inclusive or non-inclusive.
Inclusivity relations: inclusive, noinclusive
-icacheL1L2_inclusion=noinclusive
-dcacheL1L2_inclusion=inclusive

Cache allocation and write policies. *Writethrough* for write-through non-write allocate caches and *writeback* for write-back write allocate caches.
Data cache write policy: writethrough, writeback
-dcache_writepolicy=writethrough
-dcacheL2_writepolicy=writeback

TLB configuration including number of sets and ways, page size (in 4-byte words), placement and replacement policies, and whether all accesses are enforced to access the TLB. This last parameter is useful to perform experiments without the RTOS but testing the TLBs, which are not accessed if no RTOS is in place otherwise.
-force_always_tlb_accesses=yes
-itlb_ways=16
-itlb_sets=1
-itlb_words_per_page=256
-itlb_replacement=random_eviction_onmiss
-itlb_placement=random_onrun
-dtlb_ways=16
-dtlb_sets=1
-dtlb_words_per_page=256
-dtlb_replacement=random_eviction_onmiss
-dtlb_placement=random_onrun

Hit and miss latencies for caches and TLBs.
-icache_hitlatency=1
-icache_misspenalty=10
-icacheL2_hitlatency=1
-icacheL2_misspenalty=100
-dcache_hitlatency=1

-dcache_misspenalty=10
-dcacheL2_hitlatency=1
-dcacheL2_misspenalty=100
-itlb_hitlatency=1
-itlb_misspenalty=100
-dtlb_hitlatency=1
-dtlb_misspenalty=100

Parameters useful to mimic disturbing software evicting cache contents. *Addr shoots* indicated the address (in decimal) of the instruction triggering the evictions. Instruction and data shoots correspond to the number of random evictions of each type that will be performed.
-addr_shoots=0
-instruction_shoots=1000
-data_shoots=1000

Reset cache state indicates whether caches are reset when the address *addr reset cache statistics* is reached. Whenever that address is reached cache statistics are reset regardless of whether caches themselves are reset or not. *addr print cache statistics* indicates the address of the instruction triggering the printing of the statistics. Statistics are also printed when the execution ends.
-reset_cache_state=false
-addr_reset_cache_statistics=0
-addr_print_cache_statistics=0

Parameters to set the trace file generated. The PTA trace file is used for SPTA, the MBTA one generates traces with execution times to be consumed by RapiTime and the object-level branches one identifies the path traversed. Note that up to one trace can be generated simultaneously, PTA ones must have extension *.pta*, and MBTA and object-level branches ones must have extension *.rpz*.
-pta_trace_file=NONE
-mbta_rpz_trace_file=NONE
-object_level_branches_rpz_trace_file=NONE

Output files for statistics and program's output. If no files are specified (parameters commented) the standard output is used.
-statistics_file=stats_file
-simulation_output_file=outfile

3.1.3 Software Provided

The zip file proartis_sim.zip in directory *proartis_sim* includes all the source files of SoClib that have been modified and added to implement all features in the simulator and emulator.

3.2 Compiler and Run-time Libraries for Randomised Memory

3.2.1 Installation

Our compiler/run-time randomisation tool requires LLVM-gcc 4.2 front-end binaries to be installed [1] and LLVM 2.9 source code to be built and installed properly. Our software (stabilizer.zip) must be extracted. Once extracted the following commands must be run:

```
cd stabilizer2/autoconf
./AutoRegen.sh
```

AutoRegen.sh command asks for two directories: the one where the LLVM 2.9 source code is available and the one where we have built LLVM 2.9. Next, the following commands must be run to compile the compiler pass plug-in:

```
cd ..
./configure
make
```

Finally, we must generate the PowerPC library to be linked with the object code files we produce for the programs to be tested (libstabilizer.a). To do so we must do the following:

```
cd stabilizer/runtime
```

Then, we must edit the makefile incorporating the following two lines at the beginning of the file, where $CROSSCOMPILER stands for the directory where the PowerPC cross-compiler was built (see D3.3 for more details) and $CROSSAR stands for the directory where the cross AR utility was built (again, see D3.3 for more details):

```
CC=$CROSSCOMPILER/powerpc-elf-g++
AR=$CROSSAR/powerpc-elf-ar
```

Finally, we only need to type *make* to generate the stabilizer library (libstabilizer.a).

3.2.2 Usage

The process to generate a binary with software randomisation has not been automated yet, so some steps still need to be performed manually. First, the program to be evaluated must be compiled normally for *proartis_sim* as described in D3.3, and the file *main.o* must be removed. Next, we must run the following commands to generate the instrumented program in assembly where $SOCLIB stands for the path where SoCLib was installed (see D3.3) and $STABILIZER stands for the path where our stabilizer tool has been uncompressed and compiled. Note that the *opt* command will list the names of the randomised functions, which we will need later.

```
llvm-gcc -x c -mcpu=750 -Wall -O0 -I. -I$SOCLIB/soclib/utils/include -nostdinc -ggdb \
  -I$SOCLIB/soclib/soclib/platform/topcells/common -I$SOCLIB/soclib/soclib/lib/include \
  -I$SOCLIB/soclib/soclib/module/connectivity_component/vci_fd_access/include \
  -I$SOCLIB/soclib/soclib/module/verification_component/vci_simhelper/include \
  -I$SOCLIB/soclib/soclib/module/connectivity_component/vci_multi_tty/include main.c \
  -c -emit-llvm -o main.bc
opt -load=$STABILIZER/stabilizer2/Release+Asserts/lib/LLVMStabilizer.so –stabilize \
  –stabilize-code main.bc -o main_opt.bc
llc -march ppc32 -O0 main_opt.bc -o main_opt.s
```

The assembly must be edited manually with *vim* editor, which is available in most Linux distributions, by running this command:

```
vim main_opt.s
```

Within *vim* we must execute the following replacement commands so that the assembly code can be properly compiled by a GNU PowerPC assembler.

```
%s/lo16(\([^()]*\))/\1@l/g
%s/ha16(\([^()]*\))/\1@h/g
```

Still within the *vim* editor we must perform a manual process where, for each function listed by the *opt* command we search for *dummy.name_of_the_function,@f*. Right after the match we find a **blr** instruction. We must add extra lines with **nop** instructions below. In particular we need as many **nop** instructions as function calls the current function performs to other functions (counted directly from the source code) plus 3. Thus, if the function calls two functions, the we need 5 **nop** instructions to be added. Then, we can save the file and exit *vim*.

We finalise the process by running the two following commands that generate the binary to be run on top of *proartis_sim* where $CROSSCOMPILERLIBGCC stands for the path to the cross-compiler libgcc.a (it can be obtained by running *powerpc-elf-gcc -print-libgcc-file-name*).

```
powerpc-elf-as -o main_opt.o main_opt.s
powerpc-elf-ld -o test.bin *.o -T $SOCLIB/soclib/soclib/platform/topcells/common/ldscript \
  $CROSSCOMPILERLIBGCC/libgcc.a -L$STABILIZER/stabilizer2/runtime -lstabilizer
```

3.2.3 Software Provided

The zip file stabilizer.zip can be found in the directory named *stabilizer*.

3.3 RTOS and Programming Paradigms

3.3.1 Installation

While the original POK can be installed under Linux/MacOS and Windows, the version of POK that includes our modifications has been tested under Linux only.

In order to run POK on *proartis_sim* it is required to compile it using the gcc 4.5.1 for PowerPC (this compiler has been recommended by AFS), so the first prerequisite for the installation is to have that gcc properly installed. The second prerequisite is to have Ocarina installed (we use the version 2.0w); this tool is used to automatically generate the code of the examples (see later) from an AADL model, however, if you plan to not use this functionality (for example, because you run or modified existing examples) you do not need to install it.

We provide our version of POK in the file Proartis-POK.zip. After having unzip it in the location you prefer, make sure to have the above-mentioned compiler (and, if needed, Ocarina) in your path; in addition you need to set the environment variable *POK_PATH* to the POK installation directory and to add to the *LD_LIBRARY_PATH* the *lib* directory of the PowerPC gcc compiler.

In the *<pok-dir>/examples* directory we can find some applications already integrated in POK. You can choose one and build it just typing *make* from the directory *<pok-dir>/examples/<example-name>/generated-code*; if the make process is successful, in the directory *<pok-dir>/examples/<example-name>/generated-code/cpu/* you will find the executable file *pok.elf*.

However, it is also possible to modify an example or to integrate an existing application with POK; in this case it is recommended that you know some additional information. To this end, in the following paragraphs we describe how to configure the POK kernel and the partitions and how the compilation process works.

3.3.2 Usage

How to build an application running on POK

The overall architecture of POK is ideally made up of three layers: the kernel, the partitions and the applications. The kernel executes the partitions and each partition contains its application code. The POK kernel implements the core functionalities of the OS (e.g. scheduling, threads and partitions management) and provides some services that can be invoked through syscalls by the upper level. Each partition actually includes both the libpok code and user-code (application); in the current implementation, at least one partition must exist (that is to say that POK kernel cannot run in the stand-alone mode). Each part of the system (kernel, partitions) is first separately compiled and then integrated into a single executable; the compilation of each part produces a distinct binary file. Finally, all binary files are integrated to produce a single bootable ELF binary, which in fact includes different multiple distinct binaries: the code for the kernel and the code of all partitions.

POK kernel and partitions can be configured using configuration directives, that mainly consist in macros definitions (C code). For examples, it is possible (and in many cases mandatory) to define macros that specify the number of partitions, the number of threads of the whole system, and for each partition, the time slot, the size (i.e. the memory it requires), and the scheduling policy used. Moreover, using macros it is also possible to specify which parts of the system to compile and then to include in the final executable (e.g. in order to use ARINC653 code developed by PROARTIS it is required to define the $POK_NEEDS_ARINC653_AIRBUS$ macro). A more complete description of the original configuration directives can be found in the official POK documentation available at the POK site while the description of the new macros along with some other details concerning the application code development is provided in the *<pok-dir>/README* file.

In the intention of the original POK developers, the configuration code, the deployment code as well as the application (user) code could be automatically generated using the Ocarina code generator and the AADL model. This functionality has been provided in order to achieve a zero-coding development approach and to avoid potential errors introduced by the code produced by human developers. However, in case it is required to skip the automatic code generation process (for example in case we want to integrate an existing application with POK) we need to manually configure the system. In this case, to the best of our experience and as suggested in the official POK documentation, the best way to proceed is to automatically generate an example and properly modify in order to integrate the existing application.

POK compilation process

The POK compilation process is specified in hierarchical makefiles. Launching the top-level makefile (you can find one in the directory *generated-code* of every example) the compilation process first compiles the POK kernel, then the partitions and finally integrates all the binary files generated so far in a single executable file. The kernel compilation includes the files *deployment.h* and *deployment.c* (present in the application code directory, i.e. *<example-name>/generated-code/cpu/kernel*)

containing the configuration directives (previously specified by the application developer) and the deployment code of the kernel. The final result of the kernel compilation process is the *kernel.lo* file.
The partitions compilation process is executed as many time as the number of the partitions and generates an elf file for every partition. This ELF file incorporates both the user application code and the POK library. The user application code includes both the application-dependant files (that is the files created by the application developer or automatically generated by Ocarina) and the static library *libpok.a*. This file is obtained by the compilation and partial linking of the source files in the directory *<pok-dir>/libpok* and the file *deployment.h* specified for the partition directory of interest (i.e. *<example-name>generated-code/cpu/<part-name>*). It is worth noticing that the *libpok.a* is newly generated each time a partition is compiled, as it incorporates a file (*deploymen.h*) that is different for each partition. At the end of the partition compilation process, a file *<part-name>.elf* is generated for every partition.
In order to obtain the *pok.elf* executable on the *proartis_sim*, the following steps are performed:

1. All partition ELF files are padded to get aligned file size and appended to a single binary archive named *partitions.bin*.

2. The *partitions.bin* archive is then added to the object file (*sizes.o*) obtained compiling the automatically generated *sizes.c* file. This file contains an array storing the effective size of every partition (i.e. the size of the *<part-name>.elf* file).

3. The files *sizes.o* and *kernel.lo* (obtained from the POK kernel compilation process) are then linked together (using the linker script *kernel.lds*) and the *pok.elf* archive is generated.

The figure Figure 3.1 illustrates the overall creation process of the final *pok.elf* executable file.

3.3.3 Software Provided

The modified POK version can be found in the following zip file:
RTOS/Proartis-POK.zip

3.4 Analysis Tools

Some new analysis tools have been implemented and some others have been extended during this project phase. Thus, details about tool installation and use are provided mostly in this section, but some others can be found in section 6 of deliverable D5.3 and deliverable D3.3. The analysis tools can be found in the directory *Analysis_tools*. Within such directory specific directories have been set up for each tool.

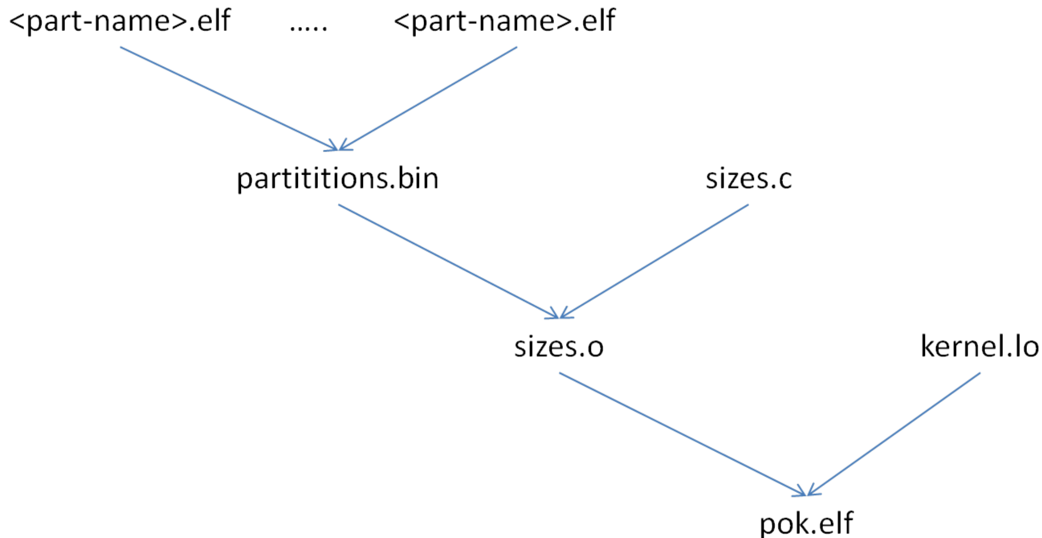

Figure 3.1: Overall POK Compilation process

```
PROARTIS analysis tool v1.3 (r553) (DIT) 29/11/2011
Main parameters are required ("<command>")
Usage: proartis [options] [command] [command options]
  Options:
    -j, –threads    Maximum number of threads
                Default: 0
    -v, –verbose    Verbose output
                Default: false
  Commands:
    pta: Process a PTA trace to obtain an execution time profile
    view: View a number of profiles as a chart
    mbta: Convert a number of MBTA run timings into a statistical profile
    help: Display usage for a specific command
    cutoff: Process an ETP and display the execution times with different probability cutoffs
    envelope: Compute the envelope of two or more input profiles
```

Figure 3.2: Options and commands for the SPTA tool and profile manipulation library

3.4.1 SPTA Tool and Profile Manipulation Library

Installation. This tool is run on top of Java. In particular it needs Java v6.0 to be installed. Other than that, no installation is needed for this tool.

Usage. This tool implements several functionalities. The basic usage of the tool is to run using the provided *proartis* script. This simply calls Java with the arguments:

```
java -jar proartis.convolutions.jar $*
```

Typing just *proartis* on the command line will display the command line usage information as shown in Figure 3.2.
SPTA ICDFs are produced with the *pta* command. In particular, information on the usage of this feature is retrieved by executing the command below, which provides information on how to select a resampling algorithm to speedup convolutions,

precision for convolutions, cache configuration for both data and instructions, instruction latencies, output file, etc.

```
proartis help pta
```

An example of how to generate a SPTA ICDF is as follows, where *epic_d-id-spta.out* is a PTA trace generated with *PROARTIS_sim/soclib* and the output file with the ICDF would be named *epic_d-id-spta.out-p32-l1024.csv*:

```
proartis pta -p 32 epic_d-id-spta.out
```

Given a *.csv* file produced from a SPTA or MBTA trace, the particular execution time for a given exceedance probability can be obtained with the *cutoff* command. An example of how to use it is provided below. Such command provides the execution time for 10^{-9} and 10^{-12} exceedance probabilities as well as the maximum execution time that could be ever reached.

```
proartis cutoff -c 9 -c 12 epic_d-id-spta.out-p32-l1024.csv
```

In order to compute the ICDF envelope for a set of ICDFs, the *envelope* command must be used. Such command generates a new ICDF whose value for any exceedance probability is exactly the maximum among those of the input ICDFs for such probability. An example of how to use it is as follows:

```
proartis envelop quicksort-spta-1.csv quicksort-spta-2.csv quicksort-spta-3.csv
```

Further details about how to generate the PDF for a MBTA trace (*mbta* command) or to plot an ICDF (*view* command) can be found in deliverable D3.3.

Software Provided. The tool can be found in the following zip file:
Analysis_tools/SPTA/Proartis_pta_1.3.3.zip

3.4.2 R-Tool for Measurement-Based Probabilistic Timing Analysis

Installation. R scripts run on top of the R-tool. R is freely distributed under the terms of the GNU General Public License; its development and distribution are carried out by several statisticians known as the R Development Core Team. We refer the reader to the original source for downloading and installing the software [3]. Some specific packages are required to run the scripts developed within PROARTIS. Those packages can be easily installed as follows:

```
install.packages("evir")
install.packages("evd")
install.packages("fExtremes")
```

Usage. Next we describe the purpose of each R script, how to run it within the R environment and the output obtained.
hd_min_run.R computes the minimum number of runs required to obtain a tight enough Gumbel distribution for the program under analysis. CRPS indexes are provided. The number of runs such that the CRPS index is below the target threshold five consecutive times is chosen as the minimum number of runs. This is script is run as follows: first the R script is loaded and then it is run with the file with the execution times (one execution time per line in decimal numbers) and the block size for block maxima as input.

```
source("Path/hd_min_run.R")
hd_min_run("Path/MBPTA_trace", block size)
```

fitting_parametrized.R generates the Gumbel distribution bounding the execution time of the single-path program whose execution times are provided as input together with the minimum number of runs and the block size. Together with the Gumbel distribution, its parameters are provided (ξ, μ and σ). This is script is run as follows:

```
source("Path/fitting_parametrized.R")
fitting_parametrized.R("Path/MBPTA_trace", block size, minimum number of runs)
```

exceedance_parametrized.R reports the execution time for a particular exceedance probability (pWCET) based on the execution time of the single-path program whose execution times are provided as input together with the minimum number of runs and the block size. This is script is run as follows:

```
source("Path/exceedance_parametrized.R")
exceedance_parametrized.R("Path/MBPTA_trace", block size, minimum number of runs,
                exceedance probability)
```

fitting_non_parametrized.R generates the Gumbel distribution bounding the execution time of the single-path program whose execution times are provided as input together with the block size. The Gumbel distribution and its parameters are provided for a different number of runs to allow comparing them. This is script is run as follows:

```
source("Path/fitting_non_parametrized.R")
fitting_non_parametrized.R("Path/MBPTA_trace", block size)
```

multivariate_fitting_parametrized.R generates the Gumbel distribution bounding the execution time of the multi-path program whose execution times are provided by hardcoding their file names in the script together with the minimum number of runs and block size, which are parameters. Indicating the files with the execution times for each path simply requires replacing "Path/MBPTA_trace" by the desired file name inside the script. Together with the Gumbel distribution, its parameters are provided (ξ, μ and σ). This is script is run as follows:

```
source("Path/multivariate_fitting_parametrized.R")
multivariate_fitting_parametrized.R(block size, minimum number of runs)
```

multivariate_exceedance_parametrized.R reports the execution time for a particular exceedance probability (pWCET) based on the execution time of the multi-path program whose execution times are provided hardcoded in the script together with the minimum number of runs and block size, which are parameters. Indicating the files with the execution times for each path simply requires replacing "Path/MBPTA_trace" by the desired file name inside the script. This is script is run as follows:

```
source("Path/multivariate_exceedance_parametrized.R")
multivariate_exceedance_parametrized.R(block size, minimum number of runs,
                exceedance probability)
```

Software Provided. R scripts can be found in the following zip file:
Analysis_tools/EVT-MBPTA/R-scripts.zip

3.4.3 Hybrid Probabilistic Timing Analysis Tool

Installation. HyPTA builds upon *proartis_sim* and RapiTime tools. Installation simply requires uncompressing the zip file with the HyPTA tool.
Usage. Usage follows the following procedure. Firstly we generate an instrumented program and simulate it on the PROARTIS hardware.

- Instrumentation of the application using RapiTime's cins program and generation of a static/structural report using xstutils as for normal RapiTime processes.
- Export of structure. Run the extract_ipoints program to generate the Ipoint remapping data;
- set up the desired parameters in the param_file;
- run the simulation as described elsewhere to produce an output as a "rpz" file.

To compute the tail projections, there are the following steps:

- Run timeparser to insert the data into the report;
- Extract the profiles;
- Run the EVT method on the profiles;
- Insert the rendered Gumbel curves into the report;
- Run the reportviewer on the report;
- Use the Tree WCET option in the Proartis tab to compute the execution time distributions.

Software Provided. The tool can be found in the following zip file:
Analysis_tools/HyPTA/Proartis-hypta.zip

3.4.4 Probabilistic Response Time Analysis

Installation. This tool is run on top of Java. In particular it needs Java v6.0 to be installed. Other than that, no installation is needed for this tool.

Usage. This tool implements several functionalities. The basic usage of the tool is to run using the provided *prt* script. Typing just *prt* on the command line will display the command line usage information as shown in Figure 3.3.
Task sets can be generated with the *sim* command. Information on the usage of this feature is provided in Figure 3.3. As shown, parameters such as the number of samples per task, tasks per task set, number of tasksets, maximum utilisation, etc. can be configured.
Probabilistic response times for a given task set can be generated with the *calc* command. Information on the usage of this feature is also provided in Figure 3.3. This feature allows controlling the tasks and number of task releases to analyse, characteristics of the resampling applied, among other parameters.

Software Provided. The tool can be found in the following zip file:
Analysis_tools/PRT/Proartis_prt_2.0.zip

```
Usage: prt [options] [command] [command options]
  Options:
    -h, –help    show usage information
               Default: false
    -v    show verbose output
               Default: false
  Commands:
    sim: Create a simulated task set
    calc: Calculate response times

Create a simulated task set
Usage: sim [options]
  Options:
    -d, –default-samples    number of samples per task
               Default: 10
    -l, –last-samples    number of samples in the last (lowest priority) task
               Default: 10
    -u, –max-utilisation    the maximum average utilisation of the generated task set
               Default: 1.0
        –sync-release    Should the releases all be synchronised at 0?
               Default: false
    -n    number of tasks to generate per task set
               Default: 2
    -o    output prefix. This will generate output files in
               the format "<prefix>""<taskset#>".txt
               Default: taskset
    -s    number of task sets to generate
               Default: 1

Calculate response times
Usage: calc [options] "<task description file>"
  Options:
    -o, –output    output file
    -x, –releases    number of releases to analyse
               Default: 1
    -r, –resample    resample to an upper limit on the number of values in a profile
               Default: 0
    -t, –task    task to analyse
    -a    Resampling algorithm to use. The default is uniform spacing
               over the range of values. Type ? to see a list of algorithms
```

Figure 3.3: Options and commands for the SPTA tool and profile manipulation library

Acronyms and Abbreviations

- API: Application programming interface.
- ETP: Execution time profile.
- EVT: Extreme value theory.
- ICDF: Inverse cumulative distribution function.
- ISA: Instruction set architecture.
- MBPTA: Measurement-based probabilistic timing analysis.
- MMU: Memory management unit.
- PDF: Probability distribution function.
- PRNG: Pseudo-random number generator.
- PTA: Probabilistic timing analysis.
- RTOS: Real-time operating system.
- SPTA: Static probabilistic timing analysis.
- TLB: Tranlation look-aside buffer.
- WCET: Worst-case execution time.

References

[1] LLVM. `http://dragonegg.llvm.org/`.

[2] POK. `http://pok.safety-critical.net`.

[3] R. `http://cran.r-project.org/`.

[4] SoCLib. `http://www.soclib.fr/trac/dev`.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.

[6] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.