D5.3. Software hardware integration, validation and evaluation

Contract No: ICT-323872
Contractual Date of Delivery: 28/02/2016 (M33)
Actual Date of Delivery: 04/04/2016
Main Authors: Konstantinos Parasyris, Vassilis Vassiliadis, Spyros Lalis, Christos D. Antonopoulos, Nikolaos Bellas
Center for Research and Technology Hellas (CERTH)
Estimated Indicative Person Months: 12
Classification: Public (PU)
Report Version: 1.0
Disclaimer:
This document reflects the contribution of the participants of the SCoRPiO project. The European Union and its agencies are not liable or otherwise responsible for the contents of this document; its content reflects the view of its authors only. This document is provided without any warranty and does not constitute any commitment by any participant as to its content, and specifically excludes any warranty of correctness or fitness for a particular purpose. The user will use this document at her/his own risk.
# Contents

1 Introduction 6

2 Programming Model - Compiler 7
   2.1 Programming Model ................................................. 7
      2.1.1 Runtime Initialization .................................. 7
      2.1.2 Memory regions allocation/de-allocation .................. 7
      2.1.3 Task definition and significance characterization ....... 8
   2.2 The Source-to-Source Compiler .................................. 8

3 Runtime System 9
   3.1 Task Creation ..................................................... 9
   3.2 Task Synchronization ............................................... 9
   3.3 Environmental Setup ............................................. 10
   3.4 System Model ...................................................... 11
   3.5 Performance, Energy and Fault Modeling ....................... 12
      3.5.1 Execution Time Modeling .................................. 12
      3.5.2 Power and Energy Modeling .................................. 13
      3.5.3 Fault Modeling ............................................... 13

4 Simulator 15
   4.1 The Gem5 Simulator ............................................. 15
   4.2 GemFI Design and Implementation ................................. 16
      4.2.1 GemFI User Interface ...................................... 16
      4.2.2 Simple Example ............................................. 17
      4.2.3 GemFI Internals and Implementation ....................... 18
      4.2.4 Simulation Checkpointing .................................. 19
      4.2.5 Simulation Campaigns on a Network Of Workstations .... 20

D5.3. Software hardware integration, validation and evaluation 36
# Evaluation

5.1 Evaluation Methodology .............................................................. 21
  5.1.1 Fault Injection ................................................................. 21
  5.1.2 Energy Estimation ............................................................ 22
5.2 Fault Tolerant Scenario ............................................................ 23
5.3 Benchmarks ............................................................................ 23
5.4 Experimental Results ............................................................... 24

# Usage Instructions

6.1 Installation instructions ............................................................ 29
  6.1.1 Installing the software stack ................................................ 29
  6.1.2 Installing GemFi ............................................................... 31
6.2 Execute the SCoRPiO framework ............................................... 31
6.3 Using the Virtual Machine ......................................................... 34
6.4 Known Issues ........................................................................ 34
Executive summary

SCoRPiO proposes to exploit the significance of computations to design energy-efficient systems which scale gracefully under the presence of errors. This deliverable discusses the integration of the software layer with the underlying hardware simulator. In the context of the SCoRPiO project the unreliable Hardware is simulated through GemFI, a fault injection tool based on the cycle accurate Gem5 simulator. On top of GemFI a Linux operating system is deployed. Finally, the SCoRPiO runtime system is utilized to provide fault tolerant enhancements to the application layer.

Deliverable 5.3 consist of four parts. The first part (Chapter 2) provides information about the source-to-source compiler. We briefly summarize the programming model pragma extension and their usage.

Chapter 3 we summarize the API calls provided by the runtime system to the source-to-source compiler. We briefly discuss the environmental set up of the runtime system. Finally, we present the assumed hardware model, the fault model and the performance/energy model.

Chapter 4 presents GemFI the simulator used by the SCoRPiO project to simulate an unreliable environment. GemFI operates on top of the cycle accurate Gem5 simulator. We discuss the implementation of GemFI as well as the usage of the tool.

In chapter 5 we evaluate our framework using five benchmarks with different computational and significance patterns. We study both the energy gains of the proposed model as well as the effectiveness of different error detection and repair mechanisms used in the framework.

Chapter 6 provides information on how to get and install the compiler, the runtime system and the simulator. The runtime system is accompanied with a set of benchmarks which can be compiled using the SCoRPiO source to source compiler and can be executed on top of the simulator. An example is also provided which installs all the necessary components and performs all the mandatory steps to perform a fault injection campaign on top of a benchmark. Finally we present known issues and limitation of our infrastructure.
Chapter 1

Introduction

This deliverable is the report accompanying the integration of the Hardware with the software stack. The software stacks comprises a source to source compiler, the runtime system and the benchmarks. The source to source compiler enables a translation of the application developed using the SCoRPiO significance aware programming model to SCoRPiO runtime system API calls. The runtime system provides fault tolerant mechanisms such as task isolation, soft-checkpointing, recovery from fatal errors etc. Finally the benchmarks provided utilize the SCoRPiO programming model to enhance their robustness against errors as well as to gracefully degrade the quality of the output in case of faults.

In deliverable D2.1 we introduced the SCoRPiO task-parallel, significance-aware programming model. It allows the programmer to express her perspective on the contribution of each computation to the quality of the final output. Computations with major contribution are characterized as significant, whereas computations with less contribution are considered non-significant. The significance of computations determines the reliability level of the hardware which can be used for their execution. More specifically, significant computations are executed reliably, whereas non-significant computations can be executed unreliably. Unreliable execution involves aggressively under-powered cores, without sacrificing performance, however at the expense of a possibility for errors. In deliverables D3.1, D3.2, and D3.3 the scheduling policies, the memory management, and the checkpointing provided by the runtime system are discussed respectively.

The main contributions of this deliverable are the following: 1. We discuss the integration of the software stack with the hardware stack. 2. We evaluate our framework using five benchmarks with different computational and significance patterns. 3. We study both the energy gains of the proposed model as well as the effectiveness of different error detection and repair mechanisms used in the framework.

This report accompanies the benchmarks, the compiler prototype, the runtime prototype and the Simulator source files. The rest of the document is organized as follows: We briefly present the programming model in Chapter 2. Chapter 3 presents the runtime system as well as a simple fault/power model used to evaluate our approach. In Chapter 4 we present GemFI the simulator used by the SCoRPiO project. In chapter 5 we present the evaluation methodology and we evaluate our framework in terms of energy gains and quality degradation. Finally in chapter 6 we provide instruction on how to get, install and use the vertical SCoRPiO infrastructure.
Chapter 2

Programming Model - Compiler

2.1 Programming Model

The SCoRPiO compiler supports all #pragma directives specified by the SCoRPiO programming model, as shown in listing 2.1. For more information about the programming model consult Deliverables 2.1 and D2.2.

2.1.1 Runtime Initialization

The #pragma omp start(threads, blockSize, memorySize) compiler directive initializes the runtime system. threads denotes the number of worker threads which will be instantiated and managed by the runtime. The runtime will also decide the reliability of the worker threads to facilitate the execution of both modes of operation. The last two clauses blockSize and memorySize define the granularity of memory blocks and the total size of memory allocated by the runtime system. This memory will hold the data structures which participate in the automatic data dependence analysis of the runtime system.

2.1.2 Memory regions allocation/de-allocation

Memory allocation and de-allocation in programs is implemented using the traditional malloc()/free() functionality offered by the C library. However, memory areas which participate in the automatic data dependence analysis by the runtime system need to be annotated and this needs to be communicated to the runtime system. The compiler directives used for this purpose are #pragma omp malloc and #pragma omp free(size). They precede the respective malloc and free calls. Essentially, they redirect calls to malloc() and free() to the implementations supplied by the

```
#pragma omp start(threads, blockSize, memorySize)
#pragma omp malloc
#pragma omp free (size)
#pragma omp task [in(...) [out(...)] [label(...)] [significant(expr...)|ratio(...)]
   [tasktolerance(taskcheck()), [redo(...)]]]
#pragma omp taskwait [label(...)] [time(...), ratio(...), all]
   [grouptolerance(groupcheck()),[redo(...)]]]
#pragma omp significant
```

Listing 2.1: SCoRPiO programming model primitives.
SCoRPiO runtime system.

### 2.1.3 Task definition and significance characterization

Tasks are characterized as either significant or non-significant at the time they are created. In our model a task is created using the `#pragma omp task` compiler directive.

The compiler directive is extended by five clauses:

- **label()** Groups of tasks are formed when tasks are named using the same `label()` at their instantiation.

- **in/out()** The SCoRPiO runtime system is capable of identifying data dependencies between tasks and enforcing the correct execution order. To facilitate this runtime mechanism we support the declaration of input/output data flows at the granularity of a single task.

- **significant()** The significant() clause allows significance characterization at the task granularity. This information is made available to both the compiler framework and the runtime system.

- **tasktolerance()** Since some tasks are executed on unreliable hardware the tasktolerance() clause enables the developer to define a function which is executed reliably after each individual tasks completes or crashes. This function tries to identify silent data corruptions (SDCs) on the result and may correct such errors using approximate computation or even default values.

### 2.2 The Source-to-Source Compiler

SCOOP [12] is a source to source compiler which uses a static dependence analysis to optimize the execution of task parallel applications by removing unnecessary runtime checks for non-conflicting tasks.

The compiler is extended to support the lowering of SCoRPiO programming primitives to calls to the SCoRPiO runtime system. Furthermore the compiler is capable of instrumenting input source code with context-aware hooks. For more information for context sensitive hooks consult Deliverables D2.2 and D2.3.
Chapter 3

Runtime System

The SCoRPiO Runtime System (RTS) is a task-parallel runtime system. We provide an API (listing 3.1) for a) initialization of the system, b) creating a task, c) synchronizing a group of tasks, and d) allocating and de-allocating memory. There is one to one mapping of the pragmas defined by the programming model as defined in Section 2.1 and the RTS code which is typically executed by the master thread\footnote{With the exception of \texttt{#pragma omp significant} which is always executed within a task-body function by a worker thread}. The functionality of each primitive is the same with the ones discussed in D5.2 besides the task creation and the task synchronization primitives which are discussed below.

3.1 Task Creation

The \texttt{tpc} call function call creates a task that will be scheduled for execution. The master will create the task-closure and will enqueue it to its personal work queue.

- \texttt{groupid} This is the id of the group in which the task belongs. The source to source compiler assings one unique id for each group, defined by the programmer using the \texttt{label()} and passes it here to the RTS.

- \texttt{significance} The significance argument ranges from 0.0 to 1.0 and informs the RTS about the significance of this task.

- \texttt{sanity} The sanity is a function pointer to the task result check function (TRC) which is a used defined function that is called after the execution of every non-significant task, to perform checks for the sanity of the task’s results.

- \texttt{args} This pointer points to a struct containing the arguments of the TRC function.

- \texttt{args.size} The size in bytes of the struct containing the argument of the TRC function.

- \texttt{redo} The maximum number of times that this task might be re-executed.

3.2 Task Synchronization

A \texttt{taskwait} can serve as a global barrier, instructing the runtime to wait for all tasks spawned up to that point in the code. Alternatively, it can implement a barrier at the granularity of a specific task group. This barrier may perform block waiting for a fixed amount of time. The block-wait duration is defined by the application programmers since it reflects application specific knowledge. Through the ratio argument, the programmer can instruct the runtime to
execute (at least) the specified percentage of all tasks either globally or in a specific group, depending on the existence of the label() clause in a reliable worker, while respecting task significance (i.e., a more significant task should not be executed unreliably, while a less significant task is executed reliably).

**groupid** The RTS will only wait for tasks of the group with id groupid. As with the tpc_call this is provided by the source-to-source compiler and it maps one-to-one with a group label.

**sanity** sanity is a function pointer to the group result-check function(GRC) which is a user-defined function that is called after the barrier unblocks to perform checks for the sanity of the groups results.

**args** This pointer points to a struct containing the arguments of the GRC function.

**args_size** The size in bytes of the struct containing the arguments of the GRC function.

**flags** These flags define the kind of synchronization that will be used. It can be an ORed expression SYNC_TIME (wait for a specific amount of time) and SYNC_ALL (wait for both a ratio and a watchdog timer).

**time_ms** The amount of time in miliseconds to wait, when a watchdog is used for the synchronization. The SYNC_TIME flag needs to have been defined, otherwise this is ignored.

**ratio** The ratio takes values in the range [0.0, 1.0] and serves as a single, straightforward knob to enforce a maximum unreliability in the system. Smaller ratios allow the runtime to create more non-reliable workers and the energy reduction opportunities, however at a potential quality penalty.

**redo_max** The maximum number of times this group may be re-executed. If this number is reached the group is marked as failed.

### 3.3 Environmental Setup

Runtime execution can be managed through environmental variables, similarly to OpenMP. This way we manage the number and the affinity of the execution threads, we can create CPU domains, that will later map to reliable/unreliable cores and the scheduling policy.

---

Listing 3.1: SCoRPiO runtime system API.

```c
/* initialize/shutdown system */
void tpc_init(unsigned int max_spes, uint32_t block_sz, uint64_t grid_sz);
void tpc_shutdown(void);

/* create a task */
void tpc_call(uint32_t funcid, uint32_t total_arguments, uint32_t groupid,
    float significance, uint8_t (*sanity)(void *), void *args,
    size_t args_size, uint32_t redo, ...);
uint32_t tpc_get_groupid(const char *lbl);

/* synchronize tasks */
void tpc_wait_all(void);
void tpc_wait_group(uint32_t gid, uint8_t (*f)(void *), void *args, size_t args_size, uint8_t
    flags, uint32_t time_ms, double ratio, uint32_t redo_max);

/* allocate/deallocate memory */
void *tpc_malloc(unsigned int size);
void tpc_free(void *mem, unsigned int size);
```
\textbf{POPT\_NUM\_THREADS} Defines the number of execution threads. It needs to be an integer. If this variable is not set, or its value is invalid, i.e. less or equal than zero or bigger than the available HW contexts, the maximum number of threads is used.

\textbf{POPT\_CPU\_AFFINITY} It maps threads to available CPUS. It contains a combination of comma-separated CPU ids (M,N) or ranges (M-N). The master thread will be mapped on the first CPU of this list. Each next thread will be mapped to subsequent CPUs from the list. This list needs to contain a number of CPUS greater or equal to the number of threads defined by \textbf{POPT\_NUM\_THREADS}

\textbf{POPT\_DOMAINS} This variable is used to group CPUs in different scheduling domains. It contains one or more lists of CPUs same as the \textbf{POPT\_CPU\_AFFINITY} separated by a colon. For example, 0-7:8-15 defines two CPU domains one containing CPUs 0 to 7 and the second one to 8 to 15.

\textbf{POPT\_SCHED\_POLICY} Contains a string defining the scheduling policy. Currently we support three such policies:
a) \textbf{SCHED\_WS} defines a simple work-stealing scheduler, b) \textbf{SCHED\_LF} defines a work-stealing scheduler that tries to steal tasks from the same CPU domain before looking at others, and d) \textbf{SCHED\_ISOLATE} that restricts working stealing only among threads running in the same CPU domain.

\section{3.4 System Model}

As a basis for our work, we consider a multi-core hardware architecture, with two types of cores: \textit{conventional} and \textit{configurable}. Conventional cores execute the code fault-free, at state $\text{Corr} = (V_c, f_c)$, where $V_c$ is the supply voltage and $f_c$ the corresponding frequency of operation. Configurable cores can dynamically switch between a reliable and an unreliable mode of operation. We refer to the respective states as $\text{Rel} = (V_r, f_r)$ and $\text{UnRel} = (V_r, f_u)$. In both modes the core operates at the same voltage $V_r$, but the unreliable mode has a higher frequency $f_u = f_r + a$. The larger $a$ is, the more faults occur during execution. The transition between these modes can be achieved, for example, based on the approach described in [5]. In the rest of this deliverable we define as a fault a deviation of the required operation of the hardware. We also define the result of a fault as an error.

For example, Figure 3.1 shows the different nominal operating points for a Quad Core i7 processor, as a function of frequency ($x$ axis) and supply voltage ($y$ axis). Each nominal point is a candidate for $\text{Corr}$. Moving below nominal points, we enter the realm of unreliable execution; the larger the distance, the higher the fault rate. Here, both $\text{Corr}$ and $\text{UnRel}$ configurations are chosen to have the same frequency. However, $\text{UnRel}$ has a lower voltage—consequently, it is more energy efficient but also vulnerable to faults. The $\text{Rel}$ configuration corresponds to the nominal configuration with the same voltage as $\text{UnRel}$, with a lower frequency. This guarantees fault-free operation, at a performance penalty compared to $\text{Corr}$.

We assume a system with at least one conventional and one configurable core. The conventional cores are used to run the Operating System (OS), the main application thread (which includes the main runtime logic), and (some) application tasks reliably. The configurable cores are used to run (some) application tasks in the unreliable mode to save energy. Configurable cores are switched to the reliable mode when running result check functions or system-level functions.
3.5 Performance, Energy and Fault Modeling

We introduce an analytical model to compute the energy consumption of an application as a function of the CPU frequency and voltage supply, and the number of tasks that are executed reliably and unreliably. Moreover, fault modeling allows us to translate voltage scaling into hardware timing faults and to drive fault injection scenarios to be presented in Section 5.1.

3.5.1 Execution Time Modeling

Equation 3.1 expresses the time for executing a task, where \( C \) denotes the number of cycles spent to execute the task code, and \( f \) is the operating frequency of the core used to run the task. Note that \( f \) varies, depending on whether the task is executed reliably on a conventional core operating at \( \text{Corr} \), or unreliably on a configurable core operating at \( \text{UnRel} \) or re-executed on the \( \text{Rel} \) state.

\[
T(f, C) = \frac{C}{f} \quad (3.1)
\]

Besides the time spent to execute the tasks themselves, the system spends additional time to schedule tasks and to manage unreliable execution. Equation 3.2 calculates the cumulative execution time of an application, for a scenario where \( N_r \) tasks are executed reliably and \( N_u \) tasks are executed unreliably. Variable \( C_{\text{sched}} \) is the average number of cycles spent by the main part of the runtime system, running on a conventional core, to schedule the execution of a task. This applies to all tasks, irrespective of whether they will be executed reliably (on a conventional core) or unreliably (on a configurable core). Finally, \( C_t \) denotes the average number of cycles spent to execute a task, while \( C_{dc} \) is the average number of cycles spent by the runtime system to prepare for an unreliable task execution and then to reliably run the result-check/repair function (only for tasks that run unreliably on a configurable core). Note that our model is for a sequential task schedule. In this deliverable, we do not study the effect of parallel task execution.

\[
T_{\text{Total}} = T_{\text{Corr}} + T_{\text{UnRel}} + T_{\text{Rel}}
\]

\[
T_{\text{Corr}} = N_r \ast T(f_c, C_t) + (N_u + N_r) \ast T(f_c, C_{\text{sched}})
\]

\[
T_{\text{UnRel}} = N_u \ast T(f_u, C_t)
\]

\[
T_{\text{Rel}} = N_u \ast T(f_r, C_{dc})
\]

Figure 3.1: Nominal operating points for an Quad Core i7 processor (green), and indicative \( \text{Corr}, \text{UnRel} \) and \( \text{Rel} \) configurations. These nominal points have been extracted by forcing the core to specific frequency values (using the \text{likwid} tool) and monitoring the resulting voltage (using the \text{sensors} tool).
3.5.2 Power and Energy Modeling

The total power dissipation of a CMOS circuit is given by Equation 3.3, where $P_{\text{dynamic}}$ is the dynamic power dissipation, $P_{\text{leakage}}$ is the power dissipation due to transistor leakage current, and $P_{\text{shortCircuit}}$ is the power dissipation due to the short circuit formed when a CMOS gate switches. With the current manufacturing technologies which use high-k dielectric materials to drastically reduce leakage current, $P_{\text{dynamic}}$ is the main portion of the total power dissipation. Therefore, our model ignores the other power components. $P_{\text{dynamic}}$ is the product of the supplied voltage squared ($V^2$), the frequency ($f$) and the capacitance ($C$) of the chip.

$$P_{\text{Total}} = P_{\text{dynamic}} + (P_{\text{static}} + P_{\text{ShortCircuit}})$$

$$P_{\text{dynamic}}(V, f) = C \times V^2 \times f$$ (3.3)

The total energy dissipation is given by Equation 3.4. In general, this depends on the hardware/core configuration and the time spent to execute the runtime management functions, the application tasks, and their result-check/repair functions, as discussed above.

$$E_{\text{Energy}} = E_{\text{Cor}} + E_{\text{Rel}} + E_{\text{UnRel}}$$

$$E_{\text{Cor}}(V_c, f_c) = P(V_c, f_c) \times T_{\text{Cor}}$$

$$E_{\text{UnRel}}(V_r, f_u) = P(V_r, f_u) \times T_{\text{UnRel}}$$

$$E_{\text{Rel}}(V_r, f_r) = P(V_r, f_r) \times T_{\text{Rel}}(f_r)$$ (3.4)

3.5.3 Fault Modeling

In this deliverable, we consider timing faults only. These are closely related to the degree the system is undervolted (or overclocked), as well as the instruction mix of the application. Instructions which activate long paths that are close to the critical path, tend to fail more frequently [10]. The failure probability of each instruction is also closely related to the micro-architectural design of the CPU, as well as to the manufacturing process. Even identical chips with the same micro-architecture, using the same technology libraries, and running identical code, can have highly different behavior [5]. Moreover, whether a fault manifests as an error does not only depend on the paths which are activated during the current cycles—it also depends on the paths which were activated during cycles in the past [11]. Modeling such non-deterministic phenomena is almost impossible, as the conclusions are specific to the particular system that was used to create the model, and cannot be generalized to other systems. To the best of our knowledge there is no model which combines all the observations in a unified and applicable method. For these reasons we ignore the instruction mix of applications, and only take into account the effects of under-volting.

As has already been shown in [6], the fault rate at the point at which circuits start to fail timing constraints (Point of First Failure, PoFF) is extremely low; approximately 1 fault every 10 million cycles. Beyond this point, the fault rate increases exponentially at one order of magnitude for every 10mV drop of the supply voltage [4, 6]. As discussed earlier, to guarantee functional correctness, designers typically account for parameter variations by imposing conservative margins that guard against worst case variations. The extent of the voltage margins required to guarantee fault-free operation for all operating conditions of the chip is typically around 15% [7]. We determine the PoFF based on Equation 3.5, where $\alpha$ is the percentage of the extra margin voltage to guarantee fault-free operation and $V_n$ is the nominal supply voltage. We select $\alpha = 15\%$ to be consistent with several observations reported in the literature [7, 4, 6].
\[ V_{PoFF} = \frac{(100 - \alpha)}{100} \times V_n \]  

(3.5)

Based on the above reports, we also model the fault-rate as an exponential function \[ Err(V_n, V_u) = \beta \times e^{\gamma \times (V_n - V_u)} \]  

(3.6)

Finally, the operating frequency is linearly dependent to the supply voltage, as per Equation \[ f(V) = \delta \times V + \varepsilon \]  

(3.7)
In this chapter we introduce GemFI\cite{9}, a cycle accurate fault injection tool based on the Gem5 simulator \cite{3}. A primary objective of the tool is to enable fault injection based on different fault models and on systems with various configurations. We target full system simulations to evaluate the impact of faults on the complete system stack, from the architectural level up to applications. A variety of system configurations and architectures can be supported without affecting the implementation of fault injection in GemFI.

4.1 The Gem5 Simulator

Gem5 is a popular open-source system simulator. It provides a modular platform for computer system-level architecture research, encompassing system-level architecture as well as processor micro-architecture.

Object oriented design enhances the flexibility of Gem5. The ability to construct configurations from independent objects facilitates multicore and multi-system design. Moreover, Gem5 provides four different CPU models, each of them representing a different point in the speed vs simulation accuracy trade-off. Atomic Simple is a single IPC CPU model. Timing Simple is similar but also simulates the timing of memory references. Minor is a pipelined in order CPU. Finally, O3 is a pipelined out-of-order CPU model. Gem5 also supports two memory system models: classic and ruby. The classic is fast and easily configurable, while the ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherence memory systems.

Gem5 operates in two modes: System Call Emulation (SE) and Full System (FS). In SE mode applications execute on simulated “bare metal”. Whenever the program executes a system call, Gem5 traps and emulates the call usually by passing it to the host OS. Currently there is no thread scheduler in SE mode. Therefore, threads are statically mapped to a core, hindering its use with multi-threaded applications. FS mode offers an environment for running an operating system (OS) on top of the simulator. There is support for interrupts, exceptions and I/O devices. Applications are executed under the control of the OS.

Gem5 supports a number of ISAs, including Alpha, MIPS, ARM, Power, SPARC and x86. The simulators modularity allows these different ISAs to be easily implemented on top of the generic CPU models and the memory system. Alpha is the most maturely supported ISA, with ARM and x86 following.
4.2 GemFI Design and Implementation

We extended Gem5 with fault injection capabilities, following the General Processor fault model described in [?]. The result, GemFI, is a configurable tool for studying the effect of faults in a processor.

GemFI was developed using C++ and Python. It fully supports the Alpha and Intel x86 ISAs. Supporting more instruction sets is rather straightforward, since the implementation of GemFI is fairly ISA-agnostic. GemFI supports full system simulation mode as well as the execution of multi-threaded applications. An architectural overview of GemFI is depicted in Fig. 4.1 whereas the following sections discuss its main features in more detail.

4.2.1 GemFI User Interface

GemFI provides an API consisted of two intrinsic functions.

- **void fi_activate_inst(int id)** is translated to a pseudo-assembly instruction. Its successive occurrences toggle (active/inactive) the manifestation of faults for the specific process/thread. The executing thread is assigned a numerical id which can be used as an identifier of the thread in fault injection configuration.

- **void fi_read_init_all()** checkpoints the simulation. Upon restoring from the checkpoint, it resets all the internal information of GemFI, allowing the same checkpoint to be used as a starting point for multiple experiments with potentially different fault injection configurations.

On GemFI invocation the user also provides – at command line – an input file specifying the faults to be injected in the upcoming simulation. Each line of the input file describes the attributes of a single fault. Faults are characterized by four attributes: *Location, Thread, Time* and *Behavior*. 

---

Figure 4.1: An architectural overview of GemFI. The red components of the architecture demonstrate the possible locations where faults can be injected, whereas the red ovals represent applications which use the extended ISA.
Location

Fault location specifies the micro-architectural modules to be targeted for fault injection. The user specifies the core, the module within the core and finally the specific bit location to be corrupted. Supported locations include registers (integer, floating point, special purpose), the fetched instruction, the selection of read/write registers during the decoding stage, the result of an instruction at the execution stage, the PC address and finally memory transactions (load/stores).

Thread

The thread attribute allows to selectively inject faults to specific threads, using the id assigned to the thread upon execution of $fi_{activate\_inst}(id)$ as an identifier.

Time

Another important aspect of the fault injection configuration is its timing. Timing is relative to a simulation milestone, marked by the execution of the $fi_{activate\_inst}$. Faults are scheduled relatively to the number of instructions already executed, or to the number of elapsed simulation ticks of the targeted thread.

Behavior

The values of the specified faulty location can be corrupted in following ways:

- by assigning an immediate value provided by the user to the location.
- by XORing the running value at this location with a user-specified constant.
- by flipping the running value at bit locations. Multiple bit flips are supported by injecting multiple faults on the same module.
- by setting all bits of the location to a value of 0 or 1.

To emulate the behavior of transient and permanent faults, the user can define how long the fault is active in terms of the number of simulation ticks or number of instructions. For example, a fault injected in the execution stage of the processor can be injected continuously for the next N instructions (or for the next N simulation cycles) if so instructed by the user.

4.2.2 Simple Example

Listing 4.1 outlines a user-provided fault configuration example. The fault is injected in the 21st bit of register $R_1$ of the CPU (location), when the application fetches the 2457th instruction after the initiation of fault injection for this thread ($fi_{activate\_inst}$). The fault is activated for a single instruction (occ:1) and only for the thread with id equal to 0.
The end user compiles (or cross-compiles) the application to be tested (Listing 4.2). Target applications must, at least, contain one call to initialize fault injection. Afterwards, the user moves the binaries into the disk image serving as the virtual disk of GemFI. Using the command line, the user provides a configuration file (Listing 4.1) describing all the faults to be injected in the simulation. After `fi_activate_inst(id)` is called, the thread identifier is stored in the internal data structures of GemFI. Simulation continues normally, until it is time for a fault to be injected. At that time, GemFI alters the state of the target hardware structure according to the fault specification in the configuration file.

### 4.2.3 GemFI Internals and Implementation

Fig. 4.2 demonstrates the main abstract steps executed by GemFI on each simulated served instruction.

Threads that have enabled fault injection are internally represented as instances of a class (`ThreadEnabledFault`), containing all per thread information necessary for fault injection, such as the number of instructions the thread has executed on each core. Each simulated core has a pointer to a `ThreadEnabledFault` object. If the thread executing on the core has not activated fault injection, the pointer is NULL. When a thread executes `fi_activate_inst()`, GemFI looks in a hash table to identify whether the specific thread has already activated fault injection. Threads are identified at the hardware/simulator level by their unique Process Control Block (PCB) address. If the thread is not found in the hash table, a new `ThreadEnabledFault` object is created and the running core is set to point to that object. On the other hand, if there was already an entry in the hash table, the invocation of `fi_activate_inst()` deactivates fault injection for the
specific thread. The thread is removed from the hash table, the corresponding `ThreadEnabledFault` object is destroyed and the core’s pointer is set to NULL. During context switches, which are identified by the change of the PCB address, GemFI checks whether the switched-in thread has activated fault injection, in order to properly set the core’s pointer to the thread’s `ThreadEnabledFault` object. Monitoring context switches allows GemFI to eliminate the overhead of checking the fault injection status of the executing thread in the hash table on each simulated clock tick.

Faults are described in the input file provided by the user at GemFI command line. The file is parsed at startup and each fault is inserted to one of five internal queues. Each queue corresponds to a different pipeline stage.

On each simulation tick, GemFI checks if fault injection has been enabled for the running thread. In such a case, it prefetches the corresponding `ThreadEnabledFault` objects. Then and for each instruction served at a pipeline stage, GemFI updates the thread’s data and scans the corresponding queue for faults targeting the executing thread at the specific simulation point. Queue entries are sorted according to the timing of each fault. If such a fault is found, the value of the targeted location is corrupted according to fault’s behavior.

### 4.2.4 Simulation Checkpointing

Checkpointing allows saving the state of a process or a system at a specific time snapshot and reverting to that later, to restart the execution from that point if needed. Checkpointing is necessary in order to avoid losing simulations in case of unexpected failures. It is particularly useful when simulation campaigns are executed to non-dedicated networks of workstations, a feature supported by GemFI.

Gem5 provides checkpointing, however with limitations. One method is to switch the simulation from O3 to atomic simple mode, create the checkpoint, and revert back to O3 mode to continue the simulation. This requires a pipeline flush, presenting a potential realism loss hazard. The second method requires simulating the MOESI hammer cache coherency protocol, which however dramatically increases simulation time.

We used DMTCP (Distributed MultiThreaded Checkpointing) to checkpoint the state of the Linux process running the simulator, instead of checkpointing the internal state of the simulator. A feature of DMTCP is its ability to take checkpoints either by programmatically invoking checkpointing from within the process to be checkpointed, or asynchronously, by setting environment variables. The ability to invoke DMTCP from within the simulator allows us to exploit the front-end checkpointing mechanism of Gem5, while altering the checkpointing back-end to use the DMTCP API.

Apart from protecting against unexpected problems in simulation campaigns, checkpointing can be used to speed-up simulations. Before starting simulation campaigns, the user executes one simulation up to the point when fault injection is activated (including booting of the operating system and application initialization). Using GemFI’s API
the user can checkpoint the simulation at this point. The saved state is then used as a starting point for all experiments in the campaign (Fig. 4.3). Upon restoring a checkpoint GemFI parses again the faults configuration file. Therefore, this strategy allows fast-forwarding of the execution to the checkpoint and spawning of multiple experiments, with different fault injection configurations from that point on.

### 4.2.5 Simulation Campaigns on a Network Of Workstations

GemFI is accompanied by a set of shell scripts which facilitate launching simulation campaigns on a Network of Workstations (NoW). The workstations need to share a network file-system, in order to store the fault description files of the experiments, the simulation checkpoints and the output of each simulation. The main steps for parallel execution of simulation campaigns on a NoW are the following:

1. The configuration files for all experiments are stored on a network share.
2. A simulation is executed up to the point fault injection is activated and the simulator process is checkpointed. The checkpoint is stored to the share.
3. Each workstation gets a local copy of the checkpoint.
4. Each workstation checks the share for experiments to be executed. It selects one of the remaining experiments and executes it locally, starting from the checkpointed state.
5. Simulation results are moved from the workstation back to the network share.
6. Steps 4-6 are repeated until there are no experiments left.
Chapter 5

Evaluation

5.1 Evaluation Methodology

Evaluation of a system operating at an unreliable configuration raises a number of challenges. Native execution on a real processor is the most favorable solution in terms of realism of results and of problem sizes used. However, commercial processors typically prohibit configuration outside their normal working envelope. Moreover, simulated fault injection offers more control in injecting faults at particular layers of the architecture. Therefore, we apply a hybrid evaluation methodology using a combination of simulated fault injection, native execution with software fault injection on a real system, and analytical energy modeling using profiling data from the native execution. The approach is described in more detail in the following paragraphs.

5.1.1 Fault Injection

We use simulation fault injection at the architectural level of a simulated CPU to characterize application behavior in the presence of faults. We use the GemFI framework [9] to execute the benchmark applications on a simulated out-of-order CPU supporting the x86 ISA. GemFI enables us to inject faults at different stages of the CPU pipeline. In the fetch stage, a fault corrupts a single bit of the instruction. In the decoding stage, the selection of registers is corrupted so that the instruction in question reads from, or writes to a different register. In the execution stage, faults corrupt a single bit of the computed result. Finally, faults in the memory stage corrupt a single bit of the value being transferred from/to memory.

Initially, we capture application behavior on top of unreliable hardware. All protection mechanisms are disabled, all tasks are susceptible to faults, and the result-check functions (if available) are ignored. We perform single fault injections. The number of fault injection experiments, for each application and fault injection stage, is decided based on the methodology described in [8], for a 99% confidence level and 1% error margin. Based on these simulation campaigns, we observe how each fault type impacts each target application. For the purpose of our evaluation, we categorize the outcome of program execution in three bins: (i) crash (the program did not terminate normally), (ii) inexact (the program produced a result that is not the same as that of a reliable execution), and (iii) exact (the result is bit-wise identical to that of a reliable execution). The output of this phase is the probability a fault will result to a crash, inexact or exact result ($P_{\text{crash}}$, $P_{\text{inexact}}$ and $P_{\text{exact}}$ respectively) for each benchmark. These probabilities are used by the software fault injection mechanisms during native execution.

In a second step, we use software fault injection during native executions of benchmarks on an Intel x86 platform. Fault injection at this phase has two possible effects: (i) it forces a crash, e.g., by causing a segmentation fault;
(ii) it corrupts a randomly chosen register of the processor. The former is done with the probability \( P_{\text{crash}} \) computed in GemFI simulation; the latter with probability \( 1 - P_{\text{crash}} \). As in simulated campaigns, faults are injected in all application tasks, and all protection mechanisms are disabled. In order to validate that software fault injection yields realistic results, we compare the outcome of software fault injection during native executions with the respective outcomes of simulated executions on GemFI. Figure 5.1 illustrates the results for all benchmarks. It can be observed that software-based fault injection during native executions has practically equivalent effects to architecture-level fault injection during simulated executions on GemFI.

To combine the effects of core configuration-related fault rate with application-specific behavior, we extend the above single-fault software-based method to inject multiple faults, at different fault rates, during native application execution. The configuration parameters (frequency and voltage) for the Corr and the \( UnRel \) state are decided as follows: (a) We select \( f_c = f_a \) in order to maximize performance, (b) \( V_c \) is derived from Equation 3.7 and (c) we derive the fault rate of the \( UnRel \) configuration from Equation 3.6, using \( V_r \) and \( V_c \) as parameters.

Given a target fault rate, the software fault injector randomly generates a set of fault-injection intervals (expressed as number of cycles between faults) using a uniform distribution with a mean value equal to the target fault rate. We then exploit the performance counter infrastructure of x86 CPUs to interrupt application execution at those intervals and invoke the software-based fault-injection logic. At execution time the performance counters also profile \( C_t \), \( C_{\text{sched}} \) and \( C_{\text{dc}} \). For each application, protection mechanism and voltage level (thus target fault rate) we perform 10,000 multiple fault injection experiments, giving a confidence interval of 95% and an error margin of 2.5%.

In the end the runtime system reports the values of the profiled quantities, result quality, and the number of tasks executed reliably and non-reliably.

### 5.1.2 Energy Estimation

We estimate the dynamic energy dissipation of the core using Equation 3.4. We use the parameters \( V_r, V_c, f_r \), and \( f_c \) corresponding to each system configuration, the number of tasks executed reliably and non-reliably (reported by the runtime) and the values of \( C_t, C_{\text{sched}} \) and \( C_{\text{dc}} \) profiled during the native execution.
5.2 Fault Tolerant Scenario

We evaluate our framework in terms of error resiliency, energy savings and the ability to gracefully trade off result quality for increased energy efficiency. We experiment with six different configurations in terms of error resiliency mechanisms:

- **No Protection (NP):** No error detection and correction mechanism and no programmer wisdom is used. All tasks are executed unreliably and are susceptible to faults. A task crash leads to application crash.

- **Basic Protection (BP):** The runtime identifies and handles errors using the standard processor/OS protection mechanisms and soft-checkpointing (as discussed in Section ??). Task crashes are caught, allowing the application to continue its execution with whatever output was produced by the crashed tasks.

- **Basic + Result Checking (B-RC):** In addition to the runtime protection of BP, the programmer provides result-check functions in order to detect and correct errors (due to crashes or silent data corruptions).

- **Basic + Significance (B-SF):** On top of BP, the user characterizes tasks according to their significance. Only non-significant tasks are susceptible to faults. However, the programmer does not provide result-check functions. Task crashes are caught and ignored as in BP.

- **Full System (FS):** The entire protection arsenal is employed, including runtime system protection, significance characterization and result-check functions. Faults are injected only to non-significant tasks.

- **Full System Re-Execution (FS-RE):** The configuration is almost identical to FS. However, if an error is detected the runtime re-executes the task reliably, instead of attempting to correct the result in the result-check function.

5.3 Benchmarks

We use a set of five benchmarks **DCT, Sobel, Blackscholes, K-means, and Jacobi**, briefly described below.

**DCT** is a module of video compression kernels, which transforms a block of image pixels to a block of frequency coefficients. The tasks that compute low frequency coefficients, close to the upper left corner of each 8x8 frequency block, are more significant than the ones computing coefficients towards the lower right corner of the block. The result-check function attempts to detect errors in the task output via a heuristic out-of-bounds check; coefficients that do not respect the bounds are set to zero. Infinite loops are handled using the timeout functionality at task group synchronization points.

**Sobel** is a 2D filter for edge detection in images. Tasks compute horizontal slices of the output image. All pixels and thus all tasks are equally significant to the quality of the result. We randomly choose the significance of each slice, so that less significant tasks are spatially distributed throughout the image. The task-result-check function executes an approximate version of the Sobel filter, using a lightweight stencil with just 2/3 of the filter taps for tasks that have crashed.

**Blackscholes** is a benchmark of the Parsec suite [2]. It implements a mathematical model for a market of derivatives which calculates the buying and selling of assets so as to reduce the financial risk. All tasks are assigned equal values of significance since there is no reliable way to know beforehand the significance of each stock option. Results are again checked with the isnormal() macro. If the check fails, we recompute the erroneous result using an approximate implementation of the Blackscholes formula by substituting costly mathematical operations (such as expr(), sqrt(), log()) with approximate versions.

**K-means** is an iterative algorithm for grouping multi-dimensional data points into $k$ clusters. Groups of data points are assigned to tasks which independently identify the cluster nearest to each point, and then assign the point to that
cluster. Afterwards, K-means computes the center of the cluster based on the points that belong to the cluster. Since all points are equally important we consider all tasks to be non-significant; errors in the assignment of individual points to clusters are tolerable. On the other hand, computing the center of the cluster should be executed precisely as it is much harder to recover from an erroneous calculations. The result-check function of non-significant tasks is minimalistic, exploiting the error-tolerant nature of this iterative application: if a point subscribes itself to a non-existing cluster then it reverts to its previous cluster. Also, if the run-time reports an error then all points computed by the task discard their computations and are subscribed to their respective previous clusters. Note that we repeat the above algorithm for a fixed number of iterations, selecting a last percentage (ratio) of iterations to be executed reliably. This is because reliable iterations can correct erroneous results due to mispositioned data points from previous iterations.

Jacobi is an iterative solver for diagonally dominant systems of linear equations described by the equation $Ax = b$. We monitor the behavior of the Jacobi solver for a fixed number of iterations. Tasks processing the initial Jacobi iterations are not significant since errors at that stage are not critical to the convergence of the method. In later steps, when the algorithm approaches the final solution, tasks become more significant. It is hard to create an error detection mechanism in Jacobi, since quality of results is often associated with the application in which the solver is used. Consequently, we use a simple result-check function which checks whether the computed outputs fail the isnormal() check. This is a glibc floating point classification macro, which returns non-zero if the argument is not a normal floating point number. If so, the current solution estimate is replaced with that of the previous iteration.

### 5.4 Experimental Results

We study the behavior of benchmarks for the six scenarios, using the hybrid methodology discussed in Section 5.1 in an Intel Core i7-4820K CPU clocked at 2.90 GHz. The y-axis of Figure 5.2 corresponds to the energy gains of a single task of DCT, when executed at an UnRel hardware state (frequency set at 2.9 GHz with a voltage lower than the nominal 1V), in comparison with an execution at the nominal voltage for the specific frequency (Corr state). At 0.83V faults occur every $10^4$ cycles on average. If we further undervolt the processor, inducing faults at an even higher rate, all tasks crash, introducing overheads due to the activation of protection mechanisms, potential re-executions etc. Moreover, system software and result check functions, which always execute reliably at the nominal frequency for each voltage setting, become slower, thus further reducing any energy gains due to aggressive undervolting. Since this effect is observed in all benchmarks, we limit the undervolting to 0.83V during the evaluation. On the other hand, when a core operates in voltage regions higher than the 0.85V, the induced failure rate is so small, that we activate the functionality of our framework rarely. Studying the voltage range from 0.83V to 0.85V demonstrates the full extent of energy gains which can be attained by our framework, while still being able to protect applications from erratic behavior.

The number of reliable and unreliable workers is configurable (to match the desired type of system). In this deliverable, we focus on the fault-tolerance and energy-efficiency of serial task execution scenarios. Therefore the runtime system uses only one reliable and one unreliable worker. The reliable worker runs, together with the main application thread and runtime logic, on the conventional core, whereas the unreliable worker runs on the configurable core. The operating states of each core are presented in Table 5.1. For the UnRel configuration we also present the fault rate.

Figure 5.3 summarizes our experimental results for different (below nominal) voltage settings (0.83V, 0.84V and 0.85V) and protection mechanisms. For each benchmark we present three diagrams. The leftmost one depicts the cumulative distribution function (CDF) of the percentage of experiments (y-axis) achieving a specific quality threshold (x-axis) under different protection mechanisms (different lines). For the media benchmarks (DCT, Sobel) the quality metric is PSNR (higher is better). For the remaining benchmarks quality is quantified by the relative error w.r.t the fully reliable execution (lower is better). The two extreme bins of the x-axis correspond for each protection configuration (line), on the one side to experiments which resulted in bitwise exact results, and on the other side to experiments producing very bad output quality. The percentage of crashed experiments can be deduced by subtracting
Significance-Based Computing for Reliability and Power Optimization (SCoRPiO)

Figure 5.2: Energy gains of a single task for DCT when executed at lower than nominal voltage.

Table 5.1: Operating parameters for each processor state for the configurable and the conventional cores. For the UnRel state we also present the corresponding average fault rate.

<table>
<thead>
<tr>
<th></th>
<th>Corr</th>
<th>Rel</th>
<th>UnRel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Freq.</td>
<td>Voltage</td>
<td>Freq.</td>
</tr>
<tr>
<td></td>
<td>2.9 GHz</td>
<td>1.0V</td>
<td>1.02 GHz</td>
</tr>
<tr>
<td></td>
<td>0.89 GHz</td>
<td>0.84V</td>
<td>0.84V</td>
</tr>
<tr>
<td></td>
<td>0.76 GHz</td>
<td>0.83V</td>
<td>0.83V</td>
</tr>
</tbody>
</table>

When no protection mechanism is active, even error-resilient applications fail to terminate properly in all or most cases for increased fault rates. Moreover, basic OS/runtime protection (BP) eliminates crashes, and can even lead to satisfactory behavior as long as the fault rate remains moderate. As expected, error resilience increases as more protection mechanisms are employed. As an exception, result-check functions (B-RC) may produce worse results compared with BP, by discarding partially good results produced by tasks before they crash. On the other hand, energy gains are typically reduced as the amount of protection increases. Finally, when combining all protection mechanisms, the breaking point of application resilience to errors is pushed towards significantly higher fault rates. In the following paragraphs, we discuss the behavior of each application in more detail.

The two image processing benchmarks (DCT and Sobel) demonstrate a similar behavior. The transition from NP to BP is adequate to completely eliminate any program crashes. However, there is no guarantee for the quality of the output. Both applications produce outputs of unacceptable quality when executed at the 0.83V step. The addition of a result check function has a positive effect on output quality, with the exception of DCT at the lowest voltage configuration. B-SF results in worse quality than the B-RC. Although significant tasks are protected in B-SF, if the erroneous outputs of non-significant tasks are not corrected they tend to destroy the output. When significance is combined with a result check function (FS configuration) both applications achieve high output quality: all experiments provide quality higher than 29 and 40 dB for DCT and Sobel respectively, at the 0.83V step, where faults occur every 10000 cycles. For Sobel B-RC behaves similar to FS. Significance assignment in Sobel simply spreads unreliability uniformly within
the output, however PSNR does not capture such effects. Finally the FS-RE configuration provides slightly better quality than FS. Only a limited percentage of results are categorized as bit-exact in the FS-RE configuration, although tasks are re-executed reliably. This is because the error detection part of the result check function of both benchmarks does not manage to identify all errors. Figure 5.4 illustrates the output of four protection configurations (excluding NP and FS-RE) for the DCT benchmark. The corrupted images show the adverse effect of multiple faults when protection is not adequate, while the image at the bottom left shows that even in a highly faulty environment, our approach can almost eliminate visible artifacts.

In DCT, when using the FS configuration, energy gains vary from 23% to 26% when the voltage is set to 0.85V and 0.83V respectively. On the other hand, for Sobel the energy gain remains stable for different supply voltages. The result check function of DCT just sets a default value (0) to the faulty output. For Sobel, an approximate version of the task is executed. Therefore, the energy gains due to undervolting are eliminated by executing the result check function more frequently due to the higher fault rate. A similar trend is observed for DCT in the case of the FS-RE configuration. Re-executing the entire task every time its output is detected as erroneous outweighs all energy savings and can often even result in energy losses. When evaluating the influence of ratio, we observe that the user can easily control the quality / energy trade off by using a single knob in the taskwait clause.

Blackscholes is a computationally intensive application with a high ratio of floating point instructions to loads/stores.

Figure 5.3: Experimental results. Percentage of experiments which achieved a certain quality (left), energy gains with each protection scenario (middle), and quality versus energy trade-offs using the Ratio knob for FS and FS-RE (right).

D5.3. Software hardware integration, validation and evaluation

Significance-Based Computing for Reliability and Power Optimization (SCoPiO)
The low probability of faults on address calculations and memory accesses reduces the probability of crashes, even in the NP scenario. As expected, this natural resiliency on crashes diminishes as the fault rate increases due to more aggressive undervolting. For example, at 0.83V all experiments result in a crash without protection. The FS configuration attains the same energy gain as FS-RE. Output quality is exceptional for FS-RE: a relative error less than 0.03 is guaranteed for all experiments.

K-means is inherently error tolerant, as it employs an iterative heuristic algorithm to optimize the classification of points. The NP scenario fails to produce any results in all tested voltage configurations. The BP and B-SF scenarios produce almost indistinguishable results. Again, due to the simplicity of the detection part of the result check function, it often fails to identify silent data corruptions. On the other hand it is essential to define the last iterations as significant, since the last iterations can fix any anomalies produced by the initial non-reliable ones. Therefore, in all scenarios which exploit significance information the relative error is negligible (< 10^{-8}). When re-executing faulty tasks (FS-RE) the energy gain drops significantly and results in energy losses when aggressively undervolting.

Finally, Jacobi demonstrates the limitations of our approach. Even though we do not suffer from crashes when even the simpler protection mechanisms are used, result quality is unacceptable in most cases, with the exception of FS and FS-RE configurations when executing at 0.85V. In these cases, relative error is less than 0.1 in 85% of the total number of experiments. In general, erroneous results quite often go undetected from result check functions and propagate forward through the iterations of the algorithm, making the result diverge from the correct solution.

The experimental results indicate that the main energy gains are obtained by shaving voltage guard bands to the point of first failure. From that point on, more aggressive downvolting results in minimal improvement of energy efficiency. This is reasonable since error detection and correction takes place too often, adding overhead and thus limiting energy gains.
Figure 5.4: DCT output at 0.84V, with one fault injected every 10,000 cycles. The images correspond (from top left and moving clockwise) to the BP, B-RC, B-SF and FS protection configurations, resulting to PSNRs of 15, 16, 20 and 29 dB respectively. NP leads to crashes.
Chapter 6

Usage Instructions

6.1 Installation instructions

Compiling the SCoRPiO compiler and GemFI requires the following packages when using an Ubuntu based distribution.

6.1.1 Installing the software stack

Create a new directory called RTS. Change directory to the RTS one and follow the upcoming instructions:

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU Make</td>
<td>3.79</td>
<td>Makefile/build processor</td>
</tr>
<tr>
<td>GCC</td>
<td>&gt;=4.7.0</td>
<td>C/C++ compiler</td>
</tr>
<tr>
<td>python</td>
<td>&gt;=2.7</td>
<td>Python interpreter</td>
</tr>
<tr>
<td>libtool</td>
<td>1.5.22</td>
<td>Shared Library Manager</td>
</tr>
<tr>
<td>zlib</td>
<td>&gt;1.2.3.4</td>
<td>Compression Library</td>
</tr>
<tr>
<td>ocaml</td>
<td>3.12.1</td>
<td>ocaml compiler</td>
</tr>
<tr>
<td>automake</td>
<td></td>
<td>Build processor</td>
</tr>
<tr>
<td>indent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>emacs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td></td>
<td>Perl interpreter</td>
</tr>
<tr>
<td>libc</td>
<td></td>
<td>The GNU C library</td>
</tr>
<tr>
<td>bison</td>
<td></td>
<td>General-purpose parser generator</td>
</tr>
<tr>
<td>flex</td>
<td></td>
<td>Lexical Analyzer</td>
</tr>
<tr>
<td>gperf</td>
<td></td>
<td>Hash function generator</td>
</tr>
<tr>
<td>SCons</td>
<td>&gt;0.98.1</td>
<td>A powerful build tool similar to make</td>
</tr>
<tr>
<td>SWIG</td>
<td>&gt;2.0.4</td>
<td>Interfaces C and C++ with high-level programming languages.</td>
</tr>
<tr>
<td>m4</td>
<td></td>
<td>The macro processor</td>
</tr>
<tr>
<td>protobuf</td>
<td>2.1</td>
<td>Trace capture and playback support</td>
</tr>
<tr>
<td>dmtpcp</td>
<td>2.2.1</td>
<td>Checkpointing mechanism</td>
</tr>
</tbody>
</table>

Table 6.1: The prerequisite packages for installing the compiler on Ubuntu Linux 14.04
### Installing the SCoRPiO Compiler

To install SCoRPiO source-to-source compiler download the distribution .zip from [http://scorpio.ireteth.certh.gr/scoop_bundle.zip](http://scorpio.ireteth.certh.gr/scoop_bundle.zip). Extract the file and enter the newly created folder. SCoRPiO compiler accesses several header files contained in the runtime source code tree. Therefore, the runtime system headers specifying the runtime API need to be available to the compiler. To facilitate the installation of the compiler we offer a bundle of the required header files in the compiler distribution. They are placed within the compiler source tree in the directory `scoop/bddt_headers`.

After installing the required packages `cd` to the `SCoRPiO compiler` directory and issue `./configure`. This script will notify you in the event of missing packages. Before you proceed to the next step make sure that the `./configure` script does not terminate due to an error.

Finally, execute `make` and the SCoRPiO compiler will be built. The compilation process will produce an executable script named `scoop` in the directory `scoop`. At this point you should modify your `~/.bashrc` file by appending the following line to the end of the file:

```
export PATH=$PATH:<full path to scoop directory>
```

This final step updates the `PATH` environment variable to enable executing `scoop` without specifying the full path to the binary.

### Installing the Runtime system

To install SCoRPiO runtime system download the distribution .zip from [http://scorpio.ireteth.certh.gr/rts.zip](http://scorpio.ireteth.certh.gr/rts.zip). Extract the files and enter the newly created directory. After installing the required packages `cd` to the `RTS` directory. The directory consists of one directory, the `runtime` Change directory to the `RTS` directory and issue the `make` command. If the runtime is to be executed on top of the simulator issue the `make static=1 queue_atomic=1 gemfi=1 nosignals=0` command. The command can take the parameters presented in Table 6.2 to build specific runtime attributes.

### Installing The Benchmarks

To install SCoRPiO runtime system download the distribution .zip from [http://scorpio.ireteth.certh.gr/benchmarks.zip](http://scorpio.ireteth.certh.gr/benchmarks.zip). Change directory to the `RTS/benchmarks` and issue the `ls` command. Each ported benchmark has it’s own directory. All benchmark directories consist of the following directories/files:

- **Makefile**: A build file which includes all the rules to build each benchmark. After issuing the `make` command

### Table 6.2: Compilation flags supported when making the runtime system

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static</code></td>
<td>0</td>
<td>build a static library</td>
</tr>
<tr>
<td><code>gemfi</code></td>
<td>0</td>
<td>The runtime system targets a simulation environment, hence the fault injection infrastructure is activated</td>
</tr>
<tr>
<td><code>nosignals</code></td>
<td>1</td>
<td>The runtime system handles traps/signals which are send by the HW/OS system due to the manifestation of faults</td>
</tr>
<tr>
<td><code>queue_atomic</code></td>
<td>1</td>
<td>Use spinlocks instead of posix mutexes to synchronize workers</td>
</tr>
</tbody>
</table>

D5.3. Software hardware integration, validation and evaluation
the executable is stored under the bin directory.

- inputFiles: Files containing all input/configuration files necessary for the benchmark to be executed.
- source/header files: Source files and header files for the benchmark.

To compile a benchmark change directory to the required benchmark directory and issue the make command. The makefiles suppose that the scoop compiler is under the RTS/scoop directory.

### 6.1.2 Installing GemFI

To install the scorpio simulator download the distribution from the [http://scorpio.ireteth.certh.gr/gemfi.zip](http://scorpio.ireteth.certh.gr/gemfi.zip). Extract the files of the zip and enter the newly created folder. Within the directory there are 2 directories Sim and X86. The Sim directory contains the simulator source code and utility files. The X86 directory contains the Linux binary and an image disk which are going to be used by the simulator. Enter the Sim directory and issue the scons -j "#Threads" build/X86/gem5.opt command. This command will compile the simulator and should take a considerable amount of time to be build.

### 6.2 Execute the SCoRPiO framework

We will use as an example the Sobel benchmark. As a first step change directory to the RTS/runtime directory and issue the make static=1 queue=atomic=1 gemfi=1 nosignals=0. This command will statically link the RTS library. Fault injection simulations assume that non-reliable tasks call the _fi_activate_inst_ to instruct gemfi to inject faults during their execution.

As a second step, the user has to change to the RTS/benchmarks/sobel/ directory and issue the make gemfi=1 command. This command compiles the binary and links it with the RTS and the gemfi libraries. After the compilation finishes the executable is found under the bin directory.

The following step copies the executable and the input files to the simulator hard disk. The disk is an image file called x86root.img and can be found under the gemfi/X86/disks/ directory. The user must go to the aforementioned directory and create a directory called mountpoint. Then the user must mount the image file using the command _sudo mount -o loop,offset=32256 x86root.img mountpoint_. At this point the simulator filesystem can be viewed under the mountpoint directory. The next step is to go to the mountpoint directory and copy the sobel executable and all the input file (peppers512.raw). Then, the user has to unmount the image file using the command _sudo umount mountpoint_.

In this step the user has to instruct the simulator to execute the Sobel benchmark. In the first step the user builds the simulator using the scons -j build/X86/gem5.opt command. After the build procedure terminates, the user creates a script which will be executed after the simulator boots the OS. The script essentially calls the executable command. First, the user goes to the gemfi/Sim/configs/boot/ directory and creates a file called sobel.rcS which contains Listing 6.1.

The script informs the runtime system about the available number of threads and the system domains. More information about the environmental variables can be found in section 3.3. Line 6 of the execution script executes Sobel with a synchronization ratio of 0.5 and 2 workers. One worker will always execute reliably and the second one will execute non-reliably, in the sense that faults might be injected on the instructions that it executes.

The simulator configuration files need to be updated about the existence of the sobel.rcS script. To do, the user switches to the gemfi/Sim/common/ directory and edits the Benchmarks.py file. In line 80 of the file, the Benchmarks directory is specified. The user has then to enter the line shown in Listing 6.2. After all the previous steps are performed, the Benchmarks.py should be similar to Listing 6.3.
At this point a fault injection simulation can be performed. The user, exports the $M5\_PATH$ variable to point to the directory $gemfi/X86$ by specifying the full path. The user then, changes directory to $gemfi/Sim/\$ and issues the command in Listing 6.4. The $dmtcp\_checkpoint$ command starts up a daemon which connects to the simulator and is responsible for checkpointing. The -r option redirects the output of the simulator to a text file called $system\_pc\_com\_1\_terminal$. The -d options redirects all the simulator output to a directory called, in this case, $sim\_out$. -debug-flags=FaultInjection instructs the simulator to print out any information relevant to the fault injection. config/example/fs.py passes a script to the simulator which is responsible to instantiate and create all the simulated objects. -n 2 instantiates 2 cores for the simulation. The option -b sobel instructs the simulation to execute the sobel.rcS script after the booting procedure finishes. The option -fi-in=input informs gemfi about the location of the fault description input file. Finally, --init-checkpoint=1 instructs gemfi to create a checkpoint when initializing the fault injector. This checkpoint can be used to inject different faults by changing the fault injection description file.

After the simulation finishes, the user must go to the $sim\_out$ directory. In this directory all the simulation output is present. The $OUTPUT$ file is the output sobel image. Opening the $simout$ file with a text editor and going to the end of the file should show text which is similar to Listing 6.5. All lines with the format Task: id CORE: core Name "Fetched Instructions" "Decoded Instructions" "Executed Instructions" "Memory Instructions" provide information about the number of instructions that each task has fetched/decoded/executed etc. There is a unique core and task id called all which captures the number of instructions which were executed from all tasks and on all cores.

Finally to create a set of fault injection experiments, the user needs to enter the $gemfi/Sim/util/createFI/$ directory and issue the $make$ command. This will create a fault generator which will be used in the next step. This fault generator expects as arguments: the number of fetched, decoded, executed, and memory instructions performed to produce a text file called $Faults.txt$. An example is presented in Listing 6.6. Each line of the file corresponds to a single fault.
injection experiment. To conduct a fault injection simulation the user copies any line from the Faults.txt file into the input file in the gemfi/Sim/ directory. Then, the user enters the simulator root directory (gemfi/Sim/) and issues the command presented in Listing 6.7. This command restarts the simulation from the checkpoint created earlier. The fault injection simulation output will be stored in the sim_out directory.

```bash
export M5_PATH=/FULL/PATH/TO/gemfi/X86/
dmtcp_checkpoint ./build/X86/gem5.opt -r -d sim_out --debug-flags=FaultInjection \ ./config/example/fs.py -b sobel -n 2 --fi-in=input --init-checkpoint=1
```

Listing 6.4: Simulation command.

```bash
5887827921500: system.fi_system: "****Fault Injection Re-Activation Instruction (261)*****"
5887848330500: system.fi_system: "****Fault Injection Re-Activation Instruction (260)*****"
5887868739500: system.fi_system: "****Fault Injection Re-Activation Instruction (259)*****"
5887889148500: system.fi_system: "****Fault Injection Re-Activation Instruction (258)*****"
5887909557500: system.fi_system: "****Fault Injection Re-Activation Instruction (257)*****"
5887929966500: system.fi_system: "****Fault Injection Re-Activation Instruction (256)*****"
5887943443500: system.fi_system: "****Fault Injection Re-Activation Instruction (255)*****"
5887963676000: global: GOT DUMP MESSAGE
5887963681000: global: Ignoring Switch request
Exiting @ tick 5888549156000 because m5_exit instruction encountered
```

Listing 6.5: Simulation output.

```bash
./generator -f <FETCHED INSTR> -d <DECODED INSTR> -e <EXECUTED INSTR> -l <MEMORY INSTR>
```

Listing 6.6: Fault experiments creation.
6.3 Using the Virtual Machine

In order to further facilitate the installation and use of the SCoRPiO development environment, we have created a virtual machine with the appropriate software packages, as well as the source code the SCoRPiO compiler, Run-Time system, benchmarks and the simulator. You can download the virtual machine files at [http://scorpio.ireteth.certh.gr/scorpiovm.zip](http://scorpio.ireteth.certh.gr/scorpiovm.zip).

The login credentials are:

```
User name: scorpio
Password: scorpio2016
```

6.4 Known Issues

In this section we discuss a number of known issues of the integration of the SCoRPiO project.

The kernel being executed on top of the Gem5 simulator freezes on an internal spin lock due to the bad implementation of the timing of the memory. This takes places only when simulating an out of order system. This is a known issue in the Gem5 community. We are actively trying to resolve the issue.

SCOOP uses a source-to-source compiler framework named CIL to parse and output C source files. This framework requires that all `#pragma` directives are coupled with a statement. However, it may be the case that a `#pragma omp taskwait` directive is placed at the very end of a block of statements. In such a scenario the input source code will fail to be parsed by CIL. We recommend placing an appropriate statement like break, continue or return in such cases to workaround this CIL requirement.

Currently the runtime system uses a non-straightforward method to acquire task arguments. As such, unnecessary restrictions arise during the compilation process. More specifically, a task argument may either be long or a pointer. We consider this not to be a major setback with regards to the functionality provided by the compiler. Our benchmarks show that most arguments can either be cast to long or be passed as pointers.

Another constraint imposed by the runtime system is that tasks accept at most 8 arguments. We have communicated this issue as well as the previous one to the team developing the runtime system. Since these are low priority issues we have decided to work on them when all higher priority features and issues have been dealt with.
Bibliography


