SCoRPiO

Significance-Based Computing for Reliability and Power Optimization

SEVENTH FRAMEWORK PROGRAMME

D2.1: Programming model specification

Contract No:	ICT-323872
Contractual Date of Delivery:	28/02/2014 (M9)
Actual Date of Delivery:	11/03/2014
Main Authors:	Konstantinos Parasyris , Vassilis Vassiliadis , Christos Antonopoulos , Nikolaos Bellas , Spyros Lalis Center for Research and Technology Hellas (CERTH)
Estimated Indicative Person Months:	9
Classification:	Public (PU)
Report Version:	1.0

Abstract

The main vision of the SCoRPiO project is to treat computation significance as a high priority concern during the development of applications. In this document we introduce a programming model that allows the programmer to express her perspective on the contribution of each computation to the quality of the final output. Moreover, the programmer can provide code that heuristically checks the output of non-reliable computations at different levels, in order to hinder the propagation of errors across the computation and limit their effects. The model also arms the programmer with methods to define tasks and their relations, to implement relaxed synchronization, and to provide insights to the system on characteristics of the input data and the quality expectations for the output. In order to demonstrate the use of the programming model we use it to re-implement four kernels from different application domains. Finally, we discuss the next steps concerning the implementation of compiler support for the model, its evaluation and the integration with other layers of the envisioned SCoRPiO software/hardware stack.

Contents

List of Figures

Listings

Chapter 1

Introduction

Energy efficiency is an increasing concern in most kinds of computer systems. Battery life is a first order constraint in mobile systems, and power/cooling costs largely dominate the cost of the equipment in data centers. Moreover, the ”power wall” is one of the major obstacles to overcome in the path towards exascale computing.

Much of the focus in reducing power consumption has been on performance/power trade offs and resource management. While those techniques are effective and can often be applied at system levels below application software, exposing energy considerations to the applications programmer may pave the way to new benefits in the performance/power tradeoff.

Improving energy consumption by controllably reducing the quality of application output is an attractive choice. The SCoRPiO project observes that part of the problem of energy inefficiency is that all computations are treated as equally important, despite the fact that only a subset of these computations may be critical in order to achieve an acceptable quality of service (QoS). A key challenge though is how to identify and tag computations of the program which must be executed reliably from those that are of less importance and can potentially be executed by a less power consuming processing core.

In this document we introduce a programming model that 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 as non-significant. Significant computations are executed reliably. Non-significant computations can be executed in aggressively under-powered cores, without sacrificing performance, however at the expense of a possibility for errors.

A core vision of the SCoRPiO project is to treat computation significance as a high priority concern during the development of applications. The programming model, being the interface between developer and underlying hardware should make the process of declaring the significance of computations as easy and intuitive as possible, without sacrificing performance or power efficiency gains. Given the proliferation of multi/manycore systems and the power/performance benefits these architectures offer, it is a natural option to offer support for the expression of parallelism as well. We opt for a task-based programming model, in which the main granularity of significance characterization is that of a task.

Figure 1.1 illustrates the position of the programming model in the higher-level picture of the vertical hardware and software stack envisioned by SCoRPiO. Beyond being an alternative to the automatic significance detection techniques developed in WP1, the programming model also serves as an intermediary between these techniques, the user provided QoS requirements and domain insight, as well as the implementation, management and execution of computations by the system software and the hardware.

The main functionality offered by the programming model is the following:

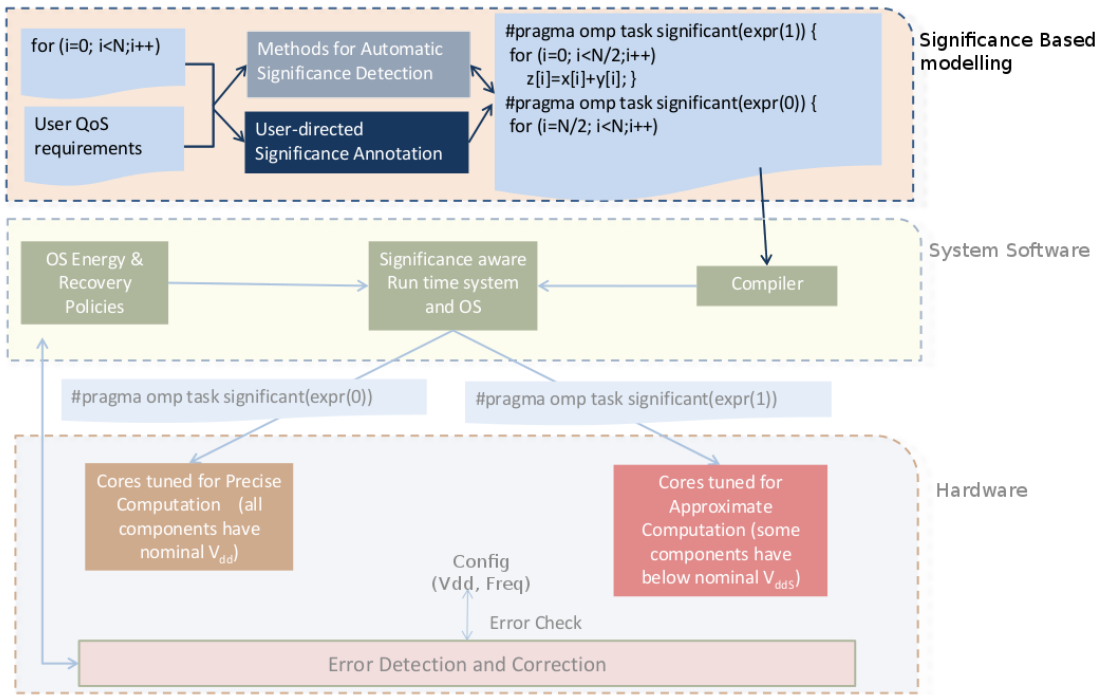

Figure 1.1: Abstract layout of the SCoRPiO Project.

- Expression of the significance of computations, which will be exploited to drive their execution at a just-right degree of reliability, with the purpose of improving performance/power efficiency at a controlled expense on the quality of results.
- Expression of computational flows in the form of tasks, which can potentially execute in parallel.
- Support for multi-level, context-sensitive checking of the results of non-significant (and thus non-reliable) computations, with the purpose of early detection and isolations of errors.
- Support for relaxed synchronization, appropriate for execution on non-reliable substrates.
- Support for information flow between the programmer, the automatic significance characterization methods and the underlying hardware and system software stack.

The rest of the document is organized as follows: In Chapter 2 we provide background information on the characteristics that make computations tolerant to errors, on the characterization of errors and of the quality perception of the end result, as well as on important characteristics of programming models, with a particular focus on parallel ones. Chapter 3 discusses the programming model. We describe the objectives, the main design decisions and define the API. In Chapter 4 we illustrate the use of the proposed programming model by re-implementing 4 popular kernels from different domains. Finally, Chapter 6 concludes the document and outlines the next steps.

Chapter 2

Background

In this chapter we lay the background required in order to introduce the SCoRPiO programming model. In section 2.1 we discuss a categorization of faults and degrees of correctness, as well as general application characteristics that contribute to increased resilience to faults. Section 2.2 provides a brief overview of programming models, with emphasis on parallel programming models. Finally, section 2.3 outlines the architectural template assumed by our programming model.

2.1 Faults, Fault-tolerant Applications & Perspectives of Correctness

Traditionally, a program has been considered to execute properly only when the produced architectural state is correct on a cycle-by-cycle basis compared with a ”golden model” of execution. A looser, though still fairly strict, notion of program correctness commonly adopted by reliability researchers is that the visible memory state after program completion should be correct on a bit-by-bit basis. A growing number of important workloads produce results that have a looser, often qualitative, user-level interpretation. These computations are referred to as soft computations. An example of a soft computation is the processing of human sensory information common in multimedia workloads. Another example is cognitive information processing, an emerging application domain that applies artificial intelligence algorithms for reasoning, inference and learning to commercial workloads. While data corruptions can change the numerical result of soft computations, they often do not change the users perception of the results. Consequently, faults that would otherwise be deemed unacceptable from a numerical standpoint may in fact be tolerable, or even imperceptible, from the users standpoint. SCoRPiO seeks to exploit such error tolerance at the user level, and offer new opportunities for performance / energy optimizations and fault tolerance.

2.1.1 Properties of Soft Computing Applications

In the past, researchers have identified properties that characterize soft computing applications and proposed exploiting them for reduced energy consumption [16, 9, 18] as well as for fault tolerance in ASIC design [22, 13]. The main characteristics of soft computations that make them resilient to errors are redundancy, adaptivity, and tolerance to reduced precision [11].

Redundancy Soft computations that are iterative often contain some degree of redundancy. These redundant computations contribute to the application result, but may not be critical for the final result quality. Programs with redundant computations are more error resilient because the redundancy can mask faults. As presented in [23]

Successive over-relaxation methods (SOR) demonstrate resiliency due to the inherent solution correction property of the SOR algorithm.

Adaptivity The possibility of errors has been taken into account during the design of many soft computing algorithms. This is particularly common in those that operate on noisy or probabilistic data. Such soft computations include code to detect certain forms of error and adapt the computation accordingly. Due to their self-healing nature adaptive codes are naturally error resilient.

Reduced Precision Requirements Soft computations often have precision requirements that are lower than the data types supported by the programming environment/hardware architecture. These soft computations are resilient to errors that modify intermediate or final values within the precision tolerance, as described earlier. For example during the JPEG compression, when computing the figure's frequency representation all computations are floating point. However, the final result is an integer value in the range [0, 255], therefore information regarding the decimal representation is discarded.

2.1.2 Faults Classification

Faults can be classified in the following categories: *non propagated*, *SDC (Silent Data Corruption)*, and *fatal*.

Non Propagated are those faults which did not ultimately manifest as run-time errors or result in erroneous final output. Such faults may, for example, lead to storing erroneous values in registers, however the corrupted registers are either not used during the execution of the application or their respective corrupted values get overwritten before being used in any computation.

*SDC*s do not result to abrupt program termination, however the produced results differ from the would-be output had there no faults manifested during the execution of the application.

Finally, *Fatal* faults corrupt the architectural state of the application to such an extent that the affected application cannot complete its execution and crashes. A special case of such errors are endless loops; during the execution of a program there exists the possibility that faults manifest in such a way that the program is forced to mistakenly execute a loop without ever terminating.

Section 2.1.3 discusses a more flexible categorization of application behavior in the presence of faults.

2.1.3 Alternative Definitions of Program Correctness

Five definitions of program correctness are listed below in decreasing strictness [11]:

1. Architectural state is numerically correct on a per-cycle (or per multiple-cycle) basis.
2. Output state (i.e., computation results visible at program completion or during system calls) is numerically correct.
3. Output state is numerically correct within some tolerance.
4. Output state is qualitatively correct based on higher-level interpretation.
5. Output state is qualitatively correct based on higher-level interpretation within some tolerance.

Definitions I,II are widely used for evaluating program correctness in existing fault tolerance research. The remaining definitions are less strict and more appropriate for soft computations. Definition III can be expressed as $|correct - computed| < \epsilon$ where ϵ is the permitted error. This definition is commonly applied to numerical codes, where floating

point arithmetic alone introduces errors. Definition IV applies to applications in which the quality of result is assessed at a higher level rather than from a purely numerical stand point. Finally definition V is quite similar to definition IV but allows some flexibility even at the interpretation level.

2.2 Programming Models

A programming model is an interface allowing the programmer to describe the algorithm and its properties, ideally separating high-level from low-level concerns. In other words, a model can be thought of as an abstract machine providing certain operations to the programmer. A programming model is of practical interest only if applications expressed in it can be mapped to the underlying architecture and executed with reasonable efficiency.

- *Programmability*: Mapping an algorithm to the underlying architecture, especially if it is a parallel one, is a tedious task. Therefore, the programming model should ideally conceal as many as possible of the following concerns from the programmer:
  - *Decomposition* of an algorithm to parallel jobs (for example threads or tasks).
  - *Mapping* of the parallel jobs to processing units.
  - *Communication* among jobs. Whenever non local data are required a communication must be initiated to transfer the data. The exact mechanism depends heavily on the underlying architecture.
  - *Synchronization* among jobs.
- *Architecture-independence*: The model should be architecturally independent so that programs can execute on multiple/different underlying architectures without having to be modified.
- *Intuitiveness*: The programming model should be easy to understand and to teach, since non-intuitiveness and a steep learning curve may discourage programmers from adopting it.

These requirements for are not easy to fulfill, while many of them are also conflicting. For example abstract models make it easy to build programs but hard to compile to efficient code, whereas low-level models make it hard to express the algorithm but easy to implement it efficiently.

2.2.1 Parallel Programming Model Categories

Most modern parallel programming models can be categorized into one of the following:

- Models in which parallelism is explicitly defined, however decomposition of programs into threads is implicit (and so is mapping, communication, and synchronization). In such models software developers are aware of the parallelism but they do not know at which degree and granularity will this parallelism actually be exploited at run time. OpenMP [4], for example, lets software developers tag computations as parallel, however parallelism is implicitly managed at run-time.
- Models in which parallelism and decomposition must both be made explicit, but mapping, communication, and synchronization are implicit. Such models require decisions to be made about the partitioning of available work into chunks. A popular programming model that falls in this category is OmpSS [7].
- Models in which parallelism decomposition communication and synchronization are explicit however mapping is implicit. Such a programming methodology is supported by POSIX threads [10].
- Models in which all details must be explicitly defined by the programmer, a typical example being OpenCL [25].

2.2.2 Expressing Parallelism

The expressiveness of programming models is usually available to the programmer in one of three ways: a) Compiler Directives, b) Run-time APIs, c) Languages or Language extensions.

Compiler Directives

Directives-based programming models offer an easy way to parallelize applications. Using pragmas the programmer tags computations that can be executed in parallel. Many popular parallel models follow this approach (OpenMP, OpenACC, and OmpSS). The changes required to the original code are usually not overly intrusive and can be performed incrementally. Moreover, as the programmer specifies algorithmic properties rather than architectural mapping details, implementations tend to be easily portable to different systems. In addition, the code is still correct for compilers that do not support the respective programming model, as in this case the directives are simply disregarded. Directives-based programming models protect the programmer for overly delving into implementation details, at the cost of lower-level control over parallelism exploitation and mapping and thus potentially reduced performance with respect to more ”intrusive” implementations.

Run-time APIs

Run-time API-based programming models provide function calls to external libraries offering services such as thread creation, work assignment and mapping. In some cases the libraries offer complete program building blocks. Such models are more intrusive; they usually require major code rewriting. The developer must typically partition the application to parallel threads and take care of synchronization and communication issues. Quite often, the implementation is tailored to a specific architecture and can not be easily ported. Popular run-time API models are POSIX Threads and the Intel Thread Building Blocks (TBB) [19].

Languages or Language Extensions

Languages or language extensions are the most flexible, yet labor-intensive approach. Functionality is offered as either a completely new programming language, or as extensions to an existing one. This approach requires extensive compiler support and major re-engineering of applications. The degree of automation vs. programmer involvement for the management and mapping of code execution depends on the language objectives and design decisions. Such languages are CUDA [17] and EnerJ [21].

2.2.3 Task- & Data-parallelism

Two widely used parallelism styles for multi/many-core computers are data- and task-parallelism. Data parallelism is typically expressed as a computation replicated over multiple cores, being applied to different data on each core. Task parallelism is typically expressed as a collection of jobs with explicit communication and dependencies between them.

Data parallelism

Data parallelism is often expressed as parallelizable loops, with different iterations performing computations on different memory locations. The compiler lowers the source code to equivalent code with calls to the accompanying run-time library. A simple example is given below:

```
int i;
int C[SIZE],B[SIZE],C[SIZE];
for( i = 0 ; i < SIZE ; i++)
    C[i] = A[i] + B[i];
```

Listing 2.1: A simple vector add example

```
int i;
int C[SIZE],B[SIZE],C[SIZE];
#pragma omp parallel for
for( i = 0 ; i < SIZE ; i++)
    C[i] = A[i] + B[i];
```

Listing 2.2: Parallel vector add exploiting data parallelism (OpenMP)

```
int i;
int C[SIZE],B[SIZE],C[SIZE];
#pragma omp parallel for
for( i = 0 ; i < SIZE ; i++)
    C[i] = A[i] + B[i];
```

Listing 2.3: Parallel vector add exploiting data parallelism (OpenMP)

Task Parallelism

Task parallelism is expressed as special code regions called tasks (or sections). The body of a task is usually limited to subroutines and loops. Each of the regions is tagged with **input, output** directives, which identify the input and the output parameters of this task. A task is a single thread of control with well defined side effects. Tasks communicate with each other through the input/output parameters, therefore the communication is explicitly identified by the data flow between the tasks. In principle, a task waits for its inputs, executes, produces its outputs and then terminates. It is common practice, to place explicit synchronization mechanisms after a group of tasks to ensure data consistency. The following example illustrates a task-parallel program:

In this example N tasks are spawned, however they will be executed sequentially due to the dependencies regarding the elements a[i], a[i-1] across successive iterations. The *omp taskwait* at the end of the code snippet is equivalent to a barrier for all threads writing to b.

2.3 Assumptions on the Architecture

Although the SCoRPiO programming model is designed with hardware neutrality in mind, some basic assumptions must be made regarding the underlying architecture. The goal of SCoRPiO is to disrupt the current performance / power consumption trade-off, by allowing certain computations to execute on hardware that is configured at a state that makes it very power efficient, yet potentially unreliable. Therefore, the first and foremost prerequisite for any underlying hardware is that it should consist of both reliable and unreliable cores.

In figure 2.1 we provide an abstract layout of the assumed underlying architecture.

he targeted architectural template comprises a host CPU and a set of many-core accelerators. The host CPU is a conventional processor. The accelerator cores can be dynamically configurable to operate in a *reliable/unreliable* mode:

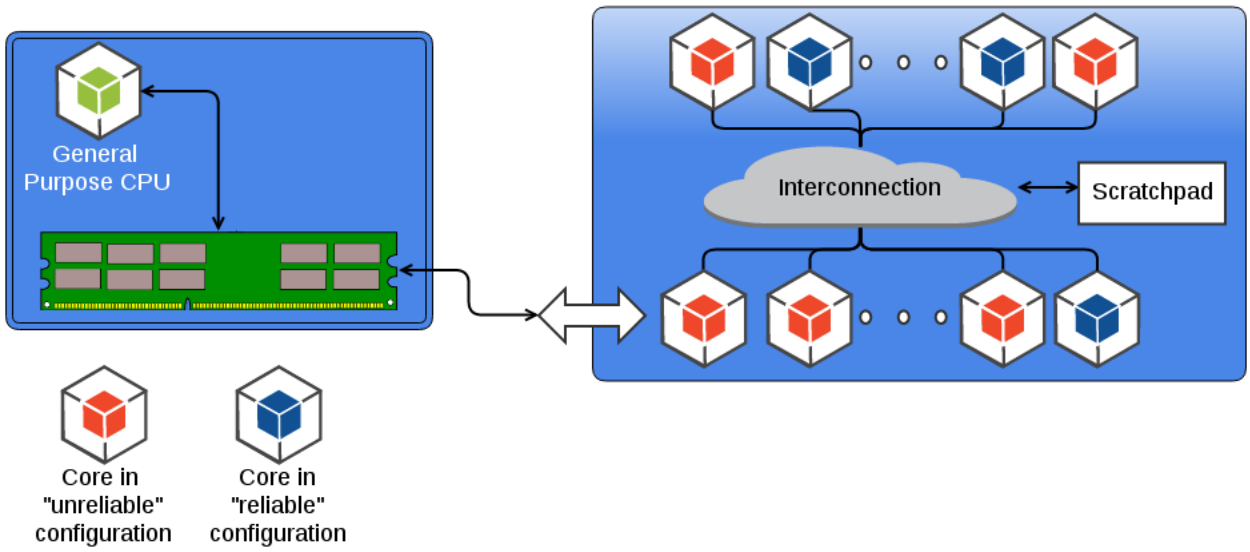

Figure 2.1: Organization of the assumed underlying architecture.

Reliable cores are used to execute computations which the developer has tagged as ”significant”. These should be critical parts of the application that directly affect the output. The programming model has been designed with the assumption that computations tagged as significant should not experience any type of faults (section 2.1.2).

Unreliable cores are used to execute non-significant parts of the application. These should be computations which either do not tremendously affect the output of the application or are inherently fault tolerant. Therefore, the programmer is willing to undertake the calculated risk of the computation producing an erroneous result or crashing, in order to improve the power / performance ratio for the application.

The cores are equipped with mechanisms that allow them to offer a subset of their functionality reliably, even when configured for unreliable operation (for example the ISA may be extended with instructions that are always executed reliably). The core reliability can be reconfigured dynamically by the operating system, mainly by changing the V_{DD} / $Frequency$ ratio. The hardware also offers limited detection functionality for critical errors and the frequency of SDCs. This information is communicated to the operating system and the run-time.

The accelerators do not necessarily share the same address space with the host CPU. They are organized in clusters. The cores within each cluster are not equipped with private caches, however they have access to a shared scratch-pad memory, with latency and bandwidth characteristics similar to those of a cache. Moreover, they can access the scratchpads of other clusters, although with increased latency.

Chapter 3

SCoRPiO Significance-Centric Programming Model

The vision of the SCoRPiO project is to elevate significance characterization as a first class concern in software development, similar to parallelism and other algorithmic properties traditionally being the focus of programmers. The proposed programming model offers programmers the expressiveness and the mechanisms to balance performance, power consumption and the quality of the end-result.

In this chapter we outline the main design objectives of the significance-centric programming model, present its basic concepts and mechanisms, and discuss key design decisions.

3.1 Objectives and Properties

In the following paragraphs we outline the most important design goals for the programming model:

3.1.1 Significance Characterization

The programming model should allow the developers to characterize computations according to their degree of significance in a straightforward and intuitive way. Significant computations will be executed correctly, at the expense of power consumption and/or performance, while computations characterized as non-significant will be executed in a way that may produce incorrect results. The significance of a code region should be expressed either statically, at compile-time, or at run-time, since significance may very well be input-related and/or context-dependent.

The goal is to exploit the underlying hardware which consists of both reliable and unreliable — yet more power efficient — modules in order to improve the performance and power consumption of applications, without sacrificing the quality of results beyond acceptable levels.

3.1.2 Safety - Isolation

The model should be safe: Computations are by-default considered significant and are thus executed correctly, unless the programmer explicitly allows imprecise computations. As a result, a silent data corruption on the output of an early part of the application can affect the execution of the following parts which are dependent on the faulty one. Such a scenario would obviously violate the isolation attribute. Our model should be equipped with mechanisms to

tackle this situation by providing functionality to the developer to detect errors early and even correct the computed values of non-reliable computations. According to this scheme, the propagation of errors across the application, can be controlled/avoided.

Moreover, both the design of the programming model and its implementation should promote — and if possible guarantee — the isolation between significant and non-significant code regions. In other words, errors manifesting on non-significant tasks should not be fatal for significant tasks or the whole application.

Although isolation is a desired property, it is not straightforward to guarantee it. Almost all realistic applications are characterized by data flow among their parts (such as objects, tasks, functions etc.).

3.1.3 Architecture Neutrality

The programming model should assume as little as possible about the underlying architecture, making applications portable with no or at least with reasonable effort to different architectures. It should be noted that our main focus is on functional portability. The power / performance / reliability balance is highly architecture dependent, therefore the power / performance / reliability efficiency is expected to differ when moving to different hardware.

3.1.4 Parallelism Expression

While the key objective of SCoRPiO is to support significance-aware programming and execution, we also wish to be able to express and handle parallelism. This is important to exploit next-generation multi-core architectures, which are quite likely to include unreliable cores or allow cores to operate at power levels that may introduce faults.

3.1.5 Relaxed Synchronization

An important prerequisite for parallel execution of jobs is the existence of synchronization mechanisms to ensure correctness of the final result. Due to the uncertain behavior of jobs executing on unreliable hardware, traditional synchronization mechanisms are overly stringent and are therefore not a feasible option. To this direction we envision mechanisms offering more elastic synchronization, by extending synchronization constructs with timing watchdogs, and more flexible synchronization achievement criteria.

3.1.6 User Friendliness

A programming model must hide most of the intricate implementation and computation mapping details from the developers. In our case, we need to simultaneously support the expression of both significance and parallelism, already putting a lot of burden to the programmer. Therefore, we should make sure that migration of parallel, or even sequential codes to our model does not pose unreasonable overhead to the programmer. Moreover, it would be desirable to allow incremental porting, without widespread and intrusive changes to the original source code.

3.2 General Features

In the next paragraphs we will utilize an abstract figure to demonstrate the basic concepts of the model. Figure 3.1 depicts an initial, single threaded application, which however has regions (dark rectangles) whose execution can be parallelized.

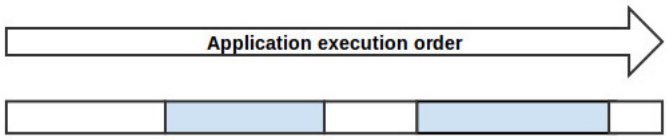

Figure 3.1: A single threaded execution of an abstract application

Task-Based Programming Paradigm

The SCoRPiO programming model adopts a task-based paradigm (section 2.2.3), similar to that adopted by OmpSS[7] and the latest version of OpenMP[4]. Task-based models are quite popular in both the academia and the industry Most developers of HPC software have prior experience in using them. Such models offer a straightforward way to express communication (data transfers) across tasks. Parallelism is expressed by the programmer in the form of independent tasks, however the scheduling of the tasks is not explicitly controlled by the programmer. The developer simply defines dependencies among tasks and the underlying run-time system schedules tasks on available hardware as soon as all their dependencies have been met. An alternative approach is to further alleviate the programmer from the burden of specifying dependencies. In this case, the run-time system automatically tracks dependencies by analyzing the input and output data declarations of tasks[26]. The programming model presented in this document adopts the later approach. This design decision assumes support from the run-time system. In the course of the project we will evaluate this decision under the light of the run-time overhead for dependence tracking, versus the programmability benefits.

Adopting a task-based model is an appealing decision:

- It can express parallelism in an efficient manner (objective 3.1.4).
- It minimizes the burden related to controlling the parallelism.
- It provides isolation between different jobs (tasks). A task is independent from others during its execution, with well defined input and output data flows (objective 3.1.2).
- Each task is a computation with well defined entry and exit points. This facilitates error detection / recovery tests at task boundaries (objective 3.1.2).

The proposed programming model will support, at least initially, single-level parallelism. In other words, a task will not be able to spawn new tasks. Supporting multiple levels of parallelism could lead to complications both at the semantic level (such as non-significant tasks spawning new tasks) and at the technical level (part of the system software — the runtime system — having to implement significant part of its functionality on potentially unreliable hardware).

Pragmas for the Expression of Parallelism and Significance

The proposed programming model supports task creation and significance characterization by annotating the input source code with *#pragma* compiler directives (see section 2.2.2). Pragma-based programming models have the advantage of facilitating non-invasive and progressive code transformations, without requiring a complete code rewrite. Adopting compiler directives to support parallelism in combination with the task based model improves the user friendliness of our proposed model to the user (objective 3.1.6).

Pragmas identify tasks and characterize them as significant or not, thus steering their execution on reliable and non-reliable cores, respectively. Each task *#pragma* construct specifies a function which is equivalent to the task body, along with its data-flow and accompanying significance information. Thus the main granularity of significance characterization is that of a task. However, in the following paragraphs we discuss programming model functionality that allows a finer-level significance characterization, as well implicit assumptions on code constructs that will always be executed reliably, even if they are part of a non-significant task.

The initial version of the programming model supports a binary mode of significance. In other words, a task can be characterized as either significant or non-significant. Significant tasks are always scheduled on reliable cores, whereas insignificant tasks may be mapped for execution on unreliable cores. Unreliable cores are typically expected to operate on lower V_{DD}. This lowers power consumption — without sacrificing performance if frequency does not follow the reduction of V_{DD} — however at the expense of potentially erroneous behavior due to timing errors.

An open question for the course of the project is whether significance characterization using multiple levels of significance would be practical, useful and beneficial. The consortium will have to specify:

- What different levels of significance conceptually mean and how they could be identified at the algorithmic level.
- Whether the hardware can support more than two levels of reliability, for example, by supporting multiple different V_{DD} levels, corresponding to different expected error rates and energy gains.
- Whether the potential gains by using multiple levels of significance would out-weight the inherent additional programming complexity.

At this point we show the decomposition of the abstracted example presented in figure 3.1 into computation chunks. These chunks are categorized as significant (white) and non-significant (dark) tasks, using the programmer intuition, knowledge, and/or domain-expertise. Tasks which have satisfied all their dependencies are ready to be executed, potentially in parallel. Note that after the parallel execution of tasks, implicit synchronization may be necessary.

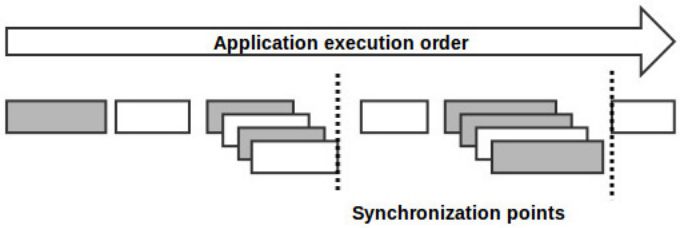

Figure 3.2: Application tasks are created and tagged with significance information

Early Fault Detection to Minimize Fault Propagation

Non-significant tasks are expected to have unpredictable behavior due to the uncertainty of their execution. Errors will manifest in the hardware which might propagate as faults at the software level. These faults can lead to arbitrary error propagation up to the final output of the program, or total program crashes if they are not identified and isolated early on. The programming model provides mechanisms that aim at identifying faults and — if possible — even correcting them (objective 3.1.2). In the event of detected faults the developer is given the opportunity to specify the recovery strategy (such as ignoring the fault, re-executing, or even assigning a default value at the task output).

The SCoRPiO programming model allows the programmer to specify *result-check functions*, which will be executed right after a non-significant task completes. In this function, which is guaranteed (by contract) to be executed reliably, the user can check for possible faults or failures and even provide a default value to the calling program as an acceptable replacement of the result of a failed execution. It should be noted that error checking and correction with result-check functions is complementary to error checking and corrections that could be implemented at the hardware-level which are expected to be developed in the context of WP4. The former have the advantage of close correlation with the algorithmic properties and the semantics of the code, whereas the latter will probably be characterized by lower overhead and higher proximity to the point in time when the error occurred.

We expect that the computational overhead of a result-check function should be minimal, preferably orders of magnitude lower than that of the corresponding task. Otherwise, the overhead introduced by its execution would cancel the performance/power consumption benefits of unreliable execution.

The running example is now modified to include the aforementioned task-level result-check functions which are illustrated as red rectangles in Figure 3.3.

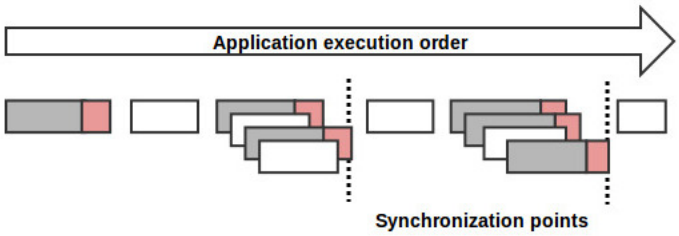

Figure 3.3: Insignificant tasks may execute fault identification and correction functions after their termination

Although result-check functions offer a degree of fault detection and isolation, in some occasions checking a single task's output is not sufficient to decide if the computation is correct. A typical example is image and video processing applications, in which the *acceptance* of the result for a pixel can be judged only in relation to the values of neighboring pixels. To support error detection/correction at a coarser granularity, tasks can be grouped, and result-check functions can also be executed at the end of each group. We extend the running example to allow for group-level result check functions as shown in Figure 3.4. The yellow rectangles are result-check functions executed right after the group's termination. Note that in this specific example multiple groups do not execute in parallel. There are no such restrictions in the programming model, this is done purely to avoid confusing the reader.

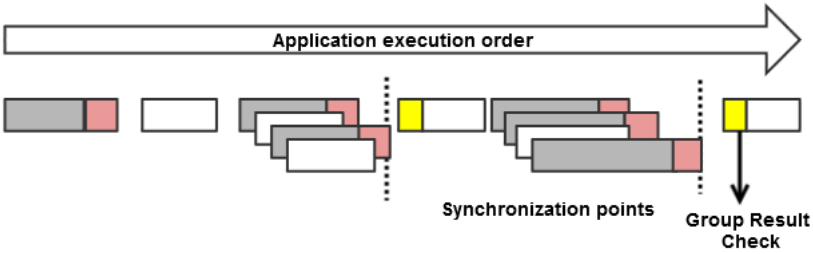

Figure 3.4: Result-checks at the group-of-tasks granularity

Synchronization Based on Data Dependencies and Elastic Barriers

Dependencies between tasks are expressed by the programmer either explicitly, or implicitly. In the latter approach, which is adopted by SCoRPiO, the programmer specifies the **in** (input) and **out** (output) arguments of each task and the system software (compiler and run-time system) exploits this information to identify RaW, WaW and WaR dependencies. Some programming models also allow the use of **inout** (both input and output) arguments. The SCoRPiO programming model does not support **inout** arguments, because they significantly hinder task re-execution in case of failures, as the state of the input arguments may have changed. Each time a new task is created, its in/out arguments are checked against those of existing tasks, in order to identify dependencies. Dependencies are, in turn, used to determine task execution order, as well as the potential for parallel task execution: A task is only available for execution after all its dependencies have been satisfied.

Another type of explicit synchronization in the SCoRPiO programming model is in the form of barriers. Traditional barriers wait for all tasks spawned before the barrier to finish and even commit changes to memory, before allowing the computation to continue past the barrier. SCoRPiO supports named barriers, in order to implement synchronization on just a subset of tasks. However, on an unreliable execution environment, traditional barriers may prove impractical. While executing under the presence of faults, it is possible that a number of non-significant tasks will never finish. Take for example the scenario of a fault resulting to a task being trapped in an infinite loop. If that task was expected

to participate to a traditional barrier or a inter-task dependency, the computation would not terminate.

In the context of the SCoRPiO programming model we support a more elastic explicit synchronization model for barriers (objective 3.1.5). Barrier-type synchronization constructs can wait for just a number or a percentage of the non-significant tasks which participate to the barrier. Alternatively, barriers can be equipped with watchdog timers which terminate the wait after the specified time frame has elapsed. In all cases when the synchronization was not successful — in a traditional perspective — the run-time system terminates all tasks that did not reach the barrier (if they have not already crashed) and resumes execution of the following code.

Given that we do not support elastic dependence checking, we highly encourage the developer to use elastic barriers. Whenever a non-significant task fails, all its successors will never have their dependencies satisfied and thus will never be scheduled for execution. The use of an elastic barrier in this case safeguards the application from the otherwise unavoidable deadlock: it ensures that the task group will terminate, either successfully, or with an error code, thus allowing the application to continue or try to recover.

The running example is now finalized as shown in Figure 3.5 and serves as an example of relaxed synchronization. Note that, due to a timing constraint which was not met, the last non-significant task was terminated by the run-time.

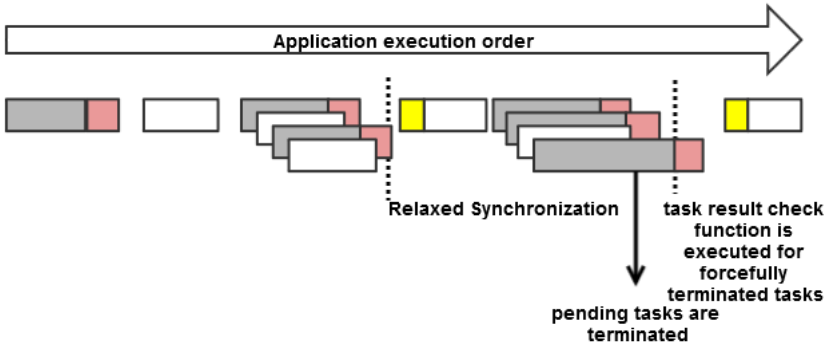

Figure 3.5: A case of relaxed synchronization which results to termination of late tasks

Significance of Data

The main point of interest in a significance-aware computing platform concerns the quality of the output results. After all, the role of computations – significant or not – is to produce output data that are acceptable to the end user even under the extended definition of correctness of section 2.1.3.

In SCoRPiO we have selected to only characterize explicitly the significance of computational tasks, and not that of data. In other words, there is no explicit tagging of significance of data structures in the code. The rationale is that early fault detection and correction using result-checking functions (as explained earlier in this section) precludes the flow of unacceptably erroneous data between tasks. If data are deemed unacceptable by a result-checking function, one option may be to substitute the erroneous value by a default value so that the program can continue. This approach greatly reduces the amount of effort that the programmer has to expend to express significance in the code.

3.3 Syntax

In this section we define the API of the proposed programming model which realizes concepts and ideas expressed in section 3.2.

3.3.1 Task Definition and Significance Characterization

```
#pragma omp task  [in(...)] [out(...)]  [label(...)] [significant(expr(...)|ratio(...))]
  [tasktolerance([taskcheck()], [redo(...)])]
```

Listing 3.1: #pragma omp task

Tasks are characterized either as 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 clauses supported in this #pragma are the following:

- label(...)
- significant(expr || ratio(...))
- tasktolerance([taskcheck(...)],[redo(...)])]
- (in||out)(array[start:end][low value, high value])

label(...) allows the developer to group tasks and name the group using a common identifier. This identifier can be used as a handle to specify parameters of group behavior. Moreover the *label* clause is necessary for relaxed synchronization methods. When defining a synchronization method the user supplies the respective task group *label* to instruct the run-time to wait for the specified group to terminate.

The clause *(significant(expr(...)||ratio(...)))* lets the developer specify the significance of the task. This can be achieved in two ways: using the *expr* or the *ratio* option. The clause *expr* encloses an expression which evaluates to a boolean result. When a task is about to be spawned and the respective *expr()* evaluates to true, the task will be characterized as significant, otherwise it will be tagged as non-significant. The *ratio* option instead, specifies the percentage of the total tasks in the specific task group that will be created as significant. For example, a *ratio(0.2f)* means that 20% of the total spawned tasks in the specific task group are created as significant. In this case, the compiler and/or runtime is responsible for maintaining this ratio. It is easy to observe that the *expr()* method provides higher control over task characterization, however at the expense of increased programmer involvement.

The clause *tasktolerance([taskcheck()], [redo(...)])* allows the programmer to define a function (*taskcheck*) which will be executed reliably after each individual task completes or crashes. The main duties of the result-check function are :

- To perform a quick estimation of the correctness of the task output, in order to detect and isolate errors early.
- To potentially assign default values to the output of the task or request a re-execution of the task, should the task execution have failed.
- To potentially change the task characterization and/or system configuration across task re-executions, should the original task execution have failed.

The result-check function has implicitly access to the input and output data of the corresponding task. Additional data can be passed as extra arguments to the result-check function. These arguments can be typically used to describe the environment in which a task was executed. It is highly recommended for the sake of performance and energy efficiency that the complexity of the result-check function is kept minimal compared to the complexity of the task. The aforementioned function return type is int and permitted return values are (a) **SGNF_SUCCESS** & (b) **SGNF_REDO.**

Should a task result-check function return SGNF_REDO, the task is re-executed by the run-time system until it reaches the permitted maximum $number$ of re-executions specified by the *redo(number)* option. It should be noted that task re-execution assumes that the underlying system layers are able to restore the original state of the execution, before the task was started.

Both clauses are optional and override the default behavior which treats all tasks as significant and allows no re-executions. Finally, if a result-check function is not present, the assumed return value is SGNF_SUCCESS[1]. Note that task-level result-check functions will only be executed for non-significant tasks.

Figure 3.6 depicts the execution life of tasks. The run-time maps the task to a core, according to its significance, and the tasks executes. After task termination, if the task is non-significant and a result-check function is defined then the function is executed. The return value of the result-check function is used by the run-time system. Depending on the returned value the task may be re-executed, potentially with a different significance characterization.

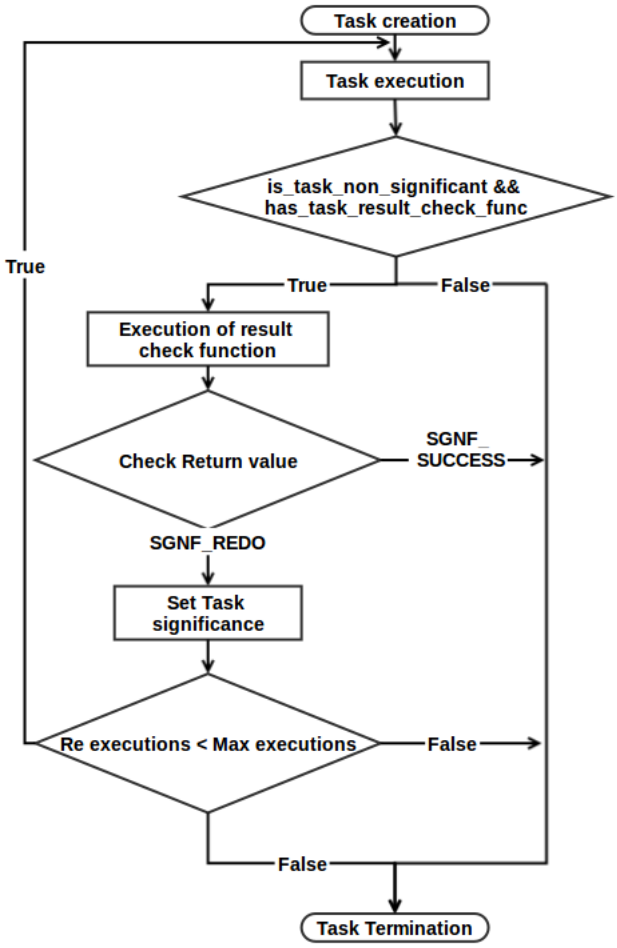

Figure 3.6: The execution life of a task.

The SCoRPiO programming model allows the specification of the input and output arguments through the *in* and *out* clauses, respectively. The information from these clauses is also used implicitly for task dependence analysis.

The in/out clauses of **#pragma omp task** are extended to specify information about the range of valid values of the in/out variables. The clauses accept arguments of the form **array_name[first_element, last_element] @range[low, high] (@absolute||@relative)(value)**, which are interpreted as follows:

- *first_element* and *last_element* refer to the range of indices being accessed by the given task.

[1] We foresee that the automatic significance characterization methods developed in WP1 will be able to contribute automatically generated result-check functions, either for checking or for default output values assignment in case of failures. In this case, the default behavior in the absence of programmer-defined result-check functions will be implementation dependent.

- *low* and *high* refer to the bounds of the range the value of array elements may lie within.
- *@absolute*/*@relative* indicates the maximum absolute/relative allowed error with respect to the result of a reliable execution.

The first two may be used in in() and out() clauses, however the @absolute and @relative tags are only valid for out() clauses. These extensions were designed to enable and assist the automatic differentiation analysis process[2].

Finer Granularity Significance Characterization within Tasks

Defining significance at the granularity of a task occasionally can be too coarse. Therefore the programming model offers one pragma which can be used to define the significance of a block of statements within a non-significant task[3].

```
#pragma omp significant
```

Listing 3.2: #pragma omp task

We expect that certain parts of the code, such as address calculations and computations that affect the control flow, will have to be executed reliably, irrespective of the characterization of the task. These computations will be automatically identified by the compiler, using backward code-slicing [28] from loads / stores and conditional control flow instructions[4].

3.3.2 Synchronization

```
#pragma omp taskwait [on(...)] [label(...)] [time(...), ratio(...), all]
[grouptolerance([groupcheck()]), [redo(...)]]
```

Listing 3.3: #pragma omp taskwait

Many task-based programming models support explicit barrier-type synchronization. For example, the taskwait pragma in Ompss instructs the program to wait for the completion of all tasks defined up to that point. We extend the *taskwait* pragma with extra clauses to support elastic synchronization for a group of tasks, as well as result-checking at the task group level.

The extended synchronization pragma supports the following clauses:

- label(...)
- [time(...),ratio(...),all]
- [grouptolerance ([groupcheck()]),[redo(...)]]

label() takes a task id as an argument, which names the group of tasks that will be synchronized.

[2]The tool estimations can be more accurate when this information is provided.

[3]If the hardware allows different levels of reliability we will extend this clauses, allowing the user to determine the level of reliability. e.g: #pragma omp significant(int level).

[4]Significance control at the instruction-level will be implemented using instruction set architecture (ISA) extensions developed in WP4. The ISA extensions will offer — at the expense of power and performance efficiency — reliable counterparts for a subset of the architecture's instructions, even when the core is configured as non-reliable.

The relaxed synchronization can be expressed in three different ways: *all*, waits for all tasks to finish. This is equivalent to the traditional, strict barrier synchronization semantics, and is the default behavior of our model. *ratio(prcntg)*, allows the user to specify a percentage of non-significant tasks that must finish before resuming the execution after the barrier (however still all significant tasks will have to finish). Finally, *time(drtn)*, allows the user to define a time watchdog. Essentially, after the specified duration, the execution resumes. Whenever the elastic synchronization is considered as achieved, the run-time forcefully terminates the remaining non-significant tasks. The terminated tasks execute the user-defined task result-check function. The run-time does not allow any re-execution of the terminated tasks and will subsequently tag them as crashed. More than one options can be specified simultaneously. In such a case when any of the conditions is satisfied and all the significant tasks have terminated their execution, the synchronization is achieved and execution resumes after the barrier.

Similarly to task-level result-checking, we allow the programmer to specify task group-level result-checking functions. A task group-level result-check function is evaluated at *omp taskwait* points, after the execution of the corresponding task group.

The result-check function return type is int; legal values are (a) *SGNF_SUCCESS* (b) *SGNF_FAILURE* (c) *SGNF_REDO*. Should the result-check function return SGNF_REDO, the computation corresponding to the entire task group will be re-executed, until the maximum re-execution limit is reached. Should the maximum number of re-executions be reached, the run-time forces a return value of SGNF_FAILURE. The programmer has no direct access to the returned value. Getting the value of a group result-check function is achieved by calling the appropriate API function (section 3.3.3). Upon returning SGNF_SUCCESS or SGNF_FAILURE the execution of the host thread resumes normally after the taskwait barrier. Note that SGNF_REDO will never be returned to the programmer since the run-time will re-execute the group until the result-check returns SGNF_SUCCESS or the max re-executions are reached. In such a case the run-time returns SGNF_FAILURE.

3.3.3 Run-time API

Compilers supporting the SCoRPiO programming model are expected to automatically define a **__SGNF_SUPPORT__** macro. The status of this macro should be checked by programmers to allow conditional code inclusion at the preprocessing level, especially when the run-time API calls introduced in this section are used.

In *listing 3.4* we introduce the eight run-time calls which are part of the API and the programmer can use directly in her code. The Run-time calls with the prefix *sgnf_task* may only be used inside a task result-check function. The rest of the API calls may be used in any part of the code which is executed by the host CPU, as well as the group result-check functions:

```
1 int  sgnf_task_status_get()
2 void sgnf_task_significant_set(int signicicance)
3 void sgnf_task_reexecutions_max_set(unsigned int number)
4 int  sgnf_task_reexecutions_get()
5 void sgnf_group_reexecutions_max_set(const char* lbl, unsigned int number)
6 int  sgnf_group_reexecutions_get(const char *lbl)
7 int  sgnf_group_return_value(const char* lbl)
8 int  sgnf_group_reexecutions_get(const char* lbl)
```

Listing 3.4: Prototypes of the programmer-accessible run-time functions.

int sgnf_task_status_get()
sgnf_task_status_get queries the run-time system to find out whether the task has crashed during its latest execution. This could either mean that the hardware has detected an unrecoverable fault and reported it to the run-time or that the task failed to complete in a timely manner with respect to user-specified time constraints. *sgnf_task_status_get* returns 0 if no crash has been detected, 1 otherwise.

void sgnf_task_significant_set(int significance)
sgnf_task_significant_set is used to set the significance mode of a given task. In the event that a task has failed, the end developer may feel that it is necessary to modify the significance characterization of a task before it is re-executed. It can only be invoked in the context of a task-level result-check function.

void sgnf_task_executions_max_set(unsigned int number)
sgnf_task_executions_max_set is used to set the maximum *number* of times a task may be re-executed. It can only be invoked in the context of a task-level result-check function.

int sgnf_task_reexecutions_get()
sgnf_task_reexecutions_get is used to find out the current number of re-executions of this task.

The next four run-time calls can be used to manage query information about the execution of a specified task group. They can be invoked from within the group result-check function, or in the code after the respective group synchronization barrier (task wait pragma).

void sgnf_group_reexecutions_max_set(const char *lbl, unsigned int number)
sgnf_group_reexecutions_max_set is used to set the maximum *number* of times a task group specified by *lbl* may be re-executed.

int sgnf_group_return_value(const char* lbl)
sgnf_group_return_value can be used to retrieve the return value of the last executed result-check function for the task group specified by *lbl*.

int sgnf_group_reexecutions_get(const char* lbl)
sgnf_group_reexecutions_get can be used to retrieve the number of times the task group identified by *lbl* has been re-executed.

int sgnf_group_status_get(const char* lbl)
sgnf_group_status_get returns the execution status for a task group which is identified by *lbl*. A group is considered to have crashed if either of the following is true:

1. a significant task has crashed, or
2. the number of insignificant tasks which completed successfully does not meet user specified constraints. These constraints are set using the *ratio()* and *all* clauses in the *taskwait* **#pragma**. For information on how to specify such constraints please refer to section 3.3.2.

Chapter 4

Examples

In this chapter we demonstrate the use of the proposed programming model by applying it to a representative set of simple kernels. These kernels correspond to different application domains. We need to point out that our purpose is to gradually demonstrate, in consecutive examples, the use of the functionality offered by the model. Therefore, the implementation of the examples — especially of the initial ones — is not optimal or even fault-torelant[1].

4.1 DCT Compression Kernel

A Discrete Cosine Transform (DCT) transforms a finite sequence of data points to an equivalent sum of cosine functions oscillating at different frequencies. The DCT algorithm is important due to its numerous applications in science and engineering, from lossy compression of audio (e.g. MP3) and images (e.g. JPEG), to spectral methods for the numerical solution of partial differential equations. In the case of image compression, for example, high-frequency components can be discarded without significant loss of image quality, as the human eye is less sensitive to those frequencies.

In Listing 4.1 we demonstrate the application of a DCT filter for jpeg [24] image compression. The image is stored in the *pic* array and the coefficients of the DCT filter are stored in the *COS* array. The image is divided into 64X64 blocks, each block containing 8X8 pixels. The outermost two for loops (lines 3,4) iterate through the blocks of the image. The next two loops (lines 5,6) iterate through the DCT coefficients. Finally the two innermost for loops (lines 8,9) iterate through each 8x8 block. In line 10 we shift the adjusted value of each pixel, from the range [0,255] to [-128,127], and multiply the pixel with the corresponding DCT coefficients.

Not all DCT coefficients contribute the same to the quality of the output. More precisely, moving from the top left corner towards the bottom right corner of the dct-coefficient matrix the contribution to the final quality decreases[27] (figure 4.1).

In Listing 4.2 we have applied our model to the DCT kernel. The reader can observe that the changes are minimal and non-intrusive. The source code was modified so that a separate task is used for the computation of each frequency coefficient. Tasks that compute the frequency coefficients included in top $SIGNIFICANT\_Y, SIGNIFICANT\_X$ box are characterized as significant whereas the remaining tasks are non-significant. Therefore the latter can be scheduled for execution on unreliable cores. All task are conceptually organized in a group, named *dct*. Each task is accompanied by its in/out data (variable / array names and index ranges for arrays). At the end of the code a strict barrier waits for the termination of all tasks in the *dct* task group.

[1] For example, the abnormal termination of a single task in DCT or Jacobi would result to deadlock, because we purposely postpone demonstrating the use of relaxed barriers after the first two examples.

```
void DCT(double pic[][512], double dct[][512], double COS[][8], double C[]) {
  int r, c, i, j, x, y;
    for (r = 0; r < 64; r++)
      for (c = 0; c < 64; c++)
       for (i = 0; i < 8; i++)
         for (j = 0; j < 8; j++) {
          double sum = 0;
          for (x = 0; x < 8; x++)
            for (y = 0; y < 8; y++)
             sum += (pic[ r*8 + x][c*8 + y] - 128) * COS[x][i] * COS[y][j];
          sum *= C[i] * C[j] * 0.25;
          dct[ r*8 + i][c*8 + j] = sum;
        }
}
```

Listing 4.1: The dct filter applied on all blocks of a 512x512 image

```
void dct_task(int r, int c, int i, int j, double pic[][512],
      double dct[][512], double COS[][8], double C[])
{
    int x, y;
   for (x = 0; x < 8; x++)
      for (y = 0; y < 8; y++)
         sum += (pic[r * 8 + x][c * 8 + y] - 128) * COS[x][i] * COS[y][j];
   sum *= C[i] * C[j] * 0.25;
   dct[r * 8 + i][c * 8 + j] = sum;
}

void DCT(double pic[][512], double dct[][512], double COS[][8], double C[]) {

   int r, c, i, j;
   for (r = 0; r < 64; r++)
      for (c = 0; c < 64; c++)
         for (i = 0; i < 8; i++)
            for (j = 0; j < 8; j++) {
               double sum = 0;
               #pragma omp task label(dct) out(dct[r*8+i:r*8+i][c*8+j:c*8+j]) \
                  in(pic[r*8:r*8+7][c*8:c*8+7]) in (COS[in(C[i:i]) in(C[j:j]) \
                  significant(expr(i<=SIGNIFICANT_Y && j<=SIGNIFICANT_X)) in(COS[0:7][i:i]) \
                  in(COS[0:7][j:j]) in(C[0:7])
               dct_task(r, c, i, j, pic, dct, COS, C);
            }
#pragma omp taskwait all label(dct)
}
```

Listing 4.2: The modified DCT filter applied on all blocks of the image

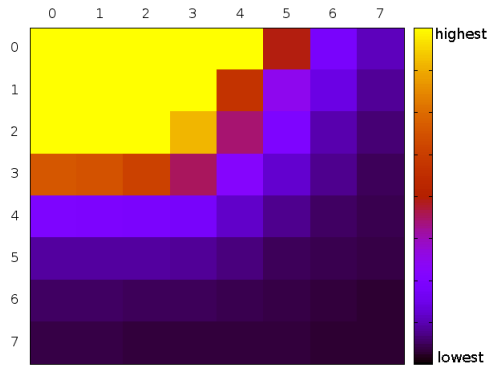

Figure 4.1: A heatmap illustrating the energy contained in the frequency coefficients after applying the discrete cosine transform on an 8x8 block of the ”Lena” picture. The energy is presented in logarithmic scale. The reader can observe that the top-left coefficients carry significantly higher energy than the lower-right.

4.2 The Jacobi Iterative Method

The Jacobi method is an iterative linear algebra algorithm for determining the solutions of a system of linear equations (equation 4.1).

$$Ax = b \tag{4.1}$$

The method guarantees convergence if the matrix A is strictly or irreducibly diagonally dominant, namely the absolute value of each element on the diagonal is larger than the sum of the absolute values of all non-diagonal elements in its row and column (equation 4.2).

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{4.2}$$

The system can be rewritten as the sum of two matrices (equation 4.3): D contains all the diagonal elements of matrix A and R contains the remaining elements.

$$A = D + R \tag{4.3}$$

Initially the method guesses a solution (x_0). The algorithm iterative refines the solution, eventually converging to the actual solution (equation 4.4) .

$$x_{k+1} = D^{-1}(b - Rx_k) \tag{4.4}$$

Listing 4.3 illustrates the implementation of the jacobi method. The algorithm is mostly sequential, with exploitable parallelism only in matrix-vector operations (lines 10-13). Given that Jacobi is a generic method used as a module in solving a multitude of problems, significance characterization should take into account the context in which the method is used. In any case, the results of a random fault injection campaign we applied on the jacobi kernel indicate that there is a relation between the timing of the fault and the final output: the earlier a fault is injected, the higher the probability the fault will not significantly affect the output.

In Listing 4.4 we use our programming model to characterize the code based on the aforementioned observation.

```
int jacobi(double *A, double *x, double *b, int size, double itol, unsigned int *iters)
{
   int iter, i, j;
   double dif, s, t, s1, s2;
   dif = itol + 1.0;
   for ( iter=0; iter<*iters && dif > itol; ++iter ) {
      dif = 0;
      for ( i=0; i<size; ++i ) {
         s = 0;
         for ( j=0; j<size; ++j ) {
            if(i!=j)
               s+= A[i*size+j] * x[j];
         }
         s = ( (double) b[i] - s ) / (double)A[i*size+i];
         t = fabs(x[i] - s );
         x[i] = s;
         if ( t>dif )
            dif = t;
      }
   }
   *iters = iter;
   return dif>itol;
}
```

Listing 4.3: The Jacobi iterative kernel

```
void jacobi_task(double A[], double x[], int i,  int size, int start, int end, double *out)
{
   int j;
   for ( j=start; j<end; ++j )
      (*out) += A[i*size+j] * x[j];
}

int jacobi(double *A, double *x, double *b, int size, double itol, unsigned int *iters)
{
   int iter, i, j;
   double dif, t, double s[2];
   dif = itol + 1.0;
   for ( iter=0; iter<*iters && dif > itol; ++iter ) {
      dif = 0;
      for ( i=0; i<size; ++i ) {
         s[0] = s[1] = 0;
         #pragma omp task out(s[0:0]) in(A[i*size:i*size+i]) in(x[0:i]) \
            significant(expr(iter<THRESHOLD)) label(jacobi)
         jacobi_task(A, x, size,i, 0, i, &s[0]);
         #pragma omp task out(s[1:1]) in(A[i*size+i+1:i*size+size]) in(x[i+1:size]) \
            significant(expr(iter<THRESHOLD)) label(jacobi)
         jacobi_task(A, x, size,i, i+1, size, &s[1]);
         #pragma omp taskwait label(jacobi)
         s = s[0] + s[1];
         s = ( (double) b[i] - s ) / (double)A[i*size+i];
         t = fabs(x[i] - s );
         x[i] = s;
         if ( t>dif ) {
            dif = t;
         }
      }
   }
   *iters = iter;
   return dif>itol;
}
```

Listing 4.4: The Jacobi iterative kernel extended with significance characterization

Notice that we require lines 24-29 to be computed in a reliable manner. An error affecting these computations may lead to infinite loops. This would result to application failure, as well as to an unnecessary increase in execution time and power consumption.

4.3 Genetic Algorithms

The concept of the ”survival of the fittest”, which originates in evolution theory, is used by genetic algorithms. These algorithms are designed to compute approximate solutions for optimization and search problems. A representative random set of candidate solutions, referred to as individuals, is allowed to iteratively evolve towards a better solution. The algorithm stops when at least one individual is considered an acceptable solution. These algorithms are widely used in computational biology, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics, pharmacometrics etc.

Genetic algorithms (GAs) typically start by randomly selecting individuals as candidate solutions. Each individual is characterized by a number of properties (genes). Each iteration of the algorithm represents a different generation. In each generation the “fitness” of all candidates is calculated. The fitness calculation function is defined according to the final objective of the optimization problem. A subset of the entire generation is selected, containing the fittest candidates of that generation. Individuals residing in the fittests subset are subsequently mated, exchanging “genes”; the mating process is tightly coupled with the optimization problem. For example when calculating the solutions of a complex polynomial, candidates can swap a subset of their estimated solutions. Afterwards, random individuals are being selected for a mutation process. Typically this is performed by randomly changing a number of “genes” in each mutation subject. The process continues iteratively until convergence is acquired or the maximum number of iterations is reached.

Listing 4.5 outlines the abstract implementation of a genetic algorithm. Typically, a random population is created (line 3) upon which a fitness function is executed to characterize each individual (line 6). For each iteration the first step is to calculate the fitness of the population (line 6). Then the population is sorted according to its fitness (line 7). Afterwards a subset of the total population is selected based on fitness criteria (line 10). Random individuals of this subset undergo a mating phase thus creating new individuals (line 11). Additionally, every individual can mutate, creating a slightly different specimen (line 12). The process continues until it converges, i.e. the fitness function determines that a suitable solution has been reached (line 8).

Genetic algorithms are inherently fault tolerant. A single entity in the population is not significant; even if an extreme mutation mistakenly makes it past the fitness filter, it will be rejected in subsequent evolutions. This is what makes GAs fit for unreliable execution environments.

In Listing 4.6 the abstract version of a simple genetic algorithm is ported to our programming model. Multiple tasks are instantiated to perform the core computations of the GA, which are mainly the fitness estimation for individuals in the population and, secondarily mating and mutations. As the number of tasks increases, more parallelism is exploitable whereas each task inherently becomes less significant.

Note that we can define only a subset of the total tasks, 20% in the specific example, as significant (lines 7-9 and 18-20) using the *ratio* clause. Lines 12 and 23 demonstrate the use of relaxed barrier synchronization. We specify that the *taskwait* barrier should wait for just a percentage of the non-significant tasks of the *GA* task group. In the specific example we opt to wait for 0% of the non-significant tasks. This may seem counter-intuitive: although the non-significant tasks are created (and some of them are potentially executed), the programmer specifies that the computation should continue as soon as all significant tasks have been executed and the non-significant tasks which have not finished (or even started) their execution should be killed. Therefore the results from the non-significant tasks do not seem to be necessary for the correctness of the computation. This use pattern however, corresponds to the scenario in which the programmer tries to achieve the best possible quality of results, given external time / power

```
Genetic_Algorithm()
{
    create_initial_population(population);
    for (i = 0; i < NUM_ITERATIONS; i++)
    {
        calc_fitness(population);
        sort(population);
        if (population[0].fitness == CONVERGED_VALUE)
            break;
        selection(population);
        mate(population, new_population);
        mutate(new_population)
        population = new_population;
    }
}
```

Listing 4.5: Pseudo-code of a genetic algorithm

```
Genetic_Algorithm()
{
    create_initial_population(population);
    for (i = 0; i < NUM_ITERATIONS; i++)
     {
       for(j = 0 ; j < NUM_TASKS; j++){
          #pragma omp task in(population[j*TASK_SIZE:(j+1)*TASK_SIZE-1]) \
              out(population[j*TASK_SIZE:(j+1)*TASK_SIZE-1]) \
              significant(ratio(0.2) label(GA))
          calc_fitness(population, j);
       }
       #pragma omp taskwait label(GA) ratio(0.0)
       sort(population);
       if (population[0].fitness == CONVERGED_VALUE)
         break;
       selection(population);
       for(j = 0 ; j < NUM_TASKS; j++){
          #pragma omp task in(population[j*TASK_SIZE:(j+1)*TASK_SIZE-1]) \
              out(population[j*TASK_SIZE:(j+1)*TASK_SIZE-1]) \
              significant(ratio(0.2) label(GA))
          mate_mutate(population, new_population, j);
       }
       #pragma omp taskwait label(GA) ratio(0.0)
       population = new_population;
     }
}
```

Listing 4.6: Pseudo-code of a genetic algorithm using the SCoRPiO programming model

```
double MonteCarlo_integrate(int Num_samples){
   int under_curve = 0;
   int i;
   double a,x,y;
   int executed = 0;
   for (i=0; i<Num_samples; ++i) {
      x= myrand();
      y= myrand();
      a= x*x + y*y;
      if ( a <= 1.0)
         under_curve ++;
   }
   return ((double) under_curve / Num_samples) * 4.0;
}
```

Listing 4.7: The Monte Carlo estimation of the value of π

constraints. Therefore, the computation will proceed until all absolutely necessary (significant) tasks have finished, however any non-significant tasks that will manage to finish in the same period are expected to contribute to the quality of results.

4.4 Monte Carlo Methods

Monte Carlo (MC) methods are a broad class of computational algorithms which use sampling techniques to approximate solutions of complex problems. Solving complex scientific problems using rigorous mathematical models and methods may be impossible or highly impractical due to the computational complexity of the problem. MC approaches can often be used to efficiently approximate a solution in such cases. Good candidates for solution with MC methods are physics applications, such as simulating the motion of fluids, as well as applications involving economics to calculate business related risks.

MC methods begin by defining a possible input domain. Afterwards this domain is randomly sampled using a predefined probability distribution. For each sampled input, deterministic methods are used to compute a result. The method finally concludes by aggregating the results.

We discuss a rather simplistic use case of the MC methodology. More specifically, we estimate the value of π by randomly selecting *N* points within a unit square and evaluating whether they reside in the unity circle. Listing 4.7 outlines the original code.

This program randomly picks points inside the square. It then proceeds to check whether the point is inside the unit circle. A point (x, y) is considered to lie within the boundaries of the unit circle when $x^2 + y^2 < 1$. During this phase, the application keeps track of how many points it has picked so far (Num_samples) and how many of those points reside inside the circle (under_curve).

π is then approximated as follows:

$$\pi = 4 * \frac{under\_curve}{Num\_samples}$$

Listing 4.8 illustrates the implementation of the MC *pi* estimation using the SCoRPiO programming model. Each task executes the *montecarlo_task()* function to inspect a set of N sample points.

Line 24 of Listing 4.8 defines a *task tolerance function* as well as a number of maximum re-executions. The code of the task-level result-check function is presented in lines 39-44 of listing 4.8.

```
void montecarlo_task(unsigned int under_curve_reduce[], int i, int Num_samples, int N)
{
   int i, count;
   double a,x,y;
   for (count=0; count<Num_samples/N; count++){
      x= myrand();
      y= myrand();
      a= x*x + y*y;
      if ( a <= 1.0)
#pragma omp significant
         under_curve_reduce[i] ++;
   }
}

double MonteCarlo_integrate(int Num_samples){
   unsigned int under_curve = 0, i, ret, N, wrong;
   unsigned int *under_curve_reduce;
   N = NUM_TASKS;
   if ( Num_samples<NUM_TASKS )
      N = Num_samples;
   under_curve_reduce = (unsigned int*) calloc(N, sizeof(unsigned int));
   for (i=0; i<N; ++i) {
      #pragma omp task out(under_curve_reduce[i:i]) significant(ratio(0.2f)) \
          label(pi) tasktolerance(pi_task_check(Num_samples/N),redo(2))
      montecarlo_task(under_curve_reduce, i, Num_samples, N);
   }
   #pragma omp taskwait label(pi) ratio(0.5) grouptolerance(pi_group_check(N, \
      Num_samples , &wrong, &under_curve))
   ret = sgnf_group_return_value(pi);
   if ( ret != SGNF_SUCCESS )
      exit 1;
   else {
      Num_samples -= wrong*(Num_samples/N);
      free(under_curve_reduce);
      return ((double) under_curve / Num_samples) * 4.0;
   }
}

int pi_task_check(unsigned int under_curve,Num_samples){
  double res = 4.0 * ((double)under_curve/Num_samples)
  if(res >= LOWER_BOUND && res <= UPPER_BOUND)
    return SGNF_SUCCESS;
   return SGNF_REDO;
}

int pi_group_check ( unsigned int *under_curve_reduce,  unsigned int  N,
     unsigned int Num_samples,
     unsigned int *wrong,  unsigned int *under_curve)
{
   int ret, i;
   *wrong = 0 ;
   *under_curve = 0;
   for ( i=0; i<N; ++i)
      if ( under_curve_reduce[i] > Num_samples/N)
         (*wrong)++;
      else
         (*under_curve) += under_curve_reduce[i];
   ret = (*wrong)>N/2;
   if ( ret )
      return SGNF_FAILURE;
   return SGNF_SUCCESS;
}
```

Listing 4.8: The monte carlo estimation of π using the significance task based model

The rationale behind the task result-check function is that each task separately estimates the π, however with less accuracy compared to the entire application, since more samples lead to a more accurate result. The $LOWER\_BOUND$, $UPPER\_BOUND$ are defined by the user and express a possible error margin.

In line 10 of Listing 4.8 we have identified that the incremental operation is significant (even within a non-significant task) and we instruct that it should always be executed in a reliable manner.

Lines 27-28 illustrate the use of a relaxed barrier, using a ratio constraint: the barrier will be achieved after a percentage (in this case 50%) of the non-significant tasks issued by the program have successfully terminated. The remaining non-significant tasks will then be terminated, even if they have not failed. Seen from a different perspective, the programmer in this case has over-created tasks by a factor of 2, expecting that some of them will fail. In the same directive we also specify a group-of-tasks result-check function to be executed after the termination of the task group. The code for the function *pi_group_check()* can be seen in lines 46-62 of Listing 4.8. The function has a dual functionality: First, it checks whether the output of each task seems rational or not. The output is obviously wrong if the task has identified that more points reside in the unit circle than the points (N) it has examined. Results that pass this check, are then used in a sum reduction, to calculate the total number of points identified to be in the circle by all tasks. If number of tasks that returned erroneous results is more than half the tasks, the result of the computation performed by the task group can not be trusted and the check function returns *SGNF_FAILURE*. Otherwise, the number of tasks that returned erroneous results is implicitly returned (via the *wrong* argument), so that the number of points these tasks sampled is not taken into account in the total number of points for the calculation of pi.

In line 29 of the Listing 4.8 we retrieve the return value of the group result-check function using a run-time call. If the result-check function indicated that the group-of-tasks completed successfully, the estimation for π is returned, otherwise the program terminates.

Chapter 5

Related Work

ANT (Algorithmic Noise Tolerance) [8] was one of the earliest proposals to leverage the inherent error resilience of algorithms in the context of hardware implementation of signal, image, and video processing algorithms, improving both energy efficiency and tolerance to deep sub-micron noise.

Other research groups have demonstrated a set of applications which are tolerant to transient faults. De Kruijf [6] injects numerous faults into the PARSEC benchmarks, pointing out that emerging applications have a high level of local, instruction-level fault tolerance. However, applications with no error tolerance mechanisms are prone to failure in the presence of faults. Error Resilient System Architecture (ERSA) [3] demonstrates that it is possible to utilize error resilient probabilistic applications for designing error-resilient systems using inexpensive yet unreliable hardware.Li [11, 12] identifies characteristics of computations that make them error resilient and conducts fault injection campaigns to validate the resiliency of computations. In [15] a new profiler is presented; it indentifies subsets of computations that can be replaced with potentially less accurate counterparts. This technique delivers increased performance in return for lower, yet acceptable quality of service. Furthermore, using techniques such as sanity checks, relaxed exception handling and checkpointing, [29] notices a drastic improvement in terms of the robustness of probabilistic applications. To this direction all research groups point out that certain parts of programs are typically more fault-tolerant than others. Currently, we exploit this property by allowing the programmer to tag computations significance in a binary mode, i.e as significant or non-significant, while allowing for simple error checking mechanisms via the sanity functions.

Flikker [14] enables developers to identify critical and non critical data. During execution, the runtime maps the data to different parts of memory. Critical data are mapped to a memory partion refreshed with the standard rate, whereas non critical data are mapped to memory regions with lower refresh rates. This partitioning leads to energy savings at the cost of data corruptions. However, the data corruptions are tolerated by the inherent error-resilience of many of the target applications. In case of fatal errors, Flikker does not provide any safety guarantees or error checking mechanisms.

Relax [5] removes the illusion of perfect hardware, and relaxes the architectural semantics. ISA extensions are encapsulated to C-like function calls, allowing to tag regions of computations as non-significant. Each region is coupled with a recovery block in case of errors. The model is applied to several applications contained in the PARSEC benchmark suite showing that more than 70% of the total computations can be non-significant. Our programming model offers a finer granularity of tagging computations as non-significant as well as two-level error checking. Moreover we plan to offer automatic correction mechanisms by integrating with the automatic differentiation tool and using its output to semi-automatically generate sanity functions.

Work by Renard *et al* [1, 20] identifies computational patterns which can be omitted in order to improve performance at the cost of accuracy. These patterns are identified by the compiler and are automatically transformed to approxi-

mate computations. This development can dramatically simplify the process of creating applications which operate successfully under unreliable circumstances. Regarding the result degradation in [20] the modified applications usually produce data of acceptable quality whereas [1] uses QOS constraints to keep the resulting output distortion within acceptable bounds . However these methods do not take into account the end-user knowledge of the applications data and computations, disregarding potential error resiliency and therefore potentially undermining the energy gains.

In the Green framework [2] the user implements a set of approximate function implementations and provides qualitative metrics to the compiler. The compiler then utilizes a two-phase compilation technique: In the first phase, referred to as the learning phase, an executable is built, profiled and subsequently quality of service (QoS) data are collected. The data are exploited during the second compilation phase, when the final binary is created. The programming model is intrusive for the programmer since it explicitly demands for different approximate implementations of the same function. These are used by the compiler to produce an "intelligent" second binary upon which an exhaustive learning phase will take place. As such, the approach followed by Green significantly increases the compilation time.

Finally, EnerJ [21] uses language extensions to the JAVA programming model to tag data as approximate. Compiler analysis passes categorize all computations as either approximate or exact, depending on the data they are applied to. Computations with approximate data are executed in lower V_{dd}. SCoRPiO operates in a different way: the end-developer directly tags computations as significant/insignificant. During the execution of the program, significant parts of the source code will always execute on reliable hardware. On the other hand, unreliable source code regions will preferably execute on unreliable hardware. Another difference is that EnerJ does not offer to the user any error correction mechanism whereas SCoRPiO provides a two-level sanity checking functionality to detect and correct faults.

Chapter 6

Conclusions and Future Work

In this document we introduced the SCoRPiO programming model. The new programming model raises computational significance as a first-class concern during application development. It is task-based and uses directives to annotate the code. It allows the programmer to express her insight on the significance of computations for the quality of the end result. At the same time, the programmer can introduce multi-level checks to the output of computations in order to block propagation of unacceptable errors throughout the computations. Moreover, the model provides the functionality to implicitly express dependencies in the task graph through the input and output data of each task, and thus provide all necessary information to the underlying system software layers to properly order tasks and exploit opportunities for parallel execution. It also equips the programmer with synchronization constructs that are more flexible, thus more robust, on non-reliable environments. Finally, it communicates user hints on the characteristics of the input data and the output error tolerance to the automatic significance analysis techniques. In order to demonstrate the use of the programming model we re-wrote three computational kernels from different application domains on order to introduce significance information and allow their execution on unreliable substrates.

The immediate next step is to implement compiler support that recognizes the programming model constructs and lowers them to appropriate calls to the underlying runtime library developed in WP3.

We will also interface the programming model / compiler with the automatic significance analysis techniques developed in WP1. The programming model will provide developer insights, concerning the acceptable error tolerance on the output and the value ranges of data, as input to the automatic significance analysis. We expect that the outcome of automatic significance analysis will be exploited by a source to source compiler in three ways: (a) as a means to automate, or at least assist the programmer in significance tagging of computations, (b) as a means to automatically rewrite parts of the code with equivalent, less computationally intensive and power consuming counterparts, given the characteristics and ranges of data the computations are applied upon and the error tolerance, and (c) as a means to automatically generate error checking functions for tasks or task groups, or to substitute erroneous results with appropriate default values.

Morever, we will both internally exploit and export to the programmer information on the interaction of software with unreliable hardware, collected at the architectural level (task crashes, detected silent data corruptions, power / performance data etc).

We also plan to apply the model to additional and more complex applications. We will evaluate the effectiveness of the programming model and its implementation towards improving the reliability of applications executed on faulty substrates. Moreover, we will evaluate application performance in the presence of faults as well as the overhead of the programming model implementation (both in faulty and reliable environments). Ultimately, we will evaluate the effect on power consumption.

In the course of the aforementioned tasks, we expect that we may have to update and / or extend the programming model. In this case, the current document will be revised accordingly.

Bibliography

[1] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical Report MIT-CSAIL-TR-2009-042, MIT-CSAIL, 2009.

[2] Woongki Baek and Trishul M Chilimbi. Green: a Framework for Supporting Energy-conscious Programming Using Controlled Approximation. 45(6):198–209, 2010.

[3] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, 2012.

[4] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-memory Programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[5] Marc De Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 497–508. ACM, 2010.

[6] Marc De Kruijf and Karthikeyan Sankaralingam. Exploring The Synergy of Emerging Workloads and Silicon Reliability Trends. In *Proceedings of the Silicon Errors in Logic - System Effects Workshop (SELSE 5)*, 2009.

[7] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[8] Rajamohana Hegde and Naresh R Shanbhag. Energy-Efficient Signal Processing via Algorithmic Noise-tolerance. In *Proceedings of the 4th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 30–35. ACM, 1999.

[9] Georgios Karakonstantis, Nikolaos Bellas, Christos Antonopoulos, Georgios Tziantzioulis, Vaibhav Gupta, and Kaushik Roy. Significance Driven Computation on Next-generation Unreliable Platforms. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 290–291. IEEE, 2011.

[10] Bil Lewis, Daniel J Berg, et al. *Multithreaded Programming With Pthreads*, volume 2550. Sun Microsystems Press, 1998.

[11] Xuanhua Li and Donald Yeung. Exploiting Soft Computing for Increased Fault Tolerance. In *Workshop on Architectural Support for Gigascale Integration*, 2006.

[12] Xuanhua Li and Donald Yeung. Application-level Correctness and its Impact on Fault Tolerance. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA 2007)*, pages 181–192. IEEE, 2007.

[13] Jane WS Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. Use of Imprecise Computation to Enhance Dependability of Real-time Systems. *Foundations of Dependable Computing: Paradigms for Dependable Applications*, pages 157–182, 1994.

[14] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. Flikker: Saving Refresh-power in Mobile Devices Through Critical Data Partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[15] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality Of Service Profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 25–34. ACM, 2010.

[16] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. Significance Driven Computation: a Voltage-scalable, Variation-aware, Quality-tuning Motion Estimator. In *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 195–200. ACM, 2009.

[17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[18] Krishna V Palem. Energy Aware Algorithm Design via Probabilistic Computing: from Algorithms and Models to Moore's Law and Novel (Semiconductor) Devices. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 113–116. ACM, 2003.

[19] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2010.

[20] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and Statistical Analysis for Understanding Reduced Resource Computing. In *ACM Sigplan Notices*, volume 45, pages 806–821. ACM, 2010.

[21] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[22] Gunter Schoof, Michael Methfessel, and Rolf Kraemer. High ASIC Reliability by Using Fault-tolerant Design Techniques. In *Proceedings of the 2nd Workshop on Design for Reliability (DFR)*, 2010.

[23] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 152–161, New York, NY, USA, 2011. ACM.

[24] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The JPEG 2000 Still Image Compression Standard. *Signal Processing Magazine*, 18(5):36–58, 2001.

[25] John E Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66, 2010.

[26] George Tzenakis, Angelos Papatriantafyllou, John Kesapides, Polyvios Pratikakis, Hans Vandierendonck, and Dimitrios S Nikolopoulos. BDDT:: Block-Level Dynamic Dependence Analysisfor Deterministic Task-Based Parallelism. *ACM SIGPLAN Notices*, 47(8):301–302, 2012.

[27] Andrew B Watson. Image Compression Using the Discrete Cosine Transform. *Mathematica journal*, 4(1):81, 1994.

[28] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[29] Vicky Wong and Mark Horowitz. Soft Error resilience Of Probabilistic Inference Applications. *Proceedings of the Silicon Errors in Logic - System Effects Workshop (SELSE 2)*, 2006.