D2.3 Advanced compiler implementation

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

1 Introduction .......................... 6

2 Background ............................ 7
   2.1 Programming model primitives ........................................ 7
      2.1.1 Modifications to the programming model ............................. 7
      2.1.2 Runtime initialization ...................................... 7
      2.1.3 Memory regions allocation/de-allocation ......................... 8
      2.1.4 Task definition and significance characterization .................. 8
      2.1.5 Synchronization .......................................... 9
      2.1.6 Interfacing the SCoRPiO programming model with dco/scorpio ............... 9
      2.1.7 Defining significant regions within tasks ........................ 9
   2.2 Runtime system .................................. 10

3 Compiler Infrastructure .................. 11
   3.1 SCoRPiO compilation workflow .................................. 11
   3.2 Front-end compiler ........................................ 12
      3.2.1 The importance of significance analysis ....................... 13
      3.2.2 dco/scorpio and task significance ............................. 13
      3.2.3 Interfacing the programming model with dco/scorpio ............... 13
      3.2.4 Technical details .......................................... 15
   3.3 Back-end compiler ...................................... 16
      3.3.1 Compiler Critical Instruction Identification Analysis ............... 17
      3.3.2 Example ........................................ 19
      3.3.3 Implementation ......................................... 19
      3.3.4 Assembly File Creation .................................. 20

4 Installation Instructions .................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Installing the front-end compiler</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Installing the back-end compiler</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Using the Virtual Machine</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Example</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Known Issues and Future Extensions</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Known Issues</td>
<td>25</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Extensions</td>
<td>26</td>
</tr>
</tbody>
</table>
Executive summary

SCoRPiO proposes to exploit the significance of computations to design energy-efficient systems which scale gracefully under the presence of errors. This deliverable discusses the design and implementation of a SCoRPiO compiler which includes a source-to-source and a source-to-binary component. The source-to-source component implements the SCoRPiO programming model described in deliverable D2.1 and enables the development of significance-aware applications. The source-to-binary compiler is responsible for identifying and marking assembly instructions that should always be executed reliably. The current report accompanies the software prototypes of the compilers, which are the main parts of the deliverable. The compiler prototypes are delivered in the form of both standalone distributions with installation instructions, and also pre-installed in a single virtual machine.

Deliverable D2.3 consists of three parts. The first part (Chapter 2) provides background information on the role of the source-to-source compiler. Sections 2.1 and 2.2 briefly summarise the programming model and runtime system, respectively. In the same Sections we also discuss minor changes to the programming model with respect to Deliverable D2.1.

The second part details the functionality and the design of the SCoRPiO compiler framework. It bridges the gap between the programming model, the runtime system, and the significance analysis framework dco/scorpio. In Section 3.1 we discuss the SCoRPiO development toolchain and conclude that the SCoRPiO compiler toolchain can be seamlessly integrated to any compilation toolchain. The compiler framework takes care of all the details required to instrument significance-aware, parallel applications and outputs efficient code, requiring minimal user intervention in the form of #pragma compiler directives. In this deliverable we also extend the compiler to interface the programming model and dco/scorpio framework providing a complete, point-to-point programming environment to analyze code significance and to generate the appropriate binary files for significance-aware execution. Chapter 3 discusses the actual implementation of both source to source and source to binary compilers. Section 3.2 discusses the front-end and section 3.3 discusses the source to binary compiler. Deliverable 2.2 discussed the modifications to SCOOP [2], an existing source-to-source compiler which parses #pragma directives introduced by the SCoRPiO programming model.

In this document we focus on the interface between the programing model with dco/scorpio as well as on the identification of the critical instructions of an application. This final version of the source-to-source compiler performs two tasks: a) it lowers the programming model directives to calls to the SCoRPiO runtime system, and b) it lowers the programming model to calls to wrapper functions of dco/scorpio. The source to binary compiler identifies and marks critical instructions. To be more precise, not all instructions are equal in terms of fault vulnerability. For example, instructions that are used to compute memory addresses or generate control flow information have to be protected because they tend to crash program execution if they are suffer erroneous execution. Such instructing are considered critical.

The final part of D2.3 (Chapters 4, 5 and 6) provides a) information on how to get the compiler prototype, b) describes a set of applications and kernels used to validate the compiler prototype, and c) enumerates known issues and future extensions of the compiler prototype.
Chapter 1

Introduction

This report accompanies the software prototype of the compiler framework implementation. The compiler comprises a front-end and back-end and has four fundamental objectives: a) to enable the translation of applications developed using the SCoRPiO significance-aware programming model to SCoRPiO runtime system API calls, b) to support context-sensitive hooks which will be exploited at execution time by the underlying software layers, c) to automate (to some extent) the annotation of applications for automatic significance analysis, and d) to identify critical instructions of applications which must be protected and never executed unreliably. Objectives a) through c) are realized by the front-end compiler, and objective d) is met by the back-end compiler. Note that deliverable D2.2 covers lowering programming model directives to runtime calls as well as the context-sensitive hooks mechanism.

In deliverable D2.1 we introduced the SCoRPiO task-parallel, significance-aware programming model. This programming model allows the programmer to express her perspective on the contribution of each computation to the quality of the final output (QoR). Computations with major contribution to QoR are characterized as significant, whereas computations with less contribution are considered non-significant. The significance of computations determines the reliability level of the hardware which can be used for their execution. More specifically, significant computations are executed reliably, whereas non-significant computations can be executed unreliably. Unreliable execution involves aggressively under-powered cores, without sacrificing performance (by retaining the same clock frequency), however at the expense of a possibility for errors.

In deliverable D2.2 we extended SCOOP [2], a source-to-source compiler which implements static data dependence analysis to optimize the execution of task parallel applications by removing unnecessary runtime checks for non-conflicting tasks. Our extensions lower #pragma directives into calls to the SCoRPiO runtime system. Furthermore, the compiler is capable of instrumenting input source codes with context aware hooks. Their functionality enables software layers to associate events related to execution mechanisms with the context of the application.

We have extended the source-to-source compiler to use the context-sensitive mechanism of deliverable D2.2 as well as the programming model introduced in deliverable D2.1 to automate the annotation of applications for the automatic significance analysis offered by dco/scorpio. Additionally, we have implemented a compiler pass in LLVM [1] to identify critical instructions in an application. Such instructions should be error-free, or the program will, most probably crash.

The rest of the document is organized as follows: We briefly present the programming model and runtime system in Chapter 2. In Chapter 3 we provide information on the implementation of the source to source compiler. In Chapter 4 we outline the distribution methods and installation instructions for the SCoRPiO compiler. In Chapter 5 we list examples from a number of applications which have been ported to the SCoRPiO programming model and compiled using our compiler. Finally, in Chapter 6 we discuss a number of known issues of the compiler and list a few possible future extensions.
Chapter 2

Background

2.1 Programming model primitives

The SCoRPiO compiler supports all #pragma directives specified by the SCoRPiO programming model. The implementation of the required code generation has been discussed in deliverable D2.2. A brief summary of the SCoRPiO #pragma directives is shown in listing 2.1. For more information please consult deliverables D2.1 and D3.1 where the programming model and runtime system respectively are discussed in detail.

2.1.1 Modifications to the programming model

The programming model has had time to mature since we first documented it in Deliverable D.2.1. This development in its design necessitates minor modification to the supported compiler directives to improve the programming mechanisms supported by the model. This subsection discusses the aforementioned changes.

Through experimenting with benchmarks we came to the conclusion that it is more user-friendly and intuitive to specify an analog value for task significance rather than a binary one. On top of that, an analog representation of significance is compatible with the automatic significance characterization framework that is developed by the SCoRPiO partners in RWTH Aachen.

An application developer uses the #pragma omp task significance(expr) directive to tag a task with an analog significance value in the range [0.0, 1.0], indicating the relative importance of the task with respect to output quality. Depending on their significance, tasks may be executed on top of reliable or unreliable hardware. These changes are also shown in Sections 2.1.4 and 2.1.5.

In this document we also discuss the interface of Programming Model and dco/scorpio via our our source-to-source compiler. The latter, is a framework that utilizes interval analysis and code differentiation to discover the significance of code at the level of operations. To this end, we extend the programming model of SCoRPiO with a couple of directives which are only used to analyse software and are discarded by source-to-source compiler unless a special option is used during its invocation. The changes are summarized in Section 2.1.6.

2.1.2 Runtime initialization

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

### 2.1.3 Memory regions allocation/de-allocation

Memory allocation and de-allocation in programs is implemented using the traditional `malloc()/free()` functionality offered by the C library. However, memory areas which participate in the automatic data dependence analysis by the runtime system need to be annotated and this needs to be communicated to the runtime system. The compiler directives used for this purpose are `#pragma omp malloc` and `#pragma omp free(size)`. They precede the respective malloc and free calls. Essentially, they redirect calls to `malloc()` and `free()` to the implementations supplied by the SCoRPiO runtime system. The runtime system, beyond allocating / de-allocating memory, also generates and manages metadata for the memory buffers involved. Note that, departing from the usual semantics of `free()`, `#pragma omp free(size)` expects an argument `size` corresponding to the the size of memory buffer to be freed.

### 2.1.4 Task definition and significance characterization

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

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

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

- **significant()** The `significant(expr)` clause allows significance characterization at the task granularity. This information is made available to both the compiler framework and the runtime system. The enclosed `expr` should produce a value within the range `[0.0, 1.0]`. Tasks with significance value of 0.0 are considered non-significant, those with significance of 1.0 are considered significant. The remaining tasks are tagged as either significant or non-significant during execution time by the run-time as specified in Section 2.1.5.

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

### 2.1.5 Synchronization

We offer the `taskwait` directive to support elastic synchronization for a group of tasks, as well as result checking at the task group level.

This synchronization pragma directive supports the following clauses:

- **label()** The developer specifies the group to synchronize by using its name as a parameter to the `label()` clause.

- **all/time()** We provide two different types of relaxed synchronization. *all* waits for all tasks in the group, and *time* defines a time watchdog. In the event that neither *all* nor *time()* is specified, the synchronization mechanism uses *all*.

- **ratio** Using the `ratio()` clause, the programmer can instruct the runtime to execute (at least) the specified percentage of tasks - either globally or in a specific group, depending on the presence of a `label()` clause - on reliable hardware, while respecting task significance (a more significant task should not be executed unreliably, if a less significant task is executed accurately).

- **grouptolerance()** Similarly to task-level result-checking, we enable the programmer to specify task group-level result checking functions. This result-check function is evaluated at `omp taskwait` points, after the execution of the corresponding task group.

### 2.1.6 Interfacing the SCoRPiO programming model with dco/scorpio

`dco/scorpio` requires a certain level of preparation before an application is ready to undergo significance analysis. We chose to involve the application developer in preparing the application source code for the analysis. She begins by identifying variables within her program which are involved in computations that take place within tasks. Then she redeclares them using a special data type. This allows `dco/scorpio` to track all computations that involve said variables. She also has to define the input data ranges as well as the variables which contain the program output.

- **`dco_input(array[start:end]@range(low, high))`** `#pragma omp dco_input(...) is used to specify the application input data as well as the value ranges of the input data.

- **`dco_output(array[start:end])`** `#pragma omp dco_output(...) informs dco/scorpio of the application output values.

Information obtained through the aforementioned changes and the `in()` and `out()` clauses is used at compile time via calls to the `dco/scorpio` framework to facilitate the significance analysis of code.

### 2.1.7 Defining significant regions within tasks

The compiler directive `#pragma omp significant` clause may only be used within the source code of task. It denotes that the following statement, or block of statements enclosed in `{ }` is to be treated as significant, regardless of significance of the whole task.
2.2 Runtime system

The SCoRPiO runtime system supports an implicit, task-based parallel programming model with significance annotations. It uses dynamic dependence analysis to guarantee that tasks are executed in correct order with respect to their individual data dependencies. However, this feature comes at the cost of additional runtime overhead. The runtime mitigates this overhead by featuring a $O(1)$ access to task and memory metadata required for such a mechanism. Tasks are scheduled to reliable, or unreliable cores depending on their significance annotation. Significant tasks will always be executed on reliable cores, whereas non-significant tasks will preferably use an unreliable core. The runtime system uses an API exposed by the hardware to manage the reliability of cores in order to meet reliability requirements for tasks of different significance levels. A non-significant task may also come with a task result check function ($\text{tasktolerance()}$). Work-groups may specify a group result check function ($\text{grouptolerance()}$).

The above functionality is offered through five runtime system API calls.

- **tpc_init()** Runtime system initialization function.
- **tpc_malloc()** The runtime system uses this function to return memory regions on which the data analysis mechanism can operate.
- **tpc_free()** Memory buffers allocated using the $\text{tpc_malloc()}$ function should be freed using the $\text{tpc_free()}$ runtime call.
- **tpc_call()** This function signifies the instantiation of a new task.
- **tpc_wait_group()** This function acts as a flexible barrier. It acts as the synchronization mechanism between the main application thread and the worker cores.
Chapter 3

Compiler Infrastructure

The compiler infrastructure of project SCoRPiO comprises a front-end, and back-end. The front-end is realised through the source-to-source compiler. It translates SCoRPiO programming model directives into calls to the SCoRPiO runtime system. The back-end compiler is responsible to compile the source files to actual binaries and annotate instructions of the application to be protected by the HW system. The back-end procedure is based on the LLVM compiler infrastructure.

3.1 SCoRPiO compilation workflow

Figure 3.1b details the SCoRPiO compilation workflow which consists of the following steps:

- The programmer produces the significant-aware application source code partitioned in tasks using the significance and dco/scorpio directives. These directives have been explained in deliverables D2.1 and will be revised in this chapter.

- The source-to-source SCoRPiO compiler recognizes the #pragma directives and lowers them to two different types of calls: a) calls to the API of the underlying SCoRPiO runtime to enforce the programming model directives, and b) calls to the API of the dco/scorpio tool that performs automatic significance analysis of the code. For a formal definition of code significance refer to deliverable D1.1.

- The dco/scorpio C++ compiler analyzes the application source code under the guidance of the API calls generated in the previous step. Those calls are in the form of MACROS that define, among others, the input, output and intermediate variables to be analyzed by sco/scorpio. The output of this phase is an enumeration of the significance for all designated variables of the source code.

- The programmer uses the information produced by dco/scorpio to a) potentially re-partition the source code into tasks, b) annotate those tasks with their significance numbers, and c) write the result check functions.

- The source code is compiled using a conventional gcc compilation flow and the resulting binary is executed in the target platform under the control of the SCoRPiO runtime system. The target platforms that have been implemented include a) the GemFI simulator and b) Intel’s x86 multicore CPU.

Note that in the case of SCoRPiO one may produce two binaries for a single application: one primary binary to compute solutions for the problem at hand and one secondary binary that performs significance analysis of the application source code. The binary generated for significance analysis is used to optimize the primary binary (Section 3.2).
The SCoRPiO automatic significance analysis framework dco/scorpio is implemented in C++. The significance analysis framework relies heavily on operator overloading to compute the significance of code at the level of operations. However, our implementation of the source-to-source compiler is agnostic when it comes to the actual tool used to perform the significance analysis. Any framework can be used as long as it uses similar interface to dco/scorpio. In fact, the virtual machine (Section 4.3) offered alongside this report as well as the source code of the source-to-source compiler do not contain the dco/scorpio framework.

![Diagram](image)

Figure 3.1: Comparison of SCoRPiO compilation workflow with a typical compilation toolchain.

### 3.2 Front-end compiler

In a previous deliverable (D2.2) discussing the prototype SCoRPiO compiler we provided details on how a program annotated with SCoRPiO programming model primitives is lowered to C code containing calls to the SCoRPiO runtime system. We also described a feature of the compiler called “context sensitive hooks”. The latter provided information at runtime to the application developer by means of events for particular interesting points within a program’s execution time such as when a task is issued, is beginning/finishing execution, etc.

The rest of this Section discusses the steps required to interface the SCoRPiO programming model with the automatic significance analysis framework dco/scorpio. Our goal is to enable an application developer to perform significance
Significance-Based Computing for Reliability and Power Optimization (SCoRPiO)

analysis on her application. The results of the analysis can be used in multiple ways, but we mostly focus on two key scenarios that have emerged through our use of the SCoRPiO software framework.

3.2.1 The importance of significance analysis

We have observed that discovering the significance of individual tasks is not a particularly trivial task. To this end, we have devised a methodology which relies on dco/scorpio functionality to report the significance of task-based codes. Inspection of the analysis results, allows the application developer to better exploit the programming model mechanisms in order to gracefully trade-off output quality with energy/power cost reduction.

A by-product of significance analysis is that a developer who reviews the significance analysis results will also gain more insight to the analysed code in the form of task significance ranking. This information can be used to optimize and fine-tune the code in order to perform better under unreliable execution conditions.

The rest of this Section discusses the technical details regarding the programming model and dco/scorpio interface.

3.2.2 dco/scorpio and task significance

The significance analysis, requires that the program input data is specified along with its respective data value ranges. Additionally, the developer that performs the analysis must also tag the data containing the final program output just after they are produced. This information is used by dco/scorpio to perform its algorithm differentiation analysis on all intermediate values computed by the application during its execution time. The significance analysis results are at the granularity of operations between program data. The analysis output, in fact, is significance ranking of all values touched by the program during its lifetime, i.e. significance values for input, intermediate, and output data. A more detailed discussion about the dco/scorpio framework can be found in Deliverable D1.1.

The first challenge we came accross was performing the automatic significance analysis on top of task-based applications, namely the programming model directives and the context-hook mechanism. dco/scorpio computes the significance of code at the level of operations which is too fine a granularity for our purposes. Ideally, we would like dco/scorpio to produce significance values at the level of tasks. Fortunately, bridging the gap between the coarseness of the significance analysis results is possible. As shown in Figure 3.2 we can expand on the principles of dco/scorpio and produce a definition for task significance: A task is as significant as its output data.

3.2.3 Interfacing the programming model with dco/scorpio

At this point for the purposes of automating the annotation of applications for significance analysis we have at our disposal a definition of task significance, as well as the programming model provisions. Since the programming model already collects information regarding task input and output via the #pragma omp task in(...) out() clauses we need only design of a mechanism to tag the input/output data as well as the input value ranges to collect all necessary information about data. This mechanism comes in the form of #pragma omp dco_input(...) and #pragma omp dco_output(...) directives and is discussed in the previous Chapter (see Section 2.1.6). Listing 3.1 serves as a comprehensive list of the dco/scorpio wrapper functions used by the source-to-source compiler.

When the source-to-source compiler detects a dco_input() or dco_output() #pragma directive, it utilizes the context-sensitive hooks event mechanism to produce code that invokes the appropriate wrapper function. The compiler delegates the communication between the application and dco/scorpio to said wrapper functions. As an example, for the dco_input(array[start:end]@range(low, high)) clause the code in Listing 3.2 is generated. Notice that, the wrapper function dco_input() internally invokes dco/scorpio and registers all data within the array that have indices in the range [start, end] as program input data. It also sets the data range of the related program input to [low, high]. Additional
Some sequence of operations

(a) Significance of operations

Some sequence of operations

(b) Significance at the level of tasks

Figure 3.2: In Figure 3.2a, we show the computation of an intermediate variable $i_0$. It is the product of the $+$ operation for input variables $x_0$ and $x_1$. This intermediate variable is used to produce the output variable $o_0$. We can claim that the operation in the red hexagon has the same significance as $i_0$. After all, data values are the results of operations. We can expand on this to provide a definition for tasks (see Figure 3.2b): A task is as significant as its output data.

The compiler will also produce calls to dco/scorpio wrapper functions for task input and output data. Following our task significance definition, output data are registered as intermediate program data. This enables the retrieval of the significance of the task which produced said intermediate program data. We also register task input data so that a human can later review the analysis results and get a complete picture of the application data-flow. This allows developers to better understand the task communication pattern of applications by means of visualizing the task data dependencies. Listing 3.3 shows an example in which a task defines its input data via a $in(array[start:end])$ clause. Index ranges of arrays specified using the $out(\ldots)$ clause are handled in a similar manner. Note that the last parameter to the wrapper function $dco\_task()$ is used to differentiate between task input and output data. In Figure 3.2b, we show a task which produces $n$ data visualised as dark green intermediate program data nodes $i_k$ with $k \in [1, n]$. The task’s significance value is a function of the significance values of its output. Our experiments have shown that for a task $t$ such an appropriate function is $task\_significance(t, \vec{i}) = \max \vec{i}$ where $\vec{i}$ is the set of intermediate variables produced by the task $t$. This way a task is as significant as its most significant output. Nevertheless, this decision is taken after the analysis has been performed. This makes our methodology broad enough to allow for experimentation with multiple functions to compute the significance of tasks.

The reader will notice two wrapper functions $dco\_malloc()$ and $dco\_free()$ in Listing 3.1 which appear out of place. However, upon closer inspection their signatures match those of $tpc\_malloc()$ and $tpc\_free()$. Indeed, $dco\_malloc()$ and $dco\_free()$ are used as substitutes of the SCoRPiO runtime memory handling mechanisms. Keep in mind that, dco/scorpio is a vast collection of interval implementations of operations and mathematical functions in C++. The significance analysis framework uses operator overloading to seemingly keep track of operations between data. This requires data to be allocated via the $new \{\}$ operator of C++. However, this operator is not a feature of the C language. To overcome this predicament, we substitute calls to $tpc\_malloc()$ and $tpc\_free()$ functions with their dco/scorpio wrapper alternatives (Section 3.2.4).
Finally, we considered various ways to signify when the analysis framework should initialize and when it should perform the actual analysis. Note that, the analysis must be performed right after all program output data are computed and registered via the dco_input() compiler directive. In the end we opted for the most user-friendly method that does not require any additions to the programming model primitives. This way, developers using our programming model need to just use two extra clauses for the purpose of significance analysis. The significance analysis framework is initialized when a #pragma omp start(...) is used. The actual analysis takes place just before the first call to a statement free() tagged with the #pragma omp free(...) directive.

### 3.2.4 Technical details

In order to keep the source code generation as simple as possible without major sacrifices to functionality we had to make some decisions fairly early in the design phase of the source-to-source compiler.

The first decision we made was to not use the SCoRPiO runtime system when producing code for significance analysis. The biggest reason for this decision is the fact that we wanted to eliminate as many free variables involved in validating the results of the analysis as possible. To be more precise, in the general case it is not easy, if at all possible, to guarantee the significance analysis results for code which executed in parallel. Executing tasks in parallel can potentially lead to slightly different results each time the significance analysis version of an application is ran due to the rounding errors that are inherent in floating point operations. Another reason that the runtime had to be completely by-passed was hinted in Section 3.2.3. The significance analysis framework requires that its classes are allocated via the new [] C++ operator. This means that data which are tracked by dco/scorpio cannot be placed within the runtime pool of memory because said memory is allocated by the run-time system prior to any data allocations. In other words, data tracked by the significance analysis cannot be used by the data-dependence analysis of the runtime to schedule the tasks. This would have the effect of erroneous program output and garbage significance analysis results.

Another issue that became obvious during the early stages of the compiler design is the fact that it is not cost-effective, in the general case, to fully automate the process of instrumenting applications for significance analysis in terms of effort spent to implement the necessary compiler functionality. In some cases, significance analysis cannot be performed at all. For example, an application might include calls to functions for which the compiler does not have access to their source code. In such a case, it would not be possible for dco/scorpio to track the operations within such
Significance-Based Computing for Reliability and Power Optimization (SCoRPiO)

functions. Therefore, instead of providing a completely automatic way of instrumenting applications for significance analysis we opt for a user guided analysis which automates the various tedious tasks one would have to perform if she ventured to manually annotate an application for significance analysis. More specifically, we require that the user decides which variables are a) program input data, b) program output data, and c) intermediate program data. The user then has to change the definition of said variables to be of type `dco_var_t`. Afterwards, he must use the compiler directives described in the previous Sections. Defining variables as `dco_var_t` instead of `dco::ia1s::type` enables the source-to-source compiler to not explicitly depend on the dco/scorpio framework. It can therefore be installed on machines without the need to distribute dco/scorpio as well. Interestingly, it also enables developers to use alternative mechanisms to dco/scorpio as long as they provide similar API interfaces.

To summarize, dco/scorpio enabled versions of code:

a. will execute all issued tasks in a serial fashion
b. require that the developer registers all input and output data as discussed in the previous paragraphs
c. must not invoke functions for which the compiler does not have access to their source code
d. expect the use of `typedef dco_var_t` which replaces `dco::ia1s::type` that is actually defined within the dco/scorpio header files to keep the compiler dependency free from dco/scorpio
e. use the directives `dco_input()`, `dco_output()`, `in()`, and `out()` on data which have been declared as `dco_var_t` manually by the developer
f. tasks must not execute after memory is freed

g. must not free allocated memory before all program outputs are registered

### 3.3 Back-end compiler

Our programming model encapsulates significant parts of the application from insignificant using a task-based approach. Nevertheless, all tasks, regardless their significance, contain certain instructions which should always be executed correctly, even if they are within an insignificant task of the application.

In Listing 3.4 and 3.5 we present a simple vector addition in C and the MIPS assembly implementation of the vector addition respectively. Line 6 of the assembly version corresponds an instruction operating on top of data, this instruction does not affect explicitly or implicitly any memory address or the control flow of the application. Lines 1, 3, 11 correspond to instructions operating on the control flow and all the remaining instructions correspond to pointer arithmetic. Errors impacting pointer arithmetic instructions may result to program failures, (application fails to terminate due to an HW/OS trap) more frequently than faults impacting data-instructions. The same applies for instructions controlling control flow.

Should someone compare the importance of instructions in relevance to application resiliency, instructions operating on top of data should be the least important. Instructions operating between data might mask a fault, or in any case they rarely result to program failures. On the other hand, pointer arithmetic instructions or instructions that may modify the control flow of the program are primary candidates to result into a failure. Such instructions are critical to the correct execution of the program, even when considering the relaxed definition of program correctness and should be protected to guarantee normal termination. Hardware mechanisms which are able to detect and correct faults due to timing violations have been proposed in deliverable X (den thimaimai pio einai). Those mechanisms try to contain hardware faults and present an error-free execution engine to the software.
This section presents a description of the \textit{LLVM} compiler pass that detects critical instructions in an application. Those instructions should be error-free, or the program will, most probably suffer from crashes. We also present linker extensions to merge information of the critical and non-critical instructions.

```
for ( k = 0; k < SIZE ; k++)
    C[k] = A[k] + B[k];
```

Listing 3.4: Vector add used a simple example.

```
add $s1 $0 $0
for
    beq $s0, $s1, end
    lw $t2, ($s2)
    lw $t3, ($s3)
    add $t4, $t3, $t2
    sw $t4, ($s4)
    addi $s2, $s2, 4
    addi $s3, $s3, 4
    addi $s4, $s4, 4
    addi $s1, $s1, 1
    j for
end
```

Listing 3.5: Vector add used a simple example.

In Figure 3.3 we present the flow chart of tools used to implement the critical instruction identification analysis. Initially the source and header files are passed to \textit{Clang}, the front end of \textit{LLVM}. \textit{LLVM} processes the output of \textit{Clang} and performs optimizations. During this phase our analysis takes place. \textit{LLVM} outputs two types of files, the object files and the meta data files, which contain information about the criticality of the instructions. Both files are fed to an extended linker, which creates an executable and a metadata file. The object file creation and the linker part is yet to be implement. In the context of this deliverable we present the algorithm to identify critical instructions. The criticality of each instruction is presented in the form of comments in the assembly output files.

### 3.3.1 Compiler Critical Instruction Identification Analysis

The analysis is similar to an upward exposed uses analysis\footnote{Upward exposed uses: For each definition of a variable, find all uses that it reaches}. Starting from the last basic block and traversing the instructions in reverse execution order we identify obvious critical instructions. Obvious critical instructions should meet one of the following criteria:

**ClassI:** During the execution of the instruction an address calculation is performed. For example the \texttt{ld} instruction of the \texttt{MIPS} architecture.

**ClassII:** The instruction has implicit or explicit impact on the control flow of the application. For example a branch instruction has explicit impact on the control flow whereas a compare instruction has implicit impact.

These instructions operate by definition on top of critical information, e.g. addresses or data flow. Therefore, in our analysis we use the operands used by such instructions to identify other instructions which operate also on top of critical information. In other words, the analysis propagates information from the obvious critical instructions to all the instruction of the application.

Obvious critical instructions are tagged as critical and depending on criteria met by each instructions some of the operands used (\textit{uses}) to compute the definition (\textit{def}) of this instructions are pushed to a bit vector, called \textit{GEN}. The vector size is equal to the number of different registers supported by the architecture. If the instructions are in \texttt{ClassI}, only the operands participating in the address calculation are pushed to the \textit{GEN} vector. If the instruction is in \texttt{ClassII}, all operands are pushed in the \textit{GEN} vector.
When traversing an instruction we check whether it defines a value contained in the GEN vector. If this is the case, the instruction is tagged as critical, the definition is removed from the vector and the uses of the new critical instruction are pushed into the GEN vector.

When reaching the entry point of the basic block the GEN vector contains all the values $x$ which are used by a critical instruction $s$ inside the basic block, however, there is no definition of $x$ between $s$ and the beginning of the basic block. Equations 3.1, 3.2 present the transfer function.

$$GEN(I_0) = \emptyset$$

$$IN_n(I) = GEN(I_n) = (GEN(I_{n-1}) - defs(I_n)) \cap f(I_n) \quad (3.1)$$

$$f(I_n) = \begin{cases} 
uses(I_n), & \text{if } I_n \in ObviousCritical \\
uses(I_n), & \text{if } defs(I_n) \in GEN(I_{n-1}) \\
\emptyset, & \text{Otherwise} 
\end{cases} \quad (3.2)$$

After the procedure traverses the entire block, it propagates the information to all the predecessors of this block using the union operator (Equation 3.3). We apply this operator to the analyzed code iteratively until there are no changes in the GEN set.

$$\forall B_i \in Predecessor_B | OUT(B_i) = OUT(B_i) \cup IN(B) \quad (3.3)$$

The analysis iterates continuously on the basic blocks of the function until there is no change between consecutive iterations.
3.3.2 Example

Figure 3.4 shows a simple example. The analysis starts from the last basic block (node 12) and the GEN set is empty. The analysis continues by processing instruction 11 which is the last instruction of the next basic block. Instruction 11 is tagged as critical since it is a control flow instruction. Instruction 11 has no operands therefore the GEN set remains empty. Instruction 10-7 are not obvious critical. Instruction 6 performs address calculation since it is a store word. The instruction is tagged as critical and the operands of the instruction that contain addresses are stored inside the GEN vector. The next instruction is not an obvious critical one and does not define a register contained in the GEN vector, hence the instruction is not recognized as critical. Instructions 4, 3 and 2 are identified as critical since they perform address calculations and branching. All operands used by these instructions are added in the GEN set. Finally the analysis moves to the first basic block and identifies instruction 1 as significant, since it sets a value to register $s_1$ which is inside the GEN vector. After the instruction is processed register $s_1$ is removed from the GEN set.

The second iteration identifies instructions 10-7 as critical because the registers defined by those instructions are in the GEN vector. Each selected instruction deletes the uses from GEN, and immediately adds back the defs. For example, instruction 9, deletes and adds register $s_4$ to bit vector GEN. The analysis continues without any other additions. The analysis terminates when no additional instructions are identified as critical.

3.3.3 Implementation

The analysis handles each function separately, therefore we register our analysis as a Machine-Function pass, a pass that operates on top of the internal LLVM machine dependent instruction representation. All instructions should be analyzed and grouped into critical and non-critical ones. To avoid loss of information due to other optimization which may modify the instruction stream, we register our analysis as a pre-emmit pass hence the analysis is performed just before emitting the instructions to their binary representation (MCInst). To identify obvious critical instructions we use member functions of the MachineInstr class. To be more precise, we use the following build-in functions: isBranch(), isCall(), isReturn(), isCompare(), mayLoad(), mayStore(). Moreover, for the x86 instruction set we manually check the opcode of the instruction against the load effective address opcodes (lea). lea instruction computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the addressing modes of the processor; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

In the x86 instruction set, branches or function calls do not have any register operands. Therefore such instructions are critical however, they do not further interact with our analysis. In the case of returning from a function call (isReturn()) again the instruction is considered as critical, however the optional use of this instruction is not recorded in the GEN vector, because we want to protect the PC address calculation of the call instruction not the returning value. In the case of compare instructions, all use-register operands are recorded in the GEN vector. For load, store instructions we consider only the operands which participate in the calculation of the source/destination address. Finally, all use/def operands are considered when processing instructions which are not contained in the obvious critical set but need to be critical due to implicit dependencies of critical instructions upon them. For example instructions 7, 8, 9 in Figure 3.4c are not contained into the obvious critical set, but are tagged as critical due to the implicit dependency of instructions 3, 4, 6 upon them.

In Listing 3.6 we provide a pseudo-C++ like implementation of our analysis.
Figure 3.4: A simple example of the compiler analysis pass for 3 iterations of the algorithm

3.3.4 Assembly File Creation

LLVM after optimizing the source code, the binary creation takes place. The binary creation in LLVM is supported by the MC interface. To start the binary creation the internal representation of instructions is changed from the MachineInstr class to the MCInst class. As the transition from MachineInstr to MCInst takes place, the criticality of each instruction is transferred to the MCInst representation.

From this point on the compiler either emits assembly files or creates an object file. The assembly file displays the criticality of the instructions in the form of comments. At the end of each instruction the compiler prints the string \#CRITICAL:X, as presented in Listing 3.7. Critical instructions have \(X\) equal to 1 whereas non-critical instructions have the value of 0. This information is produced mainly for debugging purposes and cannot be transformed back to any binary representation, since it would require modification of the assembler parser to recognize such information and encode it.
bool protectionAnalysis(Function &F) {
    //Initialize all vectors to an empty set.
    for (BB = F.end(); B != F.start(); BB++)
        BB.GEN.init(false);
    //Traverse the CFG in reverse order and apply the traverse function.
    while (!noChanges)
        for (BB = F.end(); B != F.start(); BB++)
            for (I = BB.end(); I != BB.start(); I++)
                if (IsObviousProtected(I))
                    I.Protected = true;
                    noChanges = true;
                    propagateProtection(I);
                else if (I.defs() in BB.GEN) {
                    noChanges = true;
                    I.Protected = true;
                    propagateProtection(I);
                }
    // Propagate the IN to the out of the predecessor blocks.
    for (P = BB.predecessors(); P != NULL; P++)
        P.setGen(BB.GEN);
    }
}
void propagateProtection(Instruction I) {
    for (Operands in I)
        if (Operand.isReg() && Operand.isDef())
            BB.GEN[Operand.getReg()] = false;
    for (Operands in I)
        if (Operand.isReg() && Operand.isUse())
            BB.GEN[Operand.getReg()] = true;
}

Listing 3.6: A C++ pseudo code demonstrating the main driver of our algorithm.

movslq -32(%rbp), %rax
movq -8(%rbp), %rcx
movl (%rcx,%rax,4), %edx
movslq -32(%rbp), %rax
movq -16(%rbp), %rcx
addl (%rcx,%rax,4), %edx
movslq -32(%rbp), %rax
movq -24(%rbp), %rcx
movl %edx, (%rcx,%rax,4)

Listing 3.7: x86 Assembly corresponding to the inner block of a vector add
Chapter 4

Installation Instructions

The compiler framework is split into two parts, a front-end compiler which performs source-to-source compilation and a back-end compiler which is built on top of LLVM and identifies the critical instructions of an application. Compiling the SCoRPiO compilers requires that you have several software packages installed. The table below lists all those required packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU Make</td>
<td>3.79</td>
<td>Makefile/build processor</td>
</tr>
<tr>
<td>GCC</td>
<td>&gt;=4.7.0</td>
<td>C/C++ compiler</td>
</tr>
<tr>
<td>python</td>
<td>&gt;=2.7</td>
<td>Python interpreter</td>
</tr>
<tr>
<td>libtool</td>
<td>1.5.22</td>
<td>Shared Library Manager</td>
</tr>
<tr>
<td>zlib</td>
<td>&gt;1.2.3.4</td>
<td>Compression Library</td>
</tr>
<tr>
<td>ocaml</td>
<td>3.12.1</td>
<td>ocaml compiler</td>
</tr>
<tr>
<td>automake</td>
<td></td>
<td>Build processor</td>
</tr>
<tr>
<td>indent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>emacs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td></td>
<td>Perl interpreter</td>
</tr>
<tr>
<td>libc</td>
<td></td>
<td>The GNU C library</td>
</tr>
<tr>
<td>bison</td>
<td></td>
<td>General-purpose parser generator</td>
</tr>
<tr>
<td>flex</td>
<td></td>
<td>Lexical Analyzer</td>
</tr>
<tr>
<td>gperf</td>
<td></td>
<td>Hash function generator</td>
</tr>
</tbody>
</table>

Table 4.1: The prerequisite packages for installing the compiler on Ubuntu Linux 14.04
4.1 Installing the front-end compiler

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

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

Finally, execute `make` and the SCoRPiO compiler will be built. The compilation process will produce an executable script named `scoop` in the directory `scoop`. At this point you should modify your `~/.bashrc` file by appending the following line to the end of the file `export PATH=$PATH:<full path to scoop directory>`. This final step updates the `PATH` environment variable to enable executing `scoop` without specifying the full path to the binary.

4.2 Installing the back-end compiler

To install the SCoRPiO source to binary compiler download the file [http://scorpio.ireteth.certh.gr/backend.zip](http://scorpio.ireteth.certh.gr/backend.zip). Extract the contents of the zip file and make a separate directory, in which the building of the compiler will take place. Change directory to the building directory and issue the command `/PATH/TO/BACK/END/COMPILER/configure`. This script will notify you in the event of missing packages. Make sure the `configure` script does not terminate due to an error. Finally, execute the `make` and `make install` commands. At this point the back-end compiler can be invoked by typing the `clang` command.

4.3 Using the Virtual Machine

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

The login credentials are:

```
User name: scorpio
Password: scorpio2016
```
Chapter 5

Example

We have bundled, to both the compiler distribution .zip file and the VM, a number of applications which are part of the SCoRPiO benchmark suite. The benchmark source codes can be found in the `scoop/examples/` folder in the VM. The benchmarks can be compiled using the provided `Makefile`.

The applications ported to the programming model and bundled with the distribution are:

- A kernel of a method to decompose a 2D/3D hyperplane into an MxN grid of smaller domains. Using a Monte Carlo approach, it computes solutions at interface points.
- Kmeans clustering
- Sobel filter (edge detection)
- Maclaurin series
- Jacobi iterative solver

Note that the supplied Makefiles do not produce executable binaries. The compilation process is considered finished when intermediate *.o binary files are generated. The final step would be to link those *.o files with the runtime library using a linker to produce the primary binary (Section 3.1).

There are alternative versions of the benchmarks which have been prepared for significance analysis output, which can be found alongside the original versions of the benchmarks with a suffix "_dco". The compilation process is considered finished when the intermediate file `trans.c` is produced. This file contains the required annotations to the application source code (Section 3.2) for significance analysis using the dco/scorpio framework (or any other equivalent tool which provides the same interface).

Inside each benchmark folder we include a makefile which produces the output source code file for the benchmark. It is a text file under the name `trans.c` containing C code. We also include a "golden" compilation output file named `golden_trans.c`.

To use the back-end compiler issue the `make assembly` command inside the benchmarks directory. This command will create a single assembly file. Next to each line of the assembly file the significance information will be listed in the form of comments.
Chapter 6

Known Issues and Future Extensions

6.1 Known Issues

In this section we discuss a number of non-critical known issues of the compiler:

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

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

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

Moreover, it is not possible use the automatic run-time dependence analysis pass on two-dimensional arguments to tasks. This feature would require extensive changes to the programming model and compiler for a rather limited benefit; the benchmarks indicate that even if an array is conceptually two dimensional the algorithm accessing its contents is usually accessed using the linearised form \texttt{array[row \ast columns + column]} instead of the 2D form \texttt{array[row][column]}.

Finally, the criticality of instructions is performed on the scope of a function each time. Therefore, criticality dependencies between data of different functions are not analyzed by our algorithm. Moreover, to identify the criticality of instructions requires the source files. Consequently when linking executables with external libraries there is no information about the criticality of the instruction included in the libraries. We use a conservative approach in such a case, and consider these instructions as critical.
6.2 Future Extensions

Possible extensions to future versions of the compiler prototype would include more events, or even further refined versions of the events currently offered by the compiler:

**Task error detection**  This event could be issued by the runtime whenever the hardware or the runtime itself detected an error during the execution of a task.

**Group Done**  The *Group Done* event might be further expanded to also include information about the waiting condition that was actually fulfilled for a particular *#pragma omp taskwait* directive.

**Re-executions**  The runtime could also fire events when it decides to re-execute tasks, or groups of tasks.

**Event timestamps**  All events could also be tagged with *timestamps* which would, for example, allow a profiler to combine this information into a story that illustrates the execution an application through time.

**Improve automatic annotation for significance analysis**  The source-to-source compiler can perform more intelligent analysis to automatically define variables as *dco.var.J* using the information embedded within the supported *#pragma* directives.  This would reduce even further the time required to prepare an application for significance analysis.
Bibliography
