Rigel: A DSL Use Case for Trace Analysis

Beatriz Aguilar Gallo
(beatriz.aguilar@bsc.es)

Director: Eduard Ayguadé Parra (eduard@ac.upc.edu)
Co-director: Vicenç Beltrán Querol (vbeltran@bsc.es)
"If you dare nothing, then when the day is over, nothing is all you will have gained"

Acknowledgments

I would like to thank my advisors Vicenç Beltrán and Eduard Ayguadé for giving me the opportunity to work on this exciting project.

Thanks also to Alejandro Fernández for all his support. You saved me in more than one occasion!

Special thanks to my awesome boyfriend, Ale. I really appreciate your effort to try to understand all my explanations about my research. Thank you very much for being always by my side, including in the long nights that I had to work.

Last but not least important to my family and friends for their continuous support in all my adventures.
Rigel: A DSL for trace analysis

Beatriz Aguilar Gallo

Computer Architecture Department (DAC)
Universitat Politècnica de Catalunya (UPC)

2015

ABSTRACT

Thanks to the DSLs is possible introduce compile optimizations in a flexible, scalable and simple way. However, some years ago to implement a DSL was a huge effort because it was necessary develop some complex features such as: an interpreter, a compiler, an application generator... Recently, all this effort has been reducing thanks to the development of a DSL platform that make the implementation of a DSL simple. To illustrate the potential of this platform, we have developed Rigel an use case of DSLs for trace analysis.

Keywords: DSL, LMS, Scala, Compilers, Trace, CTF, LTTng.
Rigel: A DSL for trace analysis

Beatriz Aguilar Gallo

Computer Architecture Department (DAC)
Universitat Politècnica de Catalunya (UPC)

2015

ABSTRACT

Gracias al uso de lenguaje de dominio específico (DSL) es posible aplicar optimizaciones de compilación de una manera flexible, escalable y sencilla. Sin embargo, hace algunos años implementar un DSL era un gran esfuerzo ya que era necesario desarrollar algunas características complejas como: un intérprete, un compilador, un generador de aplicaciones... Recientemente, todo este esfuerzo se ha visto reducido gracias al desarrollo de una plataforma de DSL que hace que la implementación sea relativamente sencilla. Para ilustrar el potencial de esta plataforma, hemos desarrollado Rigel el cual es un caso de uso de DSLs para el análisis de trazas.

Keywords: DSL, LMS, Scala, Compilers, Trace, CTF, LTTng.
List of Figures x

List of Tables xi

List of Code Samples xii

1 Introduction 1
   1.1 Motivation 2
   1.2 Objectives 2
   1.3 Contribution of this project 3
   1.4 Document structure 3

2 Related Work and State of the Art 4
   2.1 Domain-Specific Language frameworks 5
      2.1.1 Lightweight Modular Staging 5
      2.1.2 Jet 5
      2.1.3 OptiML 6
6.1.1 Final Users .................................................. 41
6.1.2 Indirect Users .............................................. 41
6.2 Planning .......................................................... 41
   6.2.1 Go in depth in Scala language - 4 weeks .......... 42
   6.2.2 Study of the CTF format and trace tools - 4 weeks . 42
   6.2.3 Develop a C++ library to handle CTF traces - 6 weeks . 42
   6.2.4 Study of LMS - 4 weeks ............................... 42
   6.2.5 Interface design - 4 weeks ............................ 42
   6.2.6 Code generation - 3 weeks ............................ 42
   6.2.7 Optimization implementation - 3 weeks ............ 42
   6.2.8 Test and results - 2 weeks ............................ 43
   6.2.9 Documentation - 2 weeks ............................. 43
6.3 Action plan ...................................................... 43
6.4 Methodology ..................................................... 43
   6.4.1 Development ............................................ 44
   6.4.2 Directive feedback ..................................... 44

7 Conclusions and Future Work 45

A Rigel instrumentatio 47
   A.1 Get LTTng ............................................... 48
   A.2 Get Babeltrace .......................................... 48
   A.3 Instrument the code ..................................... 48

B Installation ...................................................... 49
   B.1 Get the Simple Build Tool .............................. 50
5.1 Filter execution time ........................................... 38
5.2 Filter + Cut execution time .................................... 39
6.1 Gantt diagram ................................................... 43
A.1 Instrumentation ................................................... 48
List of Tables

5.1 Profile output .................................................. 32
List of Code Samples

1. Declaration of a generic class `TypePrinter` ........................................ 13
2. Usage example of instances of a parametrized class `TypePrinter[_]` 13
3. Pattern matching of values of type `Int` ............................................. 15
4. Simple case class declaration ............................................................... 15
5. Simple case class declaration ............................................................... 16
6. Definition of a trait `TypePrinter` with abstract type member `T` ....... 16
7. Implementation of abstract type members .............................................. 17
8. Interface of the Vector DSL ................................................................. 18
9. Implementation of the Vector DSL ........................................................ 20
10. C++ code generator for Vector DSL ..................................................... 21
11. Vector DSL example application .......................................................... 22
12. C++ code generated for the Vector DSL example application ............ 22
Domain Specific Language (DSL) are small languages develop to solve problems in a specific domain. Implementation of a DSL was not trivial and the developers’ effort was not justify because the number of users of the DSL was small. Platforms for DSLs development reduce this effort make their implementation relatively simple.
1. Introduction

1.1 Motivation

Domain Specific Languages (DSLs) are known for being small languages designed to only solve problems from a particular domain. Some years ago the effort that implied to develop a DSL was important and in some occasions this effort was not justify because the number of final users of the DSL was small. Nowadays, this effort is smaller because the use of special platforms for DSL implementation.

A DSL can take some advantages in front of library in a general purpose language:

1. DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs

2. DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes

3. DSLs enhance productivity, reliability, maintainability, and portability

4. DSLs allow validation and optimization at the domain level

1.2 Objectives

The main objective of this final master thesis is to illustrate the potential of Lightweight Modular Staging (LMS) [27] a platform for DSLs developing. To do that, we have developed Rigel an use case of DSL that analysis huge execution trace in Common Trace Format (CTF) [4] format.

To expound the power of LMS Rigel has to accomplish the following requirements:

- **Simplicity**: Rigel has to provide an easy way to apply the typical trace analysis operations.

- **Efficiency**: Rigel has to implement some compile optimizations to show the advantages of use LMS
1. Introduction

- **Flexibility:** Introduce modification on Rigel have to be simple and also the addition of new features.

1.3 Contribution of this project

This project presents Rigel, an use case of DSL for huge trace execution analysis that illustrate the potential of use a DSL development platform to generate C++ code with modest effort.

1.4 Document structure

The rest of this document is structured as follows: Chapter 2 gives a briefing over the state of the art on DSLs development, execution trace format and frameworks to trace analysis. Following on, Chapter 3 introduces the tools used to develop the project. The next chapter, Chapter 4, explains the syntax of Rigel and how to use it. To continue, the use cases chosen to validate Rigel are shown in Chapter 5. Chapter 6 details the development cost of the project, its planning, the methodology we followed and its social/economic impact. Chapter 7 concludes with some insight about the whole experience and research results obtained from the work done, overviewing some of the future lines of research this project opens.
Thanks to DSL development platforms the implementation of DSLs are growing considerably. This chapter presents some DSLs developed from different domains. In addition, there are multiple options of trace formats so some of these formats will be explained. Finally, some tools for analyzing traces will be presented.
2. Related Work and State of the Art  Rigel: A DSL Use Case for Trace Analysis

2.1 Domain-Specific Language frameworks

2.1.1 Lightweight Modular Staging

The Lightweight Modular Staging (LMS) library [27] [5] is a Scala library composed by a set of functions and type definitions that allow the programmer to define stages [29] throughout the translation process of a language. Each stage represents a translation of the application code in which the system applies the optimizations corresponding the application’s domain. The last stage is the application execution with a specific input, so all the data is defined and the program computes its results. Please note that LMS is a library written in Scala, so a DSL application is compiled together with the DSL implementation, giving place to an executable code generator. Then, once the code generator is run, the DSL application gets translated. LMS allows a modular quick development of all the part composing a DSL: Representation (language user interface), expression (representation semantics or language implementation) and code generation, where the application is finally translated to the next stage. The LMS library requires an experimental version of the Scala compiler called Scala Virtualized Compiler [10]. This compiler allows to redefine all the syntax elements of the language, thus LMS is able to change integrally the Scala default semantic for its purpose. LMS will be explained in proper detail in Chapter 3.

2.1.2 Jet

Jet is an embedded DSL for Big Data processing [11]. It has the same syntax and semantics as regular Scala with a few restrictions. It provides a high level, declarative interface similar to Spark, and it applies compiler optimizations and domain specific ones to generate highly performance code. Jet is designed in a modular and extensible way, allowing to add new modules for specific programs. The code is also portable, the same program can be compiled for use on Hadoop or on Spark.
2. Related Work and State of the Art  

2.1.3 OptiML

OptiML is a DSL for machine learning [28]. OptiML language focuses on describing what an operation should do, rather than how it should do it, deferring the how to the language implementation and runtime. OptiML describes ML operations using restricted semantics and data structures that generate efficient parallel and heterogeneous code. In this section, we describe the design and the key features of OptiML. This DSL provides the link between ML applications and heterogeneous parallel hardware.

2.1.4 Delite

Delite [3] is a Scala library built on top of LMS with the purpose of adding parallelism concepts to the LMS DSL implementation constructs. The Delite framework allows to define the semantics of DSL operations in terms of parallel constructs that run on accelerators or centralized multiprocessors by generating CUDA, MPI and C++ code. One of the most popular DSLs implemented with Delite is OptiML [28].

2.2 Trace formats

2.2.1 CTF: Common Trace Format

Common Trace Format (CTF) develop by Efficios [4]. It is a binary trace format designed to be very fast to write without compromising great flexibility. It allows traces to be natively generated by any C/C++ application or system, as well as by bare-metal (hardware) components.

With CTF, all headers, contexts, and event fields written in binary files are described using a custom C-like, declarative language called the Trace Stream Description Language (TSDL) [15]. Numerous binary trace stream layouts may be described in TSDL thanks to CTF’s extensive range of available field types.
2. Related Work and State of the Art  Rigel: A DSL Use Case for Trace Analysis

2.2.2 MTF2: MPI Trace Format

MTF2 [12] is built with expressiveness and scalability in mind. Expressiveness is achieved by providing support to a wide spectrum of concepts of the HPC trace domain. These concepts include additional MPI operations, user-defined functions, communication patterns, and process topologies. The objective is to enable the use of MTF2 in a broad range of applications with the hope of facilitating its adoption.

2.2.3 Paraver trace format

Paraver trace-file [30] is a set of three textual files containing the application activity (.prv), the labels associated to the numerical values (.pcf) and the resource usage (.row).

The trace-file is defined in ASCII and is composed by a header and a body. The header describes the process and resource model objects and the body contains the ordered list of records. There are three different types of records: Enter/Leave events for routine calls, atomic events for capturing performance counters information, and communication events for point-to-point and collective communication events.

2.2.4 OPT: Open Trace Format

OTF [18] uses different streams (files) to represent trace data for HPC applications. A stream corresponds to one process in the program. Each stream contains definitions for the trace events such as the routine names, the MPI operations used in the trace file as well as the information regarding the processes and the MPI communicators in the application. The definitions of the traces are followed by the events traced in the program.
2. Related Work and State of the Art  Rigel: A DSL Use Case for Trace Analysis

2.3 Performance analysis

2.3.1 FrameSoc: Trace Management Framework

FrameSoc [16] is a trace management infrastructure that provides solutions for trace storage, data access, and analysis flow, managing analysis results and tools. It addresses the issue of huge trace storage by using a relational database. Several pragmatic motivations led us to this choice. First, a database separates the logical data- model from the physical representation of data. Furthermore, thanks to accurate modeling and normalization, information is stored with minimal redundancy. Then, we can easily access parts of the trace or filter noise by using trivial querying. Search operations can be optimized by defining indexes: this mechanism is flexible and not limited to time or space dimensions. Finally, complex computations on trace data can be performed in the database, instead of loading the whole trace in memory and do such computations at the application level.

The core of this database solution is the generic data- model. It represents trace metadata, trace raw data and analysis results, with related tools metadata. The central entity of the model is the trace, which has metadata and can be related to files (e.g., configuration files, platform description).

2.3.2 Paraver performance analyzer

Paraver [26] performance analyzer is based on traces with a great flexibility to explore the collected data. It was developed to respond to the need to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. It supports both shared-memory and distributed-memory parallel applications.

It has three major components: a tracing facility, a trace merge tool, and a visualizer. Paraver trace includes detailed information such as states, events and communications.
2.3.3 TAU

TAU [19] is a program and performance analysis framework. It includes a suite of static and dynamic tools that form an integrated analysis environment for parallel applications. TAU includes automatic instrumentation to capture data for functions, methods, basic blocks, and program statements. In addition to automatic instrumentation, TAU provides an API for manual instrumentation. TAU can be used for either profiling (collecting cumulative data) or tracing (recording time-stamped events). TAU includes a visualizer, Paraprof, for profile data. For trace data, TAU does not include its own trace viewer, but it can convert trace files for use with other visualization tools, such as Paraver. TAU has many features and has been ported to a variety of platforms.

2.3.4 Linux Trace Toolkit next generation

The Linux Trace Toolkit: next generation [6], is an open source system software package for correlated tracing of the Linux kernel, user applications and libraries. LTTng consists of kernel modules (for Linux kernel tracing) and dynamically loaded libraries (for user application and library tracing). It is controlled by a session daemon, which receives commands from a command line interface. LTTng produces correlated kernel and user space traces, as well as doing so with the lowest overhead amongst other solutions. It produces trace files in the CTF format.

2.3.5 TraceCompass

Trace Compass [22] is a Java tool for viewing and analyzing any type of logs or traces. Its goal is to provide views, graphs, metrics, etc. to help extract useful information from traces, in a way that is more user-friendly and informative than huge text dumps. It can be integrated into Eclipse IDE or used as a standalone application (RCP). It also provides an extensible framework written in Java that exposes a generic interface for integration of logs or trace data input, analyses and views. Custom text or XML parsers can be added directly from the graphical interface by the user and also can handle traces that exceed available memory.
2. Related Work and State of the Art  Rigel: A DSL Use Case for Trace Analysis

TraceCompass supports different formats like CTF, hardware traces, GDB traces, Best Trace Format (BTF)...
Tools

To develop our use case Rigel, it was essential an intensive study of Scala, the host language to develop Rigel, and of LMS, the DSL development platform using in this TFM. In addition, Rigel use CTF format and it was necessary an important research into all the tools that allow work with this format, the most relevant were: LTTng trace, Babeltrace and Babeltrace API.
3. Tools 

3.1 Scala

Scala [23] is a statically typed, multiparadigm programming language with type inference. It supports both functional and object-oriented programming paradigms. It also supports concurrency with the actor model. Scala has a lot of mechanisms that allow the programmer to write libraries that are used like they were built-in features provided by the language itself. Thus being attractive for embedding DSLs.

3.1.1 Objects and classes

Scala provides classes, objects and traits. These language constructs are used to implement abstract data types. Objects are created through class instantiation. Scala provides singleton objects - they act as group of static functions (they operate on no particular implicit instance). If a class and an object share the same name they are called companion classes/objects. Companion objects and classes have access to each other’s private members. Scala allows declaring values, methods and types as abstract. Abstract type members are used to build generic components.

Abstract classes cannot be instantiated and its abstract members need to be defined in their subclasses. This is achieved through the extends keyword. Subclasses inherit non-private members of its superclasses. In Scala, multiple inheritance is not allowed, meaning classes cannot inherit from multiple superclasses.

3.1.2 Traits and mixin composition

Traits in Scala are the basic unit of code reuse. Traits are declared in a similar way as regular classes with the exception that the programmer uses keyword trait instead of keyword class.

Scala classes can mix in several traits with the keywords extends and with. By mixing in several traits into classes Scala provides stackable composition. The order in which traits are mixed in matters. Calls to methods on the superclass
or mixed in traits through the super call, are determined by linearization rules. Hierarchies of superclasses and mixed in traits are ordered in a linear way by the Scala compiler. Whenever super is invoked, the next method from the formed chain is called.

### 3.1.3 Generics

Scala supports type parametrization by introducing the concept of generic classes. It allows the programmer to define types that can operate in a type-safe manner on values without relying on their types. Examples of definition and use of generic classes can be seen on Code 1 and Code 2.

```scala
class TypePrinter[T: Manifest] {
  def print() = manifest[T].toString
}
```

**Code 3.1:** Declaration of a generic class `TypePrinter`

```scala
val i = new TypePrinter[Int]
i.print
val d = new TypePrinter[Double]
d.print
```

**Code 3.2:** Usage example of instances of a parametrized class `TypePrinter[_]`

On Code 2, a generic class `TypePrinter[_]` is declared. The implementation of this class was prepared for some, unspecified type T (line 1). Generics implement parametric polymorphism, meaning the implementation and interface of generics `TypePrinter[_]` is not coupled with any concrete type such as `Int` or `String`. This coupling is achieved when a generic type is instantiated and a type is specified.

Scala’s `Manifest` is used here as implicit value (line 1). Manifests are also generic, preserving type information, so they can be used at runtime. Such wrapper for type information is needed, because the JVM implements type erasure:
Type information is erased by the compiler and becomes unavailable at runtime for compatibility with previous versions of Java that did not support generics.

The generic class `TypePrinter[_]` contains only one member: A method called `print` that returns the name of the type as a String. This is achieved through a call to its generic function `manifest[_]` in line 2.

Code 2 shows how generic classes can be used. In line 1 the instance of the class `TypePrinter[_]` is parametrized with type `Int`. So, by parameterizing the generic class with a concrete type, such as `Int`, another type is constructed, namely `TypePrinter[Int]`. `TypePrinter[_]` is a type constructor. When method `print` is called in line 2, it returns the string "Int". A similar situation takes place for the code in lines 4-5, but, in this case, the output is string "Double". Please note that in this Section we are only scratching the surface of the features generics can implement. They also have a wide variety of purposes such as implementing higher-kinded types [20] and Concept Patterns [25].
3.1.4 Pattern matching

One of the functional features that Scala implements is pattern matching. Scala allows the programmer to match values of any type with a match-first policy. However, Scala, as an object-oriented language, extends this concept for objects. This is achieved with a special kind of classes called case classes. Let us look at the example Code 3:

```scala
def findRoom(n: Int): String = n match {
  case 103 => "Lab"
  case 105 => "Dean"
  case 104 => "Secretary"
  case _ => "Empty"
}
```

Code 3.3: Pattern matching of values of type Int

Function `findRoom` (line 1 on Code 3.8) takes a parameter `n` of type `Int`. This parameter is pattern-matched against several cases (lines 2-5). If `n` is equal to one of these values, the corresponding string is returned. Line 5 is the default pattern, which, if reached, is always executed.

As previously mentioned, Scala allows the programmer to pattern-match case classes. The definition of a case class is shown on Code 4.

```scala
case class Rectangle(x: Int, y: Int, w: Int, h: Int)
```

Code 3.4: Simple case class declaration

The only difference between case classes and regular Scala classes is that case classes come by default with a constructor with the same name as the case class. Regular Scala classes cannot be pattern-matched. Code 5 shows how case classes are pattern-matched.
3. Tools

Rigel: A DSL Use Case for Trace Analysis

```scala
def assertRectangle(r: Rectangle): String = r match {
  case Rectangle(0, 0, _, _) => // When rectangle is at (0,0)...
    // More cases...
  case Rectangle(_, _, w, h) => // Use w and h to...
}
```

Code 3.5: Simple case class declaration

Code 5 illustrates how case classes can be pattern-matched against values of their members (line 2). Values of case class members can be bound and used (line 4).

Case classes act like regular Scala classes in terms of class hierarchies. This means that an instance of a superclass can be matched against instances of case classes implemented as subclasses derived from a given type.

### 3.1.5 Abstract type members

Subsection 3.3.1 shows how types can be abstract members of a class. This is another way of building abstractions in Scala (next to type parametrization, described briefly in Section 3.3.3).

Abstract type members, as it was the case with generics and type parametrization, allow the programmer to abstract interfaces over implementations. The source code on Code 6 and 7 implement the same functionality as the one on Code 1 and 2: They print out the names of types Int and Double.

```scala
trait TypePrinter {
  type T
}

implicit val m: Manifest[T]

def print() = m.toString
```

Code 3.6: Definition of a trait TypePrinter with abstract type member T
class Integrals extends TypePrinter {
    type T = Int
    val m: Manifest[T] = implicitly
}

class Doubles extends TypePrinter {
    type T = Double
    val m: Manifest[T] = implicitly
}

val i = new Integrals
i.print

val s = new Strings
s.print

Code 3 7: Implementation of abstract type members

Firstly, a trait is defined so that it contains an abstract type member. We can see this in line 2 on Code 6. This type member is not defined, it will be defined in classes that mix in the TypePrinter trait.

In line 4 an abstract value \( m \) of type \( \text{Manifest}[T] \) is declared. This value is defined in classes that mix in the trait. As it was the case with examples in subsection 3.3.3, we want to avoid type erasure by using manifests.

The type member \( T \) is defined in classes Integrals and Doubles (lines 2 and 8 on Code 7). Calls to the method `print` in lines 14 and 17 give the same results as calls on Code 2.

Abstract type members can be used to build more powerful abstractions apart from what is just needed for this project. See [24] for more details.

3.1.6 TypeTags replacing Manifest

The previous sections have shown some examples on how Scala Manifests, which are directly mapped to Java Manifests, are used to work around type erasure. However, Manifest is currently deprecated as runtime type information mechanism, and is being replaced by a feature call TypeTag, which operates directly with types as the compiler understands them, not as raw strings. Nevertheless, Manifests are used for this project because one of the main libraries used, LMS, makes heavy use of it on its current version.
3. Tools

3.2 Lightweight Modular Staging

LMS [5] [27] is a Scala library for dynamic code generation. The main idea behind LMS, staging [17], is based on the observation that some computations can be lifted to the stages in which they will be performed less frequently or more information is available. The staging approach, although introduced initially as a set of compiler transformations, can be thought of as a method for embedding domain specific languages [14]. LMS is just a library, so a DSL application is compiled together with the DSL implementation, giving place to an executable code generator. Then, once the code generator is run, the DSL application gets translated. Building DSLs with LMS can be broken down into the following phases:

1. Defining interface of the DSL
2. Implementing operations performed by DSL
3. Implementing code generator for the DSL to target language

We will use the simple Vector DSL as an example for explaining the steps for building DSLs. The DSL provides a generic type VT[_] and a set of operations such as vector addition or dot product. For brevity this section will focus on addition of two vector and multiplication of vector by a scalar.

Interface of the DSL

```scala
trait VectorOps extends Base {
  class VT[O]
  def infix_+[O:Manifest](x: Rep[VT[O]], y: Rep[VT[O]]) = vector_plus(x,y)
  def infix_*[O:Manifest](x: Rep[VT[O]], y: Rep[O]) = vector_tms_scalar(x,y)
}
```

Code 3.8: Interface of the Vector DSL
The `VectorOps` trait on Code 8 defines the interface of our DSL, which consists of types and operations the programmer will use in the application. The basic type of the language is `VT[_]` represented here by an empty Scala class (line 2). In lines 4 and 5, operations for adding two vectors (`infix_+`) and multiplying vector elements by a scalar (`infix_*`) are defined. The bodies of these methods consist on calls to abstract methods `vector_plus` (line 4) and `vector_tms_scalar` (line 5). These abstract methods will be implemented in a later phase of the process. They bridge operations defined by the interface with their actual implementations.

The arguments taken by method `infix_+` are of type `Rep[VT]`, not `VT`. `Rep[+T]` is a higher-kinded type that represents a staged computation. For example, type `Rep[Int]` indicates that the staged computation will result in type `Int` in the next stage [27]. `VectorOps` is a trait mixed in with `Base`. `Base` is a part of LMS library.

**Implementing DSL operations**

Code 9 shows the exact implementation of the Vector DSL for this example. The implementation forms an expression tree [27].

The implementation is defined in trait `VectorOpsExp` (line 1). It implements the interface of our DSL, `VectorOps`. The trait `Expressions` provides an infrastructure for implementing an Intermediate Representation (IR) of our DSL as expression tree. The tree contains IR nodes of our DSL.

The IR nodes are implemented as case classes (lines 2, 3 and 4). These classes take arguments of type `Exp[_]`. This LMS type constructor represents constants and symbols. Each case class extends a parameterized type constructor `Def[_]`. This LMS type constructor represents definitions. Symbols are bound to definitions. The design choice of LMS is that each composite construct (such as case class `VtPlus`) refers to its parameters through symbols. Symbols are globally numbered so no redundant code occurs in the target application. These mechanisms also allow for some optimizations that will not be discussed here, such as common subexpression elimination.
3. Tools  Rigel: A DSL Use Case for Trace Analysis

```scala
trait VectorOpsExp extends VectorOps with Expressions {

  case class VtPlus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]])
    extends Def[VT[O]]
  case class VtTmsScalar[O:Manifest](x: Exp[VT[O]], y: Exp[O])
    extends Def[VT[O]]
  case class AXPY[O:Manifest](a: Exp[O], x: Exp[VT[O]], y: Exp[VT[O]])
    extends Def[VT[O]]

  def vector_plus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]]) = (x,y) match {
    case (x, Def(VtTmsScalar(y1,a))) => AXPY(a, y1, x)
    case (Def(VtTmsScalar(x1,a)), y) => AXPY(a, x1, y)
    case _ => VtPlus(x, y)
  }

  def vector_times_scalar[T:Manifest](x: Exp[VT[O]], y: Exp[O]) =
    VtTmsScalar(x, y)
}
```

**Code 3.9: Implementation of the Vector DSL**

Lines 6-12 show the body of methods `vector_plus` and `vector_times_scalar`. On Code 8 these methods were declared abstract. This is the way an implementation is bound to its interface. However, this binding is loose, meaning interface and implementation of a DSL are implemented as separate traits. The DSL implementer is able to provide several implementations for the same interface.

In lines 2-4, definitions are implemented as Scala case classes. They can be pattern matched. This in turn allows the implementation of optimizations such as rewriting (lines 6-10). Here the IR node `AXPY` is emitted if any of the arguments passed to the `+` operator is the result of a vector-scalar multiplication (lines 7 and 8). In other case (line 9) we emit a regular addition IR node.
DSL code generator to target language

Code 10 shows the code generator to C++ of the Vector DSL. The code generator is implemented as a trait that extends LMS’ trait CGen (line 1). Trait CGen extends the core code generation traits with code generators for regular C code, so we do not have to redefine regular assignment or looping, for example. The core code generation traits provide an infrastructure for dynamic code generation such as method quote, which obtains the symbol corresponding to a given particular definition.

```
trait VectorCppGen extends CGen {

  val IR: VectorOpsExp
  import IR._

  override def emitNode(sym: Sym[Any], rhs: Def[Any]): Unit = rhs match {
    case VtPlus(x, y) => emitValDef(sym, quote(x) + " + " + quote(y))
    case VtTmsScalar(x, y) => emitValDef(sym, quote(x) + " * " + quote(y))
    case AXPY(a, x, y) => emitValDef(sym, quote(x) + ".axpy(" + quote(y) + ", " + quote(a) + ")")
    case _ => super.emitNode(sym, rhs)
  }
}
```

Code 3 10: C++ code generator for Vector DSL

In lines 2 and 3, there is the information about the intermediate representation and the implementation of the DSL interface. The import statement in line 3 injects the types of IR nodes into the scope of the code generator so it can refer to them through pattern matching (body of method emitNode in lines 6-9).

While LMS is traversing an expression tree of our DSL, each IR node is pattern-matched (lines 6-9). If the definition is matched (left-hand side of a case statement in line 6), C code is generated for a given IR node (right-hand side of the statement). If no match is found, the symbol and definition of the IR node are passed forwarded to the next mixed-in trait (or superclass) on the chain.
An example application

```scala
object VectorDSLApp {

  trait SimpleApp {

    val v = Vector(1, 2, 3)
    val w = Vector(2, 5, 2)
    val scaledv = 5 * v
    val sum = v + w
    val sum2 = scaledv + w
    show(sum * sum2)
  }
}
```

Code 3 11: Vector DSL example application

For simplicity, we will not show all the details of the operations not described in this section (like the `Vector(...)` constructor or the `show` operation. For a complete documentation about LMS, see the referenced material and its official website [5]. By the way, the code emitted after running the generator of the application is shown in Code 12.

```cpp
#include <iostream>
#include "vectorDataStruct.h"

int main() {

  Vector x0(1, 2, 3);
  Vector x1(2, 5, 2);
  Vector x2 = x0+x1;
  Vector x3 = x0.axpy(5, x1); // Pattern recognized and optimized
  Vector x4 = x2+x3;

  for (size_t i = 0; i < x4.len(); ++i) std::cout << x4[i] << " ";
  std::cout << std::endl;
}
```

Code 3 12: C++ code generated for the Vector DSL example application
3. Tools

3.3 Scala-Virtualized

LMS allows programmers to stage any kind of computation resulting on an object of a particular type. However, the vanilla Scala compiler does not provide us with staging of control flow constructs such as if-else statements or loops.

The functionality of staging control flow is achieved through an extended Scala compiler called Scala-Virtualized [10]. The control flow constructs are compiled down into regular method calls. So, Scala statement if(a) b then c is compiled into function call __ifThenElse(a, b, c). Statement return a is compiled into __return(a). LMS requires all these features and thus DSLs need to be compiled with Scala-Virtualized. More examples and implementation details can be found in [27] and [21].

3.4 Linux Trace Toolkit: next generation (LTTng)

The Linux Trace Toolkit: next generation [6] was written by Mathieu Desnoyers after he started to maintain the predecessor called "Linux Trace Toolkit" since 2005. It offers both user space, as well as kernel space tracing. It claims to trace with a very low impact on the system. It is able to output the recorded data directly to disk or to the network.

It is an open source system software package for correlated tracing of the Linux kernel, user applications and libraries. LTTng consists of kernel modules (for Linux kernel tracing) and dynamically loaded libraries (for user application and library tracing). It is controlled by a session daemon, which receives commands from a command line interface.

LTTng is designed from the ground up to provide low overhead tracing on production systems and it provides flexible configuration options that can accommodate the system’s workload. Architectures such as x86, PowerPC, ARM and MIPS are supported, amongst others.
In addition, LTTng makes it possible to record multiple traces concurrently with different configuration options. Each user may create and configure as many tracing sessions as needed.

3.5 Babeltrace

Babeltrace project has been develop by Efficios and it provides trace read and write libraries, as well as a trace converter. Plugins can be created for any trace format to allow its conversion to/from another trace format [1].

The main format expected to be converted to/from is the Common Trace Format (CTF). The default input format of the "babeltrace" command is CTF, and its default output format is a human-readable text log. The "babeltrace-log" command converts from a text log to a CTF trace.

Babeltrace also provides an API [2] that allows handle the CTF traces in C/C++. In this Final Thesis it was used to obtain the main information of the trace: event name, time-stamp, CPU id, thread id, size of the message sent or received and number of threads in the application.
As it has mentioned in previous sections, the implementation of Rigel tries to illustrate the potential of use a DSL development platform, to do it, Rigel analyze huge execution traces with some typical operations for obtaining relevant information about the behaviour of the application. These operations are thought to be simple and flexible. This chapter explain the syntax of Rigel.
4. Rigel

4.1 CTF trace

Rigel works currently with CTF traces. Before starting work the user has to load the trace in the DSL as follows:

```scala
val execution_trace = newTrace("ctf_trace")
```

The `newTrace` constructor has the following specification:

```scala
def newTrace (path: String): CTF
```

Where "ctf_trace" is the path of the CTF trace. The return type of `newTrace` is `CTF` that is a class defined in the CTF C++ library that allows works with the information in the trace, to do that, CTF library use the Babeltrace API.

4.2 Event operations

A part from the CTF type, Rigel works with special type `Event` also defined in the C++ library. Immediately after it is explained how `Event` type is implemented in Rigel.

4.2.1 Event creation and Event fields

In Rigel the user can find some operations that give information about the traces. These operations work with type `Event` presented above. An Event defines an event that is the smallest unit in a trace.

```scala
val e = newEvent
def name = e.name
def ethreadId = e.threadId
def ecpuId = e.cpuId
def esize = e.size
def etstamp = e.tstamp
```
4. Rigel

Constructor `newEvent` creates a new `Event`, `e.name` gives the name of the previous event, `e.threadId` returns the thread that has executed the event `e`, `e.cpuId` indicates in which cpu the event has been executed , `e.size` returns the size of the message in case of a MPI call like a Send, Recv, ISend... Finally `e.tstamp` gives the time-stamp of the event.

All the fields have to be defined by the user during the instrumentation of the application except the cpu id and time-stamp that are fields defined by the CTF format.

4.3 Map and Pair operations

To instrument an application, to use Rigel it is necessary place one tracepoint at the beginning of the occurrence that we want to capture and another at the end of it, that means that we will have two tracepoints for one occurrence. We suggest the user to instrument in this way because by default the CTF format gives the time-stamp of each tracepoint and with the initial time-stamp and final time-stamp we can obtain the execution time of the occurrence without using more counters. Consequently we decided to define in Rigel our own `Pair` and `Map` type.

4.3.1 Pair

```scala
val p = newPair
define firstElement = p.first
define secondElement = p.second
```

Constructo `newPair` creates an empty `Pair`, `p.first` and `p.second` returns the first and the second element of the `Pair`. 
4. Rigel

4.3.2 GetPairs

```scala
val pairs = getPairs (trace)
```

As it was mentioned each occurrence in the user application is represented by two tracepoints that produce two events for a single occurrence. To combine each of these events, the user can use function `getPairs` that group the two events in a single pair.

4.3.3 Map

```scala
val m = newMap
val first_key = m.firstKey
val first_value = m.firstValue
val last_key = m.lastKey
val last_value = m.lastValue
```

The constructor `newMap` create an empty `Map`. Functions `firstKey` and `firstValue` returns the first key and the first value of the map, `lastKey` and `lastValue` returns the last key and the last value of the map.
4. Rigel: A DSL Use Case for Trace Analysis

4.3.4 ForEach

```
foreach(pairs,((p => m.make(p.second.tstamp-p.first.tstamp,p.first))))
```

Where `pairs` is a vector of pairs built with the start and end event of each call, `p` is a `Pair` and `m` is a Map with the key of type `Boolean` and value of type `Event`. The second parameter is the function that the user wants to apply to the pairs; as in the filter it is possible implement it in a very simple way. In this case the user get the information of the pair of events and then stores the total execution time computed by `p.second.tstamp-p.first.tstamp` as key of a map. The value of the map is the rest of the information in one of the events.

4.4 Trace operations

4.4.1 Filter

```
val filter = execution_trace.filter(_.name == "NameToFilter")
```

Filter returns a trace filtered with the conditions that the user decides on. The condition is passed to `filter` using a lambda expression, therefore the users is totally free to apply whatever condition that they want.

4.4.2 Cut

```
val cutTrace = cut(trace,ini_t,end_t)
```

Operation `cut` cuts the trace based in time. Where `trace` is the trace that will be cut, `ini_t` the minimum time to start the cut and `end_t` the maximum time to finish it. This function returns a new `CTF` only with the information between `ini_t` and `end_t`. 


After having introduced the features of Rigel in the last chapter now is the moment of showing a basic and complete example of it used, to do that we will use some different use cases: profiler, most used function, least used function and filter with and without optimization. This chapter also presents the effects of the optimizations applied by Rigel.
5. Results

5.1 Trace to analyze

To do the experiments we have used to different traces for practical reasons. For simple examples we used a trace obtained after instrument a MPI Jacobi implementation; this generated a small trace that is enough to illustrate how the user can use the different functions.

The other trace uses was bigger than the previous one and it was generated from the execution of Lulesh [7]. We chose Lulesh because it is able to generate a trace that it cannot be stored in memory completely. This trace will be used to show the comparison of using an optimize filter and use of filter without optimization.

5.2 Profile

One of the most typical use case in trace analysis is obtained the profile of the application to check the behaviour of each of the events. Table 5.1 show the output that Rigel gets.

```scala
def Profile = {
  val execution_trace = newTrace ("ctf_trace")
  val pairs = get (execution_trace)
  val m = newMap
  foreach(pairs,((p => m.make(p.second.tstamp-p.first.tstamp,p.first))))
}
```
5. Results

### Rigel: A DSL Use Case for Trace Analysis

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Function name</th>
<th>Thread ID</th>
<th>Size of message (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>MPI_Send</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>0.002</td>
<td>MPI_Recv</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>0.002</td>
<td>MPI_Send</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>0.006</td>
<td>MPI_Recv</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>0.011</td>
<td>MPI_Reduce</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.012</td>
<td>MPI_Send</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>0.013</td>
<td>MPI_Reduce</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.021</td>
<td>MPI_Send</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>7.648</td>
<td>MPI_Reduce</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10.502</td>
<td>MPI_Recv</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>10.122</td>
<td>Worker</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11.602</td>
<td>MPI_Recv</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>11.672</td>
<td>Worker</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>18.836</td>
<td>Coordinator</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5.1: Profile output**

### 5.3 MUF/LUF: Most Used Function/Least Used Function

One of the typical use cases of trace analysis is providing the Most Used Function of the application. Rigel allows the user to obtain by using some of the operations explained in the previous chapter.

```scala
def Operations = {
  val execution_trace = newTrace("ctf_trace")
  val pairs = get(execution_trace)
  val m = newMap
  val e = newEvent
  foreach(pairs, ((p => m.make(p.second.tstamp-p.first.tstamp,p.first))))
  val muf_k = m.lastKey
  val muf_v = m.lastValue
}
```
5. Results

In case that the user wants to obtain the Least Used Function the user has to change the call to `lastKey` and `lastValue` by `firstKey` and `firstValue`.

### 5.4 Filter

Rigel allows to apply the number of filters that the user wants.

```scala
def Operations = {
  val execution_trace = newTrace("ctf_trace")

  val filtered = execution_trace.filter(_.name == "start_MPI_ISend")
  .filter(_.threadId == 2).filter(_.tstamp > 10.0)
  .filter(_.tstamp < 9.0)
}
```

In case that the optimization of the filter is not available Rigel will call to `filter` three times, therefore, the trace will be rounds three times, it implies that the lineal cost of the `filter` execution will multiply by three. A pseudo code of the generated code without optimization is show below.

```scala
1 def Operations = {
2   val execution_trace = newTrace("ctf_trace")
3
4   val filtered = execution_trace.filter(_.name == "start_MPI_ISend")
5   .filter(_.threadId == 2).filter(_.tstamp > 10.0)
6   .filter(_.tstamp < 9.0)
7 }
```
5. Results

Rigel: A DSL Use Case for Trace Analysis

```c
int main(int argc, char** argv) {
    CTF trace ("ctf_trace");
    function<bool(Event)> fun1 = [&](Event e) {
        ......
    };
    CTF filteredTrace1 = trace.filter(fun1);
    function<bool(Event)> fun2 = [&](Event e) {
        ......
    };
    CTF filteredTrace2 = filteredTrace1.filter(fun2);
    function<bool(Event)> fun3 = [&](Event e) {
        ......
    };
    CTF filteredTrace3 = filteredTrace2.filter(fun3);
    function<bool(Event)> fun4 = [&](Event e) {
        ......
    };
    CTF filteredTrace4 = filteredTrace3.filter(fun4);
    return 0;
}
```

But in this case filter optimization is available, Rigel will detect that execution_trace has been filtered previously and instead of doing three calls to filter function, it will call only once, avoiding to go three times over the trace. In huge traces this represents reading the data from disk only once because large traces generated cannot be stored in memory. A pseudo code of the generated code with optimization is shown in below.
5. Results

Rigel: A DSL Use Case for Trace Analysis

```c
int main(int argc, char** argv) {
    CTF trace("ctf_trace");
    function<bool(Event)> fun1 = [&](Event e1) {
        ......
    };
    function<bool(Event)> fun2 = [&](Event e2) {
        ......
    };
    function<bool(Event)> fun3 = [&](Event e3) {
        ......
    };
    function<bool(Event)> fun4 = [&](Event e4) {
        ......
    };
    CTF x26 = x0.filter(fun4 && fun3 && fun2 && fun1);
    return 0;
}
```

To apply filter optimization, Rigel use pattern matching and group all the functions that the user wants to apply to the trace.

```python
def filterEvents (executionTrace: Exp[CTF], f: Exp[Event] => Exp[Boolean]) =
    (executionTrace, f) match {
        //Replace FilterEvents by a combination of functions f1 and f2
        case (Def(FilterEvents(executionTrace, f1)), f2) =>
            FilterEvents(executionTrace, fun({ e => doApply(f1, e) && f2(e)}))
        //Otherwise, just apply the function f to the trace
        case _ => FilterEvents(executionTrace, fun(f))
    }
```

The cost of the application of the `filter` continues being lineal but in this case Rigel will call `filter` only once.
5. Results  

5.4.1 Cut optimization

In case that the user wants to cut the trace for a specific window time and then apply a filter, Rigel applies code motion.

```scala
def Operations = {
    val execution_trace = newTrace("ctf_trace")
    val filtered = execution_trace.filter(_.threadId == 2)
    val cutter = filtered.cut(4,8)
}
```

Rigel detects that `execution_trace` has been previously filtered and reorganizes the code to apply the cut first. Following, it is shown the pseudo code of generated code:

```java
int main(int argc, char** argv) {
    CTF trace("ctf_trace");
    function<bool(Event)> x4 = [&](Event x1) {
        //......
    };
    CTF cut_trace = trace.cut (4,8);
    CTF filtered_cut_trace = cut_trace.filter(x4);
    return 0;
}
```

As in filter optimization, cut optimization is also done using pattern matching.

```scala
def cutTrace(executionTrace: Exp[CTF], ini_t:Exp[Int],end_t:Exp[Int]) =
    (executionTrace,ini_t,end_t) match{
        //Reorganize the code, first call cut and then filter
        case (Def(FilterEvents(executionTrace,f)),ini_t,end_t) =>
            FilterEvents(CutTrace(executionTrace,ini_t,end_t),f)
        //Otherwise, just apply the cut
        case _ => CutTrace(executionTrace,ini_t,end_t)
    }
```
5. Results

5.4.2 Combined Filter and Cut optimizations

Rigel can apply both optimizations at the same time. In this case, Rigel first applies a code motion and afterwards it groups filter operations.

```scala
val execution_trace = newTrace("ctf_trace")
val filtered = execution_trace.filter(_.name == "start_MPI_Send")
  .filter(_.threadId == 2).
  filter(_.tstamp > 10.0)
  .filter(_.tstamp < 9.0)
val cutter = filtered(4,8)
```

In the pseudo code of the generated code it can be observed that the result is a combination of the optimized cut and optimized filter.

```cpp
int main(int argc, char** argv) {
    CTF trace("ctf_trace");
    function<bool(Event)> fun1 = [&](Event e1) {
        ......
    };
    function<bool(Event)> fun2 = [&](Event e2) {
        ......
    };
    function<bool(Event)> fun3 = [&](Event e3) {
        ......
    };
    function<bool(Event)> fun4 = [&](Event e4) {
        ......
    };
    CTF cut_trace = trace.cut (4,8);
    CTF cut_filtered_trace = cut_trace.filter(fun1 && fun2 && fun3 && fun4);
    return 0;
}
```
5. Results

5.5 Performance Comparison

For practical reasons, traces use to do the experiments presented in this work are not so big than a normal HPC execution trace, however, they are enough to illustrate the potential of Rigel optimizations.

5.5.1 Filter

![Filter Execution Time]

Figure 5.1: Filter execution time

In 5.1 it can be appreciate that for a small trace (1.5GB) the optimization reduce the filter execution time almost by half. Therefore it can be guess that for huge traces this difference will be higher.
5.5.2 Combined Filter and Cut

As it can be observe in 5.2 the optimized version is better but in this case the difference is smaller than in the previous case.
Plan and Cost Analysis

In this chapter, the most formal aspects of this project’s management are explained. The first main section goes through the professional context and the social impact of the project. Then, the next section details the planning we followed along the almost four months of development. Finally, the last section shows the overall cost of the project and justifies its amortization in the middle term.
6. Plan and Cost Analysis

Rigel: A DSL Use Case for Trace Analysis

6.1 Context

Since this project consists in showing the power of DSL development platforms to apply compile optimizations in a simple way providing a use case of huge trace execution analysis, its context is closely bound to intensive programming in HPC systems. More specifically, the context of the project is research an efficient way of analyze huge traces produce by HPC applications. The following subsections describe the kind of users and professionals this project is intended to involve in.

6.1.1 Final Users

The final users of our DSL are domain experts, which are able to focus just on their specific domain problems, without having to deal tasks such as parallel programming or software design at all. The fact that they can just specify their systems almost literally translating from their research articles, maximizes the time they spend on solving their actual problems, and ultimately reduces their frustration, hopefully making them happier.

6.1.2 Indirect Users

Indirect users of Rigel are the actual developers of the language. Historically, domain experts and computer scientists have worked together in order to come up with supercomputing applications that solve domain problems. This is a very costly process, since it implies that the domain expert needs to be aware of what the computer scientist terms are and vice-versa. By agreeing on a common language layer, Rigel developers can work on a bounded set of features they can quickly understand and that will satisfy the domain expert’s requirements.

6.2 Planning

The development of the Rigel prototype was composed by nine different and clear stages. This chapter explain the following plan to develop Rigel and each of the stages of the project.
6.2.1  **Go in depth in Scala language - 4 weeks**

This first stage consisted in obtaining a thorough knowledge of the Scala language. By recommendation of supervisors, the Scala course taught by Martin Odersky was completed [9].

6.2.2  **Study of the CTF format and trace tools - 4 weeks**

After achieving a solid knowledge of Scala, we studied the CTF format: how we have to instrument the application, how we can handle the information of the trace and the tools available to do it.

6.2.3  **Develop a C++ library to handle CTF traces - 6 weeks**

Once CTF format and the trace tools were deeply studied, it was necessary to develop a library in C++ that allowed work with the information.

6.2.4  **Study of LMS - 4 weeks**

The study of LMS was a crucial part of this thesis because it is the main tool to develop the DSL. Some DSLs examples were developed in this stage.

6.2.5  **Interface design - 4 weeks**

This stage consisted in the selection and design of the operations that the DSL will have.

6.2.6  **Code generation - 3 weeks**

After the interface was designed, the next stage was implement the code generation.

6.2.7  **Optimization implementation - 3 weeks**

Using the results of the previous stage, we studied in which cases LMS could apply an optimization by using pattern rewriting.
6. Plan and Cost Analysis

6.2.8 Test and results - 2 weeks

Once we had the complete package we generated a big trace to compare and to test our DSL.

6.2.9 Documentation - 2 weeks

Finally, the last two weeks of the project were dedicated to writing this document, its presentation and revising it over and over again before the final delivery.

Below there is the Gantt diagram of this planification.

![Gantt diagram](image)

Figure 6.1: Gantt diagram

6.3 Action plan

This section will analyze the dependencies between the tasks just mentioned along the previous section and will try to estimate a minimum/maximum completion time for the whole project.

6.4 Methodology

This final section shows the details on our methodology and work flows to coordinate management and development.
6.4.1 Development

All the software this project is composed by was developed by Beatriz, the student presenting this project. Scala IDE was used for editing Scala code and Simple Build Tool (SBT) [8] for building the Rigel code generator and manage its dependencies.

6.4.2 Directive feedback

The student met the directors of the project weekly to ensure the project was progressing adequately and following the schedule. Since the code, calendar and planning was stored in our shared Redmine server, the directors had access to the GIT source code repository, so they could check out all the changes/additions the student made in the software at any time.
Conclusions and Future Work

This final chapter presents the conclusions obtained after developing this project and some possible future works that will make evolve further on Rigel.
Using LMS, the effort of implementing a DSL is considerable minor that to start from scratch, however, this does not become in a trivial task because it is necessary a deep knowledge in the LMS platform and how is a relative new tool is not well documented yet.

Nowadays there are a lot of different DSL for a variety of topics (Jet for Big Data [11], OptiML for matching learning [28], TraceContract for communication trace analysis [13]...). With Rigel we want to illustrate the potential of developing a DSL. We chose the use case of analyzing huge execution traces because this field has not been explored yet.

The version of Rigel we present in this document is an example of how a DSL can help to improve the application of compile optimization for that reason provides very basic features to analyze traces. However this research has demonstrated that by using simple features the user is able to obtain important information to do a performance analysis.

Once the potential of a DSL for huge trace analysis has been illustrated, there are some research lines that we can following to develop a complete DSL for traces analysis one of them is the use of hardware counters to study the behaviour of the cache, also, explore the communication between processes is really interesting. Finally another future feature that could be relevant is the used of Rigel with different trace formats, like OTF or Paraver, to increase its flexibility.

The main objective of this Final Master Thesis is show the power of develop a DSL using LMS a platform that allows to develop DSL in a flexible, scalable and simple way. To do it, it has presented Rigel an use case of DSL that analysis huge execution traces that implement some simple operations for trace analysis and also some compile optimizations.
Rigel instrumentatio
To use Rigel it is necessary to instrument the code that the user wants to analyze, this Appendix show how do that.

A.1 Get LTTng

First of all, the user has to install LTTng tracing framework and Babeltrace, both tools are available for different Linux distributions like: Ubuntu, Fedora, OpenSuse, Debian and so on. For instructions on how install LTTng and use it in a user application go to [6].

A.2 Get Babeltrace

To handle the CTF trace in a C++ code in necessary use Babeltrace API [2] provides the compress file necessary to download. The last version: babeltrace-1.24, uploaded 2014-11-04 was used in this Master Thesis.

A.3 Instrument the code

To use Rigel the user has to instrument the code using two `tracepoint` for each event, this is necessary to compute the execution time of the event using timestamp file provided by CTF format. The instrumentation would be like this:

```
tracepoint(lulesh, start_MPI_Irecv, myRank, 0, recvCount);
MPI_Irecv(domain.commDataRecv[pmid * maxPlaneComm],
recvCount, baseType, fromRank, msgType,
MPI_COMM_WORLD, domain.recvRequest[pmid]);
tracepoint(lulesh, end_MPI_Irecv, myRank, 0, recvCount);
```

Figure A.1: Instrumentation
Installation
B. Installation

B.1 Get the Simple Build Tool

SBT is a powerful build system easy configurable by just writing Scala scripts. The LMS library uses this tool to build and install itself on any system and make it available to any other project being built by SBT. For instructions on how to get and install SBT, go to [8].

B.2 Getting and installing LMS

The following steps will download and install LMS assuming you have SBT already installed on your system and available in your path by directly referring to it with the sbt command.

1. Clone the LMS GIT repository under /home/you/lms:
   https://github.com/TiarkRompf/virtualization-lms-core.git

2. Get inside the LMS folder

3. Switch to a branch called develop

4. Compile and install LMS to the SBT library folder by running sbt publish-local

Now LMS is ready to be used by any project compiled with SBT.
output.brf


B. Installation Rigel: A DSL Use Case for Trace Analysis


B. Installation

Rigel: A DSL Use Case for Trace Analysis


Acronyms

API Application Programming Interface
BSC Barcelona Supercomputing Center
CTF Common Trace Format
DSL Domain Specific Language
HPC High Performance Computing
IR Intermediate Representation
LTTng Linux Trace Toolkit next generation
LMS Lightweight Modular Staging
MPI Message Passing Interface
MTF2 MPI Trace Format 2
OTF Open Trace Format
SBT Simple Build Tool