Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis Proposal

On expressing different concurrency paradigms on virtual execution systems.

Cristian Dittamo

Supervisor
Dr. Antonio Cisternino

January 6, 2008

Abstract

The notion of virtual machine has permeated every aspect of computing systems, as witnessed by the ever growing set of virtual execution systems targeted by programming languages such as `Java`, `C#`, `Perl` and `Python` just to mention few of them. Virtual machines are appreciated not only because of portability of computer programs across different architectures; the ability of monitoring the program execution has proven important to enforce security aspects as well as tailoring execution onto specific architectures. Moreover, dynamic loading and reflection allow programs to adapt their execution depending on several environment factors, including the underlying computing architecture.
Recently microprocessor architectures are shifting from the original `Von Neumann` computational model, and including different forms of concurrent computation that cannot be hidden to the executing program; important examples are the programmable graphical processors (`GPU`), and the `Cell BE` architectures. Virtual machines provide an abstract computing model that provides heap, stack-based operations, shared memory, and multi-threaded concurrency. Although it is possible to imagine a Just In Time compiler that maps programs for the virtual machines to the underlying architecture, the program will not contain enough information to efficiently bridge the gap between computational paradigms that may differ significantly.
In this PhD work we want to investigate general approaches to bridge this gap, by providing suitable programming abstractions not affecting the general structure of the virtual machine, but consent control over the particular underlying architecture.

Contents

Introduction

In the last few years, the number of languages, applications and frameworks based on Virtual Machines (VM) has increased. Some examples are the VMPlants [10], a grid service for automated configuration and creation of flexible VM execution environments, and the Muskel [34] parallel programming library that provides users with structured parallel constructs (skeletons) usable for the implementation of efficient parallel applications. Muskel applications run on networks/clusters of workstations equipped with Java. Other examples are the CLR[1]-based frameworks such as Windows Presentation Foundation (WPF) [35], a graphical subsystem feature of the .NET Framework 3.0 that provides a unified programming model for building Windows applications that incorporate UI, media, and documents, and provides a clear separation between the UI and the business logic; the Windows Communication Foundation (WCF) [36], which is a new communication subsystem to enable applications on one machine or across multiple network-connected machines to communicate. Another example is OpenOffice [37], a free office suite based on the Java virtual machine (JVM).
One of the reason of this success is due to the (extensible) *reflection model* provided by execution environments, such as JVM and CLR. It enables dynamic access to the representation of the application, and allows the program to change its behavior while running depending on its current execution state [11, 18]. This eases cross-platform interaction and allows software to be portable between various operating systems (OP).
Although VMs in the form of *abstract machines* have been around for a long time (since the mid-1960s), the advent of Java has made them a common technique for implementing new languages, particularly those intended for use in heterogeneous environments.
A *virtual machine* was originally defined by Popek and Goldberg [1] as *"an efficient, isolated duplicate of a real machine"*. *Real* machines, i.e. computer systems, are built on levels of *abstraction*, where a higher level of abstraction hides details at lower levels (e.g. files are an abstraction of a disk). VMs enable software *virtualization* a host platform, which is similar to *abstraction* except that details are not necessarily hidden. There are many types of VMs, but the common theme between them is the idea of *virtualizing* an instruction set. Popek and Goldberg in [1] provide the notion of virtual machine map (VMMap), where any virtual instruction sequence corresponds to a real instruction sequence. This map was proposed for the conventional third generation computers. We believe that this *mapping* can be applied to the current generation of (heterogeneous) computers. The problem is how to construct such a map in order to obtain the best performance for each architecture: modify the existing VM model or just extend it? Is it possible to make a

[1]Microsoft .NET Common Language Runtime implementation of standard ECMA 335 [20] Common Language Infrastructure (CLI)

general mapping model?
Strongly typed execution environments (STEEs), such as JVM [22], CLR, Shared Source Common Language Infrastructure (SSCLI) [21], and OCaml [8], implement and expose to programmers a multi-threaded, shared memory, stack-based virtual machine and offer the opportunity to be extended by multi-stage [7] and meta-programming [6] capabilities that can be exploited by the languages that share such VMs (i.e. runtime). For example, in [4] the reflection support provided by STEE is extended allowing the programmers to mix code written in the same language, thus giving the impression that the meta-programming system performs source to source code transformations, whereas in reality the code manipulation is performed at binary level.
Modern microprocessor architectures have considerably changed from the past: from the classical *Von Neumann* architecture to the modern *Data-Flow* architecture of the Graphical Processing Unit (GPU) and the multi-core architecture of the Cell Broadband Engine (Cell BE) microprocessor [26]. The shift to multi-core processors that is currently underway provides an increase of 10 to 100 times today's computing power. Each one of these microprocessors is completely different from the other both in architecture and programming model. For example, GPU programmable parts consist of an array of *vertex processors* and an array of *pixel processors* to compute the position of vertices and the final color of each pixel respectively. Each processor has constants and temporary registers. However, there is no temporary memory. GPUs have a SIMD[2] programming model (moving towards a SPMD[3] model) that has to face the lack of resources available and has a limited support for primitive types typically found on CPUs.
In order to guarantee portability, VMs provide a single view to all these architectures by means of different implementations of the STEE. This view provides a shared memory, multi-threaded programming model along with synchronization mechanisms for managing *race conditions* such as locks. It is the Just-In-Time (JIT) compiler that is in charge of translating the intermediate machine-independent code into native CPU code at runtime. Therefore, there is a JIT compiler for each computer architecture supported by a VM. Each specific JIT compiler emits optimized code based on its knowledge of the specific instructions (e.g. SSE2[4]) provided by the ISA of the host CPU.
In the last few years, we've observed a progressive branching between the programming models provided by VMs and the ones provided by modern microprocessors. We call the problem of filling this gap the **mapping problem**. Researchers have recently tried to reduce this gap. For example, Tarditi, Puri and Oglesby [23] have developed a library, named Accelerator, that uses data parallelism to program GPUs for general purpose uses under .NET CLR. The main advantage is that no aspect of the GPU is exposed to the programmers, only high level data-parallel operations: the programmer must use specific data(-parallel) types to program the GPU instead of the CPU.
Another example is the RapidMind Development Platform [30] that allows the developer to use standard C++ to create concurrent/parallel applications that run on GPUs, the Cell BE, and multi-core CPUs. It provides a single unified programming model, adding types and procedures to standard ISO C++ and relying on a dynamic compiler and runtime management system for parallel processing. However, Accelerator only extends the APIs

[2]Single Instruction Multiple Data
[3]Single Program Multiple Data
[4]Streaming SIMD Extensions 2, is one of the IA-32 SIMD introduced by Intel with the initial version of the Pentium 4 in 2001.

allowing programmers to perform computations on GPUs. Whereas in RapidMind, there is a reliance on a specific programming language along with extension of the APIs. Actually both don't introduce a new programming model to solve the mapping problem.
We believe that by leveraging meta- and multi-stage programming features and STEE features, such as CLR, SSCLI or Mono, it will be possible to map the different programming models, by:

- providing a single and unified programming model without losing expressivity nor forcing the use of a single source language, because SSCLI is a *cross-language* VM as well as *cross-platform*;
- staging the compilation, optimization, and specialization processes by introducing programming constructs for controlling the generation process and by allowing programmers to continue developing and debugging at source level;
- providing a mechanism to supply specific operations of the runtime host architecture at a higher level (i.e. VM level) in order to permit more control over the computation to programmers.

Outline of the work

This thesis proposal is organized as follows. In Chapter 1 we present the state of the art in Virtual Machine programming models, providing some useful details about STEEs. We then focus the discussion on meta-programming, runtime bytecode generation, and multi-stage programming techniques which allow to extend STEEs with features useful for maximum performance exploitation of new microprocessors architectures. For each of these techniques we argue relative features, advantages and disadvantages.
Chapter 2 details our proposal. We outline some ideas about how to solve the mapping problem.

Chapter 1

Basic concepts

In the following sections we introduce the state of the art in programming models both VMs and microprocessors. These basic concepts will be useful to delineate the (virtual-to-real machine) **mapping problem**.

1.1 Virtual Machine programming model

Virtual Machines, in the form of *abstract machines*, have been around for a long time (since the mid-1960s), but the advent of Java has made them a common technique for implementing new languages, particularly those intended for use in heterogeneous environments. There are many types of VMs, but the common theme between them is the idea of *virtualizing* an instruction set. Each VM uses a *virtual instruction set* (VIS) mapped to the *real instruction set* (RIS) of the computer.
Currently the VMs are separated into two major categories, based on their use and degree of correspondence to any real machine: **System virtual machine** (SysVM) and **Process virtual machine** (ProcVM). The former provides a complete system environment constructed at ISA level which supports the execution of a complete operating system (OS). On the other hand ProcVMs, sometimes called *application virtual machine*s, are designed to run a single program, which means they support a single process. Their purpose is to provide a platform-independent programming environment that abstracts the details of the underlying hardware or OS, so that every program executes in the same way on every platform.
One ProcVM subclass is the **High Level Language VM** (HLLVM). Its main goal is complete platform independence for applications. The major difference between ProcVM and HLLVM is the specification level: HLLVMs provide a Virtual ISA + APIs[1]. Major modern examples of HLLVMs are the Java JVM [22] and Microsoft Common Language Infrastructure (CLI[2]) [20]. The latter is also *cross-language* since it allows languages that target the CLI to be integrated with one another, so that it is possible to use data types of another language as if they were your own. CLI is an open specification[3] that has a number of implementations including CLR, SSCLI, and Mono. All of them provide a *type-oriented*, *multi-threaded*, *stack-based* VM, named Virtual Execution System (VES), that offers many services such

[1] Instead of ISA + OS interface with ProcVM
[2] Recently standardized by ECMA and ISO as well as the C# language specification [19]
[3] Published under ECMA-335[20] and ISO/IEC 23271) developed by Microsoft.

as dynamic loading, garbage collection and Just-In-Time (JIT) compilation.
An essential characteristic of a VM is that the software running on it is limited to the resources and virtualizations provided by the VMs; we focus on how to overcome this limit by providing more functionality based on runtime host architecture features.
In the remainder of this proposal, we consider the type oriented VMs only, named STEE, such as CLR, SSCLI and Mono, because their execution environments contain information about program types and their structure, and are able to reflect it to running programs.

1.1.1 Strongly Typed Execution Environment (Model)

A STEE is a VES which implements a VM providing an extensible type system and reflection capabilities. It is *strongly-typed* because it guarantees both that value types can always be established, and that values are only accessed by using the operators defined on them. The execution is *managed* since STEE has complete information about all aspects of a running program. This includes knowledge of: the state and liveness of local variables in a method; all extant objects and object references, including reachability information.
It provides direct support for a set of built-in data types, and defines a VM with an associated *machine model*, shown in (figure 1.1), a state, a set of control flow constructs, and an exception handling model. The purpose of the STEE is to provide the support required to execute the intermediate instruction set: IL for CLI and bytecode for JVM. The IL expresses a program executed by a thread. Each method invocation corresponds to the addition of a *stack frame* wich contains local variables and the input arguments.
As for many programming technologies and environments, such as OS with *processes* and the Java VM with the *class loaders*, the CLR defines its own unique model for scoping the execution of code and the ownership of resources, called *Application Domain* (AppDomain). AppDomain is a sub-process unit of isolation for managed code, which fills many of the same roles filled by an OS process since it scopes the execution of code, provides a degree of fault isolation, provides a degree of security isolation and owns resources on behalf of the programs it executes. However a process is an abstraction created by the OS, whereas an AppDomain by the CLR. A given AppDomain resides in exactly one OS process, whereas a given OS process can host multiple AppDomains.
The CLR has its own abstraction for modeling the execution of code that is conceptually similar to an OS thread. The CLR defines a type that represents a schedulable entity in an AppDomain, called *soft thread*. It is a construct that is not recognized by the underlying OS. A given AppDomain may have multiple soft thread objects. A one-to-one correspondence doesn't exist between soft threads and AppDomains. When a soft thread in one AppDomain calls a method in another AppDomain, the thread transitions between the two AppDomains. Whenever a hard thread winds up executing code in multiple AppDomains, each AppDomain will have a distinct soft thread object affiliated with that thread. When unloading an AppDomain, the CLR knows which threads are in it, and forces them to unwind out of it. Objects in one AppDomain can communicate with types and objects contained in another AppDomain. However, access to these types and objects is made only through well-defined mechanisms (e.g. .NET remoting). .NET exposes AppDomains to programmers via appropriate types in the APIs.

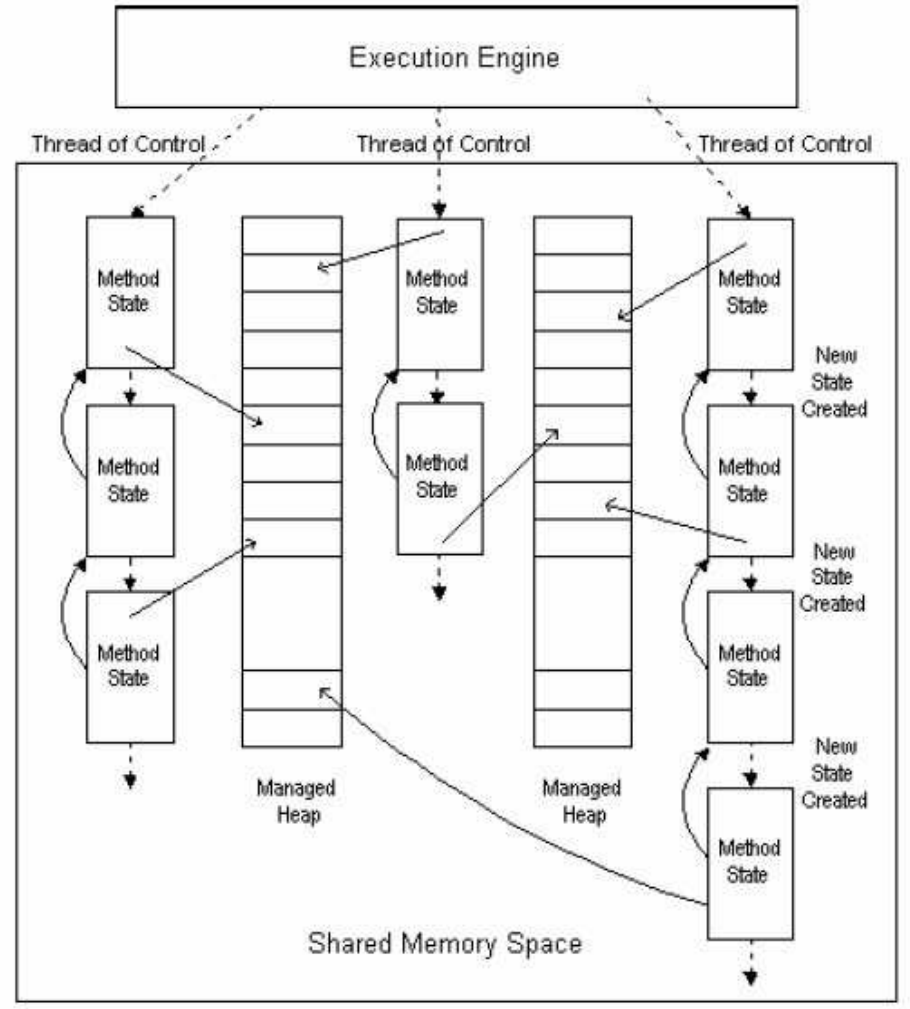

Figure 1.1: Machine state model, `VES`, manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space.

Just In Time compilation

`STEE` compiles the `IL` code using `JIT` into native `CPU` code. `CLR` (as well as `JVM` and `Mono`) adopts a *2-stage compilation*. At the *1st stage*, a compiler that targets the `CLR` forms program files (i.e. assembly) in a standard *machine independent* format containing both code (i.e. intermediate language (`IL`)) and *metadata*. At the *2nd stage*, the `JIT` compiler converts the `IL` as needed during execution and stores the resulting native code for subsequent calls. The loader creates and attaches a *stub* to each one of a type's methods when the type is loaded. On the initial call to the method, the stub passes control to the `JIT` compiler, which converts the `IL` for that method into native code and substitutes the stub with the native code location address. The choice of representing types and code in intermediate language form, rather than machine code, is somewhat constrained because of design goals. Without information on types it is almost impossible to have general

support for dynamic loading of modules, thus reducing reuse of software.
Furthermore, the compiler verifies the IL code and relative *metadata* it receives as input to find out whether the code is type safe or not, which means that it only accesses the memory locations it is authorized to access.
A key aspect of the CLR programming model is the heavy reliance on metadata.

Metadata and Reflection

Metadata contains all the information necessary to describe and reference types defined by the type system. It provides a common interchange mechanism for use between tools that manipulate programs (such as compilers, debuggers, and runtime code generators (RCG)), as well as between those tools and the VES. When types become a shared abstraction between the execution environment and the programming language a larger amount of information is made available about a program to the runtime and to all the other programs interested in code analysis. After compilation, programs may manipulate the output looking for special patterns inside the intermediate language, types and metadata [5]. Post processing may be done for several reasons: in [4] it is done for runtime code generation; in [12] the meta-program Particular uses metadata to make a sequential C# coded method parallel.
Another key aspect of the CLR is the (extensible) model for Reflection. It makes all aspects of a type's definition available to programs through metadata, both at development time and at runtime. The model is extensible because it supports arbitrary metadata attributes, named Custom Attributes (CA), without introducing new keywords into the programming language. In [12] a set of CAs are defined so that programmers can suggest which section of a method body must be analyzed for parallelization purposes.
In the following section, we illustrate how the execution environment can be extended, in order to overcome the limits of current STEEs.

1.1.2 Extending the STEE

STEEs, such as the CLR or SSCLI, offer the opportunity to be extended with meta-programming [6] and multi-stage [7] capabilities that can be exploited by the languages that share such a runtime.

Meta-programming and Runtime Code Generation

Meta-programming refers to the class of programs manipulating other programs, called *object programs* (OP). A meta-program may construct OPs, combine OP fragments into larger OPs, and observe the structure and other properties of OPs. Examples of meta-programs are compilers, partial evaluators, AOP[4] weavers, FOP[5] systems, and many others.
In [6] Sheard proposes a meta-program taxonomy whereof we'll introduce those aspects relevant to this proposal only. All meta-programs can be divided into two categories: *program analyzer*s and *program generator*s (RTCG[6]). The former observes the structure and environment of an OP and computes some value as a result (e.g. optimizers, partial evaluation systems, etc.). The latter is used to address a whole class of related problems, with

[4] Aspect Oriented Programming
[5] Feature Oriented Programming
[6] RunTime Code Generator

a family of similar solutions, for each instance of the category. It does this by creating an OP that solves a particular instance.
Furthermore, RTCGs can be further categorized into *manually* and *automatically* annotated. The body of a program generator is partitioned into static and dynamic code fragments. The static code comprises the meta-program, and the dynamic code comprises the OP being produced. Annotations are used to separate the pieces of the program. *Manually* annotated systems are those where the programmer places these annotations directly. Whereas *Automatically* annotated ones have the annotations placed by an automatic process.
We are especially interested in RTCGs since they have some useful benefits:

- *performance*, it is a common objective of many meta-programming system, since computations can be sped up by preprocessing (i.e. *partially evaluating*) some of the static data before consuming the remainder of the dynamic data;
- *code compaction*, some parts of code may become unnecessary at runtime;
- *code adaptation*, system can adapt itself to the host environment changes;
- *programming language expressiveness*, programmers should be able to express not only the programming language implementation, but its design as well.

Each language attempts to do this within its own domain and under its unique constraints. We believe that the potential to provide high-level abstractions previously unavailable to programmers is a RTCG area of application.
However, there are some important issues to be addressed within RTCG:

- *speed*, there is a tradeoff between generating code quickly or generating code that runs fast. Generating source code at runtime, through a runtime compiler, entails post-processing phases, such as transformations and register allocations, which requires time, thus preventing quick code generation. On the other hand, generating machine code directly is more complex, but potentially quicker to perfom, since an intermediate step is not required. Whatever the choice, it is hard to make RTCG generic and cross-platform compatible, since the machine architecture is involved in the code generation phase.
- *garbage collection* (GC) for generated code; most GC systems are designed to collect *data*, not *code*, which are often resident in different portions of memory; furthermore, there is the problem of keeping track of all different access methods of different kinds of variables.

Multi-stage programming

We leverage meta-programming to overcome the limitations of existing programming language targeting runtime such as CLR: either performance or expressivity problems. One way of overtaking them using meta-programming is *staging*. *Staging* is a program transformation that involves reorganizing the program's execution into stages. The goal is to improve a program based on a priori information about how it will be used. *Staged systems* are useful because in the life-cycle of a program there are different stages where computations can be performed to partially specialize a program. The idea comes from the observation that a result from a run of an early stage might be reused several times in

later stages, and the cushioned cost of generating specialized code is much smaller than that of the repeated computation.
A new execution model has been introduced whereby a program is processed by a sequence of processors which take as input an `OP` and generate a new `OP`. At each step, called *stage*, the input program is evaluated and manipulated. An example of that is the `Yacc` parser generator. *First*, it reads a grammar and generates its relevant `C` code; *second*, it compiles the generated program; *third*, the user runs this compiled program.
Multi-stage programming languages, such as `MetaML` [7], `MetaOCaml` [25], and `CodeBricks` [4], have been developed to provide a good basis for the multi-stage programming paradigm. These languages allow users to write code constructions directly in the form of source code instead of numerous data constructors for a specific "code" type. They are characterized by annotation mechanisms which explicitly specify the evaluation order of the various computations, namely which portion of the program should be executed, so that

$$\text{Stage program} = \text{Conventional Program} + \text{Staging Annotations}$$

The type systems of these languages guarantee that the generated code is well-formed, thereby guaranteeing that the entire execution will not fail.
Following is a description of some important languages and frameworks for meta-programming and code generation. For each of them we'll argue the main features.

Languages

MetaML

An example of `RTCG` is `MetaML` [7]. This is a homogeneous, manually annotated, runtime generation system. It guarantees the syntactic correctness in specifying `OP`s, and type (and semantic) correctness of `OP`s. It provides support for multi-stage programming by three types of staging annotations, static type-checking, a polymorphic type-inference, and static scoping for both meta-level and object-level variables. The staging annotations provided are: `brackets` (`< >`), which can be inserted around any expression to delay its execution; `escape` (`~`), that combines delayed computations, splicing-in the argument in the context of the surrounding `brackets`; `run`, for executing delayed computations, thus forcing a piece of code to be evaluated.
`MetaML` is based on two main principles: *cross-stage persistence* and *cross-stage safety*. The former allows the programmer to use a variable bound at the current level in any expression to be executed in a future stage. To the user, this means the ability to stage expressions that use variables defined at a previous stage. The latter, on the other hand, prevents developers from staging programs where a variable is used at a level lower than the level of the lambda-abstraction in which it is bound.
In `MetaML` the *cross-stage persistence* comes at a price. Because most compilers do not maintain a high level representation for values at runtime, being able to inject any value into the code type means that some parts of this code fragment may not be printable. So, if the first stage is performed on one computer, and the second on another, we must "port" the local environment from the first machine to the second. Since arbitrary objects, such as functions and closures, can be bound in this local environment, this can cause portability problems. Currently, `MetaML` assumes that the computing environment does not change between stages. Thus, `MetaML` currently lacks *cross-platform portability*.

F#

F# [14] is a multi-paradigm .NET language explicitly designed to be ML suited to the .NET environment. F# was modeled on Objective Caml (OCaml) [8] and then tweaked and extended to mesh with .NET. It fully embraces .NET and enables users to do everything that .NET allows: garbage collection, JIT compilation, interoperable generics, and .NET libraries.
Its main contribution is the integration of three programming language paradigms (functional - imperative - object oriented) with each other and with the type system imposed by the .NET virtual machine: F# types and code can be used directly from other CLI languages. Moreover it provides the ability to execute programs interactively and across multiple platforms since it is supported by Mono.
F# supports meta-programming[7] by means of F# *quotations* (<@ @>). The quote operator instructs the compiler to generate data structures representing code rather than IL. The F# parser and type checker statically guarantee the syntactic validity of quoted fragments and the typing of quoted literals. *Quotations* allow capturing of type-checked expressions as structured terms. They can be interpreted, analyzed and compiled to alternative languages. An interesting feature is that F# quotations can also be generated programmatically at runtime.
Quotations allow heterogeneous execution of F# programs since a single program written entirely in F# can run not only as .NET code, but also in various other environments. They make it possible to "take" part of the program, process it and execute it somewhere else. For example, in the following code a query comprehension is wrapped in <@ @> marks, which allows the SQL function to analyze the F# code and translate it to the appropriate SQL code.

```
let CustomersList =
  SQL <@ { for c in (db.Customers)
           when c.Country = "Italy"
           -> c } @>
```

This feature is very interesting for our aims since we want to process and analyze a high-level language implemented code and execute it where the best performance is provided; for example, matrix operations can be executed faster on a GPU rather than on a CPU.

Frameworks

Execution environments such as CLR and JVM provide many features needed by multi-stage programming languages, though there is no explicit support for them. Follows we present two examples of possible runtime extensions to provide support for multi-stage languages.

Jumbo

A tool for building runtime code generators for Java [17]. It provides a dynamic compiler for Java capable of pulling together fragments of Java code into a single program and compiling the combined program at runtime. The basic idea behind Jumbo is *compositional*

[7] It is still at an experimental stage.

compilation, i.e. the compilation of each language construct is a function of the compilation of its sub-constructs only. The advantage is that any particular piece of syntax can be easily abstracted from and filled in at a later time/stage. `Jumbo`'s main disadvantage is that it is tied to the `Java` language: it accepts plain source `Java` code as input and produces (virtually) the same code as the `javac` compiler.

CodeBricks

It provides a framework for code generation that allows programs to manipulate and generate code at source level, while the joining and splicing of executable code is carried out automatically at intermediate code (`VM` level). This framework introduces a data type `Code` to represent code fragments: methods/operators from this class are used to reify a method from a class, producing its representation as an object of type `Code`. `Code` objects can be combined by partial application to other `Code` objects. `Code` combinators, corresponding to higher-order methods, allow *splicing* the code of a functional actual parameter into the resulting `Code` object.
`CodeBricks` is a library implementing the framework for the `.NET CLR`, where code manipulation primitives are provided through a library rather than as a language extension, allowing cross language code generation as well as a more robust and maintainable approach than language extensions. Programs can be specialized more than once, for instance as more information becomes available to the program. Since no language processor is involved, specialization can happen at different stages throughout the lifetime of a program, even beyond the program build. Advantages of the `CodeBricks` approach are: the generated code is expressed and manipulated in a high-level language; no high-level language processor is required to run the generated code; efficient binary code is produced and code generation overhead is minimal; finally the solution is not tied to a programming language.

1.2 Microprocessors programming model

In this section we present some of the most important architectures and programming models of the modern microprocessors, so that we can outline the differences between them and with the `VM`s models.

1.2.1 Von Neumann architecture

The `von Neumann` architecture is a computer design model that uses a processing unit and a single separate storage structure to hold both instructions and data. This architecture is characterized by a sequential control flow resulting in a sequential instruction stream. This operating principle is still the basis for today's most widely used high-level programming languages, and more astounding, of the `ISA` of all modern microprocessors. The main goal of the `von Neumann` architecture, minimal hardware structure, is today far outweighed by the goal of maximun performance.
The separation between the `CPU` and `memory` leads to the *von Neumann bottleneck*; the limited *throughput* (data transfer rate) between the `CPU` and `memory` compared to the amount of memory. In modern machines, *throughput* is much smaller than the rate at which the `CPU` can work. This seriously limits the effective processing speed when the `CPU` is required to perform minimal processing on large amounts of data. The `CPU` is

continuously forced to wait for vital data to be transferred to or from memory. As CPU speed and memory size have increased much faster than the *throughput* between them, the bottleneck has become more of a problem. The performance problem is reduced by a memory hierarchy of *cache* between CPU and main memory, and by the development of *branch prediction* algorithms.
Future performance improvements will predominantly come from parallelism rather than from an ever-increasing uni-processor clock speed.

Multi-core CPU

The CPU speed is reaching a *bottleneck* because of the number of transistors that can be integrated on a chip. Solutions can be found in the future using nano technology, but in the short term using the dual- or multi-core machines, clustered CPUs, even grid computing and supercomputing. As we'll argue, GPUs face the same problem, but still have space to press on due to their task specific designs and parallelism paradigm.

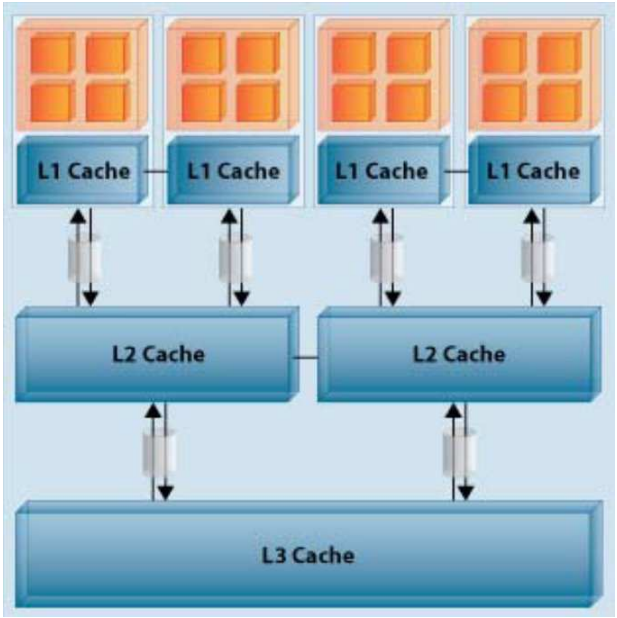

Figure 1.2: Multi-core CPU architecture

A multi-core CPU combines two or more independent *cores* into a single package composed of a single integrated circuit, called a *die*, as shown in figure 1.2. *Cores* may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD[28] dual-core processors). The processors also share the same interconnect to the rest of the system. Each "core" implements optimizations independently such as superscalar execution, pipelining, and multi-threading.
Software benefits from multi-core architectures because code can be executed in parallel. Under most common operating systems this requires code to execute in separate threads or processes. Each application running on a system runs in its own process so multiple

applications will benefit from multi-core architectures. Each application may also have multiple threads but, in most cases, this must be specifically written. However, programming multi-threaded code often requires complex co-ordination of threads and can easily introduce subtle and difficult to find bugs due to the interleaving of processing on data shared between threads (thread-safety). Consequently, such code is much more difficult to debug than single-threaded code when it breaks.
In some existing parallel programming models either shared-memory (such as `OpenMP`) or message passing (such as `MPI`[8]) can be used on multi-core platforms. They require specific knowledge on parallel computing to program these platforms because developers must manage non-functional aspects of the computation such as thread synchronization, scheduling, etc. In the past, many parallel programming paradigms were proposed to hide those aspects, freeing the programmers to focus on the computation features.

1.2.2 Data-Flow architecture

The *data-flow* principle states that an instruction can be executed when all operands are available. Such an execution is said to be *data-driven*. The parallelism in this architecture is limited only by the actual data dependences in the application program. Software written using a dataflow architecture consists of a collection of independent components running in parallel that communicate via *data channels*; such a design can be succinctly depicted graphically; they use *data-flow graphs* as their machine language, which specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instruction.
Data channels provide the sole mechanism by which nodes can interact and communicate with each other, ensuring lower coupling and greater reusability. *Data channels* can also be implemented transparently between processors to carry messages between components that are physically distributed.

Graphical Processing Unit

A Graphics Processing Unit (or `GPU`) is a dedicated graphics rendering device for a personal computer or game console. `GPUs` contain many parallel processing units, as shown in figure 1.3, and are capable of sustaining computation rates greater than ten times that of a modern `CPU`. They can in fact be thought of as highly parallel `Single Instruction Multiple Data` (`SIMD`) type processors. `GPUs` differs from multi-cores because the latters provide coarse-grain parallelism, heavyweight threads which gain better performance per thread, whereas `GPUs` provide more fine-grain parallelism and lightweight threads.
The `GPU` programming model, however, is very different from traditional `CPU` models. Recent advances in programmability and architectural design have enabled the use of `GPU` processors for general purpose computations (named `GPGPU`) such as linear algebra, geometric computing, database and stream Mining, `GPU` ray tracing, advanced image processing and others. The `GPU`'s rendering pipeline has become programmable both in geometry stage and rasterization stage. In order to use the `GPU` the problem is how to map general purpose computing onto the `GPUs` programming architecture. Traditional `GPGPU` programming has demonstrated some limitations, of which high-level programming by using specialized shading languages, no scatter support, lots of specific limitations on program complexity,

[8]Message Passing Inteface

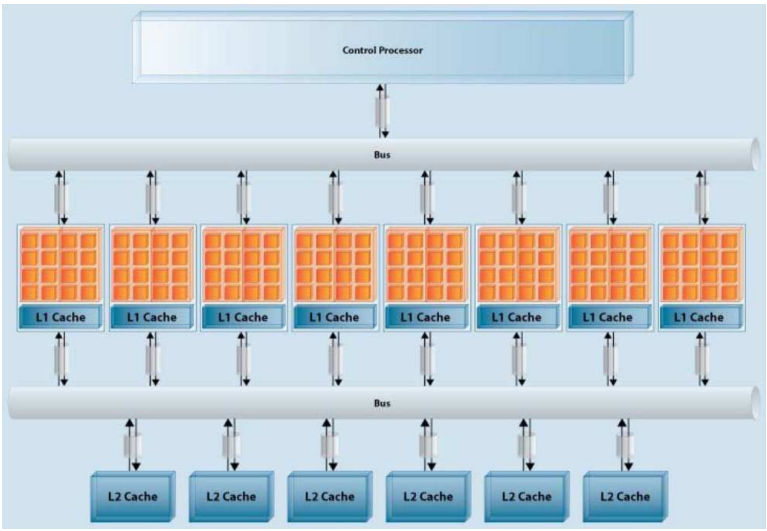

Figure 1.3: Nvidia GPU architecture, G80

no support for integer types, and limited support for arrays. Recently Nvidia has developed the CompUte Driver Architecture (CUDA) programming model, that is a general programming model where the GPU is viewed as a compute device that is a coprocessor to the CPU or host, has its own DRAM (device memory), and runs many threads in parallel.

1.2.3 Multi-thread hardware, multi-level caches

The Cell architecture grew from a challenge posed by Sony and Toshiba to provide power-efficient and cost-effective high-performance processing for a wide range of applications, including the most demanding consumer appliance: game console, such as Sony Playstation 3. Cell is not limited to game systems. IBM has announced a Cell-based blade, which leverages the investment in the high-performance Cell architecture. Other future uses may include HDTV sets, home servers, game servers and supercomputers.
Cell is a heterogeneous chip multiprocessor that consists of an IBM 64-bit Power Architecture core, augmented with eight specialized co-processors based on a novel SIMD architecture called Synergistic Processor Unit (SPU), which is for data-intensive processing, like that found in cryptography, media and scientific applications. The system is integrated by a coherent on-chip bus, as shown in figure 1.4. The main computing power of the Cell BE is provided by the eight *synergistic processor elements* (SPE). The SPE is a processor designed to accelerate media and streaming workloads. The local memory of the SPEs is not coherent with the PowerPC processor element PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using a Direct Memory Address DMA engine.
As for higher level programming model (HLPM), a Cell BE based architecture has the Octopiler compiler, that implements techniques for optimizing the execution of scalar code in SIMD units, using subword optimization and other techniques. It is also able to overlap data transfers with computation, by allowing SPEs to process data that exceeds the local memory capacity. Beside the other lower level optimizations, this compiler also enables the OpenMP programming model. This approach provides programmers with the

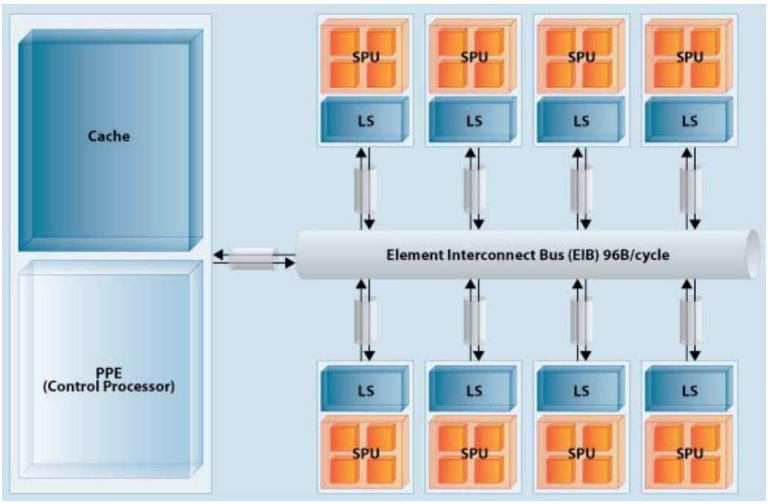

Figure 1.4: Cell BE architecture

abstraction of a single shared-memory address space. Using `OpenMP` directives, programmers can specify regions of code that can be executed in parallel.
Another `HLPM` example is the `Cell Superscalar framework` (`CellSs`). It provides a programming model the programmers write sequential applications and the framework is able to exploit the existing concurrency and use the different components of the `Cell BE` (`PPE` and `SPEs`) by means of an automatic parallelization at execution time. The only requirement we place on the programmer is that *annotations* (somehow similar to the *OpenMP* ones) are written before the declaration of some of the functions used in the application. The similarity with the `Octopiler` approach is that an *annotation* (or directive) before a piece of code indicates that this part of code will be executed on the `SPE`s.

1.3 Problems and Existing solutions

In the previous sections we saw how the `VM` programming model evolution has diverged from the microprocessor programming model. The result is a significant gap between the two, which results particularly evident when comparing the fast performance improvement of modern microprocessors with the slow one of the `VM`-based applications, as shown in the figure 1.3. In the past, `VM` programming models spent most of their effort to solve problems such as programmability, productivity, and portability. However, the new microprocessors performance capabilities require that a good programming technology should:

1. provide a more accurate conceptual model of the hardware;

2. clearly expose the most important policy decisions and architectural elements of the hardware;

3. provide structure and modularity, other than productivity and portability;

4. expose a suitable high level programming mechanism, such as code annotations, that allows exploiting specific underlying architecture features.

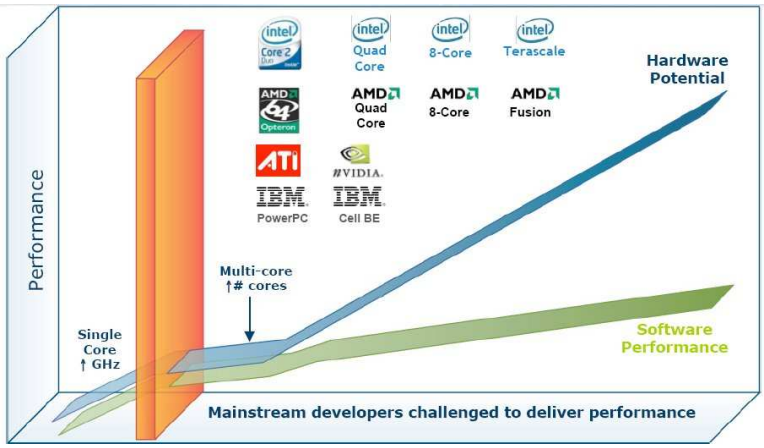

Figure 1.5: Progressive branching between hardware potential performance and actual software performance.

The key to performance improvement is therefore to run a program on multiple processors in parallel. Unfortunately, it is still very hard to write algorithms that actually take advantage of those multiple processors. In fact, most applications use just a single core and see no speed improvements when run on a multi-core machine. We need to write our programs in a new way. Some proposals have been made, both `VM`- and non VM-bsaed.

RapidMind

`RapidMind` provides a software development platform for multi-core processors, with a single-source solution for portable, high performance parallel programming based on the general-purpose programming model `SPMD`. It can be used to target `GPUs`, multi-core processors and `Cell BE` as well. It integrates with existing `C++` compilers, so programmers don't need to learn new languages or tools, but just use specific types, vectors and arrays. User directly specifies parallel algorithm using abstract parallel programming model, then the platform maps algorithm onto parallel execution mechanisms, selecting the most appropriate mapping for each architecture.
`RapidMind` supports two operating modes: *immediate* mode, which simply reflects standard practice in numerical programming under `C++`; and *retained* mode, where operations on specific types are recorded and dynamically compiled into a "*program object*" rather than being immediately executed. For example, in the case of `GPUs`, applying such a function to an array of values, causes he platform automatically invokes a massively parallel computation on the video accelerator.
`RapidMind` introduces two new stages between the compilation and execution. The first, named `RapidMind Collection`, takes, as input, a standard executable with embedded `RapidMind operations`, and transforms such operations into parallel computations that best exploit the host architecture. This trasformed code is taken as input by the second (new) stage, named `RapidMind compilation`, in order to produce platform specific code. In the following example, a sequential matrix multiplication `C++` routine is converted to a parallel implementation by using `RapidMind` framework.

```
#include <cmath>
const int w = 512, h = 512;
float f;
float a[w][h][4], b[w][h][4];
void compute() {
 for (int i = 0; i < w; i++)
  for (int j = 0; j < h; j++)
   for (int k = 0; k < 4; k++) {
    a[i][j][k] += f * b[i][j][k];
 }
}
```

Sequential version.

```
#include <rapidmind/platform.hpp>
#include <Rapidmind/shortcuts.hpp>
using namespace rapidmind
const int w = 512, h = 512;
Value1f f;
Array<2,Value4f> a(w,h), b(w,h);
Program compute_prog;
void init_compute_prog () {
 compute_prog = BEGIN {
   In<Value4f> r, s;
   Out<Value4f> t;
   t = r + f * s;
 } END;
}
```

Parallel version using `RapidMind`.

This is a very interesting proposal for solving the mapping problem, but it is tied up to the `C++` language, thus cross-language is not supported.

.NET Parallel Extensions

Lately Microsoft has introduced `Parallel Extensions` [31] to its `.NET Framework` technology. They run on `.NET FX 3.5`, rely on features available in `C# 3.0` and provides imperative data- and task-parallelism `API`s in a declarative way. The new concurrency runtime is used across the library to enable lightweight tasks and effectively map and balance the concurrency expressed in code to available concurrent resources on the execution platform.
A major component of the `Parallel FX` library is the `Task Parallel Library` (`TPL`) designed to write managed code that can automatically use multiple processors. The library uses *work-stealing techniques* for dynamic work distribution and automatically adapts to the workload and particular machine. Meanwhile, the primitives of the library only express potential parallelism, but do not guarantee it. For example, on a single-processor machine, parallel for loops are executed sequentially, closely matching the performance of strictly sequential code. On a dual-core machine, however, the library uses two worker threads to execute the loop in parallel. Unfortunately, the library does not help to correctly synchronize parallel code that uses shared memory. It is still the programmer's responsibility to ensure that certain code can be safely executed in parallel. Other mechanisms, such as `locks`, are still needed to protect concurrent modifications to shared memory.
For example, in the following code the outer `for` loop of the matrix multiplication is replaced by a call to the static `Parallel.For` method:

```
using System.Concurrency;

void ParMatrixMult(int size)
{
  double[size,size,4] a, b;
```

```
  double f;
  ...
  Parallel.For( 0, size, delegate(int i) {
    for (int j = 0; j < size; j++) {
      for (int k = 0; k < 4; k++)
        a[i, j, k] +=  f *  b[i, j, k];
    }
  };
}
```

The `delegate` takes the iteration index as its first argument and executes the unchanged inner loop body. No changes to the original loop body are necessary since delegates automatically capture the free variables of the loop body.
The library provides a *task manager* that, by default, uses one worker thread per processor, which ensures minimal thread switching by the `OS`.
The parallel patterns provided are: delegate-based parallel loop, which ensures the correct exceptions management by properly propagating them and by cancelling all iterations; *reduce* (called *aggregation*); *fork-join parallelism*, and *replicable tasks*.

Chapter 2

Research Proposal

In order to take advantage from the new architectures, presented in the section 1.2, the most effort has been spent on `VM API` extensions. As introduced in section 1.3, these are technological solutions which work well with the current architectures but that don't provide any new conceptual model, thus solution, for filling the models gap. They entail more mandatory framework extensions in the future whenever new architectures come, with an increase of `API` complexity which leads to loss of programmability, thus productivity.
For these reasons the aim of this thesis is to define a model that can map the `Virtual Machine` stack-based programming model to the different microprocessor (parallel) programming models in order to obtain the maximun performace available without expecting specific `HPC`[1] knowledge from mainstream developers. The problem is that virtual machines allow representing computations in such a way that they don't contain enough information to efficiently map them onto all underlying architectures. The basic idea is to provide a simple way for developers to express specific features (e.g. parallel execution) in their algorithms without losing control over the generated code, and being aware of the cost in term of performace hit and code transformations overhead. It will be the meta-programming framework in charge of the source code adaptations for each host architecture. Therefore in our opinion a new model should have the following features:

- **simple to code**, so that mainstream developers don't need to be aware of the microprocessors programming models, the parallel/concurrent programming paradigms and runtime code generation techniques;
- **syntax and semantic correcteness** of `OPs`, so that less controls need be performed at runtime, and best performance can be obtained;
- **completeness**, the `VM` *stack-based* model should be mapped to all parallel programming paradigms, such that developers can develop and debug the program in its sequential form, focusing on the functional aspects of the application;
- **maximun performance** for each microprocessor architectures;
- **scalability**, no assumption should be made neither on the specific microprocessor architecture features, which are evaluated during the runtime code generation, nor on specific parallel programming paradigm;

[1] High Performance Computing

- **cross-platform portability**, since there is a wide range of current available heterogeneus platforms both on `OSs` and `microprocessors`;
- **cross-language integration**, programmers should choose whatever programming language allows them to express their intentions most easily. Frameworks, such as the `CLI`, should be in charge of different languages integration.

The methodology that we advocate in our proposal merges the two wide areas presented in the previous sections into a unique programming framework. We propose to explore the existing meta-programming techniques over virtual machines since we must treat *code as data* in order to manipulate it as needed. There are many techniques for representing *code as data*. Many make code an `abstract type` in order to provide a more usable presentation of it. Since this representation hides the internal structure of code, some other interface to the internal structure of code is necessary, if code is deconstructed or observed as for our aims. Moreover we want generate code efficiently at runtime, so that it will be possible process and execute it somewhere else without performance decrease. Since specialization can happen at different stages through the lifetime of a program, especially in order to translate a `VM`-language code into microprocessor-language code, we rely on multi-stage programming. By using it, developers can inspect the code produced by their generators, so that it can be either printed or compiled, because a delayed computation maintains an intensional representation.
Therefore we consider to leverage on an existing programming language such as `F#` and the `CodeBricks` library which exploit the cross-platform and cross-language portability features of `.NET` framework. Although these tools help our investigation, we espect that results will not depend on them.
`F#` *quotations* allows obtaining data structures that represents the code with the security of quoted fragments syntactic validity by means of parser and type-checker static checking. The question now is, what exactly should they represent? For example, every parallel aspect of an algorithm so it can be preserved for later stage specializations based on specific microprocessor features. For instance, we can consider the well known *data-flow* model, where data structures should represent a data-flow execution graph of the algorithm with all intrinsic parallelism. Due to the array handling and representation problem raised by the data-flow model we propose to explore the *macro data-flow* model [38]. These data structures can be then interpreted, analyzed and compiled to alternative languages (e.g. at lower level, microprocessors programming languages).
The `CodeBricks` library provides support for multi-stage programming which allows expressing and manipulating the generated code in a high-level language, limiting the code generation overhead, and guaranteeing that only type-safe code-fragments can be produced.
As for the implementation, we consider to leverage on the `AppDomain` modularity for executing small *jit-compiled* program fragments on different microprocessor cores or `Cell SPE`s, having the main program and the `JIT` compiler in execution on central processing unit or `Cell PPE`. In addition we can take advantage of the parallel extensions to the `.NET` runtime to obtain the maximun performance from host architectures. This is an example of how it is possible to define a mapping among `VM` semantic objects and underlying computing elements.
In order to illustrate our idea, as introduced in section 1.1.2 on `F#`, we provide the following example. Developers should code their applications, for example to perform some

matrix operations, in a normal way without using any special types or extensions to the used language, but simply wrapping parts of the code between `<@ @>` marks.

```
let moltMatrices =
  <@ fun a b ->
    Matrix.init 512 512 (fun x y ->
     ((Matrix.get a x y) + (Matrix.get b x y)) / 2) @>
```

Then, after syntax analysis, the framework determins that this code can be executed faster on a `GPU` (using pixel and vertex shaders) rather than on a `CPU`, thus it translates it to `GPU` code and processes matrices on the graphics card.
In providing an adequate solution to the *mapping problem*, our aim is to follow several steps:

- In the first place, we evaluate the existing solutions to the computational models mapping problem.
- Subsequently, we consider how to represent programs as data to hide unnecessary details, yet make their important structure evident, and their common operations easy to express, and efficient to implement.
- The final crucial step will concern the study of a (new) general model that can map the `VM` stack-based model on all microprocessor architecture programming models.

To conclude, we argue a formal definition of that new model that will be correct, in the sense that the semantics expected by the transformation is satisfied, and complete.

Bibliography

[1] Popek, G.J.; Goldberg, R.P.: *Formal requirements for virtualizable third generation architectures*, Communications of the ACM, Vol. 17, Issue 7, New York, NY, USA, (1974)

[2] Smith, J.E.; Nair,R.: *The Architecture of Virtual Machines*, IEEE Computer Society, (2005)

[3] Demers, F.N.; Malenfant,J.: *Reflection in logic, functional, object-oriented programming: a short comparative study*, IJCAI, Workshop on Reflection and Metalevel Architectures and their applications, (1995), 29-38

[4] Cisternino, A.: *Multi-stage and Meta-programming Support in Strongly Typed Execution Engines*, Ph.D. thesis, Department of Computer Science, Pisa University, (2003).

[5] Attardi,G.; Cisternino, A.; Colombo,D.: *CIL + Metadata ¿ Executable Program*, in Journal of Object Technology, vol. 3, no. 2, Special issue: .NET: The Programmers Perspective: ECOOP Workshop 2003, pp. 19-26.

[6] Sheard, T.: *Accomplishments and Research Challenges in Meta-Programming (invited paper)*, Springer Berlin/Heidelberg, Lecture Notes in Computer Science, Vol. 2196, (2001), 2-44.

[7] Taha, W.; Sheard, T.: *MetaML and multi-stage programming with explicit annotations*, Theoretical Computer Science, Vol. 248, (2000), 211-242.

[8] Leroy, X.: *The Objective Caml system*, See http://caml.inria.fr/pub/docs/manual-ocaml/, (2007).

[9] Aycock, J.: *A brief history of just-in-time*, ACM Computing Surveys (CSUR), Vol.35, Issue 2, (2003), 97-113.

[10] Krsul, I.; Ganguly, A.; Zhang, J.; Fortes, J.A.B.; Figueiredo, R.J.: *VMPlants: Providing and Managing Virtual Machine Ezecution Environment for Grid Computing*, ACM/IEEE SC 2004 Conference (SC'04), (2004).

[11] Karsai, G.; Sztipanovits, J.: *A Model-Based Approach to Self-Adaptive Software*, Intelligent Systems and Their Applications, IEEE, Vol. 14, Issue 3, (1999), 46-53.

[12] Dittamo, C.; Cisternino, A.; Danelutto, M.: *Parallelization of C# programs through annotations*, proceedings of Practical Aspects of High-Level Parallel Programming Workshop (PAPP), "Computational Science ICCS 2007", LCNS 4488/2007, (2007), pp. 585-592.

[13] Shiel, S.; Bayley, I.: *A Translation-Facilitated Comparison Between the Common Language Runtime and the Java Virtual Machine*, Electronic Notes In Theroretical Computer Science, 141, Elsevier Ltd., (2005), 35-52.

[14] Syme, D.: *Leveraging .NET Meta-programming Components in F#*, The 2006 ACM SIGPLAN Workshop on ML (ML 2006), Portland, Oregon, (2006).

[15] Bawden, A.: *Quasiquotation in LISP (invited talk)*, ACM SIGPLAN Workshop on Partial Evaluation end Semantic-Based Program Manipulation, ACM, BRICS Notes Series, (1999), 4-12.

[16] Taha, W.: *Multi-Stage Programming: its theory and applications*, Ph.D. thesis, Oregon Graduate Institute of Science and Technology, (1999).

[17] Kamin, S.; Clausen, L.; Jarvis, A.: *Jumbo: Run-time Code Generation for Java and Its Applications*, Proceedings of the International Symposium on Code Generation and Optimization (CGO 2003), San Francisco, California, (2003), 48-56.

[18] Sestoft, P.: *Runtime Code Generation with JVM and CLR*, Unpublished, Available at http://www.dina.dk/sestoft/publications.html, (2002).

[19] ECMA International: *ECMA Standard 334: C# language specification*, See http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[20] ECMA International: *ECMA Standard 335: Common Language Infrastructure*, See http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[21] Microsoft Research: *SSCLI: Shared Source Common Language Infrastructure*, See http://research.microsoft.com/rotor.

[22] Lindholm, T.; Yellin, F. : *The JavaTM Virtual Machine Specification*, Sun Microsystems, (1999).

[23] Tarditi, D.; Puri, S.; Oglesby, J.: *Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism*, Technical Report MSR-TR-2005-184, Microsoft Research, (2005).

[24] Syme, D.; Margetson, J.: *The F# website*, See http://research.microsoft.com/fsharp, (2006).

[25] Taha, W. and other contributors: *MetaOCaml: a compiled, type-safe multi-stage programming language*, See http://www.metaocaml.prg, (2006).

[26] Pham, D.; Asano, S.; Bolliger, M.; Day, M. N.; Hofstee,H. P.; Johns,C.; Kahle,J.; Kameyama,A.; Keaty,J.; Masubuchi,Y.; Riley,M.; Shippy,D.; Stasiak,D.; Suzuoki,M.; Wang,M.; Warnock,J.; Weitzel,S.; Wendel,D.; Yamazaki,T.; Yazawa,K.: *The Design and Implementation of a First-Generation CELL Processor*, Proceedings of the Custom Integrated Circuits Conference, (2005).

[27] Wechsler, O: *Inside Intel Core Microarchitecture*, White paper, Intel corporation,(2006).

[28] *AMD Multi-core*, White paper.

[29] Elrad,T.; Filman,R.E.; Bader,A.: *Aspect-oriented programming*, Communications of the ACM, Vol.44 (2001).

[30] Monteyne, M.: *RapidMind Multi-Core Develpment Platform*, RapidMind (2007).

[31] Microsoft corporation: *Parallel Computing Developer Center website*, See http://msdn2.microsoft.com/en-us/concurrency/default.aspx, (2007).

[32] Craig, I.D.: *Virtual Machines*, Springer, 1st edition, (2005).

[33] Rosenblum, M.: *The Reincarnation of Virtual Machines*, ACM Queue, Vol. 2, no. 5, (2004).

[34] Danelutto, M.: *QoS in Parallel Programming through Application Managers*, 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), (2005), 282-289.

[35] Petzold, C.: *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation*, Microsoft Press, (2006).

[36] Smith,J.: *Inside Windows Communication Foundation*, Microsoft Press, (2007).

[37] Sun Microsystems, Inc.: *OpenOffice suite*, See http://www.openoffice.org/index.html.

[38] Sharp, John A.: *Data Flow Computing: Theory and Practice*, Intellect Books, (1992).