

A Generic Model of Functional Programming With Dynamic Optimization

Martin Grabmüller

Technische Universität Berlin

Abstract

Modern virtual machines for object-oriented machines use dynamic (run-time) compilation in order to ensure fast execution while maintaining security and portability of program code. Several virtual machine implementations using this compilation model have been implemented and are successfully used in practice, but to date no formal model of program execution and dynamic compilation has been published. This paper presents a formal framework for describing dynamically optimizing virtual machines in the context of purely functional programming languages.

1 INTRODUCTION

Using virtual machines to implement high-level programming languages is quite common today, as it promises several advantages when compared to traditional (static machine-code compiler) approaches. The code of programs is stored in a machine-independent format and the virtual machine provides a portable interface to the applications running on it, so that applications are automatically portable across different machine architectures and operating systems. Another important aspect is security: the virtual machine has complete control over the applications it runs and may restrict their operation to conform to some security policy.

The main drawback of virtual machines is, when naively implemented, their poor performance which results from safety verification of program codes and the overhead of interpreting the portable code representation. A lot of work has been done on dynamic translation from portable to machine code at run-time (just-in-time, or JIT compilation). Virtual machines employ dynamic profiling and optimization features in order to balance the time required for compilation with the performance benefits of optimized code.

Despite the wide application of virtual machine techniques in practice, no publications which formalize aspects of virtual machines – such as dynamic analysis and transformation – are known to the author, even though most principles from static program analysis and transformation can be carried over almost unchanged. The framework presented in this paper aims at formalizing some aspects of virtual machines and dynamic program optimization. We concentrate on the essence of dynamic analysis and transformation by restricting the programs running on the virtual machine to purely functional ones.

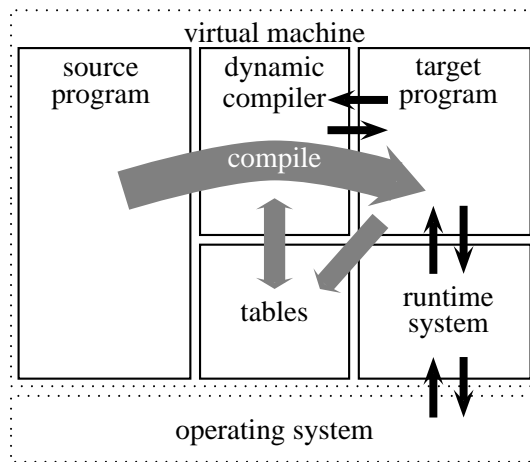


FIGURE 1. Components of a virtual machine

1.1 Contributions

This paper makes the following contributions:

- We present a formal model of dynamic translation, abstracted over source language, target language and execution model as well as the dynamic analysis and transformation algorithms.
- In order to illustrate our model, we apply it to a well-known optimization, namely, profile-driven dynamic inlining.

1.2 Outline

The rest of the paper is organized as follows: Section 2 gives an introduction to the principles of virtual machines and dynamic compilation. Section 3 describes our formal approach to dynamic analysis and transformation. We describe the components of the framework, their interplay and how existing transformations can be formulated in our terminology. In order to illustrate the use of the framework, Section 4 describes how a well-known dynamic optimization can be treated in the proposed notation. Section 5 compares to other work and Section 6 outlines possible future extensions to the system. Finally, Section 7 concludes.

2 VIRTUAL MACHINES AND DYNAMIC TRANSLATION

In this paper, we concentrate on virtual machines which do some kind of dynamic translation during the course of executing user programs. This section describes the principles of these machines and defines the terminology used throughout the paper.

2.1 Virtual Machines

Figure 1 shows the basic structure of a virtual machine and its interaction with the underlying operating system. Data flow is indicated using grey arrows and control flow with black arrows.

The *source program* is a program in some portable representation (e.g., Java bytecode [LY99] or CIL code [ECM05]), whereas the *target program* consists of machine code which can be directly executed by the underlying machine architecture (e.g., IA-32 or SPARC machine code). The *dynamic compiler* translates source program parts (methods, functions, or basic blocks) from source to target language, storing it into the so-called *code cache*, which holds the target program fragments. The code fragments in the code cache are not required to correspond to the code fragments in the source program: the former may be superblocks made up of several source code basic blocks, for example. Note that the code cache in real implementations is limited in size, so that *code cache management* is necessary to act properly when the code cache overflows. Control is transferred from the target program to the dynamic compiler whenever some code is to be executed which has not yet been translated. The compiler then translates the corresponding source program part and continues execution in the newly created target code. The tables hold various information about the dynamic translation state, such as source-to-target address mappings or profiling information. This information is added both by the compiler and the target program when it has been instrumented to collect data about its own execution. This includes counts of function calls or loop iterations, for example. The compiler can base translation decisions on this analysis of the dynamic program behavior.

The *runtime system* is responsible for virtualizing machine resources such as processes, files or network connections and provides the only interface to the operating system.

In general, virtual machines map from expected machines to actual machines. This includes both translation and the mapping of resources. This paper considers only translation, because the source language is expected to abstract over resource representations.

2.2 Dynamic Compilation

Dynamic compilation translates the input to the virtual machine (source language) to the language actually executed (target language). This process naturally decomposes into two phases: *analysis* and *transformation*. In contrast to static (off-line) compilers, a dynamic (on-line) compiler has more information about the program, for example certain actual input values and dynamic execution profiles. An additional difference between static and dynamic compilers is that the former normally translate complete programs (or modules), while the latter can operate on much smaller parts of the program. This reduces compile times and therefore interruptions of the user program. It also reduces compilation to the parts of the program

which are executed in the actual program run. For many programs, this is a big advantage, since no time is wasted on the compilation of unneeded program parts.

2.3 Feedback-driven Transformation

An important aspect in dynamic optimization is to find the parts of the program which are most often executed and which would therefore benefit from (possibly expensive) optimizations. In a *feedback-driven* optimization architecture, the target code is instrumented so that it maintains counters of how often each part of the program has been run. When a counter reaches some predefined threshold, the dynamic compiler is invoked to perform optimization on this program part. This can eventually happen several times through a single program run. Profiling information can also be gathered by sampling the active code blocks at regular intervals. This incurs less overhead than counting, but is less precise and requires support from the hardware and/or operating system.

A complete feedback-driven (also called *adaptive*) optimization architecture thus requires the collection of profiling data, a decision procedure for determining candidates for recompilation and a run-time compiler supporting several modes of code generation: at different optimization levels and with or without code instrumentation.

3 FORMAL FRAMEWORK

Based on the terminology introduced in the previous section, we will now describe our formal framework in detail. First, the virtual machine model will be introduced, including its components: source program, target program, dynamic transformation and so on. After that, we will provide details about the interplay between the various components and describe how the genericity of the framework is obtained by abstracting over source and target languages as well as over the supported analyses and transformations.

In the following, we will use the following notation: a sequence of items (possibly empty) will be overlined, as in \overline{D}_t .

3.1 Dynamic Transformation Machine

The source and target programs are written in source and target languages, respectively. Figure 2 shows the basic structure of these languages: programs consist of a (possibly empty) sequence of definitions (e.g. function definitions), followed by an expression, which is interpreted as the main program giving the final result of running the program. The structure of expressions is not defined and can vary between source and target languages (and often will do so dramatically, for example for high-level source and machine target language). The identifiers appearing in definitions are expected to be unique. The target language must define a reduction function which takes the set of definitions and an expression and reduces

P_s	$::= \overline{D_s} E_s$	source program
D_s	$::= x = E_s$	source definition
E_s	$::= \text{depends on source language}$	source expression
P_t	$::= \overline{D_t} E_t$	target program
D_t	$::= x = E_t$	target definition
E_t	$::= \text{depends on target language}$	target expression
\rightarrow	$: \overline{D_t} \times E_t \times K \rightarrow E_t \times K$	target reduction
K	$::= \text{depends on source/target language}$	knowledge base
τ	$: \overline{D_s} \times E_s \times K \rightarrow E_t$	dynamic translation

FIGURE 2. Source and Target Language

some part or the whole expression to a new expression, using and possibly updating the knowledge base (see below). This function defines how target programs are executed in the virtual machine, specifying the target language's operational semantics.

A knowledge base contains analysis results, both static (compile-time) as well as dynamic (run-time). Note that in this context, compile-time is actually part of the run-time of the user program and interleaved with user program execution, but we keep the classic distinction and use the term run-time for the actual (*productive*) execution of the user program and compile-time for the (*non-productive*) work performed by the dynamic compiler. The structure of the knowledge base depends on the source and target language as well as on the supported analyses and transformations.

The function τ defines the compilation of source to target expressions. This translation takes place in the context of the set of source definitions and the current knowledge base, so that optimization decisions can be made with respect to the program execution so far. The translation function τ together with the reduction function \rightarrow defines the semantics of the source language.

The overall structure of a virtual machine is shown in Figure 3. It consists of a source program, a target program, a target reduction function, a transformation function, a compilation strategy and a knowledge base. The compilation strategy (also shown in Figure 3) is a function which maps the virtual machine state to a target program. The strategy decides, based on the information contained in the knowledge base, which definitions should be recompiled, that is, should be translated from the source to the target program, possibly replacing already compiled functions. Since the outcome of the decision might be to update (or reset) the knowledge base, the knowledge base is also an output of this function. Note that it would be possible to define this function as the identity, which means that no recompilation would ever take place. The problem of code cache management

$$VM ::= (P_s, P_t, \rightarrow, \tau, \sigma, K) \quad \text{virtual machine}$$

$$\sigma : P_s \times P_t \times \tau \times K \rightarrow P_t \times K \quad \text{strategy function}$$

FIGURE 3. Machine Model

mentioned in Section 2 could also be modelled by appropriately defining the strategy function.

3.2 Machine Execution

The execution of a user program written in the source language proceeds as follows:

1. The program is loaded into the machine, giving the source definitions and main source expression.
2. The expression of the source program (the main program) is translated to the target program's expression, using the translation function τ .
3. The knowledge base is initialized to be empty.

When no incremental compilation is desired, Step 2 of the program initialization above would also translate all source definitions to corresponding target definitions. After that, the virtual machine evaluates the target program using the following steps:

1. The strategy function is called in order to decide whether recompilation is necessary, and
2. the reduction function of the target language performs one reduction.

These steps are repeated in alternation until no recompilation occurs anymore and no reducible expressions can be found in the target program.

The delayed recompilation used in real virtual machines can be modelled in our framework by a strategy function which compiles a definition whenever it detects that a reference is made to a source program definition which has not yet been compiled. Section 4 (below) provides an example. The information needed to detect this case is held in the set of target definitions and corresponds to a source-address to target-address mapping table as used in real virtual machines.

3.3 Modelling Common Transformations

Many commonly used program analyses and transformations can be defined using the framework described in this section. The strategy function plays several roles: first, it is responsible for detecting when a source definition needs to be compiled

		<i>reduction function</i>		
		no reduction	reduce only	reduce and profile
<i>strategy function</i>	compile eagerly	static compiler	dynamic compiler	dynamic compiler + profiler
	compile on demand		incremental compiler	incremental compiler + profiler
	compile on demand and recompile		incremental compiler	feedback-driven incremental compiler + profiler

TABLE 1. Modelling common compilation techniques

for the first time, that is, on the first invocation, or recompiled when needed. Second, it has to decide at which optimization levels the compilation should proceed. The distinction of several optimization levels is common in real virtual machine implementations, because optimization is mainly a tradeoff between fast compilation and slow execution on the one hand and slow compilation and fast execution on the other. Between the two extremes of compiling one definition at a time and compiling the whole program at once, it is also possible to compile several definitions at once so that possible initial startup costs of the strategy function can be amortized over multiple definition compilations.

Table 1 shows the relation between various definitions of strategy and reduction functions and how they can be used to perform some classical kinds of compilers: when the strategy function compiles all definitions eagerly (at program startup) and the reduction function is the identity, we get the equivalent of a static (off-line) compiler. The two on-demand compilation schemes only make sense when the reduction function performs useful work, therefore two of the table entries are left blank. By varying the strategy function to compile on demand and to allow recompilation, and by letting the reduction function perform some useful work or even to gather profile information, the system is able to perform dynamic, incremental or even feedback-driven recompilation.

3.4 Properties of the Virtual Machine Model

Since the model is simple and mainly serves as glue between the language definitions of the source and target language as well as analysis and transformation functions, most properties of the system depend heavily on the properties of the given languages. If properties like type preservation and progress hold for the language semantics, they will hold for the dynamically optimizing system as well, as long as the used transformations from source to target language also preserve these properties. A detailed study of the machine's properties and its interaction with the languages implemented on top of it are topics of future research.

3.5 Genericity of the Virtual Machine Model

Our claim is that the proposed virtual machine model is generic across multiple source and target languages as well as the analysis and transformation algorithms used. The machine execution is decoupled from all these aspects (except for the minor requirements of source and target language structure given in Figure 2) by requiring them to be packaged up in the reduction, translation and strategy functions. Because this model is so abstract, it is not obvious how it can be used to model real-life applications of dynamic compilation techniques. This is the reason for presenting a concrete example in the next section, where the model is instantiated with concrete languages and a concrete optimization.

4 CASE STUDY: FEEDBACK-DRIVEN INLINING

The formal framework presented in the previous section will now be used to model an optimization which is successfully used in practice: feedback-driven inlining. *Inlining* (also called procedure integration) works by replacing a function application by the called function's definition, substituting the actual arguments for the formal parameters in the process. This optimization avoids the overhead of a function call and return and additionally allows other optimizations to work over larger parts of the code, thereby specializing the called function for the calling context.

In this section, we use the following notation: the operation \oplus adds an element to a sequence of definitions, overwriting any definition of the same name. The operation \downarrow retrieves the expression of a definition with a given name. These will be used to manipulate the target program. The notation $E_1[E_2/x]$ stands for the capture-free substitution of expression E_2 for variable x in expression E_1 .

Figure 4 contains the definitions of the source and target languages. For simplicity and in order to concentrate on the optimization used, the languages are identical in this example. The language is an untyped λ -calculus extended with the operations `zero`, `pred` and `succ` as well as with a conditional expression (`if0`) which tests for equality to `zero`. Note that application expressions are labelled using unique markers. This is required for referencing application expressions appearing in the source and target programs.

For notational convenience, we define source and target *contexts*, which are, like expressions, identical for the source and target cases. Contexts are expressions which may contain *holes* (written \bullet). Each hole stands for a position in the expression where another expression may be placed. Filling the hole in a context C with an expression E is written as $C[E]$. Contexts are used to select subexpressions for reduction and their grammar essentially defines the reduction order.

The knowledge base consists of a possibly empty sequence of label/number pairs, where the number counts the times a function application with a given label has been performed. We define the operations \oplus and \downarrow (introduced above) specially on knowledge bases. The addition operation \oplus adds a label/number pair to a knowledge base, setting the count to one if the label does not appear in the

$$\begin{aligned}
E_s & ::= x \mid \lambda x. E_s \mid E_s^l E_s \mid \text{zero} \mid \text{succ } E_s \mid \text{pred } E_s \mid \text{if0 } E_s E_s E_s \\
E_t & \stackrel{\text{def}}{=} E_s \\
C_s & ::= \bullet \mid \bullet E_s \mid v \bullet \mid \text{succ } \bullet \mid \text{pred } \bullet \mid \text{if0 } \bullet E_s E_s \\
C_t & \stackrel{\text{def}}{=} C_s \\
K & ::= \cdot \mid (l, n), K \\
K \oplus l & = (l, 1), K \quad \text{if } l \notin K \\
K \oplus l & = K_1, (l, n+1), K_2 \quad \text{if } K = K_1, (l, n), K_2 \\
K \downarrow l & = 0 \quad \text{if } l \notin K \\
K \downarrow l & = n \quad \text{if } (l, n) \in K
\end{aligned}$$

where $v ::= \text{zero} \mid \text{succ } v \mid \lambda x. E_s$

FIGURE 4. Example source and target language

$$\begin{aligned}
(\overline{D}_t, x, K) & \rightarrow (\overline{D}_t \downarrow x, K) & (r1) \\
(\overline{D}_t, \text{if0 zero } E_{t1} E_{t2}, K) & \rightarrow (E_{t1}, K) & (r2) \\
(\overline{D}_t, \text{if0 (succ } v) E_{t1} E_{t2}, K) & \rightarrow (E_{t2}, K) & (r3) \\
(\overline{D}_t, \text{pred(succ } v), K) & \rightarrow (v, K) & (r4) \\
(\overline{D}_t, (\lambda x. E_{t1})^l v_2, K) & \rightarrow (E_{t1}[v_2/x], K \oplus l) & (r5) \\
(\overline{D}_t, E, K) \rightarrow (E', K') & \Rightarrow (\overline{D}_t, C_t[E], K) \rightarrow (C_t[E'], K')
\end{aligned}$$

FIGURE 5. Reduction function

knowledge base and incrementing the count otherwise. The lookup operation \downarrow retrieves the count for a given label and returns 0 if the label does not appear in the knowledge base.

The reduction function for the example target language is shown in Figure 5. It defines a call-by-value variant of the λ -calculus with conditionals. Note that **SUCC** is actually an uninterpreted data constructor whereas **PRED** is an operation which removes one **SUCC** node from a value. The reduction function performs profiling while reducing an expression: it counts applications of λ -expressions by adding the application expressions' labels to the knowledge base. The last line of the figure generalizes the reduction function by stating that if some expression E reduces to E' , then a context containing E reduces to the same context where E is replaced by E' .

Initial compilation and recompilation is triggered by the strategy function shown in Figure 6. The first case handles references to identifiers whose definitions have not yet been translated. The use of a context in the rule makes sure that only definitions which will be needed are translated.

The second rule checks on function applications whether the call site is “hot”,

$$\begin{aligned}
\sigma((\overline{D}_s, E_s), (\overline{D}_t, C[x]), \tau, K) &= (\overline{D}_t \oplus (x = \tau \overline{D}_s \llbracket \overline{D}_s \downarrow x \rrbracket K), K) \\
&\text{if } x \notin \overline{D}_t & (s1) \\
\sigma((\overline{D}_s, E_s), (\overline{D}_t, C[(\lambda x. E_2)^l v_2]), \tau, K) &= (\overline{D}_t \oplus (x = \tau \overline{D}_s \llbracket \overline{D}_s \downarrow x \rrbracket K), K) \\
&\text{where } x \text{ contains } l \text{ and } K \downarrow l \geq T & (s2)
\end{aligned}$$

FIGURE 6. Strategy function

which means that it has been executed more than a predefined number of times T . If this is the case, the containing definition of the function application is recompiled. The translation function (see below) is responsible for performing the actual inlining operation.

The translation function from source to target code is displayed in Figure 7. It mainly maps source terms to their direct target code equivalents, except for possible inlining opportunities. When a function application is to be translated, the knowledge base is consulted to find out whether the function called is an inlining candidate. When an application expression is encountered, the translation function considers several cases. When the expression in function position is a variable, the invocation count of the application expression is retrieved from the knowledge base. When the number of invocations is less than threshold T , the application is mapped to a target language application. Otherwise, the definition's expression is compiled in place of the variable. Because the definition is most certainly a λ -expression, an additional translation is used which performs β reductions at compile time whenever the argument of an application is a variable or a simple value. This helper translation is called τ' and also appears in Figure 7.

Note that in this setup, all optimizations are performed by the translation functions, whereas the necessary profile data collection is done by the reduction function and decisions concerning recompilations are defined by the strategy function. This separation of concerns is not required and other uses of the formal framework could be defined differently, since the framework is sufficiently flexible to encode several policies. The distribution of responsibilities between the different components chosen here allows for modular extensions of the supported optimizations.

Figure 8 contains an example program written in the source language of this section. The figure represents the machine state after initialization, which consists of loading the source program and translating the main program to the target language. The first two boxes contain the source program (both definitions and the main program), the third and fourth boxes contain the target program and the last box contains the knowledge base. The target definitions are empty, because no definition has been translated yet. The knowledge base is also empty when execution starts.

Beginning with the initial state shown in Figure 8, evaluation proceeds as follows:¹ First, the strategy function triggers rule (s1) in order to translate function

¹Because of space constraints, we do not show the complete reduction sequence here.

$$\begin{aligned}
\tau \overline{D}_s \llbracket x \rrbracket K &= x \\
\tau \overline{D}_s \llbracket \text{zero} \rrbracket K &= \text{zero} \\
\tau \overline{D}_s \llbracket \lambda x. E_s \rrbracket K &= \lambda x. \tau \overline{D}_s \llbracket E_s \rrbracket K \\
\tau \overline{D}_s \llbracket x^l E_{s2} \rrbracket K &= (\tau \overline{D}_s \llbracket x \rrbracket K)^l (\tau \overline{D}_s \llbracket E_{s2} \rrbracket K) \quad \text{if } K \downarrow l < T \\
\tau \overline{D}_s \llbracket x^l E_{s2} \rrbracket K &= \tau' \llbracket (\tau \overline{D}_s \llbracket \overline{D}_s \downarrow x \rrbracket K)^l (\tau \overline{D}_s \llbracket E_{s2} \rrbracket K) \rrbracket \\
&\quad \text{if } K \downarrow l \geq T \\
\tau \overline{D}_s \llbracket E_{s1}^l E_{s2} \rrbracket K &= (\tau \overline{D}_s \llbracket E_{s1} \rrbracket K)^l (\tau \overline{D}_s \llbracket E_{s2} \rrbracket K) \\
\tau \overline{D}_s \llbracket \text{succ } E_s \rrbracket K &= \text{succ } (\tau \overline{D}_s \llbracket E_s \rrbracket K) \\
\tau \overline{D}_s \llbracket \text{pred } E_s \rrbracket K &= \text{pred } (\tau \overline{D}_s \llbracket E_s \rrbracket K) \\
\tau \overline{D}_s \llbracket \text{if0 } E_{s1} E_{s2} E_{s3} \rrbracket K &= \text{if0 } (\tau \overline{D}_s \llbracket E_{s1} \rrbracket K) (\tau \overline{D}_s \llbracket E_{s2} \rrbracket K) (\tau \overline{D}_s \llbracket E_{s3} \rrbracket K) \\
\tau' : E_t \rightarrow E_t & \\
\tau' \llbracket (\lambda x. E_{t1})^l y \rrbracket &= E_{t1} [y/x] \\
\tau' \llbracket (\lambda x. E_{t1})^l \text{zero} \rrbracket &= E_{t1} [\text{zero}/x] \\
\tau' \llbracket E \rrbracket &= E \quad \text{if } E \text{ is complex}
\end{aligned}$$

FIGURE 7. Transformation rules

$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$g = \lambda c. \text{pred } c$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
.

FIGURE 8. Example program

f , because f is a free variable in reduction context and has not yet been translated. This results in a state where function f has been translated to target code.

$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$g = \lambda c. \text{pred } c$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
.

Following several applications of reduction rules, the machine collects information about the active function applications in the code, resulting in the following state:

$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$g = \lambda c. \text{pred } c$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$g = \lambda c. \text{pred } c$
zero
$l1 = 1, l2 = 2, l3 = 1, \cdot$

We suppose that the recompilation threshold T has been set to 2. In this case, the presence of the entry $l2 = 2$ in the knowledge base causes a recompilation of the function containing call site $l2$, which happens to be f . The translation function τ uses the profiling data in the knowledge base and finds that call site $l2$ is “hot” and should be inlined. The resulting machine state after applying rule (s2) is shown below:

$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (g^{l2} n))$
$g = \lambda c. \text{pred } c$
$f^{l3} (\text{succ } (\text{succ } \text{zero}))$
$f = \lambda n. \text{if0 } n \text{ zero } (f^{l1} (\text{pred } n))$
$g = \lambda c. \text{pred } c$
zero
$l1 = 1, l2 = 2, l3 = 1, \cdot$

In this simple example, there is no performance win because the program has terminated before it could use the improved definition, but for larger inputs to the function f , each following application of f will be faster because function g has been inlined. Additionally, it would be possible to purge the definition of g from the target program definitions because it is not needed anymore, thus freeing up memory to be used for other function translations.

This example of using our framework for defining a dynamically optimizing system illustrates the flexibility of our approach. Other optimizations could be easily added by appropriately extending the strategy and/or translation functions. By adapting the reduction function, other profile data could be collected and used as the basis for optimizations.

An obvious limitation of the optimization presented in this section is that it can only inline functions which are called using the name of their top-level definition (first-order calls). This problem can be solved by applying so-called *guarded inlining*. Instead of inlining only functions known to be called at certain call sites, the reduction function collects information about which set of functions are called at a given call site and counting how often each member of this set has been called. The “hottest” function (e.g., with the highest call counter) can then be inlined, guarded by a conditional which tests whether the called function is indeed the expected one. If it is, the inlined code is used, otherwise the function call is performed.

5 RELATED WORK

A comprehensive description of virtual machines can be found in Smith and Nair [SN05], who treat both low-level (binary and hardware) and high-level language virtual machines. Examples of low-level virtual machines are binary translators [CH97], binary optimizers [LD97, BGA03] and co-designed hardware/software systems which execute machine code on different processor architectures using integrated translation software [EA96, DGB⁺03]. High-level virtual machines provide additional services for user programs, such as abstraction over machine resources, garbage collection and multiple threads of execution [LY99, ECM05]. An earlier approach of portable code files and load-time compilation was that of Franz [Fra94] for the Oberon system. The portable code files are actually compressed abstract syntax trees which are compiled when a module is loaded into the system. The Self system also relies heavily on dynamic compilation [Höl94] and pioneered aggressive feed-back directed optimizations. Self is a dynamically typed object-oriented language where all operations are dispatched dynamically, and thus requires good optimization in order to execute efficiently. For similar reasons, but not as aggressively, Deutsch and Schiffmann [DS84] have used dynamic compilation in their Smalltalk implementation.

The design of the formal model was inspired by several virtual machine architectures for the Java programming language. The Jalapeño virtual machine (later renamed to Jikes RVM) at IBM [BCF⁺99, SOT⁺00] first introduced the notion of a *controller*, which is the component making optimization decisions based on a cost-benefit model of adaptive recompilation. This component loosely corresponds to our strategy function and appears also in other Java VMs, such as the StarJIT system from Intel [ATBC⁺03], where it is called *Profile Manager*.

Arnold et al. [AFG⁺04] propose a model-driven policy for detecting recompilation opportunities which is used in the Jikes Research Virtual Machine. They model both expected recompilation costs and the expected benefits of running optimized code, basing their heuristics on the compile times and profile data collected up to that point in execution. This seems to be the only published attempt to capture aspects of dynamic optimization systems formally.

Wakeling [Wak98] used dynamic compilation of a lazy functional language in order to reduce the memory requirements of compiled code. In his system, Haskell source code is compiled to a compact intermediate format prior to execution and a dynamic compiler translates this code to machine code while the program is running. When the storage reserved for compiled code is exhausted, all compiled code is discarded and required code is re-generated (throw-away compilation [Bro76]). Other uses of dynamic code generation in the context of functional languages include those in meta programming, for example MetaML [She98], which does not require but benefits from dynamic code generation. Lee and Leone [LL96] translate code written in a subset of ML into programs which automatically specialize programs at run-time. Run-time code generation as a user-level library was proposed by Lomov and Moscal [LM02]. They provide a library which allows the

translation from abstract syntax trees to bytecode for the Caml system at run-time.

Other work related to the presented model includes several works on defining language semantics which reflect aspects of evaluation such as the cost associated with each reduction step. Hope and Hutton [HH05] presented how a step-counting semantics can be derived from purely functional programs. Sands [San90] investigated cost semantics for several languages, from simple first-order to higher-order languages.

6 FUTURE WORK

The work presented here can only be seen as a first step towards the formalization of dynamic optimization systems. Several topics for future work have been identified while developing the formal model and the example optimization presented here.

Modern virtual machines support dynamic loading of program code. This should be included in our framework by adding some primitive mechanism for replacing (parts of) the source program and triggering the recompilation for replaced definitions.

An important aspect is whether other language constructs besides the concept of a *definition* are useful enough across various source and target languages so that they should be included into the basic framework instead of being defined in the concrete languages used with the model. This would allow more work on the core framework to be applied to all possible source and target languages. One possible example is to define some general cost semantics which can be used across all (or at least most) reduction-based languages. The advantage is that the description of individual analyses and transformations is moved into the general framework, making the concrete language definitions smaller and easier to work with.

Another interesting path for future work is in lifting the restrictions on the current framework: we would like to allow languages with more complicated semantics than just reduction-based small-step semantics. Although we suppose lifting some of the restrictions will not impose major problems, we have not yet studied it in detail.

As already mentioned in Section 3, we plan to study the effect of dynamic optimization on the properties of the programs running on top of it. Though we expect no difficulties, since the machine is designed to be transparent to the running programs, it may be necessary to state specific requirements for the transformation and strategy functions used in order to ensure this transparency.

7 CONCLUSION

In this paper, we have proposed a simple generic theoretical model of virtual machine execution with support for dynamic compilation and optimization. In order to keep the model simple, we have restricted input programs to purely functional lan-

guages which can be defined using reduction. The model is generic in that it does not prescribe specific source or target languages and gives freedom to the analysis and transformation algorithms used for dynamic compilation and optimization.

Using the model, we have shown how to formulate a well-known optimization and have demonstrated the evaluation of a simple dynamically optimized program. The easy formulation of various optimizations in a unified formal model is expected to encourage the formal treatment of such transformations and to increase the understanding of dynamically optimizing systems in general.

REFERENCES

- [AFG⁺04] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Research Report 23429, IBM Research, November 2004.
- [ATBC⁺03] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, and Tatiana Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), February 2003.
- [BCF⁺99] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.
- [BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275. IEEE Computer Society, 2003.
- [Bro76] P. J. Brown. Throw-away compiling. *Software Practice and Experience*, 6(4):423–434, 1976.
- [CH97] Anton Chernoff and Ray Hookway. Digital FX!32 – Running 32-Bit x86 Applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, Washington, August 1997.
- [DGB⁺03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.
- [DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, 1984.
- [EA96] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Technical Report 8502, IBM, 1996.
- [ECM05] ECMA International. *Standard ECMA-334: Common Language Infrastructure (CLI)*. 3rd edition, June 2005.

- [Fra94] Michael Steffen Oliver Franz. *Code-Generation On-the-fly: A Key to Portable Software*. PhD thesis, ETH Zürich, 1994.
- [HH05] Catherine Hope and Graham Hutton. Accurate step counting. In *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, Dublin, Ireland, 2005 2005.
- [Höl94] Urs Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Computer Science Department, Stanford University, 1994.
- [LD97] Mark Leone and R. Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Indiana University Computer Science Department, September 1997.
- [LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 137–148. ACM Press, 1996.
- [LM02] Dmitry Lomov and Anton Moscal. Dynamic Caml v. 0.2 – Run-Time Code Generation Library for Objective Caml. Available on the World Wide Web at <http://oops.tercom.ru/dml/files/dml-0.2.1.ps.gz>, last visited: 2006-01-31, May 2002.
- [LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.
- [San90] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.
- [She98] Tim Sheard. Using MetaML: A Staged Programming Language. In Swierstra et al. [SHO98], pages 207–239. Third International School, AFP’98.
- [SHO98] S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors. *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998. Third International School, AFP’98.
- [SN05] James E. Smith and Ravi Nair. *Virtual Machines – Versatile Platforms for Systems and Processes*. Morgan Kaufman, 2005.
- [SOT⁺00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.
- [Wak98] David Wakeling. The dynamic compilation of lazy functional programs. *Journal of Functional Programming*, 8(1):61–81, January 1998.