University of Texas at San Antonio, Department of Computer Science, Technical Report 2006-005

SSA-based Mobile Code: Implementation and Empirical Evaluation

WOLFRAM AMME
Friedrich-Schiller-Universität Jena
and
JEFFERY VON RONNE
The University of Texas at San Antonio
and
MICHAEL FRANZ
University of California, Irvine

Although one might expect transportation formats based on static single assignment form (SSA) to yield faster just-in-time compilation times than those based on stack-based virtual machines, this claim has not previously been validated in practice. We attempt to quantify the effect of using an SSA-based mobile code representation by integrating support for a verifiable SSA-based IR into Jikes RVM. Performance results, measured with various optimizations and on both the IA32 and PowerPC, show improvements in both compilation time and code quality.

Categories and Subject Descriptors: D.3.4 [**Programming Language**]: Processors—*Code generation*; *Compilers*; *Optimization*; *Run-time environments*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: virtual machines, static single assignment form, SafeTSA

1. INTRODUCTION

When we introduced SafeTSA [Amme et al. 2001], an inherently-safe typed and dominator-scoped representation for Java that is based on static single assignment form (SSA), we conjectured that such a code format should have several advantages over the more conventional stack-machine design of the Java Virtual Machine bytecode language (JVML). In particular, in the context of just-in-time (JIT) compilation, an optimizing compiler will be able to save the time that would be needed to construct SSA form at runtime. It also may not need to perform as much optimization during just-in-time compilation to get the desired code quality, because many optimizations, including redundancy eliminations that are difficult to express in JVML (because the stack-machine will require compensation code to carry temporaries), can be performed cleanly on SSA code. This is further enhanced by SafeTSA's type system, which allows null- and bounds-checks to be safely removed by these optimizations when the SafeTSA code is produced. In addition, SafeTSA's validity is enforced as an inherent property of an encoding that also reduces file size.

Elsewhere, we have described the details of SafeTSA's design and encoding, and experimentally validated its compactness and load-time decoding performance [von

Ronne et al. 2006]. This paper, focuses on an empirical evaluation of the compilation speed and overall execution performance of a prototype virtual machine supporting SafeTSA.

We report on how we have taken the SafeTSA format and integrated it into IBM's Jikes Research Virtual Machine (RVM) [Alpern et al. 2000] for the PowerPC and IA32 architectures. This undertaking was more ambitious than we had anticipated and required adding tens of thousands of of lines of code to Jikes RVM. The result is a Java execution environment that is capable of processing both Java class files and SafeTSA files interchangeably. It can even execute programs in which some of the classes have been compiled into Java class files and others into SafeTSA files, which are then all combined during dynamic class loading.

Using this system, we ran a series of benchmarks in order to evaluate the performance characteristics of the resulting system on both platforms and under a variety of differing optimizations in order to assess whether the conjectured benefits of an SSA-based intermediate representation format provide a practical benefit over JVML.

In the following sections, we first introduce some of the key features of the SafeTSA format. We then give a brief overview of the Jikes RVM system, particularly its code generator and internal data structures. Following this, we describe the implementation of our SafeTSA compiler and its integration into the Jikes RVM system. This is followed by a discussion of the benchmark results, reporting on both code-generation time and on the generated code's performance. The paper concludes with a summary of our findings.

2. THE SAFETSA REPRESENTATION

SafeTSA[1] is an intermediate representation designed to be target-machine independent, simple to verify, and easy to translate into optimized native code. SafeTSA achieves this through a novel combination of several key features: the use of static single assignment form, a tree representation of structured control flow, a typed language-specific instruction set, a type system that facilitates the safe optimization of runtime checks, and an inherently type-safe encoding.

2.1 Static Single Assignment Form

Static single assignment form (SSA) [Cytron et al. 1991] is a standard representation for optimizing compilers. SafeTSA is based on SSA form, leveraging its benefits during the JIT compilation phase but shifting off-line the costs of producing SSA form. In SSA form, the target operand of each instruction is unique, such that each of these "values" is created (assigned to) at exactly one (a single) static location and remains constant throughout the value's scope. This is possible, because in SSA, program locations at which multiple original program variable definitions converge are represented by using explicit ϕ-functions to create new values at merge points.

As an example, consider the program in Figure 1. The left side shows a source program, and the right shows how it might look translated into SSA form. The SSA instructions are grouped into basic blocks which are connected to form a control

[1]The name SafeTSA stands for *Safe Typed Single Static Assignment Form* and predates the formation of the U.S. Transportation Security Administration.

flow graph. (In this figure, there is no explicit representation of goto, branch, or jump instructions, these are to be inferred from the directed edges making up the control flow graph.) All of the instructions shown produce output values whose name is shown on the left-hand side of the arrows. Since each variable definition produces a uniquely named value, there can be several value names for each variable in the original program. By convention, these values are named by combining an original source program variable name with a unique numerical subscript. After conversion to SSA, values with the same name but different subscripts are treated as distinct values despite their common origin. For example, the SSA values $\mathtt{x}_1$, $\mathtt{x}_2$, and $\mathtt{x}_3$ all contain values that would have been held in the original program variable `x`, but in SSA, they can be acted on independently by future analyses, optimizations, and transformations.

In Java programs, as in those of most procedural and object-oriented programming languages, there are often operations that use variables whose value may have been defined at different locations depending on how the program's execution reached that operation. For example in the the program in Figure 1, the add operation should use the initial value of x (obtained from the getfield) during the initial execution of the while loop, whereas on subsequent iterations through the loop, the add operation should act on the value produced by the add on the previous iteration through the loop. In SSA, these are distinct values ($\mathtt{x}_1$ and $\mathtt{x}_3$), and it would be impossible for the add operation to refer directly to both of these simultaneously. Instead of referring directly to either of these, the add operation refers back to yet a third distinct value $\mathtt{x}_2$. This value $\mathtt{x}_2$ is the output value of a ϕ-function which can be thought of as selecting $\mathtt{x}_1$ on the first iteration (when control flow comes from the first block) and $\mathtt{x}_3$ on subsequent iterations (when control flow comes from the third block). These ϕ-functions always appear at the top of blocks that have two or more incoming edges in the control flow graph, and which are called join nodes. The ϕ-functions within a join node have one parameter corresponding to each control-flow graph edge that goes into their join node. These parameters indicate the value that should be selected when the execution reaches the join node on that control-flow graph edge.

Actual instruction set architectures do not have direct support for ϕ-functions, so during generation of machine code, it is necessary that ϕ-functions be "resolved" by removing the ϕ-functions and inserting equivalent register-to-register move instructions into the control-flow graph predecessors of the join nodes. Assuming there are sufficient registers and no mutual dependencies among the ϕ-functions (c.f., [Cooper and Torczon 2003]), the simplest way to resolve a ϕ-function is to allocate a fresh register for the result of the ϕ-function and to insert, into each of the predecessor basic blocks, a move operation that transfers the contents of the register containing the ϕ-function's corresponding parameter into the fresh result register from a register. Our implementation translates from SSA into an intermediate representation with virtual registers, so we are able to allocate a distinct virtual register for each value including those produced by ϕ-functions. For example, the ϕ-functions of method foo could be resolved by inserting move instructions $\mathtt{x}_2 \leftarrow \mathtt{x}_1$ and $\mathtt{j}_1 \leftarrow 1$ at the end of the block before the while loop and move instructions $\mathtt{x}_2 \leftarrow \mathtt{x}_3$ and $\mathtt{j}_1 \leftarrow \mathtt{j}_2$ at the end of the block that represents the loop body.

```
public class A {
  int f1;
  int f2;
}

public class B {
  static int foo (A a) {
    int x = a.f1;
    int j = 1;
    while (j < 10) {
      x = x + x;
      j++;
    }
    return x * a.f2;
  }
}
```

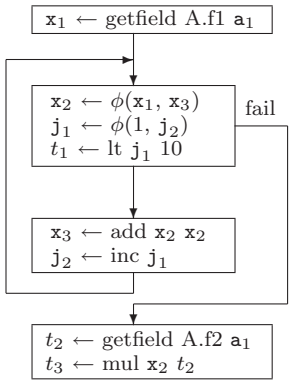

Fig. 1. Example program and method foo in SSA.

This naive resolution of ϕ-functions will, however, often produce more moves than is required (most of these moves were not in the original program). A more sophisticated resolution can often coalesce the virtual register for some ϕ-instructions with one or both of their input registers. To reduce the number of move instructions that have to be inserted, our JIT compiler can be instructed to perform a ϕ-move optimization. In doing so, the JIT compiler checks before inserting each move instruction if the target register can be coalesced with the source register of the instruction. If the live ranges of the target and source register do not overlap, the compiler will coalesce both registers into a single virtual register instead of inserting a move instruction. For our example program, with an applied ϕ-move optimization the insertion of move instruction into foo() can be reduced to one, namely the initialization of j with $\mathtt{j}_1 \leftarrow 1$ at the end of the block before the while loop.

2.2 Control Structure Tree

Unlike JVML and most other mobile code representations, which represent control flow with low-level branch statements. SafeTSA retains each source program's high-level control structures. These high-level control structures are expressed as a tree data structure with individual instructions embedded in basic blocks found at the leaves of the tree. Figure 2 shows this tree structure with the embedded instructions that make up a SafeTSA method. As a first approximation, this *control structure tree* can be thought of as the method's abstract syntax tree with its expressions removed and replaced with basic blocks of three-address code instructions in SSA form. The use of control structure trees restricts SafeTSA methods' control flow graphs to a well defined subset of reducible control flow graphs. This simplifies

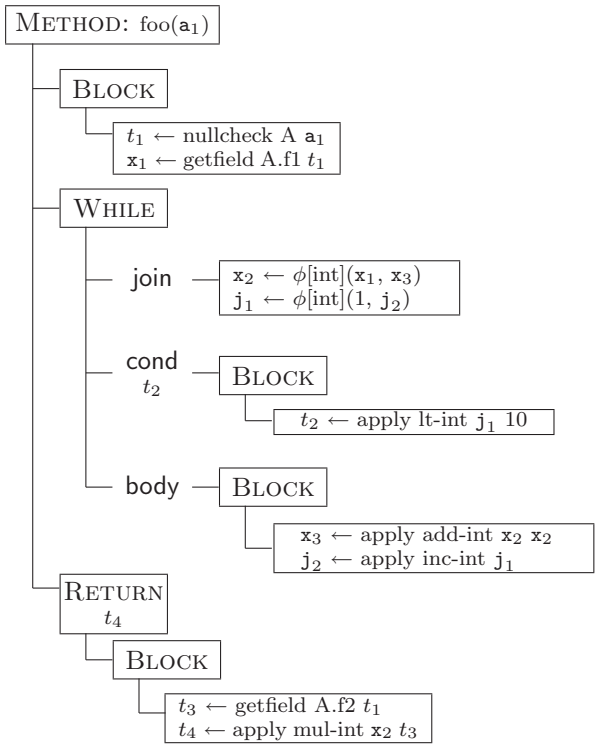

Fig. 2. SafeTSA representation of method foo().

the machine-specific code generation and optimization as well as dominator tree derivation [Hecht and Ullman 1974].

2.3 Instructions

SafeTSA's instruction set is specialized for the Java language. SafeTSA uses Java's heap memory model without modification allowing SafeTSA and JVML classes to be intermixed in a single type safe runtime environment. SafeTSA provides high-level instructions for interacting with Java classes and objects including instructions for getting and setting values stored in fields and for calling Java methods of various sorts. These operations closely follow the semantics of their JVML counterparts and enforce the same type and memory safety invariants.

In addition, SafeTSA provides a comprehensive set of "primitive functions" providing unary and binary numerical operations over the primitive types of boolean, char, short, int, long, float, and double. Examples of such operations include, "add two integers," "convert a double to an integer," "shift a short to the left." All of these simply-typed first-order functions can be used with the "apply" instruction. This "apply" instruction takes one operand specifying the primitive function and additional operands carrying the inputs to the primitive function and produces a result value containing the result of applying the primitive function to its argu-

ments. Thus the type system is able to treat all of the primitive functions with a uniform rule.

One interesting aspect of SafeTSA's instruction set design is that the namespace of SSA values is partitioned according to a discipline of *type separation* [von Ronne et al. 2006]. Although there is runtime subclassing and interface subtyping of objects on the heap, there is no subtyping on SafeTSA's instruction operand and result values; thus, all SSA values belong to exactly one type. The implicit type coercions found in source programs (e.g., when a reference to an java.lang.String is passed as a parameter to a method that expects a reference to a java.lang.Object) are converted to explicit coercion instructions in a manner similar to Penn Translation [Breazu-Tannen et al. 1991]. In addition, the exact types of all of an instructions operand and result values are only influenced by its static non-value parameters (e.g., class and field identifiers) [von Ronne et al. 2006]. Thus, values of different types may be treated as belonging to separate namespaces, greatly simplifying type checking.

2.4 Type System Support for Safe Optimizations

At its core, SafeTSA's type system closely follows that of Java and Java bytecode; it allows the same types of objects in the garbage-collected virtual machine's heap, and the SSA value types include all of the Java primitive types (int, float, etc.) and reference types (which are restricted to point to instances of a particular class, an interface, or array type according to the same rules as Java). But the SafeTSA type system also includes some additional types intended to facilitate ahead-of-time optimization during the generation of the machine-independent SafeTSA code.

In particular, for each Java reference type, SafeTSA adds a 'safe' reference type that can only be produced by a null-check operation. All operations that act on the heap object require the null-checked 'safe' reference type as input. This safely decouples null-checks from their corresponding dereferencing instructions (e.g., getfield) using variables holding references of the null-checked 'safe' reference type. This allows some of the redundant null-checks to be eliminated during the compilation into the SafeTSA representation by using standard redundancy elimination optimizations (such as common subexpression elimination) in the code producer.

An example of this elimination can be seen in Figure 2: The first getfield is immediately preceded by its required null-check (producing t_1) which either throws an exception or creates a new safe reference t_1 out of the original unsafe reference passed as the argument a_1, but the second getfield (in the last block) is able to use the already null-checked reference t_1, and thus, a second null-check was avoided. SafeTSA also has separate types to represent the results of bounds-checked array element address computations. The introduction of safe reference and array element types allows the elimination of redundant null- and bounds-checks when performing simple common subexpression elimination (CSE) [Amme et al. 2000].

2.5 Inherent safety

Unlike typical intermediate representations in SSA, which is untyped and trusted to be correctly produced, SafeTSA provides intrinsic and tamper-proof type safety as a well-formedness property of its encoding [von Ronne 2005]. The on-the-wire SafeTSA format is, in fact, a prefix code of valid SafeTSA classes. Several tech-

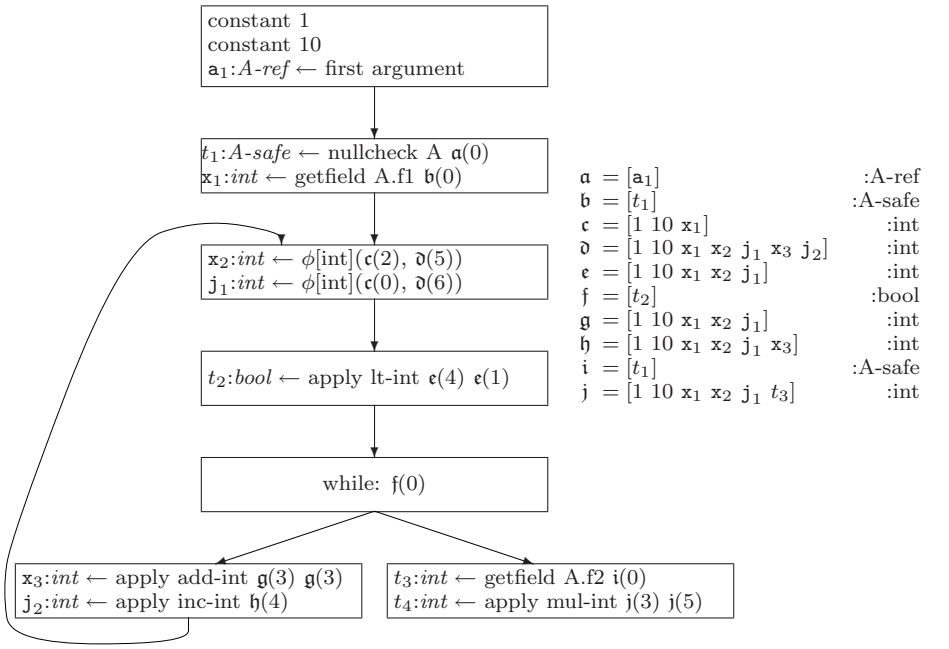

Fig. 3. Operand-Encoding for foo().

niques are used to serialize each class in a SafeTSA program as sequences of symbols, such that each symbol can be selected from a set whose membership can be enumerated knowing only the parts of the class that precede the symbol in the encoding. A low-level prefix encoding can then be used to represent the selection of one of the valid candidates from that enumerated set.

The most interesting of these techniques relate to guaranteeing the type safety of operands. In the encoding, SafeTSA values are not explicitly named. Instead, each operand selects from the enumerated list of reaching definitions of the correct type in a manner similar to the de Bruijn notation for λ-calculus [de Bruijn 1978]. Two filters are applied to obtain the enumerated list: values are scoped so that only the values guaranteed to be defined are including in the list, and the candidate values whose type does not match the type required by the operand are excluded from the enumerated list. And since only an index into this list is transmitted, not even hand-crafted malicious code can undermine type safety and concomitant memory integrity.

Figure 3 shows how these operand encoding rules work out for our example program. In this figure, the individual SafeTSA instructions are grouped into basic blocks, which are laid out according to their place in the method's control flow graph. In this example, if the upward pointing "backwards edge" is ignored, this arrangement of basic blocks also reflects the method's dominator tree, such that any instruction can be executed only when all of the instructions "above" it in this dominator tree have been executed.

The input operands of the SafeTSA instructions do not refer back to SSA values by name, but instead use an index into a context-sensitive, enumerable list of legal

candidate values; these lists are identified by $\mathfrak{a}, \mathfrak{b}, \ldots, \mathfrak{j}$ and are shown explicitly to the right of the basic blocks. The list for a particular instruction operand, contains those SSA values that dominates that instruction and has the correct type. For example, consider the instruction defining t_2 as the result of "apply lt-int $\mathfrak{e}(4)\mathfrak{e}(1)$;" the *lt-int* function returns true whenever a first integer operand is less than a second integer operand. This instruction's enumerated candidate operands list are chosen from the list denoted by $\mathfrak{e}$ and consisting of the SSA values 1, 10, x_1, x_2, j_1. Thus, the first operand j_1 can be identified using the index 4, and the second operand, the constant 10, can be identified using the index 1. But there are no indexes that could be used to identify an operand that may not always be defined (e.g., x_3) or that is not of the correct type (e.g., a_1). The requirement that the candidates dominate this instruction, precludes the definitions x_3, j_2, t_3, t_4, and the requirement that the operands have an integer type, excludes a_1 and t_1.

Special care must be taken in the handling of ϕ-function parameters. Since each parameter is only used when control flow arrives from a particular predecessor block, the last instruction of that block should be considered the immediate dominator for that parameter. For example, the ϕ-function defining the integer j_1 has two parameters. The first is used during the initial iteration when the control flow continues from the instruction defining x_1, so x_1 is considered the immediate dominator, and the candidate list $\mathfrak{c}$ contains x_1 and the other integers that dominate that definition. The second parameter is used when the loop is repeated and control flow returns to the top of the loop after executing the instruction defining j_2.

Neither the SSA value names to the left of each instruction nor the operand candidate lists to the right of the instructions are explicitly represented in SafeTSA's inherently safe wire format. At each point in the program, the decoding algorithm is able to enumerate the set of dominating instructions of the correct type and associate an operand index number with each of these. These lists can be maintained efficiently using a stack data structure during syntax directed program traversal [von Ronne et al. 2006].

2.6 Related Program Representations

The Java language is normally compiled into Java bytecode (JVML), which is a stack-based language that is designed to be executed by a Java Virtual Machine (JVM) implementation, such as the one found in Sun's Java Platform Standard Edition. Prior to execution of JVML, the JVM bytecode must be verified using an iterative dataflow analysis and is either interpreted or translated from its stack-oriented form into register-based machine code. SafeTSA replaces JVML's stack-storage for temporaries and local variables with virtual registers in SSA form.

Besides SafeTSA, the Low Level Virtual Machine (LLVM) [Lattner and Adve 2004] is the only persistent program representation based on SSA form we are aware of. It differs from SafeTSA in that it is designed for use with a local compilation and optimization infrastructure rather than for transporting safe code. So unlike SafeTSA, it is not Java specific, does not retain source program control structures, and does not enforce a high-level type system that can be used for fine-grained language-based security.

Many optimizing compilers, including those found in Java Virtual Machine implementations, utilize SSA based intermediate representations. The Intel Research's

StarJIT Java Virtual Machine is particularly noteworthy for having recently introduced a type-system to verify the correctness of null- and bounds-check optimizations [Menon et al. 2006]. Compared to SafeTSA's null-checked object reference and bounds-checked array element reference types, their approach is more general than SafeTSA's (which can only support fully-redundant eliminations) but results in a more complex type system. The StarJIT intermediate representation includes additional "proof values" as SSA operands and result values, and potentially unsafe instructions require "proof value" operands whose type matches the property to be proved. The complexity of such a type-checking system is dependent upon the complexity of the proofs: Menon et al. suggest that an integer linear programming tool would be sufficient to check the proofs needed to perform most common optimizations [Menon et al. 2006]. StarJIT, however, does not actually need to check the proofs, since they are produced internally and only used to convey dependencies between optimization phases.

Another internal compiler representation, λJVM [League et al. 2001], has some similarities to SafeTSA. Like SafeTSA, it is Java specific, and it uses A Normal Form (ANF) which is similar to SSA [Appel 1998]. As an internal representation, however, λJVM lacks a verifiable externalization.

Besides SafeTSA, Compressed Abstract Syntax Trees (CAST) [Stork et al. 2000; Amme et al. 2001; Stork 2006] is the only other inherently safe program representation that we are aware of. Unlike SafeTSA, CAST is only concerned with the transmission and verification of the abstract syntax tree of source programs. This was inspired by our earlier work on encoding the syntax trees of Oberon programs using Slim Binaries [Franz and Kistler 1997; Kistler and Franz 1999].

The biggest commercial competitor to the Java language and platform is Microsoft's .NET platform with its Common Intermediate Language (CIL) [ECMA 2002]. Like Java's bytecode CIL is a stack-based representation which needs to be verified using a dataflow analysis and translated into register based machine code by the .NET runtime. CIL, however, has some restrictions on the use of the stack across block boundaries that reduces the complexity of these analyses and transformations compared to Java bytecode.

3. OVERVIEW OF JIKES RVM

Jikes RVM is a Java Virtual Machine developed at IBM Research [Arnold et al. 2000]. Jikes RVM possesses several unique features, three of which are particularly relevant to the work presented here: it is compile-only (i.e., it has no interpreter), it is itself written almost entirely in Java, and the optimizing compiler makes use of a linear scan register allocator.

Instead of having an interpreter, Jikes RVM features multiple JVML to native code compilers. One is the 'Baseline' compiler, which exists for debugging and verification purposes; it produces native code that directly implements JVML's stack model as closely as possible and is in many ways comparable to an interpreter. Jikes RVM also has an 'Optimizing' compiler [Burke et al. 1999], which consists of multiple phases and can be operated at various levels of optimization.

The phases of the Jikes RVM optimizing compiler communicate through a series of intermediate representations: a high-level intermediate representation (HIR), a

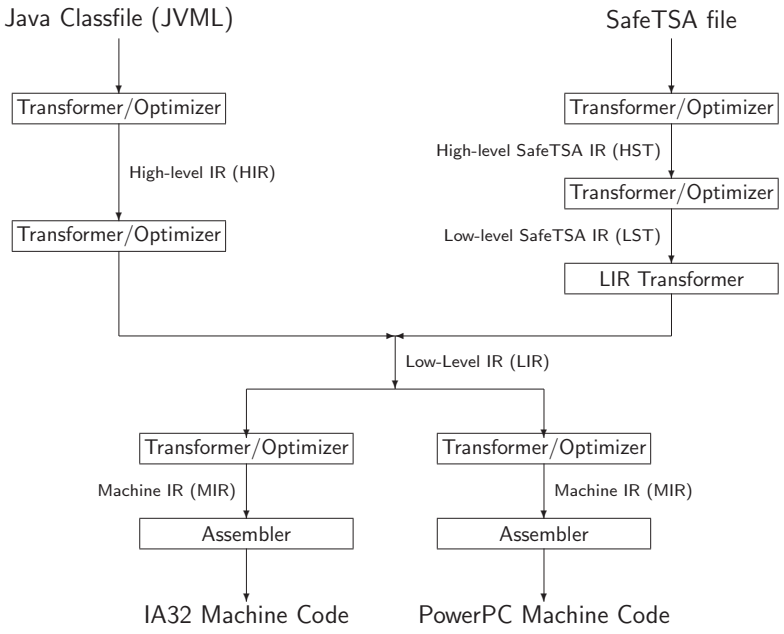

Fig. 4. Compiling JVML and SafeTSA methods in Jikes RVM.

low-level intermediate representation (LIR), and a machine-specific intermediate representation (MIR). As can be seen in Figure 4, a JVML method is initially translated into HIR, which can be thought of as a register-oriented transliteration of the stack-oriented JVML. The LIR differs from the HIR in that certain JVML instructions are replaced with Jikes RVM-specific implementations (e.g., an HIR instruction to read a value from a field would be expanded to LIR instructions that calculate the field address and then perform a load on that address). The lowering from LIR to MIR renders the program in the vocabulary of the target instruction set architecture (PowerPC or IA32). The final stage of compilation is to produce native machine code from the method's MIR. Depending on the configuration of the optimizing compiler, optimizations can be performed in each of these IRs.

Because Jikes RVM is written in Java, the key data structures are accessible to the VM as Java arrays. The most important of these is the Jikes RVM's table of contents (JTOC). The JTOC is an array containing (or containing references to) all globally-accessible entities (i.e., all of the constants, each type's type information block (TIB), meta-objects for static fields and methods, etc. can be found as an index in the JTOC). Loading a class consists of instantiating the appropriate TIB and meta-objects and adding them to the JTOC or the class's TIB.

Methods are compiled the first time they are invoked. When the compiler finds references to fields or methods that have not yet been loaded, it includes code to resolve the field or method just before using the field or method the first time. Jikes RVM facilitates dynamic class loading by maintaining two offset tables, Off-

setTableField and OffsetTableMethod. There is an entry in one of these tables for every known field and method. When resolution of a field or method is required, the appropriate entry is accessed. If it is valid, the offset is used to calculate an address. If it is not valid, the class loader will be called to load the appropriate class and write a valid offset into the appropriate offset tables.

Another noteworthy aspect of the Jikes RVM is that the optimizing compiler uses a linear scan register allocation algorithm [Poletto and Sarkar 1999]. In contrast to other commonly used register allocation strategies [Briggs et al. 1994; Chaitin 1982; Chaitin et al. 1981; Fraser and Hanson 1992], this algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables according to the beginning and ending of their live intervals (a conservative estimate of live ranges) in a depth-first traversal of the program's flow graph. The linearity of the algorithm is based on the fact that each variable is assigned a register only once, at the starting point of its live interval. If a variable X that has already been allocated a register needs to be spilled to make room for a new variable, then that variable X will remain spilled to memory for the rest of X's live interval. Therefore, the primary function of Jikes RVM's register allocator can be thought as being to decide whether, at the start point of a variable X's live interval, it should be initially assigned a register or permanently assigned a place in memory. In situations in which not all registers have yet been allocated to program variables, the value of X will always be placed in a register. In other cases, the decision of which variable to spill is based on the remaining length of each variable's live interval, so that the value of X will be stored in a register only if one of the other variables already allocated to registers has a live interval that ends after that of X.

Thus, Jikes RVM's register allocation tends to assign registers to variables with short live ranges and should result in acceptable code quality for simple programs that use short-lived variables. For programs in which each variable will be assigned a value only once and will be used only within the same basic block, such a register allocation strategy is optimal [Belady 1960; Motwani et al. 1995]. In contrast, in programs that make extensive use of long-lived global variables, it is likely that the register strategy used by Jikes RVM will result in suboptimal performance, especially since the frequency of variable use is not considered during register allocation.

One further feature of the Jikes RVM system is that null-check instructions, which are explicit in HIR and LIR, can be replaced by hardware checks. To reduce the number of null-check instructions that have to be executed at runtime, a so-called null-check combiner examines each null-check instruction during the generation of MIR to determine whether the instruction can be eliminated and combined with a directly following load or store instruction. When an explicit null-check instruction has been eliminated in this way, the reference will still be null-checked implicitly at the time of memory access, because the execution of a load or store instruction with base address *null* will throw a hardware interrupt that is trapped by the Jikes RVM system.

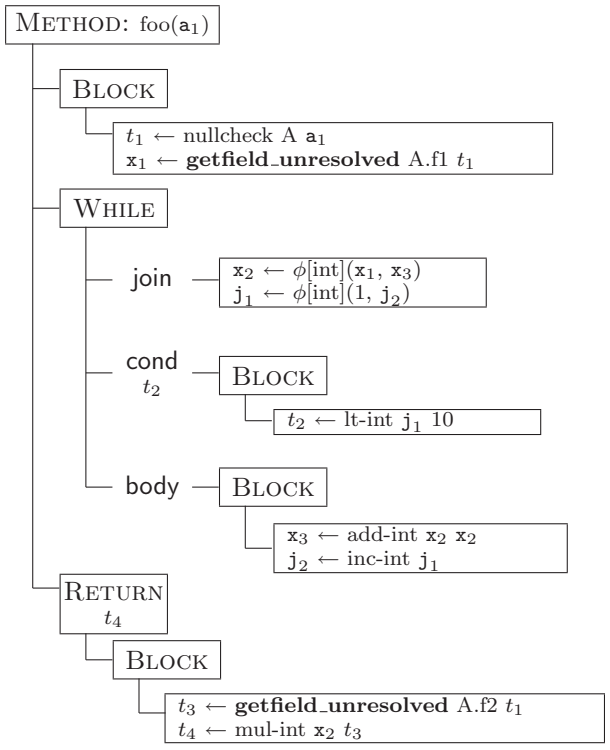

Fig. 5. High-level SafeTSA (HST) representation of method foo().

4. INTEGRATING SAFETSA

By adding SafeTSA class loading and a SafeTSA compiler to the Jikes RVM system, we have built a virtual machine that can execute both SafeTSA- and JVML-compiled Java programs. Thus, functionally, it does not matter whether the whole program has been compiled to SafeTSA, or if it exists as JVML class files, or if the program is provided as a heterogeneous mix of both SafeTSA and JVML classes.

With respect to dynamic class loading, the modified Jikes RVM system treats SafeTSA classes in a manner analogous to traditional JVML classes. This was accomplished by modifying the class loader so that whenever it loads a new class, it will check the class file repositories for SafeTSA classes. Whenever the modified class loader finds a SafeTSA file, it will load the SafeTSA file and set up the necessary JTOC and TIB entries; method invocations on classes loaded from SafeTSA files result in the SafeTSA compiler being invoked to produce executable code. If no SafeTSA file exists in the classpath, the class loader simply loads the appropriate Java class file from the repository, and any method invocation on the JVML class will result in the standard JVML optimizing compiler being invoked to compile the method.

Figure 4 shows the internal flow of data in Jikes RVM while compiling a method. Initially, the compiler transforms the method into its high-level SafeTSA repre-

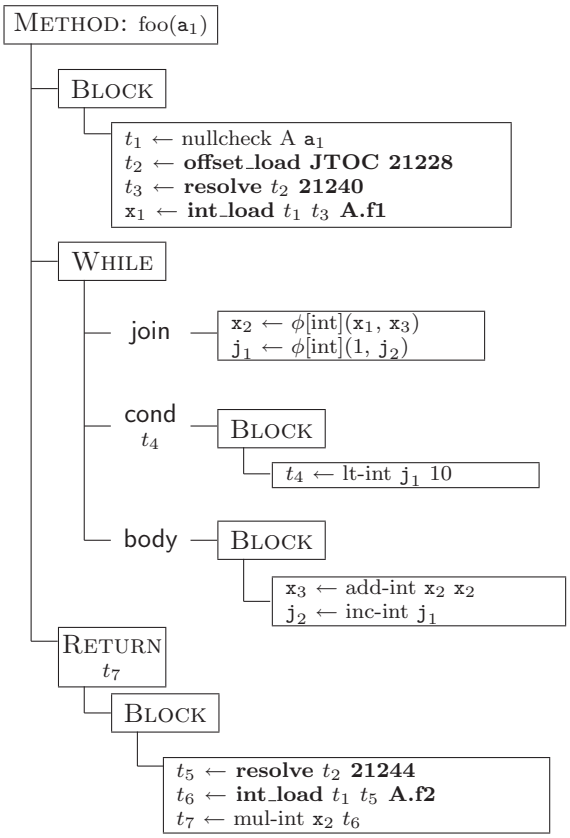

Fig. 6. Low-level SafeTSA (LST) representation of method foo().

sentation (HST). An HST representation of a SafeTSA method is an intermediate representation that is largely independent of the host runtime environment but differs from the original SafeTSA method in that there is some resolution of accessed fields and methods. Next, the SafeTSA method is transformed from HST into the low-level SafeTSA representation (LST). This process expands some HST instructions into a host-JVM specific LST operations specializing them for Jikes RVM's object layout and parameter passing mechanisms. After this transformation, the LST method is optimized and transformed into the same LIR that is used by Jikes RVM's JVML optimizing compiler. Jikes RVM's existing LIR to MIR compilation phase is used unmodified to perform instruction selection, scheduling, and register allocation.

A consequence of Java's dynamic class loading is that a method may refer to fields, methods, and types of classes whose implementation has not yet been loaded into the VM. When the JIT compiler processes one of these references, it will be unable to resolve the reference immediately, and so instead of inserting code to directly access the data structure, the JIT compiler must insert special "resolve" instructions that cause the implementation to be loaded and the appropriate VM

data structures to be instantiated. The SafeTSA compiler inserts these resolution instructions during the creation of the HST IR. This is, in fact, the main difference between the serialized SafeTSA representation and HST: in HST, a getfield_unresolved, setfield_unresolved, or call_unresolved will be substituted for each of the getfield, setfield, or call instructions that operate on classes that are not yet loaded. Figure 5 shows what the HST for the method foo would look like if class A had not been loaded prior to foo's compilation as is evidenced by the getfield_unresolved instructions.

Once the method is in HST, it can be lowered to LST by expanding certain high-level instructions to Jikes RVM specific implementations. Mostly, this consists of performing address computations using offsets from Jikes RVM's JTOC and TIBs. It also involves translating SafeTSA check and cast operations into their Jikes RVM equivalents, materializing constant operands, and translating high-level storage accesses into low-level load and store instructions. Figure 6 shows the optimized LST that would be created from the example program. In the LST, high-level getfield_unresolved instructions have been lowered to sequences of instructions that perform resolution and access the appropriate fields: The offset_load instruction uses the JTOC to find the address for the field offset table, and the resolve instruction finds the offset of the field within its object by loading the class containing the field—if necessary—and then looking the offset up in the field offset table at the appropriate field dictionary index (in the example program, a.f1 has the field dictionary entry 21240 and a.f2 the entry 21244). The final low-level int_load instruction is used to actually load the field once the offset within the object has been determined. Because of common subexpression elimination, the second getfield_unresolved does not require an offset_load instruction when translated to LST.

The translation from LST to LIR is the final phase of the SafeTSA compiler and is composed of three main tasks: the translation of the control structure tree into branch instructions, the straightforward translation of LST instructions into LIR instructions, and the translation from SSA variables into LIR's virtual registers. Figure 7 shows our example program translated into LIR, which consists of 9 basic blocks. Instructions 5–11 resolve 'a.f1'; instructions 21–26 resolve 'a.f2'; instructions 15–20 represent the while-loop.

During the translation of SSA values to virtual registers, ϕ-functions will be replaced with move instructions on the predecessor nodes of the control flow graph. Optionally, the SafeTSA compiler can be instructed to minimize the number of these moved by performing a "ϕ-move optimization." This ϕ-move optimization performs a simple backwards traversal of instructions in the basic block list associated with the body of a loop to determine if the target register of the candidate move instruction is ever used after the definition of the source register; if it is not, then the source register will be renamed (i.e., all users of the source register will be altered to use the target register), and the move will be suppressed.[2] For our example program, this simple algorithm reduces the number of move instruction

[2]A generic register coalescing pass would subsume this ϕ-move optimization, but Jikes RVM's LIR to MIR backend does not perform coalescing pass, so a specialized ϕ-move optimization is used in the SafeTSA compiler.

```
1  LABEL0
2          prologue l0pi(A,x,d) =
3  LABEL1
4          null_check t2v = l0pi(A,x,d)
5          int_load t3i([I) = JTOC(int), 21228
6  LABEL3
7          int_load t4i(int) = t3i([I), 21240
8          int_ifcmp t4i(int), 0, !=, LABEL2
9  LABEL4
10         resolve A.f1
11         goto LABEL3
12 LABEL2
13         int_load t6i(int) = l0pi(A,x,d), t4i(int), A.f1, t2v
14         int_move t7i(int) = 1
15         goto LABEL6
16 LABEL7
17         int_add t6i(int) = t6i(int), t6i(int)
18         int_add t7i(int) = t7i(int), 1
19 LABEL6
20         int_ifcmp t10v = t7i(int), 10, <, LABEL7
21 LABEL8
22         int_load t4i(int) = t3i([I), 21244
23         int_ifcmp t4i(int), 0, !=, LABEL10
24 LABEL9
25         resolve A.f2
26         goto LABEL8
27 LABEL10
28         int_load t11i(int) = l0pi(A,x,d), t4i(int), A.f1, t2v
29         int_mul t12i(int) = t6i(int), t11i(int)
30         return t12i(int)
```

Fig. 7. LIR representation of method foo().

inserted into the LIR of foo() to the minimum, the insertion of one move instruction (instruction 14 in Figure 7) which is need to initialize the register *t7i*.

5. EXPERIMENTS

5.1 Procedure

SafeTSA provides a mechanism for the safe transport of optimized code, but in order to empirically assess whether SafeTSA delivers the expected performance benefits, we compiled a series of benchmarks from Java source code into JVML bytecode and SafeTSA files using a static producer-side ahead-of-time compiler, and then ran them through the Jikes RVM system. For each program, we measured the time required by the Jikes RVM system for JIT compilation and the total time required to execute the benchmark (including JIT compilation time).

For our benchmarks we used the programs contained in Sections 2 and 3 of the Java Grande Forum Sequential Benchmarks (JGF) [Bull et al. 2000], which are freely available in source code and appropriate for measuring the compilation time and performance of the generated code. Table I lists these benchmark programs, provides a short description for each program, and list each program's size

Table I. Java Grande Forum Sequential Benchmarks.

Name	Description	Size (bytes)			Instructions	
		Source	JVML	SafeTSA	JVML	SafeTSA
Section 2						
Crypt	IDEA encryption	25152	5038	3254	1050	721
HeapSort	Integer sorting	15181	3395	2009	301	194
LUFact	LU Factorization	23318	5924	3659	1246	848
SOR	Successive over-relaxation	11016	3466	2130	279	181
SparseMat	Matrix multiplication	9571	3956	2288	312	237
Section 3						
Euler	Fluid Dynamics	41664	18234	15834	7722	7296
Moldyn	Molecular Dynamics	20106	7315	4213	1952	1302
MonteCarlo	Monte Carlo	120911	31771	16987	2970	1878
RayTracer	3D Ray Tracer	43566	17181	8132	1989	1340
Search	Alpha-beta pruned search	27906	9213	6508	2367	1580

Table II. Abbreviation of Mobile-Code Formats and Optimizations.

Mobile-Code Formats			
Java Bytecode (JVML)	J		
SafeTSA	S		
Producer-side Optimizations			
local CSE		C	
global CSE		G	
global dead code elimination		D	
global constant propagation		P	
Optimizations During JIT Compilation			
ϕ-move optimization			Φ
local CSE			C
local dead code elimination			D
local constant propagation			P
guarded method inlining			I
global code motion			M
global value numbering			N
redundant load elimination			L
redundant store elimination			S

as well as number of instructions for different versions. (It should be noted that Java Grande benchmark's calls to start and end timing are designed such that it normally includes compilation time but not JVML verification, SafeTSA decoding time, loading, or linking time so we do not report on them here. In any case, Jikes RVM does not include a bytecode verifier, and these times are small [von Ronne et al. 2006].)

In the following sections, we have utilized two different mobile-code formats, with various optimizations being applied by both the producer-side compiler and also by the just-in-time compilers. In order to facilitate the discussion of these different combinations, we will use a uniform notation in which we will describe them using a tuple of the form *a:b/c*, whereas the first component *a* indicates the mobile-code format, the second component *b* indicates the optimizations performed on the program in the mobile-code format at the producer side, and the third component *c* indicates the optimizations being performed at runtime by the just-in-time compiler.

Table II shows the abbreviations we use in this notation for mobile-code formats and optimizations. JVML is used in the table to denote Java class files produced using version 1.2.2 of Sun javac. As an example, a benchmark run described as *S:GDP/ΦI* would represent the execution of a SafeTSA version of the benchmark programs to which global common subexpression elimination, dead code elimination and constant propagation had been applied during the translation into SafeTSA, and to which ϕ-move optimization and method inlining had been performed by the JIT compiler during the execution of the benchmark program.

In order to better assess to what extent any performance gains attributed to SafeTSA transfer to different architectures, benchmarking was conducted on both of Jikes RVM's target architectures (PowerPC and IA32), Measurements for the PowerPC architecture were obtained on a PowerMac with a 733 MHz PowerPC G4 (7450), 1.5GB of main memory, 64KB L1, 256KB L2, 1MB L3 caches running Mandrake Linux 7.1 (Linux 2.4.17 kernel). As a representative of the IA32 architecture, we have chosen a standard PC with a 1.333GHZ AMD Athlon XP 1500+ processor, 2GB of main memory, 128KB L1 and 256KB L2 caches running SuSE Linux 9.0 (Linux 2.4.21 kernel). One of the main differences between these two target architectures is the number of available registers: the PowerPC has a total of 32 registers, whereas the IA32 has only 8 general purpose registers and an additional 8 floating point registers.

All results were obtained running each machine's operating system in single user mode, and the Jikes RVM systems used for our measurements were generated from a modified version of Jikes RVM 2.2.0 using the FullOptNoGC option (i.e., the Jikes RVM bootimage was itself produced by the Jikes RVM optimizing compiler, all JVM classes were included in the bootimage, and the garbage collection was disabled). This configuration does not include the adaptive optimization system, which is the recommended configuration for newer versions of Jikes RVM, because we found that the adaptive optimization system performed poorly on the Java Grande Forum benchmarks. This finding was consistent on both Intel and PowerPC and across different versions of Jikes RVM, and is a consequence of Jikes RVM spending most of its time compiling and executing a single method in each benchmark. This results in the adaptive optimization system merely postponing full optimization and causes the virtual machine to waste time compiling at lower optimization levels and running under-optimized code. This configuration also disables garbage collection, because we do not expect garbage would have a meaningful effect on the results since the SafeTSA and JVML code should be semantically equivalent producing the same number of memory allocations and garbage, but garbage collection would increase the amount of noise in our measurements.

We ran each benchmark several times and report best result,[3] but in this stable configuration, all of the overall execution times varied by less than 0.005s, and in most cases, the variation was less than we could measure.

We were interested in evaluating the relative merits of JVML and SafeTSA across the spectrum of varying optimization levels. Therefore, our initial measurements were made running both the SafeTSA compiler and Jikes RVM's optimizing byte-

[3]It should be emphasized that benchmark executions have been performed independently, so that each run incorporates a complete JIT compilation of the program.

Table III. Minimal Optimization on PowerPC: Execution and Compilation Times.

Benchmark	**Exec. Time (s)**					**Comp. Time (s)**		
	J:P/-	S:P/-	Δ%	S:P/Φ	Δ%	J:P/-	S:P/-	S:P/Φ
Crypt	5.498	5.370	-2.33	5.392	-1.93	0.121	0.102	0.102
HeapSort	3.084	3.063	-0.68	3.051	-1.07	0.044	0.041	0.040
LUFact	3.320	3.247	-2.20	3.232	-2.65	0.129	0.120	0.118
SOR	10.420	10.388	-0.36	10.377	-0.47	0.044	0.036	0.036
SparseMat	23.266	23.040	-0.97	23.019	-1.06	0.045	0.046	0.046
Euler	59.671	58.132	-2.58	58.154	-2.54	1.592	1.290	1.294
Moldyn	14.785	14.642	-0.97	14.641	-0.97	0.279	0.278	0.278
MonteCarlo	38.175	37.580	-1.56	37.287	-2.33	0.283	0.279	0.278
RayTracer	55.932	56.597	1.19	56.476	0.97	0.225	0.216	0.215
Search	20.067	20.322	1.27	20.324	1.28	0.220	0.160	0.163

Table IV. Minimal Optimization on IA32: Execution and Compilation Times.

Benchmark	**Exec. Time (s)**					**Comp. Time (s)**		
	J:P/-	S:P/-	Δ%	S:P/Φ	Δ%	J:P/-	S:P/-	S:P/Φ
Crypt	3.544	3.609	1.83	3.597	1.5	0.079	0.066	0.064
HeapSort	2.153	2.109	-2.04	2.131	-1.02	0.032	0.029	0.028
LUFact	2.610	2.440	-6.51	2.430	-6.90	0.084	0.077	0.075
SOR	6.177	6.099	-1.26	6.119	-0.94	0.027	0.026	0.026
SparseMat	14.596	14.042	-3.80	14.059	-3.68	0.031	0.032	0.032
Euler	48.123	47.279	-1.75	47.384	-1.54	1.247	0.911	0.915
Moldyn	9.598	9.161	-4.55	8.987	-6.37	0.156	0.154	0.153
MonteCarlo	27.767	27.002	-2.76	27.941	-0.63	0.210	0.200	0.200
RayTracer	28.936	29.105	0.58	29.809	3.02	0.156	0.151	0.150
Search	15.071	15.101	0.20	15.184	0.75	0.130	0.099	0.099

code compiler with no unnecessary optimizations. Next, we examined the performance benefits that can accrue using SafeTSA's producer-side optimization when JIT optimization is still disabled. We then measured the effect of enabling comparable optimizations in the consumer-side JVML and SafeTSA JIT compilers. After that, we evaluated several additional JIT optimizations individually. Finally, we benchmark JVML against SafeTSA utilizing all available optimizations.

5.2 Minimal Optimization

Our initial experiment ran the JVML and SafeTSA JIT compilers utilizing only constant propagation (which is required for static final fields by the Java Language Specification [Joy et al. 2000]) in the code producer. Table III compares the execution times in seconds on PowerPC for the benchmark programs compiled with Jikes RVM's JVML JIT compiler (J:P/-) to the execution times measured when using baseline SafeTSA files (S:P/-) as input for the SafeTSA compiler. It also shows the timings for execution with the SafeTSA JIT's ϕ-move optimization (S:P/Φ), which is really just a more sophisticated translation out of SSA form.

5.2.1 *Execution Time.* The SafeTSA-based version (S:P/-) outperformed the JVML version (J:P/-) in 8 out of 10 benchmarks, but only slightly (speedups between 0.36% to 2.58%). Similar results were observed on the Athlon XP (Table IV); there the largest speedup for SafeTSA is larger (6.51% for LUFact), but only 7 of the 10 benchmark were faster for SafeTSA than JVML.

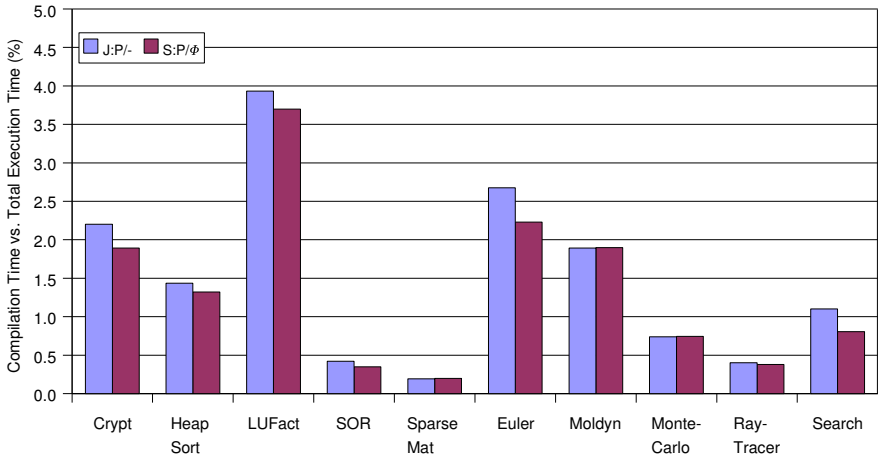

Fig. 8. Compilation Time as a Percentage of Execution Time.

It should not be surprising that there is little difference in execution times, since the only substantive difference between these two is that SafeTSA uses SSA form. The SSA discipline often splits multiply-defined local variables into several distinct virtual registers. This splitting interacts with the register allocator and sometimes results in a better register allocation, especially on IA32, where there is a shortage of registers. In particular, the benchmark programs *LUFact* and *Moldyn* contain a number of local variables with live ranges spanning the entire body of a method that are assigned different values at spaced intervals. The SSA splits these live ranges creating more shorter-lived variables that are less likely to be spilled. But in some other cases, the creation and resolution of ϕ-functions during the translation into and out of SSA can also increase the number of live temporaries at a program point increasing register pressure and hurting the register allocation.

The effect of ϕ-move optimization is interesting. On PowerPC, there is generally a slight additional speedup (up to 0.8% on MonteCarlo), but on the Athlon XP, ϕ-move optimization actually hurt the performance slightly on seven of the ten benchmarks with the worse being an additional 2.44% performance degradation for the RayTracer benchmark. This degradation arose because the ϕ-move optimization coalesces the source and target operands of the ϕ-functions, and this transforms many short-lived variables into longer-lived variables, which will often be assigned by Jike RVM's register allocator to memory locations instead of registers.

5.2.2 *Compilation Time.* The execution times described above include JIT compilation, but Table III and Table IV also list the JIT compilation time separately. It can be seen that the results for S:P/- were not significantly different than those for S:P/Φ.

While important for interactive and short-live programs, compilation time was not a significant component of the total execution time for our benchmarks at this optimization level. As can be seen in Figure 8 the compilation times were always less than 4% of the total compilation time, and about half of the compilation times

came in under 1% of the total execution time. In general, the SafeTSA compiler was slightly faster, but because compilation is only a small component of total execution time, any improvements in compilation time for SafeTSA had only a marginal effect on the total execution time. The same pattern holds for the compilation times on the Athlon [Amme 2004].

5.3 Producer-side Optimization

One of the key advantages of representing programs using an SSA-based format is the ease of utilizing producer-side optimizations. (In JVML, temporaries are carried on the stack and optimizations such as CSE are awkward, require compensation code, and are not generally used.) To ascertain the effect of such producer-side optimizations for short-lived programs which cannot afford extensive JIT optimization, we kept JIT optimization disabled, but created SafeTSA files optimized with different producer-side optimizations and compared their execution time and JIT compile time with that of unoptimized JVML class files.

In particular, we examined constant propagation, dead code elimination, global common subexpression elimination. And on IA32 we also examined local common subexpression elimination. The SafeTSA code producer's constant propagation algorithm identifies arithmetic and move instructions whose operands are constant, and replaces references to them with direct references to the constant result of that computation, and then removes the no-longer referenced instruction. The dead code elimination algorithm identifies all of the instructions of operator types that can cause side effects, marks all of those instructions and—transitively—all of those that they use, and then finally deletes all the instructions that remain unmarked. The global common subexpression elimination algorithm identifies all the instructions (processing them according to a pre-order traversal of the dominator-tree) that are dominated by another instruction with the same operator and operands. It then deletes the dominated instruction replacing all uses of the deleted instruction with references to the identical dominating instruction. In contrast, local common subexpression elimination only identifies and eliminates those instructions that have the same operators and operands as another instruction that precedes it in the same basic block.

5.3.1 *PowerPC Execution Times.* Figure 9 shows the execution time speedup for different SafeTSA versions (S:-/Φ, S:P/Φ, S:DP/Φ and S:GDP/Φ) of each benchmark relative to the execution time of the J:P/- version of the same benchmark on the PowerPC G4. The measurements show that constant propagation and dead code elimination tend to result in rather minor performance improvements. Moreover, in some cases the application of these optimizations can lead to performance degradation (particularly in HeapSort), which is probably caused by reduced program locality. Whereas, except for one benchmark program (SparseMatrix), the global common subexpression elimination consistently improved performance, and was the main contributor to the speedup of the three benchmark programs that showed speedups of over 9%.

To better understanding of the cause of these improvements, we performed additional experiments to determine what instruction eliminations were responsible for the improvements in these three benchmark programs (LUFact, Euler, and Mol-

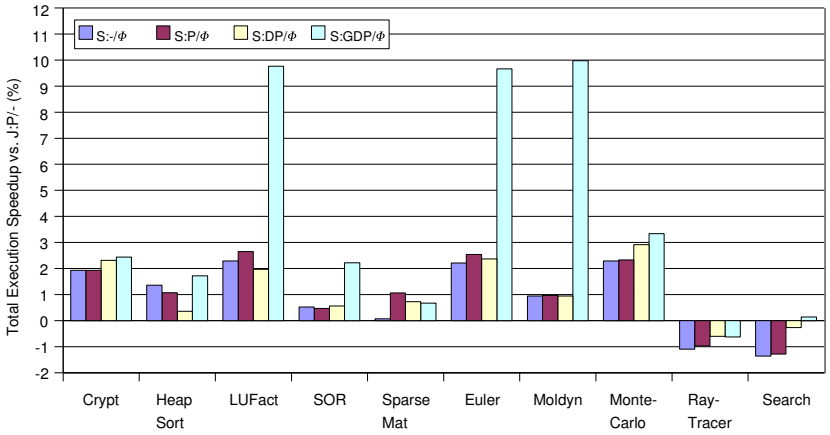

Fig. 9. Execution time of SafeTSA after Producer-side Optimization on PowerPC.

Table V. Instructions, null-checks and bounds-checks eliminated by producer-side CSE

Benchmark	Instructions			Null-checks			Bounds-checks		
	Before	After	Δ%	Before	After	Δ%	Before	After	Δ%
Crypt	721	681	-5.55	86	63	-26.74	81	81	0
HeapSort	194	185	-4.64	24	24	0	21	21	0
LUFact	848	770	-9.20	94	69	-26.60	96	81	-15.63
SOR	181	165	-8.84	23	15	-34.78	23	22	-4.35
SparseMat	237	231	-2.53	29	28	-3.45	32	32	0
Euler	7296	6698	-8.20	1666	1422	-14.65	1230	1103	-10.33
Moldyn	1302	1293	-0.69	108	108	0	49	49	0
MonteCarlo	1878	1773	-5.59	181	140	-22.65	50	46	-8.01
RayTracer	1340	1177	-12.16	203	121	-40.39	14	14	0
Search	1580	1486	-5.95	111	99	-10.81	283	274	-3.18

dyn). We first examined what instructions were removed by the common subexpression elimination. Table V shows the number of total instructions as well as null-checks and bounds-checks that could be eliminated from the SafeTSA files at the producer side. As the table shows, 7% of all instructions, 17% of all null-checks, and 8% of all bounds-checks could be eliminated from the programs. We investigated further, by selectively disabling the elimination of certain operator types, and found that the elimination of bounds-checks is the primary cause of the speedup in the execution of LUFact and Euler while Moldyn's speedup is caused by the elimination of redundant floating point operations in the inner-loop. The null-check eliminations—it turns out—had no effect; Jikes RVM uses hardware memory protection to implement the null-checks, so in most cases, null-check removal does not change the generated machine code.

5.3.2 *IA32 Execution Times.* Figure 10 shows the execution time measured on the Athlon optimized SafeTSA versions relative to unoptimized Java class files on the Athlon XP. In order to avoid the occasional performance degradation caused by ϕ-move optimization, all measurements shown in Figure 10 were measured with the ϕ-move optimization disabled. Surprisingly, on the Athlon XP, only one bench-

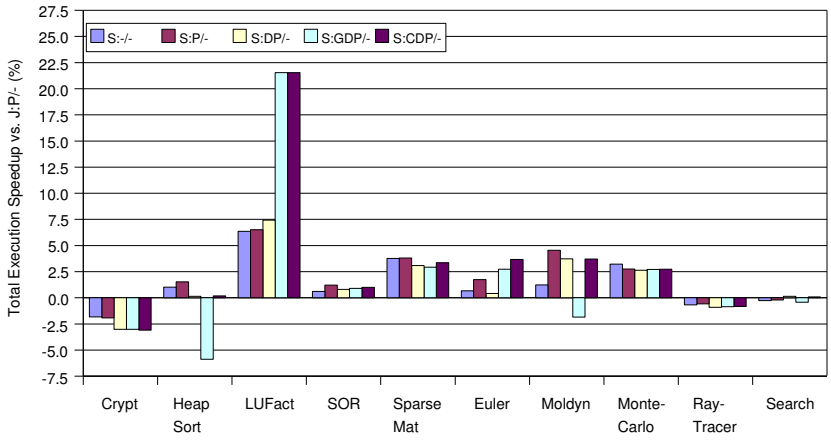

Fig. 10. Total Execution Time Speedup with Producer-side Optimization on IA32.

mark (LUFact) was sped up by common subexpression elimination. In most cases, the execution time was unaffected by common subexpression elimination, but for two benchmark programs (Heapsort and Moldyn), there was a noticeable degradation. Like the performance degradations due to ϕ-move optimization, these can be explained by the reduced number of registers that are available on the IA32 architecture and Jikes RVM's register allocation strategy.

Common subexpression elimination usually increases the live range of a temporary variable so that it can be reused without being recomputed. The spill heuristic of Jikes RVM's register allocator, which favors variables with shorter live ranges, can interact badly with loop invariant subexpression eliminations. In such a case, a variable that is used heavily in an inner-loop can be spilled and allocated a memory location because it is defined outside of the loop and has a long live range.

Based on this observation, further measurements were performed in which common subexpression elimination was restricted to only occur within basic blocks.[4] These measurements, shown as the S:CDP/- bars in Figure 10, indicate that even on the IA32 architecture, local common subexpression elimination leads, in most cases, to a decrease in total execution time and avoids the performance degradation that would sometimes result from global common subexpression elimination, especially for the benchmark programs *Heapsort* and *Moldyn*. In fact, in no benchmark did global common subexpression elimination significantly outperform local common subexpression elimination (S:CPD/-), and S:CPD/- outperformed JVML (J:P/-) by over 2.5% in five of benchmarks, outperformed J:P/- by over 20% for LUFact, and seems to provide the best overall performance that could be attained without JIT optimization on the IA32 architecture.

5.3.3 *Compilation Time.* Figure 11 shows the JIT compilation time speedups on the PowerPC for differently optimized SafeTSA files relative to J:P/- for all

[4] A more general solution to the problem of balancing redundancy elimination and register pressure is found in [Gupta and Bodik 1999]. Also, c.f., rematerialization [Briggs et al. 1994].

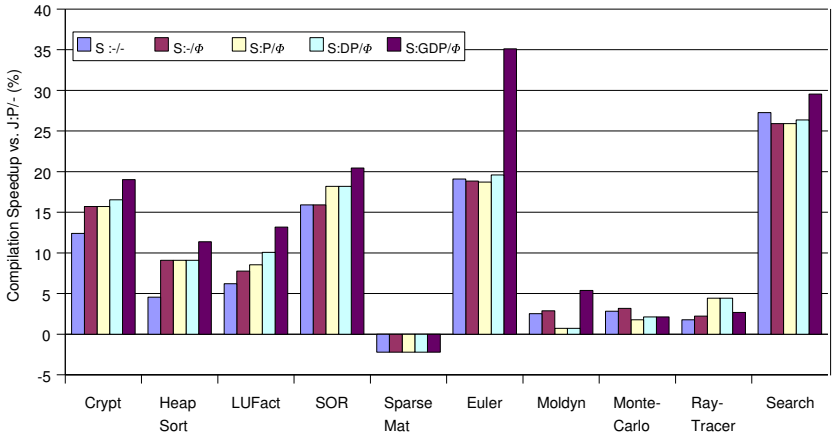

Fig. 11. JIT Compilation Time Relative to Baseline JVML on PowerPC.

benchmark programs. For baseline SafeTSA (S:P/Φ), compilation was up to 27% faster than the compilation of J:P/-, and compilation of fully optimized SafeTSA was up to 35% faster. It is, perhaps, counter-intuitive that increased optimization should reduced compilation time, but the time spent optimizing at the code producer is not counted, and the optimized SafeTSA files have less instructions for the SafeTSA JIT compiler to process during execution.

5.4 Basic Optimization During JIT Compilation

In further experiments, we investigated whether producer-side optimized SafeTSA files retained any benefit over Java classfiles when the equivalent optimizations are performed at runtime. The optimization considered were constant propagation, dead code elimination, and common subexpression elimination. These three optimizations were applied locally to the basic blocks of the JVML files during JIT compilation (J:P/CDP), and two sets of SafeTSA files were generated with either local optimizations (S:CDP/ΦC) and global optimization (S:GDP/ΦC) applied ahead-of-time during the generation of the SafeTSA file. Additionally, during JIT compilation, before generating the LIR, local common subexpression elimination was applied again by the SafeTSA JIT compiler.

Figure 12 and Figure 13 shows the percentage change in execution times measured on the PowerPC G4 and the Athlon XP, respectively, relative to the execution of unoptimized baseline JVML files (J:P/-). In general, with the exception of the MonteCarlo benchmark, on the PowerPC, the measurements indicate that JIT optimizations of JVML code led to smaller improvements in execution time than code-producer optimization of SafeTSA code. Furthermore, shorter execution times (between 0.3% and 3.5%) were only observed for half of the JVML-compiled benchmark programs, whereas for the other half of the benchmarks, optimization of the JVML code during JIT compilation resulted in a performance degradation of between 0.9% and 4.2%. On the Athlon XP, the results were worse, with the only significant improvements resulting from the SafeTSA execution of the LUFact

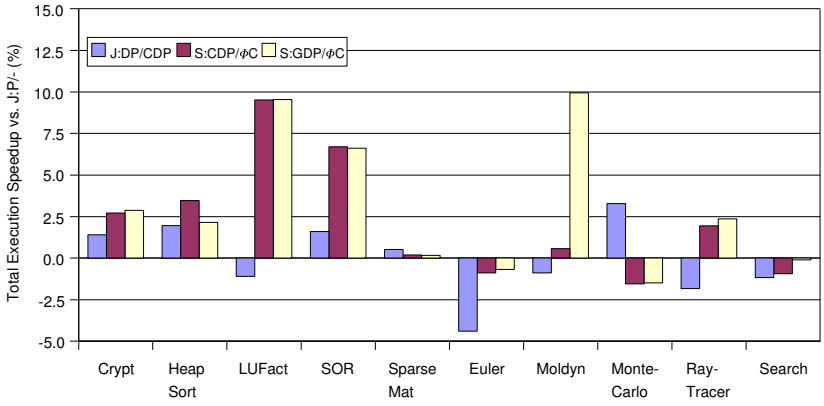

Fig. 12. Total Execution Time Speedup with JIT Optimization on PowerPC.

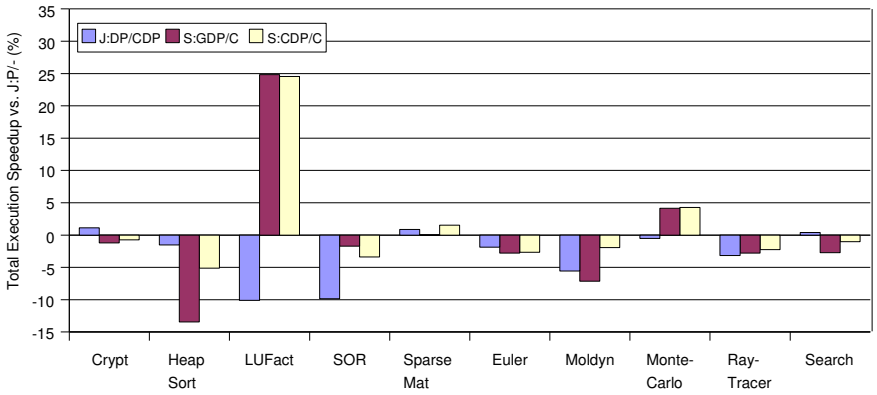

Fig. 13. Total Execution Time Speedup with JIT Optimization on IA32.

(25%) and MonteCarlo benchmark (5%) and many of the other benchmarks showing substantial performance degradations for JVML, SafeTSA, or both. An inspection of code generated by Jikes RVM for these benchmarks showed that an excessive elimination of common subexpressions often leads to an increase in the number of long-lived global values resulting in excessive spilling. This occurs frequently on IA32, but it also caused the performance degradation in Euler on PowerPC when optimized by the JVML JIT compiler.

Table VI contains the absolute compilation times in milliseconds required by Jikes RVM's bytecode compiler and the SafeTSA compiler when these optimizations are utilized during JIT compilation. In most cases, the Jikes RVM bytecode compiler does not need more than 11 ms in order to execute all of the three optimizations; only the optimization of the Euler benchmark requires more time, and in that

Table VI. JIT Compilation Time: Basic Optimizations.

Benchmark	Time in ms	
	J:P/CDP	S:GDP/ΦC
Crypt	11	4
HeapSort	1	1
LUFact	6	2
SOR	1	1
SparseMat	1	1
Euler	940	518
Moldyn	8	4
MonteCarlo	11	6
RayTracer	9	4
Search	11	3

case, an optimizing compilation requires 0.94s. When we examined the compilation more closely, we found that the time required to perform constant propagation and dead code elimination was negligible, and nearly all of the total 0.94s was spent performing common subexpression elimination. The time required by the SafeTSA JIT compiler to accomplish its local common subexpression elimination was generally about half as much as the Jikes RVM bytecode compiler needed and varied between 0.01s and 0.518s. Compilation times for S:CDP/ΦC and S:GDP/ΦC versions of the benchmarks were the same within the precision of our measurements, so only the S:GDP/ΦC times are included in Table VI.

5.5 Extensive JIT Optimization

In order to evaluate SafeTSA's utility when performing more extensive runtime optimizations, we implemented several additional optimizations into the SafeTSA JIT compiler: method inlining, global code motion, global value numbering, and unnecessary load and store elimination. Except for inlining (which is performed at optimization level 0), these optimizations are performed by default in Jikes RVM bytecode JIT compiler at optimization level 2 after it translates the program into SSA form. Therefore, the measurements using these optimizations demonstrates how the dynamic generation of SSA influences the execution and compilation times of the benchmark programs. In order to ensure a fair comparison between dynamic optimizations performed on the SafeTSA and JVML programs, we implemented these optimizations in the SafeTSA compiler using algorithms as close as was practical to those already found in Jikes RVM's bytecode compiler.

5.5.1 *Inlining.* Paralleling the implementation of inlining in the Jikes RVM we integrated ordinary and guarded inlining [Arnold et al. 2000] into the SafeTSA compiler. We found that the inlining of a method does not invalidate the producer-side SafeTSA optimizations, and our measurements show that the improvements in execution time resulting from inlining was nearly the same for SafeTSA and JVML files. Unlike typical object-oriented code, the Java Grande benchmarks spend most of their time in a single large procedure, so it is not surprising that ordinary inlining improved execution time negligibly on both the PowerPC G4 and Athlon XP with the largest improvement on PowerPC measuring 2.56% for the RayTracer benchmark. An application of the more aggressive guarded inlining on the PowerPC,

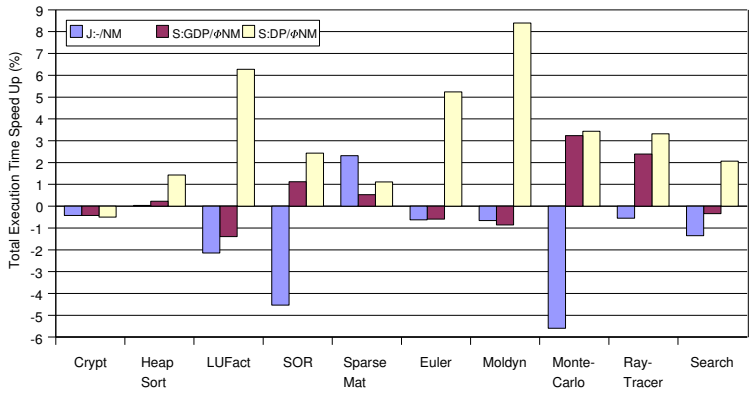

Fig. 14. Total Execution Time Speedup with GVN and GCM.

however, resulted in a considerable improvement (of over 22%) for RayTracer, but even the application of a guarded inlining did not improve the execution time of the other benchmark programs. On the Athlon XP, the effect of aggressive guarded inlining was slight, ranging from a 3.02% performance degradation (*SOR*) to a 1.72% speedup (*RayTracer*). (The performance degradations can be explained by the increased live ranges of temporary variables that surround the inlined methods, which thus become more likely to be spilled.)

When we examined the compilation times on PowerPC, we found that, for both strategies, better performance behavior could be observed for the SafeTSA versions. For guarded inlining the time required by the SafeTSA compiler varied between 0.01s and 0.09s, and the Jikes RVM bytecode compiler needed between 0.02s and 0.29s.

5.5.2 *Global Value Numbering and Global Code Motion.* Global code placement strategies are popular optimizations which are applied in several existing optimizing JIT compilers. In Jikes RVM's bytecode compiler, global code placement, which is performed on the HIR as well as the LIR of a program, consists of a simplified global value numbering (GVN) followed by global code motion (GCM). Global value numbering starts with the transformation of the considered program representation (HIR or LIR) into its SSA form and the construction of the program's dominator tree. In a subsequent step, an analysis finds identical instructions using a technique based on Alpern, Wegman and Zadeck's [Alpern et al. 1988]. During this analysis the instructions are divided into numbered congruence classes, i.e. equal numbers represent congruent instructions. Information about these congruence classes will be used for finding and eventually eliminating common subexpressions in the program. The global code motion implementation[5] in the Jikes RVM's bytecode compiler examines each instruction of a program and checks the earliest and latest scheduling point that will not violate program semantic and moves that instruc-

[5]the version the GCM algorithm used in Jikes RVM—notwithstanding the documentation—was not limited to acting upon loop invariant code

tion to the block between the scheduling points expected to be executed the least frequently.

The implementation of GVN and GCM in the SafeTSA compiler is based on the algorithms of Click [Click 1995], which are broadly comparable to the techniques used in Jikes RVM's bytecode compiler. Global code motion starts with early scheduling, which determines, for each instruction, the first basic block at which it is still dominated by its inputs. The next phase is late scheduling; this phase finds each instruction's last permissible basic block that still dominates all of that instruction's uses. After finishing these two stages, the first and last possible basic blocks for instruction placement are given, and thus a safe-placement range is defined. Finally, it is necessary to find the optimal block with regard to loop nesting depth and control dependence, and place the instruction in that block.

In our implementation, the scheduling process for each instruction starts with the last permissible basic block. During a backward traversal of basic block from this one to the first permissible basic block, the algorithm searches for the block with the lowest loop nesting depth. If more than one such blocks exist, the instruction will be placed at the beginning of the basic block that has been found first. When no basic block with a lower loop nesting depth than the last permissible basic block exists, the instruction is placed at the beginning of that basic block.

Global value numbering in SafeTSA's compiler is also based on the work of Click [Click 1995] and finds identical expressions, algebraic identities and performs constant folding. In contrast to the algorithm used by Jikes RVM's bytecode compiler, it uses hash table techniques to identifying equivalent instructions and replaces them with a single occurrence. To better match the algorithm in Jikes RVM, but in contrast to Click's original work, our implementation does not eliminate identical expressions that reside on divergent execution paths, and thus—like Jikes RVM's GVN—is really a form of global common subexpression elimination.

Preliminary performance measurements indicated that GCM does not have a visible influence on our benchmarks' execution times for either JVML or SafeTSA. Therefore, performing GVN and GCM on benchmark programs has nearly the same effect on execution time as global common subexpression does. Figure 14 depicts the relative improvement in execution times that was actually measured when performing both GVN and GCM on both JVML and SafeTSA programs. Measurements of SafeTSA program execution confirmed our expectation, and on the PowerPC, the SafeTSA programs generated without common subexpression elimination on the producer side (S:DP/ΦNM) showed a relative improvement of up to 8.39% (for Moldyn), but the SafeTSA programs that were fully-optimized at the producer side (S:GDP/ΦNM) resulted in an maximum improvement of only 3.23% (MonteCarlo). Measurements for SafeTSA programs on the Athlon XP which were generated without common subexpression elimination on the producer side (S:DP/ΦNM) showed relative improvement of 19.14% for *LUFact*, whereas the other benchmarks were unchanged or showed degradations of up to 4.13%. Contrary to our expectations, GVN and GCM was not as effective for JVML programs (J:P/NM) and, on the PowerPC, only improved the execution of one benchmark *SparseMat* (by 2.32%). For all other benchmark programs the execution time became between -0.42% (for Crypt) and -5.59% (for MonteCarlo). On the Athlon XP, the only JVML (J:P/NM)

Table VII. Compilation time for GCM and GVN (in ms).

	J:P/NM			S:GDP/ΦNM	S:DP/ΦNM
BENCHMARK	SSA	GVN*	TOTAL	TOTAL	TOTAL
Crypt	84	54	156	46	47
Heapsort	34	18	64	19	26
LUFact	97	78	187	52	57
SOR	24	12	50	18	18
Sparse	28	16	56	20	22
Euler	1108	488	2501	538	589
Moldyn	199	152	477	72	73
MonteCarlo	169	62	323	125	140
RayTracer	144	52	290	89	93
Search	159	98	361	82	83

program to show an improvement (of 9.66%) was *LUFact*, and the other benchmarks showed performance degradations of up to 7.02%.

The disappointing effect of GVN and GCM on the execution of JVML programs can be explained by the compilation time needed by Jikes RVM's bytecode compiler to perform these optimizations. Further investigation showed that the number of instruction eliminated by Jikes RVM bytecode compiler and the SafeTSA JIT compiler were nearly identical, however, these improvements were not enough to make up for the extra compilation times to perform these optimizations. As Table VII shows, for the benchmark programs *Euler* and *Moldyn*, these compilation times rose to 2.501s and 0.477s, respectively, on the PowerPC. One reason for such large compilation times is that before applying GVN the intermediate representation of the program must be transformed into SSA. Since GVN is performed on the HIR as well LIR of a program, this transformation is actually performed twice for each benchmark program. Total costs for transformation of HIR and LIR into SSA form (see column SSA in Table VII) varied from between 0.024s (for SOR) to a very long 1.108s (for Euler). Furthermore, the process of identifying equivalent instructions, that again must be performed on HIR and LIR, also turns out to be very time-consuming (see column GVN* in Table VII) and required up to 0.488s for the benchmark *Euler*. In contrast, the measured overhead of GVN and GCM—which in the SafeTSA compiler are performed on LST and HST—was in most cases nearly negligible and fell between 0.018s and 0.538s for producer-optimized (i.e., S:GDP/ΦNM) and between 0.018s and 0.589s for non-producer-optimized (i.e., S:DP/ΦNM) programs. Similar compilation performance was observed on the Athlon XP and lay between 0.029s (*SOR*) and 1.497s (*Euler*) for JVML files and 0.011s (*HeapSort*) and 0.289s (*Euler*) for SafeTSA files. These reduced compilation times for SafeTSA can be explained by the fact that GVN and GCM are performed directly on the more compact SafeTSA-derived intermediate representation, which among other things, made finding equivalent instructions less expensive.

5.5.3 *Load and Store Elimination.* The implementation of redundant load and store elimination in Jikes RVM's bytecode compiler is based upon a unified analysis technique of array and object references that had been developed by Fink, Knobe and Sarkar [Fink et al. 2000]. With this technique, load elimination and store elimination are performed separately, and each of these consist of three different passes through program's intermediate representation. The first step of redundant

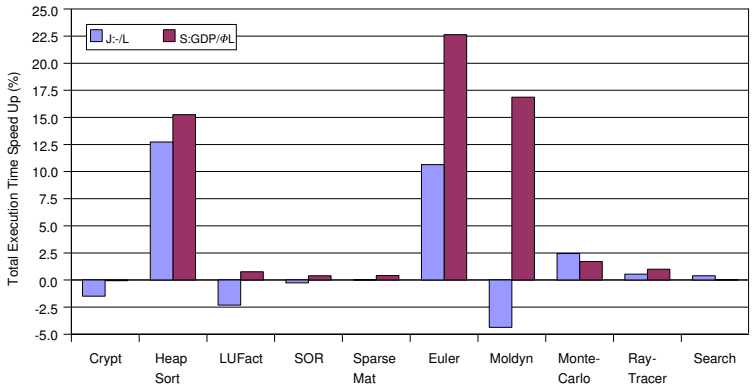

Fig. 15. Total Execution Time Speedup with Load Elimination.

load eliminations is to translate the program into an extended Array SSA form [Knobe and Sarkar 1998]. Then global value numbering is used to determine for each load instruction *inst* the set of reaching load and store instructions that could be considered for a replacement of *inst*. Lastly, based on this information, redundant load instructions are eliminated. Redundant store elimination can be accomplished in similar fashion, but this optimization identifies, for each instruction *inst*, the set of store instructions that can be executed after *inst*; this information is then used to eliminate the redundant stores.

Rather than generating the Array SSA form of a program, the SafeTSA JIT compiler performs both the load and the store eliminations directly on the SafeTSA program format. Like the Jikes RVM's bytecode compiler, the SafeTSA compiler performs load and store elimination separately, but it only needs two passes for each optimization. The first pass gathers the necessary program information (i.e., set of reaching load and store instructions, or the set of subsequently executed store instructions), and the second pass eliminates the redundant load or store instructions. Unlike the JVML bytecode compiler, the SafeTSA compiler does not employ global value number to determine which load and store instructions are redundant; instead, it uses a native data flow analysis.

Figure 15 shows the improvements in execution time, relative to non-optimized program execution, that we measured when performing redundant load elimination onto the benchmark programs. Except for the benchmark program *Crypt*, redundant load elimination in the SafeTSA JIT compiler, consistently lead to a significant improvement in runtime performance on the PowerPC and even greater improvements on the Athlon XP, on which loads are relatively more expensive. The highest runtime speedups on the PowerPC were measured for the benchmark program *Euler* (22.64%), *MolDyn* (16.86%) and *HeapSort* (15.25%), and the redundant load elimination for the *Euler* Benchmark resulted in an absolute speed up of 12.36s. On the Athlon XP, the largest speedup was 33.63% for the *Euler* benchmark.

In contrast, the application of this optimization to the JVML benchmarks programs resulted in considerably smaller speedups than for SafeTSA programs. For four benchmark programs, this optimizations even increased the execution time of

Table VIII. Compilation time for redundant load elimination (in ms).

	J:P/L			S:GDP/ΦL
BENCHMARK	SSA	GVN*	TOTAL	TOTAL
Crypt	21	29	62	11
Heapsort	13	5	28	1
LUFact	46	18	92	8
SOR	7	3	15	2
Sparse	10	3	24	2
Euler	1085	705	2270	397
Moldyn	128	42	331	20
MonteCarlo	80	22	147	15
RayTracer	62	26	127	7
Search	73	44	162	32

the benchmark programs. For the JVML benchmark programs highest speedups on PowerPC could be measured for the *HeapSort* (12.73%) and *Euler* (10.65%), with an absolute speedup for the *Euler* Benchmark of only 7.43s.

The main reason for the disappointing speedup when performing redundant load elimination on the JVML benchmark program is the high compilation time required to performing this optimization on bytecode programs. Table VIII contains the absolute compilation times in milliseconds that Jikes RVM's bytecode compiler and the SafeTSA compiler require when performing a redundant load elimination on the benchmarks. The measurements show that the Jikes RVM bytecode compiler needed between 0.015s and 2.270s in order to execute this optimization. In the case of *Euler*, the extra 2.270s of compilation time had a substantial negative effect on the overall performance. In contrast, compilation times required from the SafeTSA compiler for the optimization were significant lower and lay between 0.001s and 0.397s. On the Athlon XP, the compilation times required by the SafeTSA compiler for load elimination lay between 0.001s (*HeapSort*) and 0.181s (*Euler*), whereas the compilation times for JVML lay between 0.008s (*SOR*) and 1.320s (*Euler*).

The superiority of SafeTSA's load elimination is to be found—among other things—in its use of the SSA form. Optimizations that are executed for the SafeTSA files on the producer side (S:GDP/ΦL) make subsequent optimization more efficient, and the SafeTSA compiler as a whole uses a more efficient internal SSA form representation. A further inspection of Jikes RVM load elimination algorithm discovered, that in contrast to the description in [Fink et al. 2000], an accurate load elimination cannot be done in one iteration of the algorithm. In facts, for discovering all redundant load instructions the algorithm must performed iteratively, whereby the number of its iterations depend on the maximal depth of the access paths used by the load instructions in the program. Therefore, for some of the benchmark programs Jikes RVM's load elimination algorithm does not terminate for five iterations. Since the program information can become invalidated, each iteration must update and rebuild the Array SSA form, perform global value numbering, and recalculate reaching load and store instructions. As shown in Table VIII, the repeated (re-)construction of SSA form and (re-)examination of global value numbering (GVN*), in particular, influence the execution time required by Jikes RVM for load elimination. For example, for the *Euler* Benchmark, these passes required 1.085s and 0.705s, respectively.

Table IX. Full Optimization Comparison of SafeTSA and JikesRVM.

	Compilation time (in ms)			execution time (in secs)	
	JVML		SafeTSA	JVML	SafeTSA
BENCHMARK	SSA	TOTAL	TOTAL	TOTAL	TOTAL
Crypt	478	1298	142	6.626	5.511
Heapsort	139	357	53	2.601	2.628
LUFact	635	1549	150	4.304	3.010
SOR	108	0274	50	8.755	9.718
Sparse	145	0375	68	22.592	23.064
Euler	2869	8083	1275	50.45	42.006
Moldyn	1426	9359	289	25.175	11.763
MonteCarlo	554	1576	352	37.68	36.690
RayTracer	651	1755	273	40.66	38.957
Search	875	2119	458	20.088	19.632

Since redundant store instructions appear very little or not at all in the JGF benchmark programs, the measurements for this optimization are not really meaningful. In general, the measurements of the influence of a store elimination on a program's execution time indicate that the execution time of the benchmarks increased by the same amount that was required by the compiler to perform the store elimination. On the PowerPC, the additional overhead of a store elimination fell between 0.033s and 1.561s for the Jikes RVM compiler, and between 0.001s and 0.353s for the SafeTSA compiler.

5.6 Full Optimization

As we've seen, construction of SSA form in Jikes RVM's optimizing bytecode compiler is very time-consuming, especially since during different optimization phases SSA form can be destroyed and therefore must be rebuilt. In contrast, the SafeTSA compiler uses an internal SSA form representation that is based on pointer structures that always remain valid and don't need to be reconstructed. We performed our final experiment, in order to ascertain the effect this has when extensive optimization is used.. In this experiment, we ran Jikes RVM bytecode compiler in optimization level 2,[6] and we compared its performance with that of SafeTSA's optimizing compiler when activating all possible optimizations (i.e. S:GDP/ΦDPILNM). Although Jikes RVM's bytecode compiler performs some more optimizations when running in optimization level 2 than the SafeTSA compiler, the effect of these additional optimizations is small, and therefore is—nevertheless—a fair overall comparison of the two compilers.

Table IX reports execution time and SSA construction time for fully-optimized benchmarks. In most cases, the SafeTSA programs outperform the JVML programs, except in three of the benchmark, in which the JVML version is slightly faster. But as can be seen in Figure 16, there are several benchmarks programs for which the SafeTSA version performs significantly faster than the JVML version (*Crypt, LUFact, Euler, and Moldyn*). For *Euler* the difference in absolute execution

[6]Besides different method inlining strategies, global code placement, and redundant load eliminations, in optimization level 2 Jikes RVM bytecode compiler performs local common subexpression elimination, local copy propagation, code restructuring applied inside and outside of basic blocks, and local as well as global constant propagation

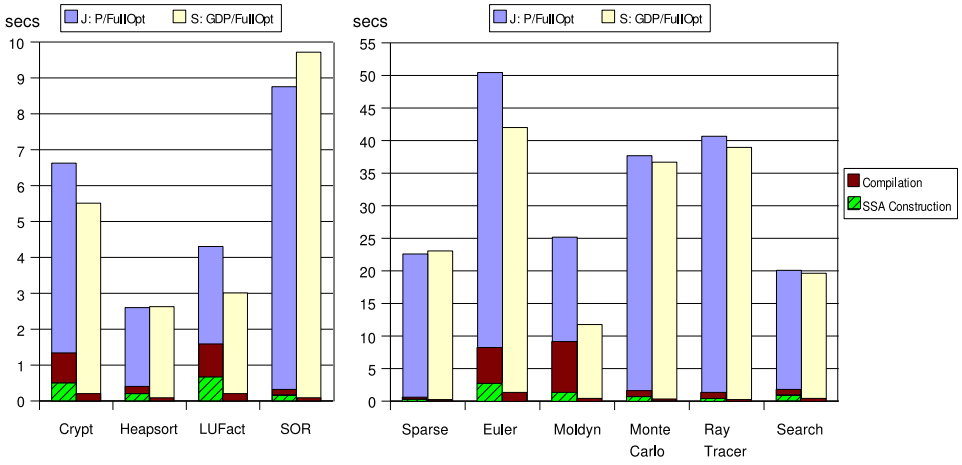

Fig. 16. Full Optimized Program Execution: Absolute Execution and Compilation Times.

time between JVML and SafeTSA rose to 8.444s on the PowerPC. The reason for these differences was the substantial compilation times required by the Jikes RVM bytecode compiler is easily seen by looking at the portion of execution time spent in compilation, which is also shown in Figure 16. For the *Euler* benchmark this measured 8.083s, and a significant amount of this compilation time (2.869s) was spent in SSA construction. In contrast, the SafeTSA compiler needed only 1.275s for JIT compilation of *Euler* with full optimization.

6. CONCLUSION

By integrating SafeTSA support into the Jikes Research Virtual Machine, we created a complete runtime environment that supports a heterogeneous mix of SafeTSA and JVML class files, and that we could use to investigate the practical ramifications of using an SSA-based mobile code representation. We ran the Java Grande Sequential Benchmarks using the JVML and SafeTSA compilers with a range of producer-side and consumer-side optimizations on both the PowerPC and the IA32 architectures in an attempt to assess the relative merits of SafeTSA vs. JVML as inputs to a JIT compiler.

In the process, we found that the scarcity of registers in the IA32 architecture is a major factor in determining the effect of redundancy elimination on execution time. In particular our experiments show that on Jikes RVM, local—rather than global—redundancy elimination provides the best balance between avoiding unnecessary computations and avoiding increased register pressure. The conditions under which this occurs and the means to ameliorate this effect deserve further investigation.

On both architectures, however, these experiments show that the conjectured advantages of SSA for mobile code hold in practice. When JIT compilation time is constrained, SafeTSA's natural producer-side optimizations results in faster-executing code, and when more extensive JIT optimizations, such as load elimination, are used, SSA form gives SafeTSA code a head start allowing it to outperform JVML code. Thus, a system based on SafeTSA can indeed produce as good or better

performing native code with a shorter compilation time across a wide range of optimization levels.

ACKNOWLEDGMENTS

We wish to thank Tino Mai for his assistance in preparing some of the figures, Andreas Finn for implementing load elimination, and Hartmut Arlt for implementing GVN/GCM. We would also like to thank the anonymous referees, Fermín Reig, Andreas Hartmann, and Ning Wang for their comments and suggestions which have greatly improved this paper. In addition, we would like to acknowledge everyone else who has contributed to the SafeTSA project in some way, especially Philipp Adler, Alexander Apel, Hartmut Arlt, Niall Dalton, Andreas Finn, Andreas Hartmann, Thomas Heinze, Tino Mai, Marc-André Möller, Jan Peterson, Prashant Saraswat, and Ning Wang. We also wish to extend our thanks to Michael Hind and the other contributors to the Jikes Research Virtual Machine at IBM Research and elsewhere.

This investigation has been supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants AM-150/1-1 and AM-150/1-3, by the National Science Foundation (NSF) under grant CCR-9901689, and by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-99-1-0536.

REFERENCES

Alpern, B., Attanasio, D., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, C., Fink, S. J., Grove, D., Hind, H., Hummel, S. F., Lieber, D., Litvinov, L., Mergen, M., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, S., Smith, S., Sreedhar, V. C., Srinivasan, S., and Whaley, J. 2000. The Jalapeño virtual machine. *IBM Systems Journal 39,* 1 (Feb.), 211–238.

Alpern, B., Wegman, M. N., and Zadeck, F. K. 1988. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 1–11.

Amme, W. 2004. Effiziente und sichere Codegenerierung für mobilen Code. Habilitation, Friedrich-Schiller-University, Jena, Germany.

Amme, W., Dalton, N., Franz, M., and von Ronne, J. 2000. A typed-safe mobile code representation aimed at supporting dynamic optimization at the target side. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO'2000)*. ACM Press, Monterey, CA.

Amme, W., Dalton, N., Franz, M., and von Ronne, J. 2001. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*. ACM SIGPLAN Notices, vol. 36. ACM Press, Snowbird, Utah, USA, 137–147.

Amme, W., Dalton, N., Fröhlich, P., Haldar, V., Housel, P. S., von Ronne, J., Stork, C. H., Zhenochin, S., and Franz, M. 2001. Project transprose: Reconciling mobile-code security with execution efficiency. In *Proceedings of the Second DARPA Information Survivability Conference and Exposition*. IEEE Computer Society, Los Alamitos, California, 196–210.

Appel, A. W. 1998. Ssa is functional programming. *SIGPLAN Not. 33,* 4, 17–20.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA'2000)*. ACM SIGPLAN Notices, vol. 35. ACM Press, New York, 47–65.

Arnold, M., Fink, S., Sarkar, V., and Sweeney, P. F. 2000. A comparative study of static and profile-based heuristics for inlining. In *DYNAMO '00: Proceedings of the ACM SIGPLAN*

workshop on Dynamic and adaptive compilation and optimization. ACM Press, New York, NY, USA, 52–64.

Belady, L. A. 1960. A study of replacement algorithms for virtual storage computers. *IBM Journal of Research and Development 5,* 2, 78–101.

Breazu-Tannen, V., Coquand, T., Gunter, C. A., and Scedrov, A. 1991. Inheritance as implicit coercion. *Information and Computation 93,* 1 (July), 172–221.

Briggs, P., Cooper, K. D., and Torczon, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16,* 3 (May), 428–455.

Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. 2000. A benchmark suite for high performance Java. *Concurrency: Practice and Experience 12,* 6 (May), 375–388.

Burke, M., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M., Sreedhar, V. C., and Srinivasan, H. 1999. The jalapeño dynamic optimizing compiler for java. In *ACM Java Grande Conference*. ACM Press, New York, 129–141.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the Symposium on Compiler Construction (CC'1982)*. ACM, ACM Press, New York, 98–105.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via coloring. *Computer Languages 6,* 1, 47–57.

Click, C. 1995. Global code motion/global value numbering. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 246–257.

Cooper, K. and Torczon, L. 2003. *Engineering a Compiler*. Morgan Kaufman, San Francisco.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

de Bruijn, N. G. 1978. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Indagationes Mathematicae (Proceedings) 81,* 3, 348–356.

ECMA 2002. Common Language Infrastructure (CLI), Standard ECMA-335.

Fink, S. J., Knobe, K., and Sarkar, V. 2000. Unified analysis of array and object references in strongly typed languages. In *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, J. Palsberg, Ed. Lecture Notes in Computer Science, vol. 1824. Springer, Heidelberg, 155–174.

Franz, M. and Kistler, T. 1997. Slim Binaries. *Communications of the ACM 40,* 12 (Dec.), 87–94.

Fraser, C. W. and Hanson, D. R. 1992. Simple register spilling in a retargetable compiler. *Software – Practice and Experience 22,* 1 (Jan.), 85–99. C. Fraser and D.R. Hanson, "Simple Register Spilling in a Retargetable Compiler", *Software - Practice and Experience*, Vol.22, No.1, 1992, pp. 85-99.

Gupta, R. and Bodik, R. 1999. Register pressure sensitive redundancy elimination. *Lecture Notes in Computer Science 1575*, 107–121.

Hecht, M. S. and Ullman, J. D. 1974. Characteristics of reducible flow graphs. *Journal of the ACM 21,* 3 (July), 367–375.

Joy, B., Steele, G., Gosling, J., and Bracha, G. 2000. *Java Language Specification*, 2 ed. Addison-Wesley Professional, Boston, MA, USA.

Kistler, T. and Franz, M. 1999. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming 27,* 1 (Feb.), 21–34.

Knobe, K. and Sarkar, V. 1998. Array ssa form and its use in parallelization. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 107–120.

Lattner, C. and Adve, V. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, Palo Alto, California.

League, C., Trifonov, V., and Shao, Z. 2001. Functional Java bytecode. In *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics*. International Institute of Informatics and Systemics, Orlando, FL.

Menon, V. S., Glew, N., Murphy, B. R., McCreight, A., Shpeisman, T., Adl-Tabatabai, A.-R., and Petersen, L. 2006. A verifiable ssa program representation for aggressive compiler optimization. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 397–408.

Motwani, R., Palem, K. V., Sarkar, V., and Reyen, S. 1995. Combining register allocation and instruction scheduling. Technical Note CS-TN-95-22, Stanford University, Department of Computer Science. Aug.

Poletto, M. and Sarkar, V. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems 21,* 5 (Sept.), 895–913.

Stork, C. H. 2006. Compressed abstract trees and their applications. Ph.D. thesis, University of California, Irvine.

Stork, C. H., Haldar, V., and Franz, M. 2000. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Information and Computer Science, Univeristy of California, Irvine. Nov.

von Ronne, J. 2005. A safe and efficient machine-independent code transportation format based on static signle assignment form and applied to just-in-time compilation. Ph.D. thesis, University of California, Irvine.

von Ronne, J., Amme, W., and Franz, M. 2006. An inherently type-safe ssa-based code format. Tech. Rep. 2006-004, Computer Science, The University of Texas at San Antonio.