

Using the SafeTSA Representation to Boost the Performance of an Existing Java Virtual Machine

Wolfram Amme
Friedrich-Schiller-Universität Jena
Institut für Informatik
Ernst-Abbe-Platz 1-4
D-07743 Jena, Germany
amme@informatik.uni-jena.de

Jeffery von Ronne
University of California, Irvine
Information and Computer Science
127B Computer Science Trailer
Irvine, CA 92697-3430, USA
jronne@ics.uci.edu

Michael Franz
University of California, Irvine
Information and Computer Science
444 Computer Science Bldg.
Irvine, CA 92697-3425, USA
franz@uci.edu

Abstract

High-performance just-in-time compilers for Java need to invest considerable effort before actual code generation can commence. SafeTSA, a typed intermediate representation based on SSA form, was designed to ease this burden, decreasing the time required for dynamic compilation, without sacrificing safety or code quality.

We report on our experience integrating support for loading and compiling safeTSA files into the Jalapeno Java Virtual Machine. The net result of this integration is a Java runtime which is capable of executing classes from Java class files, safeTSA, or a heterogeneous mixture of the two. This system's safeTSA compiler is patterned after the Jalapeno optimizing compiler and currently shares its low-level code generator.

Preliminary performance results are very encouraging and show simultaneous improvements in both compilation time and code quality relative to Jalapeno's standard optimizing compiler for JVM class files. This supports the hypothesis that SSA-based intermediate representations, such as safeTSA, offer unique advantages in the context of just-in-time compilation.

Introduction

SafeTSA, introduced in [3], is a type-safe intermediate representation based on static single assignment (SSA) form. It is hypothesized that, compared to the stack model found in the Java Virtual Machine's bytecode language (JVML), SSA form possesses significant advantages as the underlying model of intermediate representations for mobile code, especially in the context of just-in-time (JIT) compilation, where dynamic compilation time is critical. Because dataflow is explicit in SSA form, type checking and certain classes of optimization become computation-

ally inexpensive for programs in a representation such as safeTSA. In addition, SSA, being an intermediate representation originally developed for use in optimizing compilers, is compatible with a greater range of expensive optimizations, which can be applied offline.

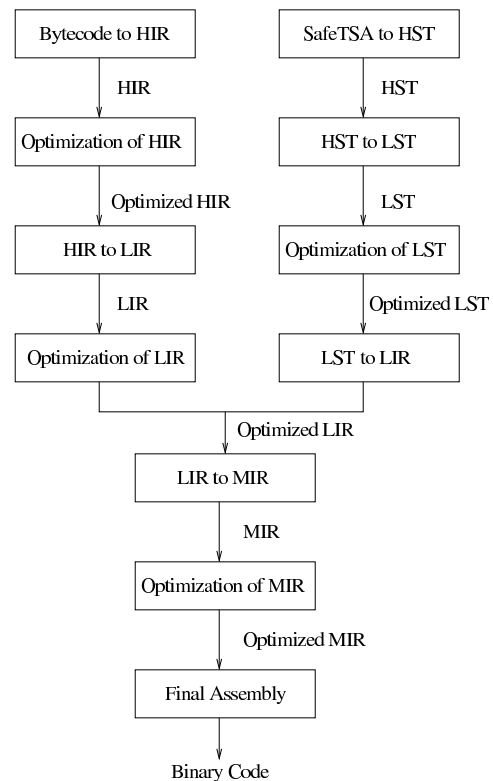


Figure 1. Compiling JVM and safeTSA methods in Jalapeno

To assess whether an SSA-based intermediate representation format can serve as a feasible replacement for JVML and to quantify any of its benefits, a prototype compila-

tion system for safeTSA was developed. An initial Java to safeTSA compiler was reported in [2] and [3]. This paper, reports the integration of a safeTSA to PowerPC dynamic compiler into IBM's Jalapeno JVM¹. Currently this safeTSA to PowerPC compiler utilizes some of Jalapeno optimizing compiler's facilities, including the low-level code generation, as seen in Figure 1.

The result is a Java virtual machine, which is capable of executing both Java class files and safeTSA files. It can even execute programs in which some of the classes were compiled to Java class files and others were compiled to safeTSA files; the file format is completely transparent to the executing program.

In an attempt to quantify the relative merits of JVM and safeTSA (and by extension stack-based vs. SSA-based IRs) in the context of JIT compilation, a series of benchmarks were compiled into Java class files and safeTSA files, which were used as input for Jalapeno's JVM optimizing compiler and our safeTSA compiler, respectively. This allows one to compare both the code-generation time required by the compiler and the performance of the compiled code.

In the following sections, some of the key features of the safeTSA format are introduced, and a brief overview of the Jalapeno system is given, particularly its code generator and internal data structures. Following this is a description of the the safeTSA compiler's implementation and its integration into the Jalapeno system. Next is a discussion of the benchmark results, reporting on both code-generation time and on the generated code's performance. A related work section follows, and the paper concludes with a summary and discussion of future work.

The SafeTSA Representation

SafeTSA[3] is a type safe intermediate representation designed as a machine-independent program representation which is both trivial to verify and easy to translate into optimized machine code. SafeTSA achieves this through the novel combination of several key features: the control structure tree, instructions in SSA form, dominator-based referential integrity, type safety through type separation, a type system extended to support key optimizations, and a carefully chosen instruction set.

Figure 3 contains a graphical representation of the safeTSA file produced from the source program in Figure 2. It will be referred to in the following discussion of safeTSA's key features.

Rather than allowing arbitrary branch instructions, safeTSA conveys the program's control flow through a tree

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

public class t1 {
    static int foo (A a) {
        int sum = a.f1;
        int i = 1;
        while (i < 10) {
            sum += sum;
            i++;
        }
        return sum * a.f2;
    }
}
```

Figure 2. An example program

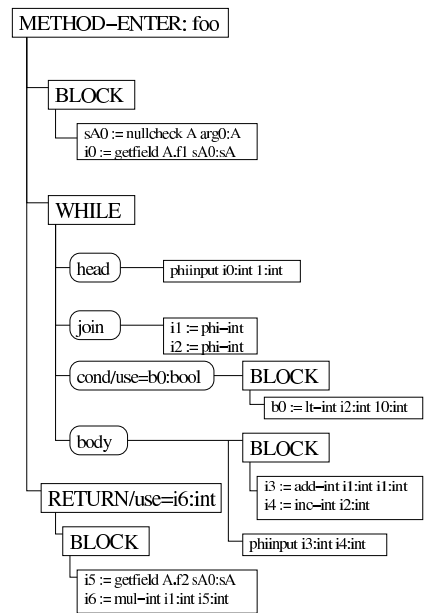


Figure 3. An example program in safeTSA

¹Our current prototype is based on IBM Research's university release of Jalapeno 1.1B. Newer versions of the Jalapeno system have been released as an open source project under the name Jikes Research Virtual Machine.

of high-level control structure elements closely paralleling those of the Java source language. This *control structure tree* can be seen depicted as the large-font boxes and the connecting lines in Figure 3; as a first approximation, the control structure tree can be thought of as a method’s abstract syntax tree with its expressions removed. The use of control structure trees restricts safeTSA methods’ control flow graphs to well defined subset of reducible control flow graphs. This simplifies the machine-specific code generation and optimization as well as the dominator tree derivation.

Static single assignment form is a state-of-the-art intermediate representation for optimizing compilers. SafeTSA is based on SSA form, leveraging its benefits during the JIT compilation but shifting offline the costs of producing SSA form. The key feature of SSA form is that each ‘variable’² which may be used in the program’s SSA representation may only be defined at a single location in the representation. This can be seen depicted in Figure 3 as the unique variable on the left-hand side of each instruction.

Sometimes, however, the input of an operation should come from different source instructions depending on the control flow path through which the execution arrived at that operation (for example, the use of a loop variable in the loop condition should refer to the loop variable initializer on the first iteration, but the loop variable increment on subsequent iterations, but these are separate SSA variables), so SSA provides special ϕ instructions at CFG join nodes whose result variable represents alternative input variables depending on which edge of the CFG node the execution followed to reach the join node. As can be seen in Figure 3, safeTSA has a separate “phiinput” at each location where control can be transferred to the join node; each ϕ input has as many operands as there are ϕ s in the join node.

An important property of a correct SSA program representation is that, for all instructions A and B, if A uses the result variable of B, A must be dominated by B³. This property, which we call *referential integrity*, is the necessary and sufficient condition that all variables in an SSA program are defined before they are used on every possible path through the CFG. The serialized safeTSA encoding enforces this property statically by referencing the input variables of an operation using a relative addressing scheme, described in [3], which only assigns addresses to variables defined by operations which dominate this using operation. (Because this relative addressing changes for every instruction, it is not depicted in Figure 3.)

²A program’s SSA representation will usually require several SSA ‘variables’ for each original program ‘variable’

³For ϕ instructions the use of the input variable is considered to occur at the end of each of the blocks which precede the join node in the CFG. The output variable defined by the ϕ instruction(s), however, is considered to occur at the top of its join node. In safeTSA, this is implemented through the existence ϕ -inputs that are separated from the ϕ at the join node.

SafeTSA simplifies type checking through *type separation* and explicit cast operations. Object oriented source languages will normally allow a subtype to be used anywhere the parent type is used. In contrast, safeTSA maintains a separate name space for variables of each type: every operation that defines a result variable defines that variable to be of a particular type, and every operation that uses a variable can only refer to variables of the correct input type. Type separation is depicted in Figure 3 in the naming convention of the SSA output variables, and in the “:type” notation on the input variables; type separation requires that input and output variable types match each instruction’s type signature exactly. If a subtype must be used as a type an explicit cast is placed in the program representation through an instruction that takes the subtype variable as input and produces the type as its output variable. If, when the JIT compiler processes the method, this cast can be verified to always be correct (by consulting the virtual machine’s class hierarchy), then it will produce no executable code, but if the correctness cannot be guaranteed (e.g. a cast from a type to a subtype), a dynamic check will be necessary. This combination of SSA form and syntactic type separation is trivial to verify, but it, along with referential integrity, replaces the complex stack-based type-inference required by Java bytecode verification.

The type system of safeTSA is, at its core, the same as that of Java and Java bytecode. It allows the same types of objects in the garbage collected virtual machine’s heap, and the SSA variable types may be any of the Java primitive types (int, float, etc.) or a reference type restricted to instances of a particular class type, a particular interface type, or a particular array type, according to the same rules governing Java reference types. But the safeTSA *type system* has also been *extended to support optimization*. In particular, for each Java reference type, safeTSA adds a ‘safe’ reference type, which can only be produced by a null-check operation. All operations which act on the heap object require the null-checked ‘safe’ reference type as input. As a consequence the null-check can be safely separated from the access using the null-checked reference, allowing some classes of redundant null-checks to be optimized away when the safeTSA representation is produced. An example of this can be seen in Figure 3, where the first getField requires a nullcheck to convert the argument from type ‘A’ to ‘sA’, but the second is able to use the ‘sA0’ variable, which is already known to be safe. Similarly, separate types are added to represent the results of array bounds-checks.

While the SSA representation’s type safety is preserved through type separation, the *instruction set* of safeTSA was carefully *chosen to maintain the type/memory safety of the heap space*. The safeTSA instruction set can be divided into two main classes. The first class contains those operations which are functional (i.e. take zero or more inputs

and produce an output based solely on those inputs); this includes primitive computation (e.g. integer add), casts, and checks; the effects of these instructions are entirely captured the SSA model. The second class of instructions interacts with the virtual machine (in particular, the garbage collected heap space and the class loader). This class includes both field and array manipulation, as well as method and constructor invocation. The `getField` in Figure 3 is an example of this second class. These operations closely follow the semantics of their JVMML counterparts and enforce the same type/memory safety invariants.

SafeTSA combines control structure trees, instructions in SSA form, dominator-based referential integrity, type safety through type separation, a type system extended to support key optimizations, and a carefully chosen instruction set to provide an intermediate representation that is both easy to verify and from which it is easy to generate code.

Overview of Jalapeno

The Jalapeno system is a Java virtual machine developed by IBM Research[1]. Jalapeno, possesses several unique features, two of which are particularly relevant to the work presented in this paper: it is compile-only (i.e. it has no interpreter), and it is written almost entirely in Java.

Instead of having an interpreter, Jalapeno features three different JVMML to native code compilers. The first is the ‘Baseline’ compiler, which exists for debugging and verification purposes; it produces native code that directly implements JVMML’s stack model as closely as possible and is in many ways comparable to an interpreter. The second is the ‘Quick’ compiler, which exists to produce normal register-oriented native code without taking extra time to perform expensive optimizations. The third, the ‘Optimizing’ compiler[5], consists of multiple phases and can be operated at various levels of optimization.

The phases of the Jalapeno optimizing compiler communicate through a series of intermediate representations: a high-level intermediate representation (HIR), a low-level intermediate representation (LIR), and a machine-specific intermediate representation (MIR), as can be seen in Figure 1. A JVMML method is initially translated into HIR, which can be thought of as a register-oriented transliteration of the stack-oriented JVMML. The LIR differs from the HIR in that certain JVMML instructions are replaced with Jalapeno-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. 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 Jalapeno is written in Java, the key data structures are accessible to the VM as Java arrays. The most important of these is the Jalapeno table of contents (JTOC). The JTOC is an array containing (or containing references to) all globally-accessible entities, i.e. all the constants, a type information block (TIB) for each type, 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 fields or methods are not yet loaded, it includes code to resolve the field or method just before using the field or method the first time⁴.

Integrating SafeTSA

By integrating safeTSA class loading and a safeTSA compiler into the Jalapeno system, we have built a VM that can execute both safeTSA- and JVMML-compiled Java programs. Thus, functionally, it doesn’t matter whether the whole program has been compiled to safeTSA, or if it exists as JVMML class files, or if the program is provided as a heterogeneous mix of both the safeTSA and JVMML classes.

With respect to dynamic class loading, the modified Jalapeno system treats safeTSA classes in a manner analogous to traditional JVMML 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 setup the necessary JTOC and TIB entries; method invocations on classes loaded from safeTSA files, result in the safeTSA compiler being invoked to produce the executable code. If no safeTSA file exists, the class loader simply loads the appropriate Java class file, and any method invocation on the JVMML class will result in the standard JVMML Optimizing compiler being invoked to compile the method.

Figure 1 shows the internal structure of our safeTSA compiler when compiling a single method. In a first step the compiler transforms the method into its high level SafeTSA representation (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

⁴In the Optimizing compiler’s IRs, instead referring directly to the symbolic name of the field or method, the resolution code contains an offset to the method call in the JVMML code. This is one example of where there is an assumption of a JVMML backing which must be worked around to support a different input language for Jalapeno.

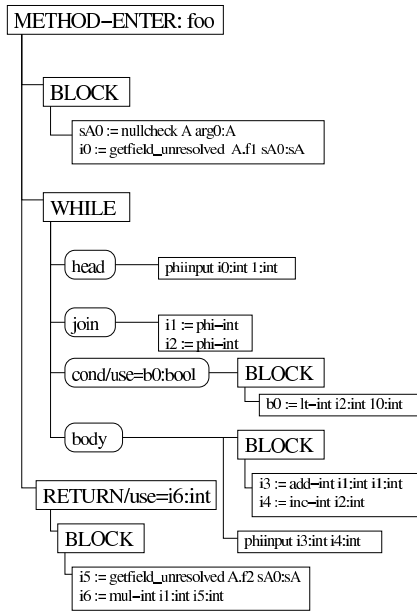


Figure 4. High Level SafeTSA representation of method foo().

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 the method for Jalapeno’s object layout and parameter passing mechanisms. After this transformation the LST method is optimized and transformed into the same LIR used by Jalapeno’s JVML optimizing compiler. The JVML compiler’s LIR to MIR phase is used 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. In this case the JIT compiler will be unable to resolve the access at compile time. Instead of directly accessing the data structure, the JIT compiler must insert special “resolve” instructions that cause the implementation to be loaded and the appropriate VM data structures 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: a `getfield_unresolved` `setfield_unresolved` or `call_unresolved` will be substituted for each `getfield/setfield` or `call` instruction of the safeTSA representation that operates on a class which is not yet loaded. Figure 4 shows what the HST for the method `foo` would look like if class `A` has 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 Jalapeno spe-

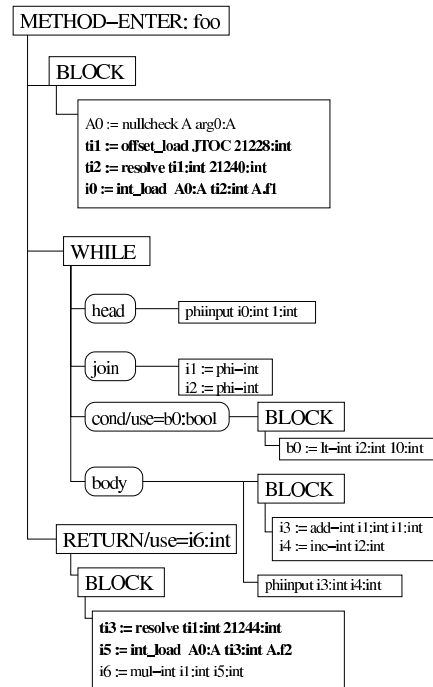


Figure 5. Low Level SafeTSA representation of method foo().

cific implementations. Mostly, this consists of performing address computations using offsets from Jalapeno’s JTOC and TIBs. It also involves translating safeTSA check and cast operations to their Jalapeno equivalents, materializing constant operands, and translating high-level storage accesses into low-level load and store instructions.

Jalapeno facilitates dynamic class loading by maintaining two offset tables, `OffsetTableField` 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.

Figure 5 shows the optimized LST that would be created from the program in Figure 2. In the LST safe types have been converted back into normal Jalapeno types and the high-level `getfield_unresolved` instructions have been lowered to sequences of instructions that perform resolution and accesses the field: An `offset_load` instruction delivers the address of the offset table `OffsetTableField`. The `resolve` instruction checks attempts to load the offset based on the field’s field dictionary index (in the example program `a.f1` has the field dictionary entry 21240 and `a.f2` the entry 21244). A final low-level `int.load` instruction is used to ac-

```

1 LABEL0
2 prologue l0pi(A,x,d) =
3 bbend BB0 (ENTRY)
4 LABEL1
5 null_checkt2v(Guard) = l0pi(A,x,d)
6 int_load t3i(I) = JTOC(int), 21228
7 bbend BB1
8 LABEL3
9 int_load t4i(int) = t3i(I), 21240
10 int_ifcmp t4i(int), 0, !=, LABEL4
11 bbend BB3
12 LABEL4
13 resolve A.f1
14 goto LABEL3
15 bbend BB4
16 LABEL2
17 int_load t6i(int) = l0pi(A,x,d),t4i(int), A.f1, t2v(Guard)
18 int_move t7i(int) = 1
19 goto LABEL6
20 bbend BB2
21 LABEL7
22 int_add t6i(int) = t6i(int), t6i(int)
23 int_add t7i(int) = t7i(int), 1
24 bbend BB7
25 LABEL6
26 int_ifcmp t10v(GUARD) = t7i(int), 10, <, LABEL7
27 bbend BB6
28 LABEL8
29 int_load t4i(int) = t3i(I), 21244
30 int_ifcmp t4i(int), 0, !=, LABEL10
31 bbend BB8
32 LABEL9
33 resolve A.f2
34 goto LABEL8
35 bbend BB9
36 LABEL10
37 int_load t11i(int) = l0pi(A,x,d), t4i(int), A.f1, t2v(Guard)
38 int_mul t12i(int) = t6i(int), t11i(int)
39 return t12i(int)
40 bbend BB10

```

Figure 6. LIR representation of method foo().

tually load the field once the offset 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, straightforward translation of LST instructions into LIR instructions, a translation from SSA variables into LIR’s virtual registers. Figure 6 shows our example program translated into LIR, which consists of 9 basic blocks. Instructions 6-15 resolve ‘`a.f1`’; instructions 28-35 resolve ‘`a.f2`’; instructions 19-26 represent the while-loop. This LIR is very similar to what the Jalapeno optimizing compiler would produce for the same program in JVMML. But the `safeTSA` compiler only performed common subexpression elimination, while the Jalapeno JVMML optimizing compiler

had to perform 8 optimization passes, taking significantly more time, to produce LIR of similar quality.

During the translation of SSA values to virtual registers, each of the ϕ instructions will normally be translated into a move at each of the ϕ instructions’ ϕ -inputs. This move transfers the contents of the appropriate ϕ -input operand into a result register correspond to the ϕ ⁵. Many of these moves, however, can be removed by the LIR to MIR phase, when the actual register allocation occurs.

Results

`SafeTSA` provides a mechanism for the safe transport of optimized code. To assess whether `safeTSA` delivers the desired benefits, we ran a series of benchmarks through the Jalapeno system in which we compare the runtime performance and required compilation time of programs compiled to JVMML and `safeTSA`. As our current system can only handle Java source programs we chose to utilize programs from the Java Grande Benchmarks (JGF) [14]. The programs reported here come from section1 of the benchmark suite⁶. In the following discussion, where we refer to JVMML we denote Java class files produced using version 1.2.2 of Sun `javac` using options to generate no debug information (`javac -g:none`). `SafeTSA` files have been produced with constant propagation, common subexpression elimination and local dead code elimination.

All compilation times in this section are given in milliseconds (ms), and the performance of the resulting code is given in numbers of operations/sec as outputted by the benchmark suite. All results were obtained on a PowerMac with a 733 MHz PowerPC G4 processor running Mandrake Linux 7.1. The system has 1.5 GB of main memory and a 256KB L2 Cache.

The only optimization phase performed during the translation from `safeTSA` to LIR, is common subexpression elimination in the LST representation. In contrast, JVMML optimizing compiler performed several optimizations, which were comparable to those performed ahead of time by the Java to `safeTSA` compiler. The following optimizations are performed by the Jalapeno optimizing compiler when transforming a JVMML representation into its corresponding LIR: branch optimizations, limited constant propagation, local common subexpression elimination, unreachable code elimination, and local redundant null and bound checks. Because the compiler does not yet support

⁵Occasionally if the same virtual register would be used as both input and output in the ϕ instructions of the join node, an extra move is needed to maintain the SSA semantics that all of the ϕ instructions at a join node have the effect of being executed simultaneously.

⁶A few of the benchmark programs required slight modifications to work correctly with the Jalapeno system; the most significant modification was the removal of the explicit invocations of the Java garbage collector between benchmark runs in `JGFCreateBench.java`.

Benchmark	JVML	SafeTSA	<i>Improvement</i>
Arith:Add:Int	7.22359E08	7.27773E08	+0.75%
Arith:Add:Long	2.31208E08	1.94045E08	-16.10%
Arith:Add:Float	1.44512E08	1.45636E08	+0.78%
Arith:Add:Double	1.44416E08	1.45603E08	+0.78%
Arith:Mult:Int	3.61379	3.10638	-13.8%
Arith:Mult:Long	1.44607E08	1.45668E08	+0.73%
Arith:Mult:Float	1.44767E08	1.45668E08	+0.61%
Arith:Mult:Double	1.44671E08	1.45668E08	+0.68%
Arith:Div:Int	3.08144E07	3.10656E07	+0.8%
Arith:Div:Long	5.22582E06	5.3057E06	+1.5%
Arith:Div:Float	1.44416E08	1.457E08	+0.9%
Arith:Div:Double	1.44416E08	1.457E08	+0.9%
Assign:Same:Scalar:Local	1.15507E09	2.91271E09	+250.16%
Assign:Same:Scalar:Instance	1.36108E00	1.30941E08	-4.58%
Assign:Same:Scalar:Class	1.37623E08	1.29441E08	-5.1%
Assign:Same:Array:Local	2.31535E08	242771E08	+4.85%
Assign:Same:Array:Instance	3.63093E07	3.53295E07	-2.87%
Assign:Same:Array:Class	1.0733E08	1.07931E08	+0.5%
Assign:Other:Scalar:Instance	4.81741E07	1.20073E08	+149.24%
Assign:Other:Scalar:Class	4.82166E07	1.16467E08	+140.54%
Assign:Other:Array:Instance	3.57807E07	1.14214E08	+219.21%
Assign:Other:Array:Class	3.57807E07	5.94593E07	+66.17%
Cast:IntFloat	3.69342E07	3.73042E07	+1.00%
Cast:IntDouble	3.68843E07	3.7202E07	+0.86%
Cast:LongFloat	2.32622E06	2.35294E06	+1.14%
Cast:LongDouble	2.32833E06	2.35078E06	+0.96%
Loop:For	1.55667E08	2.42726E08	+55.92%
Loop:ReverseFor	1.58223E08	3.64089E08	+130.11%
Loop:While	1.80839E08	1.82839E08	+1.1%
Math:AbsInt	2.40376E07	2.65028E07	+10.2%
Math:AbsLong	2.4023E07	2.60023	+8.3%
Math:AbsFloat	2.05829E07	2.31609E07	+12.52%
Math:AbsDouble	1.94584E07	2.24438E07	+15.34%
Math:MaxInt	1.339E07	1.32858E07	-0.8%
Math:MaxLong	2.17814E07	2.37725E07	+9.14%
Math:MaxFloat	6.38404E06	6.41002E06	+0.4%
Math:MaxDouble	3.47001E06	3.53043E06	+1.7%
Math:MinInt	1.33812E07	1.32815E07	-0.8%
Math:MinLong	2.65371E07	2.51707E07	-5.29%
Math:MinLong	2.18162E07	2.65371E07	+21.63%
Math:MinDouble	6.43418E06	6.47077E06	+5.6%

Figure 7. Generated code performance (in operations/sec)

inlining between JVMML and safeTSA methods we globally disabled the inlining of method calls. This limitation is partially ameliorated, because most of our benchmarks do not require inlining in order to perform effective optimization,

After the transforming of a safeTSA file into the LIR representation, both compilers perform the same tasks, i.e. a dependence analysis, and afterward live range analysis and a linear scan register allocation, before the machine code is finally produced.

These benchmarks support the hypothesis that a program in safeTSA is better suited for the generation of optimized target code than a stack-based intermediate representation. During the execution of 41 benchmarks the safeTSA based version outperforms the JVMML version in 33 cases. Figure

7 shows the runtime behavior in the number of operations per second for the benchmark programs that has been compiled with Jalapeno’s JIT compared to the runtime that can be achieve when compiling the programs with our safeTSA compiler. On average the SafeTSA versions of the benchmarks could execute about 33% more operations than the JVMML compiled version, but varies from between -16.10% and 250.16%. This variation is not surprising, when one realizes that these are micro-benchmarks, executing set sequences of operations many times. In some of these benchmark programs, the operations are very simple, e.g. an addition or multiplication, so there is not much possibility for optimization or variation of the resulting code, particularly since both compilers used the same low-level instruction se-

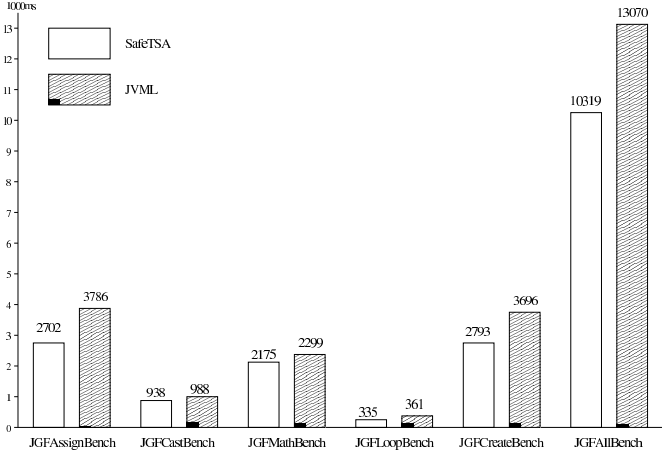


Figure 8. Overall compilation time (in ms)

lector. In contrast, some of the other benchmark programs contain more complicated operations there is a greater variation between the compiler chains. The target code generated from safeTSA was nearly optimal for the benchmarks containing assignments of array elements, assignments to object fields, and assignments of scalar variables. Some of the improvements for the safeTSA compiler chain, seem to be a result of a more efficient identification and elimination of common subexpressions, which may be due to the consistent use of SSA throughout the safeTSA to LIR compiler chain, in contrast to the JVMIL to LIR compiler chain, which switches back and forth between SSA and virtual register-based representations, possibly disguising common subexpressions.

The deterioration of the runtime behavior in some benchmark programs, notably for the program *Arith.Mult.long*, is due to an interaction of the ϕ resolution algorithm and Jalapeno’s linear scan-register allocator, resulting in the safeTSA compiler producing more *long move instructions* than the JVMIL compiler.

Even though the target code produced from safeTSA is often significantly better than its JVMIL counterpart, the measured compilation times for both compilers indicate that a safeTSA compilation needs the same or less time than the JVMIL-based compilation process. Figure 8 contains the compilation times for safeTSA and JVMIL-based compilation for all benchmark programs. On average the SafeTSA compilation times are nearly 19% below those of JVMIL, but the improvement varies from 5.06% and 28.63%. This fluctuation of these measurements can also be explained by the simple structure of most of the JGF benchmark programs.

Although these performance gains are already very encouraging, certain deficiencies in the current implementation may be obscuring even greater gains. The safeTSA compilation times are inflated by the use of the standard

Jalapeno Optimizing compiler phase to translate our LST representation into the corresponding machine code. For this, both an additional compilation pass that transforms the LST representation into its corresponding LIR and a later data dependence analysis for register allocation has to be performed. Neither of these would be necessary if the LST representation would be immediately translate into machine code. This may result in substantial gains, because the code generation of machine code from LIR took nearly 50% of the overall compilation time.

In Figure 9 we depict the amount of time spent by the respective compilers to translate from safeTSA or JVMIL to LIR. The time required for the safeTSA compiler was, on average, 30.4% less (varying from 8.75% to 60.6% less) than what was required for the JVMIL compiler.

Related Work

The Architecture Neutral Distribution Format (ANDF[4]), originally developed by the Defence Research Agency in the UK (DRA) uses a tree based intermediate representation, the TDF intermediate language. TDF is a tree structured language, defined as a multi-sorted abstract algebra, which preserves more program structure information than other languages. However, it has a weaker type system than typical high level languages. It was originally designed for the compilation of sequential languages such as C and Lisp. Indeed, the overall level of the language is similar to C but also includes support for other features such as for garbage collection. Programs represented in ANDF are compiled to native code at installation time. As such, ANDF was designed solely as a distribution format rather than a mobile code format.

Slim Binaries[8, 12], an intermediate representation developed by Franz and Kistler are also based on a tree structured language. In contrast to ANDF, Slim Binaries were designed with mobile code applications in mind and to be suitable for dynamic code generation on target

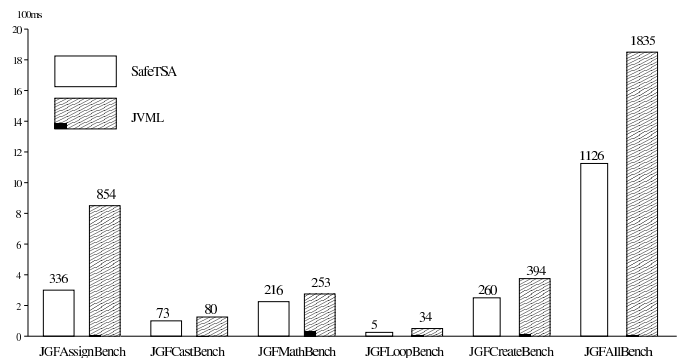


Figure 9. Optimization time (in ms)

machines. Slim Binaries have been shown to be much smaller than ANDF and Java bytecode files—an important feature for mobile code applications. Slim Binaries have been used successfully in studying continuous program optimization[11]. Unlike safeTSA, Slim Binaries are based on encoding the source-level abstract syntax tree of a program. As a consequence of being so far removed from the internal representations typically used in compilers, Slim Binaries cannot communicate the results of program analyses and target machine independent optimizations as well as safeTSA can.

Because of the simplicity of code generation, 0-address architectures have been a popular intermediate target architecture in compiler design. A well known example of such a stack-based intermediate language is pCode[15], used for code generation in the earlier Pascal compilers. pCode has recently received renewed attention because of its influence on JVMML.

JVML is either interpreted by a virtual machine (resulting in unacceptable performance) or compiled by a just-in-time compiler at the target site. Unfortunately, time spent in just-in-time compilation translates directly into end-user latency, so minimizing this time is essential. JVMML’s stack-based model (making certain optimizations more difficult[10]), and its approach to memory safety (i.e. making nullchecks and boundschecks indivisible parts of object and array operations) hinders offline optimization, from which safeTSA receives a benefit. It must also be noted, that while Jalapeno does not perform JVMML verification, doing so would require a relatively expensive dataflow analysis which is completely unnecessary to verify safeTSA.

CIL, the intermediate representation for Microsoft’s .NET platform is a further improvement of the stack-based virtual machine. It is also designed to support being targeted from many different source languages. To do this efficiently, it has many escape hatches for “unmanaged” trusted code to trade safety for performance. It also has a mechanism for including SSA information along side the stack-based representation. We have not seen any provision or verifying the SSA information.

λ JVM[13] is, perhaps, the closest IR to safeTSA. λ JVM is an intermediate representation based on λ -calculus and has properties similar to SSA form. It is designed as an intermediate step bridging the semantic gap between JVMML (and Java) and FLINT[17]. Unlike safeTSA, it has no defined serialization or file format, but only exists inside the FLINT-based compiler. By translating programs from JVMML to λ JVM and from λ JVM to FLINT Java classes can exist in the same type safe environment. Because Java and the languages typically used with FLINT, such as ML, have significantly different semantics, modules compiled from λ JVM and other FLINT modules will not communi-

cate transparently. Thus the result of this co-existence is quite differs from the commingling of JVMML and safeTSA classes within our Jalapeno-based system, where the JVMML and safeTSA classes interact transparently. The work on λ JVM also differs in that it focuses on type safety rather than performance.

Besides Jalapeno, there are many other JIT compilers implementing the Java Virtual Machine. The one most closely related to our work is Sun’s HotSpot Server compiler[9], which uses an SSA-based internal representation similar to the one described in Click’s dissertation [6]. It turns Java exceptions into explicit control flow in its IR and uses a full flow pass to discover types from the Java bytecode. In contrast our representation explicitly marks all types.

There have been several traditional static compilers utilizing intermediate representations based on SSA. Each of them requires either Java source code or Java bytecode as input and produces native machine code as output. However, there is no compiler that conserves the information of the SSA based intermediate representation in any kind of class file that could support a dynamic code generation process. Below we will discuss in more detail two optimizing compilers of particular interest.

The Swift compiler[16] has been designed and implemented at the Western Research Laboratory of Compaq Computer Corporation. Swift translates Java bytecode to optimized machine code for the Alpha architecture and uses SSA form for its intermediate representation. The intermediate language used by the compiler is relatively simple, but allows for straightforward implementation of all standard scalar optimizations and other advanced optimization techniques, i.e. method resolution and inlining, interprocedural alias analysis, elimination of run time checks, object inlining, stack allocation of objects, and synchronization removal. Each value in the SSA graph also has a program type. Comparable with our approach, the type system of Swift can represent all of the types present in Java program. In contrast to our approach the Swift intermediate representation contains no structural information, i.e. the control structure tree of our SafeTSA graphs. Also, compared to the instruction set of our SafeTSA, the instructions used by Swift are very specialized and adapted to its target architecture.

Marmot[7] is a research compiler from Microsoft that transforms Java bytecode into native machine code. The organization of the compiler can be divided into three parts: conversion of Java class files to a typed high level intermediate representation based on SSA form, high level optimization, and code generation. Type information in the high level representation of the Marmot compiler (there is also a low-level IR) is derived by type elaboration. This process produces a strongly-typed intermediate representa-

tion in which all variables are typed, all coercion and conversions are explicit, and all overloading of operators is resolved. Marmot doesn't support Java's essential facility of *dynamic class loading*.

Summary and Future Work

This work, describes the integration of safeTSA support into Jalapeno, an existing dynamic optimization system, creating a complete runtime environment transparently supporting safeTSA and Java class files. We have used this system, along with the Java Grande Benchmarks, to attempt to assess the relative merits of safeTSA and JVMML as JIT compiler input. The results from these benchmark runs show that the safeTSA compiler was able to produce, on average, 33% faster code in 19% less time.

While the results so far are very impressive, they are still preliminary. Current plans for improving the compiler, include writing an SSA aware register allocation and final machine code generator, which will take advantage of LST being in SSA form. [18] discusses the possibility of incorporating more offline analysis and optimizations into the safeTSA files improving the code quality and/or reducing the compilation time of the generated code, while still retaining safety. It should also be pointed out, that because Jalapeno does not verify class files, the compilation times reported here do not include verification time, which would result in additional speedup if safeTSA were to be used in a production JVM.

Unfortunately, the range of benchmarks reported here is somewhat limited. This is due to both a lack of robustness in the compiler and the requirement that the benchmarks be available in Java source. The lack of compiler robustness is being remedied, and the requirement of benchmarks available in Java source will be alleviated after the completion of a JVMML to safeTSA translator, which will allow us to utilize benchmarks for which Java source is unavailable.

While much of safeTSA's potential still remains unexplored, what has been explored, shows that safeTSA can simultaneously provide faster compilation time, better performing generated code, and simpler verification, while at the same time maintaining interoperability with the existing JVMML code base.

Acknowledgement

We would like to thank Fermín Reig, Andreas Hartmann, and the various anonymous reviewers, for their comments on earlier versions of this paper. We would also like to thank Ning Wang for his work on a Java class file to safeTSA translator and his feedback on earlier versions of this paper. In addition, we would like to thank Michael Hind and the rest of the Jalapeno JVM team at IBM Research for providing us with the Jalapeno Java Virtual Machine.

Parts of this effort are sponsored by the National Science Foundation (NSF) under grant CCR-9901689, by the Deutsche Forschungsgemeinschaft (DFG) under grant AM-150/1-1, and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536.

References

- [1] B. Alpern, C. R. Attanasio, J. J. Barton, et al. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.
- [2] W. Amme, N. Dalton, M. Franz, and J. von Ronne. 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-3*, Monterey, CA, Dec. 2000.
- [3] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.
- [4] Architecture Neutral Distribution Format (XANDF) Specification. Open Group Specification P527, January 1996.
- [5] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Ser-rano, V. C. Sreedhar, and H. Srinivasan. The jalapeño dynamic optimizing compiler for java. In *ACM Java Grande Conference*, San Francisco, June 1999.
- [6] C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, Texas, 1995.
- [7] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, Mar. 2000.
- [8] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.
- [9] Sun Hotspot compiler for Java. <http://java.sun.com/products/hotspot/>.
- [10] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, Nov. 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.
- [11] T. Kistler. Continuous Program Optimization. *Phd Dissertation, University of California, Irvine*, 1999.
- [12] T. Kistler and M. Franz. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb. 1999.
- [13] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.
- [14] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and Development of Java Grande Benchmarks. In *Proc. of the ACM 1999 Java Grande Conference, San Francisco.*, June 1999. Also available as DHPC Technical Report DHPC-063.
- [15] K. V. Nori, U. Ammann, et al. Pascal-P implementation notes. In D. W. Barron, editor, *Pascal – The Language and its Implementation*, pages 125–170. John Wiley and Sons, Ltd., 1981.
- [16] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean. The Swift Java Compiler: Design and Implementation. WRL Research Report 2000/2, Compaq Research, April 2000.
- [17] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
- [18] J. von Ronne, A. Hartmann, W. Amme, and M. Franz. Efficient online optimization by utilizing offline analysis and the safetsa representation. In *2nd Workshop on Intermediate Representation Engineering for Virtual Machines*, Dublin, Ireland, June 2002.