

Efficient Online Optimization by Utilizing Offline Analysis and the SafeTSA Representation

Jeffery von Ronne†
jronne@ics.uci.edu

Andreas Hartmann‡
krs@informatik.uni-jena.de

Wolfram Amme‡
amme@informatik.uni-jena.de

Michael Franz†
franz@uci.edu

†Information and Computer Science
University of California, Irvine
United States

‡Institut für Informatik
Friedrich-Schiller-Universität Jena
Germany

Abstract

Conventional mobile-code representations, e.g. Java bytecode, provide machine-independence and type-safety, but do so at the expense of performance. This performance hit can be taken in the form of *decreased throughput* or in *increased latency*. SafeTSA was designed to reduce this performance hit, especially when producing high-quality optimized machine code. It does this by utilizing SSA form and thus shifting dataflow analysis effort from the online JIT compiler to the offline producer of the safeTSA program.

This paper describes the extension of safeTSA to support a greater shifting of optimization effort away from the online JIT compiler, increase the performance, without sacrificing type-safety or machine-independence. Specifically, we describe mechanisms to reduce the cost of online register allocation through offline analysis, and to use offline escape analysis so the online JIT compiler can produce less heap allocations.

1 The SafeTSA Representation

SafeTSA[ADvRF01] is a verifiable type safe intermediate representation designed as a machine independent intermediate 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 1(b) contains a graphical depiction of the safeTSA representation produced from the source program in Figure 1(a). 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 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 1(b); 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. By basing safeTSA on SSA form, we leverage the benefits of SSA during the JIT compilation but shift offline the costs of producing SSA form. The key feature of SSA form is that each 'variable' 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 1(b) as the unique variable on the left-hand side of each instruction.

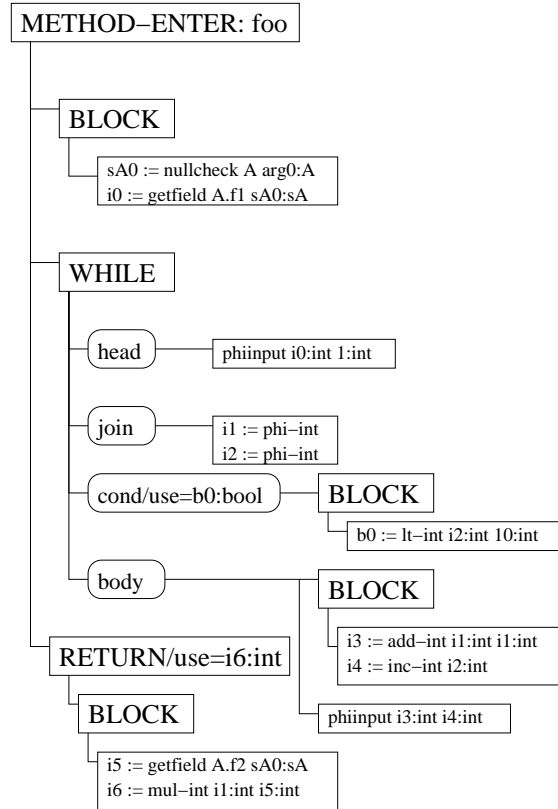
Sometimes, however, the input of an operation should come from different source instructions depending on the control flow path through which the execution arrived at that operation

```

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

```

(a) source code



(b) translated to safeTSA

Figure 1: An Example program

(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 1(b), safeTSA makes a separate “phi-input” 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, as an input, the result variable of B, A must be dominated by B¹. 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[ADvRF01] enforces this property statically by referencing the input variables of an operation using a relative addressing scheme 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 1(b).)

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

¹For 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 phi at the join node.

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 1(b) 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 as 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 1(b), where the first getfield requires a nullcheck to convert the argument from type ‘A’ to ‘sA’, but the second is able to use the ‘sAO’ 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 second class includes both field and array manipulation, as well as method and constructor invocation. The getfield in Figure 1(b) 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.

2 Register Allocation

Because safeTSA is a machine-independent intermediate representation, register allocation must be performed online by the JIT compiler. This does not mean, however, that the offline producer cannot provide information to the online register allocator. This paper proposes two such mechanisms. The first is a special kill function which marks the live range end(s) of each SSA variable. The second is a spill ranking for each value.

Dynamically, the semantics of the *kill function* can be thought of as deleting a particular

SSA variable. Statically, it makes the variable inaccessible from any potential-use point in the program for which there exists a path through the program's CFG which starts at the SSA variable definition, goes through the kill function, and reaches the potential-use point without passing through the definition again.

These functions can then be used to mark the end of live ranges. In linear non-branching code, they should be placed immediately after the last use of each variable. In the presence of join nodes, however, in order to ensure that dead variables are not used, we modify our SSA representation and require that all live SSA variables be explicitly mentioned at every join node, so that it can be verified that these live variables have not been killed on any of the branches into the join node. This can be added to the existing safeTSA checks for referential integrity with little additional complexity.

The live range information provided by the code producer can err conservatively (in that it doesn't kill off variables as soon as they could be), but this results in wasting registers rather than using one twice, and thus poses no risk to the virtual machine's type and memory invariants.

Thus, utilizing these kill functions (and assuming no spills are necessary), the register allocator can simply walk through the program track the live variables at each point, adding at definitions, and deleting at kills, assigning registers and producing, in linear time a good register allocation, similar to that found in [PS99]. Alternatively, the live range and conflict graph can be used as input for a more time consuming graph coloring allocation.

The other proposed mechanism to enhance register allocation is with what we call *spill ranking*, that is ranking at each variable according to an estimate of how much its spilling will hurt program performance. The offline code producer can perform expensive analyses to try to determine accurate estimates. The only thing that is required of the JIT compiler, however, is to utilize its knowledge of the number of available registers and its knowledge of how many variables are live at each point, and spill as many variables as necessary. As suggested in [KC01], this may be useful for a more efficient prioritized graph coloring. Spill ranking may be combined with a copy function, to allow the offline safeTSA producer to split live ranges and assign different spill ranking to each live range segment.

3 Escape Analysis

Escape analysis is supported by extending the safeTSA type system with additional 'bound' reference types, which are treated similarly to the null-checked 'safe' types. There are two varieties of bound reference types: object-bound and call-stack-bound, each of which exists in null-checked and non-null-checked versions. Offline escape analysis is used to specialize regular reference types into the bound reference types. The safeTSA type system is then used to restrict the uses of these bound reference types. Call-stack-bound references are the most restricted; they can be used within a method or passed as arguments of the appropriate call-stack-bound reference type, but they may never be written to any field or returned from a method. Object-bound references may be cast to call-stack-bound reference types, stored to object-bound fields of the 'this' object, and passed as arguments of the appropriate object-bound reference type to other methods of the same class. These extra types are coupled with special allocation instructions: a `stackalloc`, an `objectalloc`. The `stackalloc` instruction results in a variable of the null-checked call-stack-bound reference type, and the `objectalloc` results in a variable of the null-checked object-bound reference type. Thus, the type system enforces the invariants that `stackalloc`'ed objects do not live longer than the creating method's call-frame and that the `objectalloc`'ed objects do not outlast the 'this' object.

a)

```
public class A {
    int y;
    public void foo(){
        SObject O = new SObject();
        O.do1();
        O.x = 15;
        O.do2();
        subdo(O);
    }
    private void subdo(SObject O){
        y = O.x;
    }
}
```

b)

```
method foo
3: xcall eob-SObject #Class init
4: xupcast eob-SObject #eob-SObject #(3)
5: xdispatch #eob-SObject #(4) do1
7: setfield #eob-SObject #(4) x 15
9: xdispatch #eob-SObject #(4) do2
10: xdispatch #A #this subdo #(4)

method subdo
11: xupcast eob-SObject #eob-SObject p
12: getfield #eob-SObject #(11) x
13: setfield #A #this y (12)
```

Figure 2: Example program (a) and its annotation with escape information (b)

Consider the example given in Figure 2(a). Offline escape analysis can be used to determine that ‘SObject O’ has an object-bound reference type. Further analysis shows that ‘obj’ is bound for the method ‘subdo(SObject O)’, as the variable x is only read by the method. Method ‘subdo()’ then can be marked as bound for its argument. Using this information, the JIT compiler may check if ‘subdo()’ is called with an argument, that is object- or call-stack-bound (as it is in figure 2(b)) and object allocate ‘O’ accordingly. Figure 2(b) partially depicts the resulting SafeTSA representation of class A, where #- marks ‘SObject O’ to be safe and -eob- to be object-bound.

4 Related Work

Recently, various annotations have been proposed for enhancing the performance of JIT compilers using Java bytecode and other intermediate representations [ANH00, KC01, BFHS02, Rei01]. The annotations of [KC01] and [BFHS02] will not compromise type safety. Both [ANH00] and [KC01] suggest utilizing annotations for register allocations. Their proposed mechanisms are not dissimilar to our spill rankings. They do not, however, convey information about live ranges as our kill functions do. [BFHS02] suggests the use of annotations to convey the results of escape analysis in order to reduce heap allocation and its ‘captured’ variables and is similar to our ‘call-stack-bound’ types, but it proposes nothing similar to our ‘object-bound’. Proof carrying code[Nec97] and TAL [MWCG99] are annotation techniques that are in parts

semantically equivalent to our concept, however its annotation forms are not machine independent. Unlike previous annotation work, our work leverages the pre-existing SSA form of the safeTSA representation.

5 Summary

This paper presents three extensions to the safeTSA representation, which will reduce the work which needs to be performed by an optimizing JIT compiler. To enhance the performance of register allocation, the kill function and spill rankings are provided. Kill functions explicitly mark live ranges and, indirectly, register pressure. Spill rankings prioritize the allocation of registers to SSA variables. The third mechanism, conveys a reduction in heap allocations resulting from alias analysis; it safeguards this through the use of specialized types, similar to those which safeTSA already provides for null- and bounds-checks. These mechanisms further the goal of shifting optimization effort away from online JIT compilation.

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 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

- [ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael 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.
- [ANH00] Ana Azevedo, Alexandru Nicolau, and Joseph Hummel. An annotation-aware java virtual machine implementation. *Concurrency - Practice and Experience*, 12(6):423–444, 2000.
- [BFHS02] Matthew Beers, Michael Franz, Vivek Haldar, and Christian H. Stork. Tamper proof annotations by construction. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, March 2002.
- [KC01] Chandra Krintz and Brad Calder. Using annotation to reduce dynamic optimization time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, 2001.
- [MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Prog. Lang. and Sys.*, 23(3):528–569, May 1999.
- [Nec97] George C. Necula. Proof-Carrying Code. In *POPL '97*, Paris, France, January 1997.
- [PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.
- [Rei01] Fermín Reig. Annotations for portable intermediate languages. In *First Workshop on Multi-Language Infrastructure and Interoperability (BABEL 01)*, 2001.