A Virtual Machine for Interpreting Programs in Static Single Assignment Form

Jeffery von Ronne *Ning Wang* *Alexander Apel* *Michael Franz*

Technical Report 03-19
School of Information and Computer Science
University of California, Irvine, CA 92697-3425

October 23, 2003
Revised November 4, 2003

Abstract

Optimizing compilers, including those in virtual machines, commonly utilize Static Single Assignment Form as their intermediate representation, but interpreters typically implement stack-oriented virtual machines. This paper introduces an easily interpreted variant of Static Single Assignment Form. Each instruction of this Interpretable Static Single Assignment Form, including the Phi Instruction, has self-contained operational semantics facilitating efficient interpretation. Even the array manipulation instructions possess directly-executable single-assignment semantics. In addition, this paper describes the construction of a prototype virtual machine realizing Interpretable Static Single Assignment Form and reports on its performance.

Contents

1 Introduction **1**

2 Interpretable SSA Form **2**
2.1 Unique Naming 2
2.2 Choosing Φ-function Operands 4
2.3 Simultaneous Execution of Φ-functions 5
2.4 Arrays 8
2.4.1 The Cytron et al. Array Model 8
2.4.2 ISSA's Implementation of Arrays 10
2.4.3 Optimizations 11

3 Implementation and Performance **11**

4 Future Work **14**
4.1 A Faster ISSA Virtual Machine 14
4.2 A SafeTSA Interpreter 15

5 Related Work **15**

6 Conclusion **16**

7 Acknowledgements **17**

References **17**

A Implementation of the Interpreter Core **21**
A.1 ssa_vm.c 21
A.2 ssa_vm.h 27
A.3 ssa_array.h 28
A.4 inst.h 29

B Benchmarks **31**
B.1 Factorials 31
B.1.1 factorial.ssa 31
B.1.2 factorial.c 32
B.1.3 Factorial.java 33
B.1.4 factorial.pl 34

B.2 Fibbonacci Sequence (in scalars) 35
B.2.1 fibonacci.ssa 35
B.2.2 fibonacci.c 36
B.2.3 Fibonacci.java 37
B.2.4 fibonacci.pl 38
B.3 Fibbonacci Sequence (in an array) 39
B.3.1 fibonacci_array.ssa 39
B.3.2 fib_array.c 41
B.3.3 FibArray.java 42
B.3.4 fib_array.pl 43

List of Figures

1 a simple program 2
2 execution of a simple program 2
3 a program using the if-then-else construct 3
4 execution of if-then-else construct 4
5 a program computing the Fibonacci sequence 5
6 computing the Fibonacci sequence 6
7 tree depiction of the array model 8
8 a program exhibiting array manipulation 9
9 dynamic execution of array manipulation 9
10 performance slowdown 12

List of Tables

1 execution times 11
2 relative execution time 11

1 Introduction

Over the past decade, intermediate representations based on Static Single Assignment (SSA) Form [Alpern et al., 1988, Rosen et al., 1988] have been utilized inside many research and industrial compilers. More recently, Amme et al. [2001] proposed SafeTSA, a verifiable external program representation based on SSA Form, which is well suited for just-in-time compilation. Just-in-time compilation, however, imposes a startup delay, which may not be justified for short-lived applications and infrequently executed methods. Java Virtual Machine implementations, such as Sun's HotSpot Performance Engine [Agesen and Detlefs, 2000], typically use mixed-mode interpretation and compilation to combine interpretation's shorter startup times with compiled code's better throughput. Perhaps as a consequence of several problematic features of SSA, conventional imperative interpreters have not been written for SSA representations. Consequently, Krintz [2002] recently proposed storing and transporting programs in both Java class files (which use a stack-oriented virtual machine) for interpretation and also in SafeTSA classes for compilation, allowing both compilation and interpretation at the cost of supporting two program representations.

If an SSA interpreter were available, however, it would be possible to build a virtual machine supporting both compilation and interpretation using only SSA representations, providing the same benefits as the hybrid virtual machine Krintz [2002] proposes without the overhead of supporting two input program representations. Incidentally, the same interpreter technology could be used as a debugging and testing tool for executing the SSA intermediate representations of optimizing compilers.

While an interpreter for SSA is desirable, several features of SSA Form are particularly challenging for direct imperative interpretation: the large number of variable names, the selection of ϕ-function operands, the simultaneous execution of mutually dependent ϕ-functions, and the handling of non-scalar variables (such as arrays) with single-assignment semantics.

In the next section, this paper presents Interpretable SSA (ISSA) Form, an SSA variant in which each instruction has directly-interpretable operational semantics, demonstrating how ISSA handles each of these problematic features of SSA. After this, it reports on the performance of a prototype ISSA virtual machine. Finally, the paper concludes after discussing future improvements and related work.

(a) source code	(b) stack-based code	(c) SSA Form	(d) ISSA code
x = 3	iconst_3		
y = 2	iconst_2	x_0 := mov 3	0 const 3
x = x + y	iadd	y_0 := mov 4	1 const 2
x = z * y	iconst_2	x_1 := iadd x_0 y_0	2 iadd (0) (1)
print x	imul	x_2 := imul x_1 y_0	3 imul (2) (1)
exit	print	print x_2	4 print (3)
	exit	exit	5 exit

Figure 1: a simple program

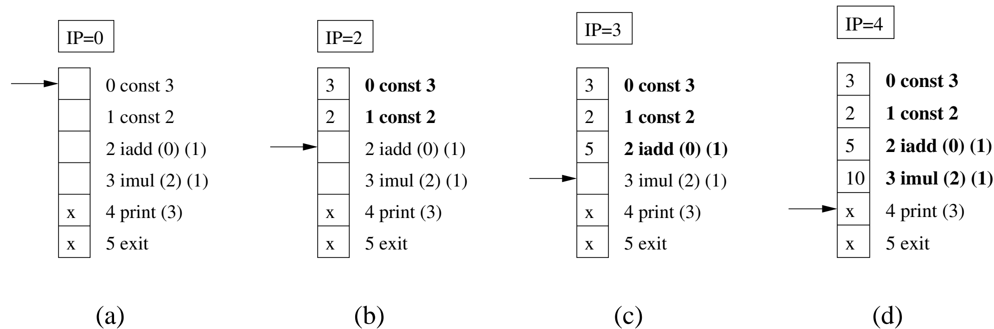


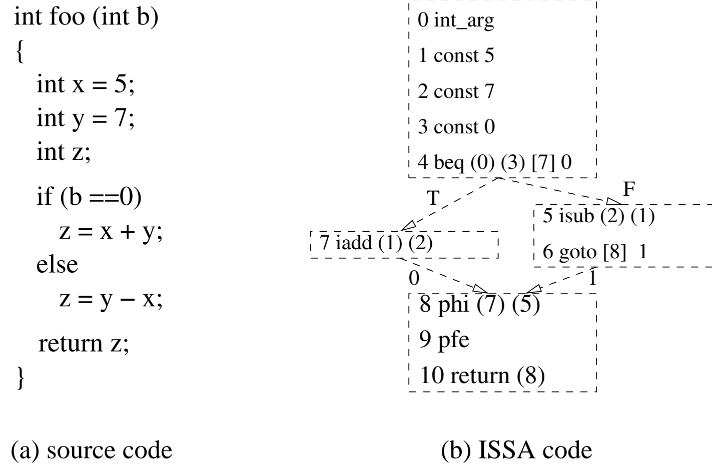
Figure 2: execution of a simple program

2 Interpretable SSA Form

2.1 Unique Naming

The most important characteristic of Static Single Assignment is that the left hand side of each and every variable assignment must have a unique name. As a result, each original program variable has several corresponding SSA variables (often distinguished with subscripts).

Since each SSA variable is defined by exactly one program instruction (the right hand side of the assignment), ISSA instantiates an abstract machine for each program containing one *result register* per instruction. Figure 1(d) shows the program in Figure 1(a) translated into ISSA. Each instruction in ISSA is labeled (on the left) with a consecutive instruction number. A few instruction types, such as



(a) source code (b) ISSA code

Figure 3: a program using the if-then-else construct

the `const` instructions, take immediate integer values, but most have indirect operands which refer to the result of previously executed instructions. Each of these operands (syntactically distinguished by parentheses) denote the result register by indicating the definition's instruction number.

Figure 2 shows the dynamic execution of this program's abstract machine. The instructions' result registers are shown by the boxes to the left of the instructions; an auxiliary *instruction pointer* (IP) register is used to indicate the instruction to be executed next. As each instruction executes, it retrieves its inputs from from the indicated registers, performs its computation, and writes to the appropriate result register (RR) on its left. For example, before instruction 3 executes, the machine state is as show in Figure 2(c); as it executes it reads the values of RR2 (i.e. 5) and RR1 (i.e. 2), multiplies them together, and writes the result (i.e. 10) to RR3.

This solution is simple to implement, and while it may seem to waste a large amount of memory. The memory used is limited to fraction of that which is required to hold the instructions and is offset by not needing to perform stack manipulation (in conventional stack-oriented virtual machines) or designate a result register (in virtual register machines).

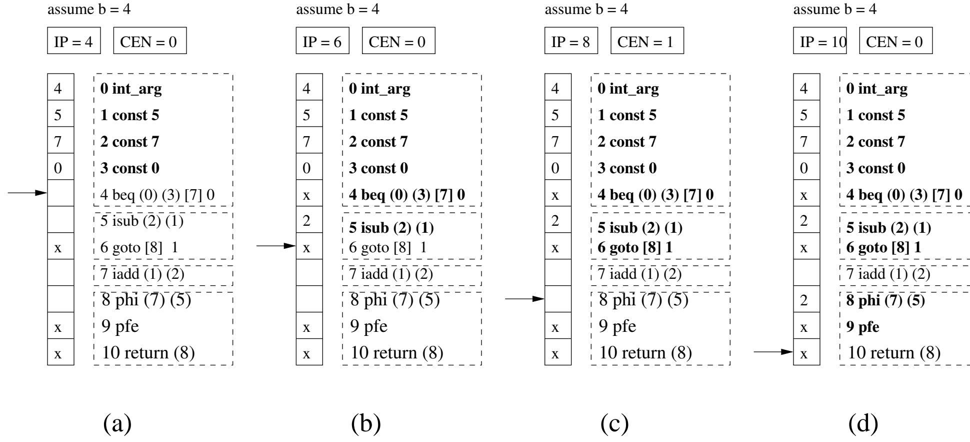


Figure 4: execution of if-then-else construct

2.2 Choosing Φ-function Operands

While SSA naming can be handled by instantiating result registers, ϕ-functions pose a greater obstacle to direct imperative interpretation of programs in SSA Form. In standard SSA Form, each ϕ-function resides in a basic block (where more than one control flow edge converges) and selects one of its input operands (using the value denoted by that operand as its result value) depending on the control flow edge through which the dynamic execution entered the basic block.

Figure 3 shows a simple program with converging control flow translated into Interpretable SSA. The ϕ-functions (that would exist in standard SSA Form) are replaced by `phi` instructions. Since the basic-block control flow graph (CFG) (which is shown as the dashed boxes and arrows in Figure 3(b)) is not explicitly represented, it is not clear how an interpreter should decide whether the `phi` instruction is to copy from RR7 or RR5.

For this reason, ISSA provides an auxiliary CFG-Edge Number (CEN) register, which is set on branching instructions and is used by `phi` instructions to select among their operands. Figure 4 shows several snapshots of this program's execution. Consider the execution of instruction 6 (transforming the state of Figure 4(b) into that of Figure 4(c)); this corresponds to the traversal of the CFG edge labeled "1" in Figure 3(b). ISSA's `goto` instruction takes two immediate operands, the first is the target instruction number (in this case, 8) and the edge number (in this case, 1). When instruction 6 is executed the CEN register is set to

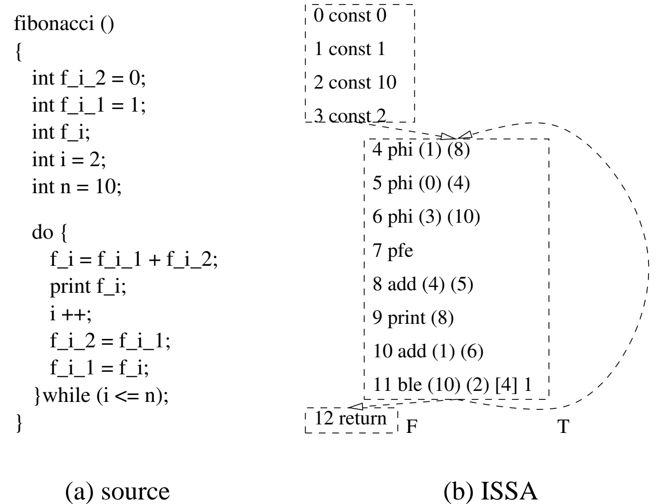


Figure 5: a program computing the Fibonacci sequence

1 and the control is transfered to instruction 8 (the `phi` instruction). Because the CEN register is 1, the second operand of the `phi` instruction is selected, and 2 is read from RR5 and placed in RR8. After this the CEN register is reset to 0; the resulting state can be seen in Figure 4(d).

2.3 Simultaneous Execution of Φ-functions

The observant reader will have noticed that the previous section did not mention the `pfe` (Phi-Function End) instruction marking the end of the `phi` instructions within a basic block. The `pfe` instruction is needed because standard SSA Form ϕ-function semantics require that ϕ-functions be "executed" at the beginning of the basic block in which they reside [Cytron et al., 1991]. An often overlooked consequence of this rule manifests itself when one or more ϕ-function (in a loop) reference the result values of ϕ-functions within the same basic block. In this case, they must be implemented, so that the virtual machine behaves as if they were all executed simultaneously using the previous iteration's result values [Morgan, 1998].

A concrete example of this problem occurs during the execution of the program shown in Figure 5(a), which calculates the first 10 numbers of the Fibonacci sequence. This program has a ϕ-function (instruction 5) that references the result

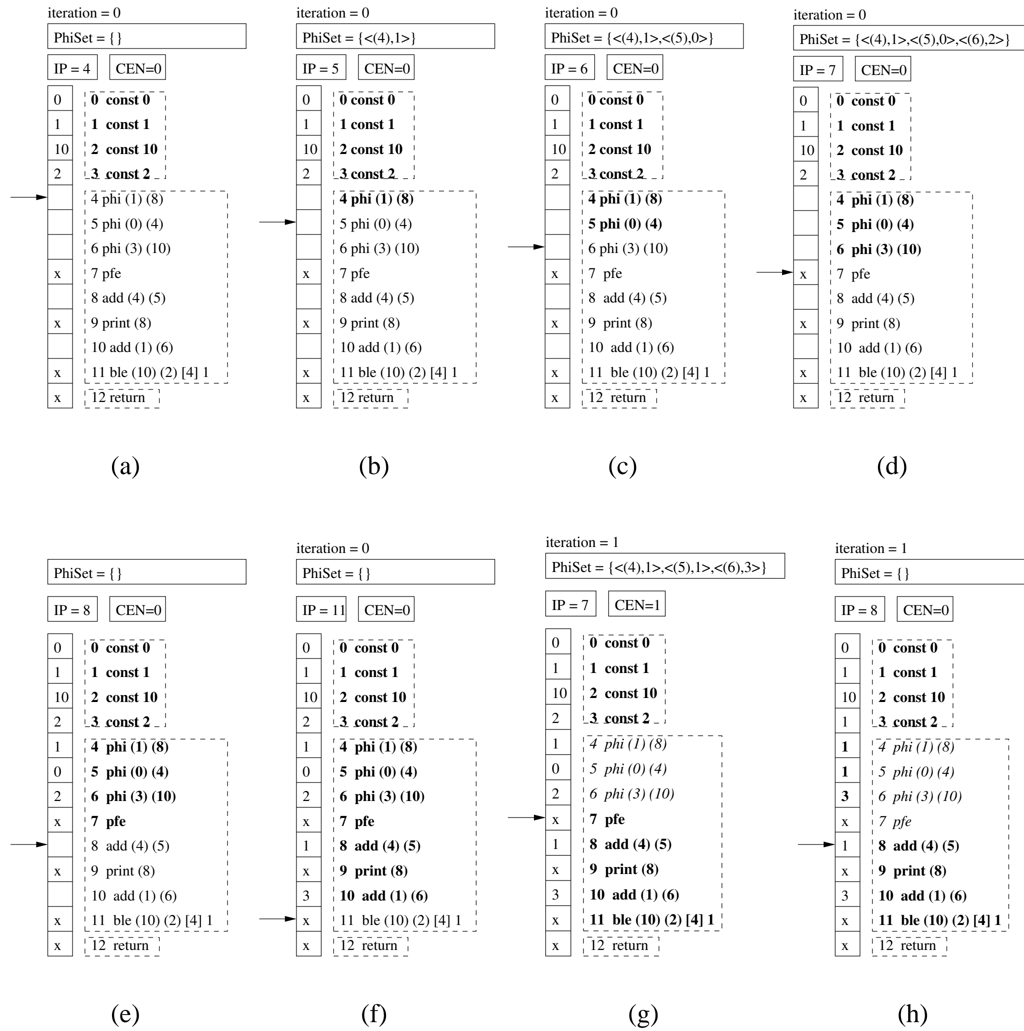


Figure 6: computing the Fibonacci sequence

of a ϕ-function (instruction 4) from the previous iteration. (This happens, because the previous iteration's n-1 becomes the new iteration's n-2; in more complicated examples, there could be multiple mutually dependent ϕ-functions.) If these ϕ-functions were to be executed sequentially simply copying the results from the correct input operand into the result register, instruction 5 will erroneously copy the value placed in RR4 during the current iteration instead of the previous iteration. For example, at the end of the second iteration RR4 and RR5 will both be 1; for the third iteration, the new value of RR4 is 2, but the new value of RR5 should still be 1.

For this reason, an ISSA virtual machine will buffer `phi` instruction transfers until it executes the `pfe` instruction, which commits the transfers stored in the *PhiSet* buffer and resets the CEN register. This solves the problem because all of the reads associated with the SSA ϕ-functions occur at the ISSA `phi` instructions before performing any of the writes (at the `pfe` instruction).

For an example of the `pfe` instruction in action, consider the first two iterations of the Fibonacci program shown in Figure 5(b). Figure 6(a) show the state of the program on initially entering the loop body; the CEN register is 0, indicating that the first operand of the `phi` instructions should be used. As each `phi` instruction is executed, a pair, containing the `phi`'s instruction number and the selected operands current value, is added to the *PhiSet* buffer. When instruction 4 executes, it selects the first operand, reads in the contents of RR1 (i.e. 1), and places (4, 1) into the *PhiSet* buffer (Figure 6(b)); for 5 it reads in RR0 and places (5,0) into the buffer (Figure 6(c)); for 6 it reads in RR3 and places (6,2) into the buffer (Figure 6(d)). After this, the *pfe* instruction executes; it removes each of the pairs out of the PhiSet (the order does not matter) and writes to the appropriate result registers (1 to RR4, 0 to RR5, and 2 to RR6) and resets the CEN register to 0.

The second iteration is entered from the conditional branch of instruction 11 (Figure 6(f)). Because the value of RR10 (i.e. 3) is less than the value of RR2 (i.e. 10), the test succeeds, control transfers control back to instruction 4, and the CEN register is set to 1. Thus, in this iteration, the second operand of each `phi` instruction is selected, and the virtual machine reads in the current values of RR8, RR4, and RR10 and placing (4,1), (5,1), and (6,3) into the *PhiSet* buffer (Figure 6(g)). When the `pfe` instruction executes, the appropriate result registers are written to and the CEN register is reset; the result is shown in Figure 6(h).

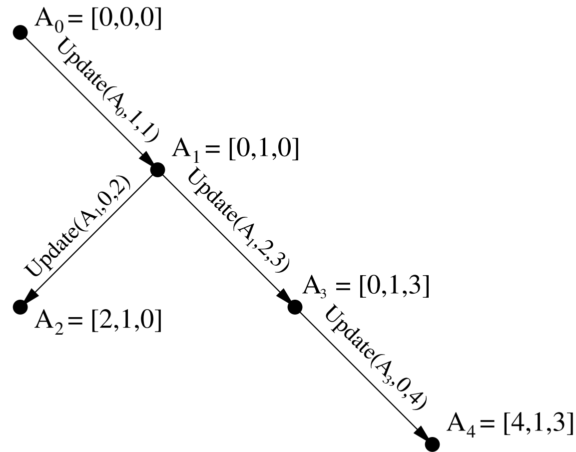


Figure 7: tree depiction of the array model

2.4 Arrays

2.4.1 The Cytron et al. Array Model

Support for non-scalars has long been problematic in SSA, and many extensions have been proposed for supporting arrays and other non-scalars (e.g., Array SSA Form [Knobe and Sarkar, 1998]). The simplest array semantics consistent with the single-assignment property are found in the seminal description of SSA by Cytron et al. [1991]. Cytron et al. treat each array as a single scalar variable with multiple instances and describe two primitives for accessing and manipulating these arrays: *Access(*A_x*,*i*)* and *Update(*A_y*,*j*,*V*)*. The *Access* primitive merely fetches the value of index i in array instance A_x. The *Update* primitive creates a new array instance A_z, which is equivalent to A_y except that element j is changed to V. This model can be viewed as creating a tree (Figure 7) where each array instance is a node, and each *Update* creates a new child instance derived from a parent instance. All instances remain accessible to future *Update*s and *Access*es.

Maintaining multiple instances of each array may seem expensive but it is needed to maintain proper SSA semantics and avoid output dependencies. The output dependencies would not be a problem if the SSA code was produced by a straightforward translation from a source program. If, however, code motion was performed as part of the program's optimization in SSA Form (for example, when debugging compiler output after partial redundancy elimination), an SSA interpreter supporting non-destructive array semantics must be prepared to deal

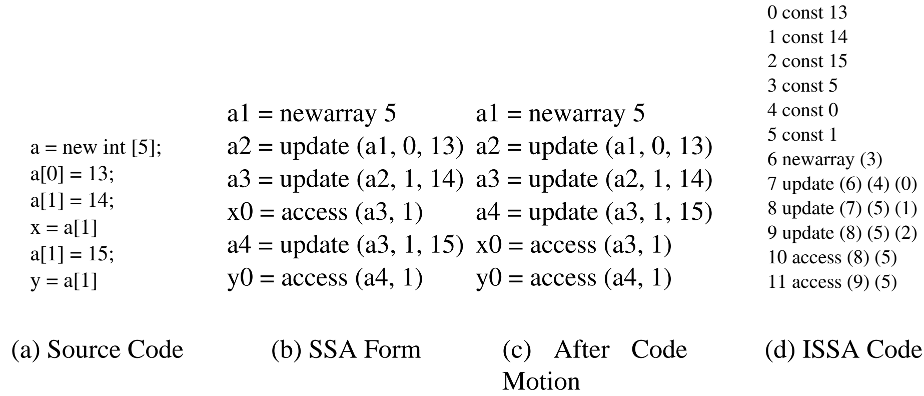


Figure 8: a program exhibiting array manipulation

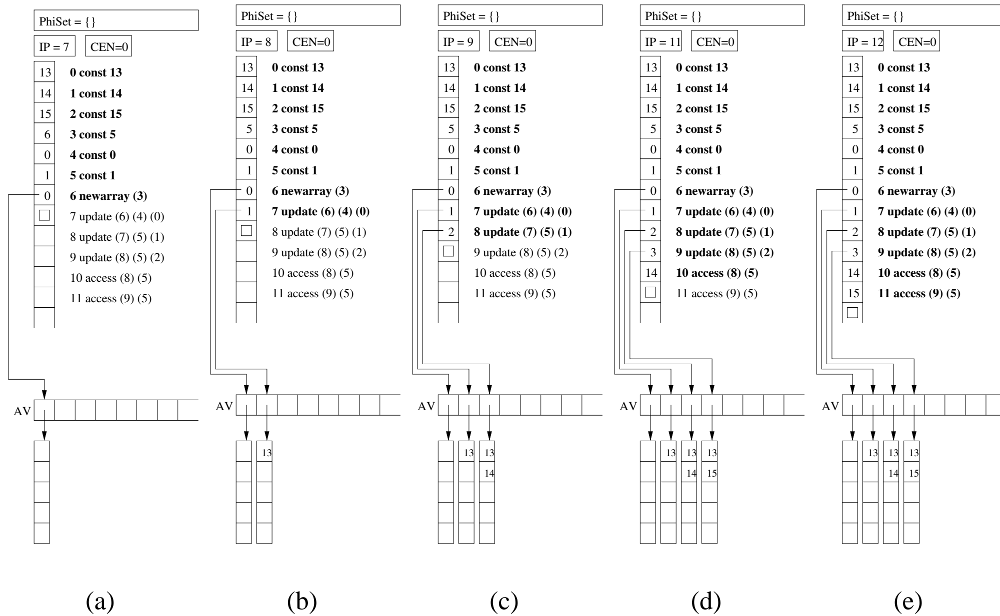


Figure 9: dynamic execution of array manipulation

with the possibility of an array access being moved below an update[1].

An example program requiring non-destructive *update* semantics is shown in Figure 8. Figure 8(b) shows the direct SSA translation of the source program in Figure 8(a). Figure 8(c) which is semantically equivalent to the program in Figure 8(b) but has been altered by legal code motion and as a result accesses an old version of an array (i.e. a3) even after it has been updated (becoming a4). Thus, a3 and a4 have overlapping live ranges and, for this reason, cannot share the same storage space.

2.4.2 ISSA's Implementation of Arrays

ISSA supports array manipulation through `newarray`, `access`, and `update` instructions modeled after the *Update* and *Access* functions of Cytron et al. [1991], which treat the entire array as a single SSA variable. Each `newarray` instruction takes as an operand the number of elements, creates a new array of that size, and places a reference to it in the `newarray`'s result register. Every `access`, takes as operands a reference to an array and the index into the array, fetches the appropriate element from that array, and places a copy of its value in the `access`'s result register. Each `update` instruction takes as operands a reference to an array, an index into that array, and a value, copies the array, writes the value to the element in the new array identified by the index, and places a reference to the new array in the `update`'s result register.

As a concrete example, we will now describe the dynamic execution of the program shown in Figure8(d); Several steps in this program's execution are illustrated in Figure 9. Array references are implemented as indexes into a dynamic data structure called the *array vector* (AV), which contains pointers to all of the arrays instantiated during program execution. The execution of each `newarray` or `update` adds a new array to the *array vector* for each instantiation of the original program array (Figure 9(a), Figure 9(b), and Figure 9(c)). The `access` instructions select array versions by referencing the result register of the instruction which produced that instance; this result register contains an index to the array vector which in turn contains a pointer to the actual array instance. The `access` of instruction 10 uses the array produced by instruction 8, which was unaffected by the `update` at instruction 9 (Figure 9(d)), so retrieves the "old" value of element 1 (i.e., 14). Instruction 11, however, uses the "current" version

[1] Alternatively, the optimizer could be made aware of output dependencies for non-scalars, or output dependencies could be fixed-up by another code motion phase just prior to interpretation, but this goes beyond single-assignment semantics.

Program		Elapsed Time (in sec)					
		ISSA		C	Perl	Java	
		destructive	non-dest.			with JIT	no JIT
Factorial	($12! \times 10^7$)	-	42.57	0.35	115.57	1.04	13.01
Fibonacci	(first 47×10^7)	-	166.12	1.02	719.67	3.20	76.64
Fibonacci (Array)	(first 47×10^5)	2.95	8.98	0.04	11.64	0.37	1.29

Table 1: execution times

Program		Slowdown (Relative to Optimized C Code)				
		ISSA		Perl	Java	
		destructive	non-dest.		with JIT	no JIT
Factorial	($12! \times 10^7$)	-	$122\times$	$330\times$	$3.0\times$	$37\times$
Fibonacci	(first 47×10^7)	-	$163\times$	$706\times$	$3.1\times$	$75\times$
Fibonacci (Array)	(first 47×10^5)	$74\times$	$225\times$	$291\times$	$9.3\times$	$32\times$

Table 2: relative execution time

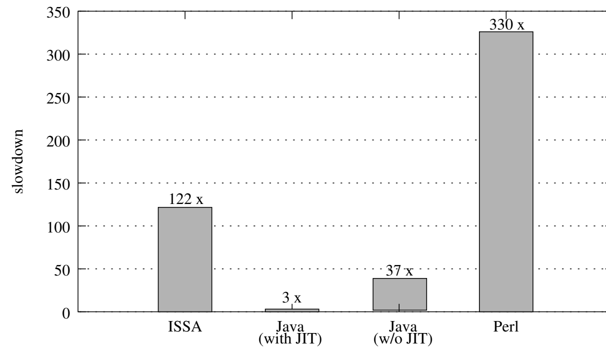
of the array produced by the `update` at instruction 9 (Figure 9(e)) and retrieves the the "current" value of element 1 (i.e., 15).

2.4.3 Optimizations

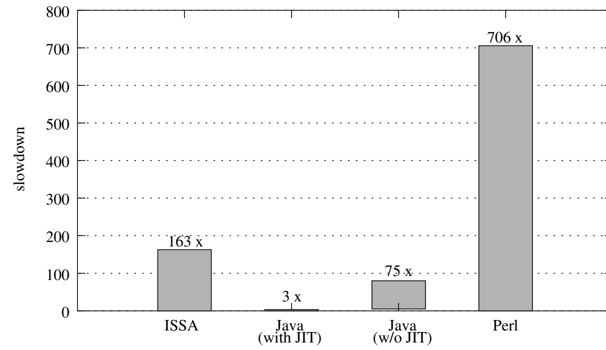
As noted above, each array `update` results in a copy. Most of the time, these are unnecessary. A live range analysis could be used to identify cases where the `update`'s input array is never used again. (For most programs, this would be all of them.) In those cases, the `update` can safely avoid the copy and instead destructively modify the array in place and output the a reference to that same array.

3 Implementation and Performance

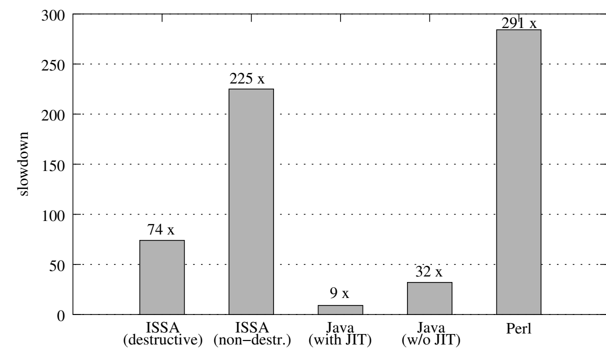
We have implemented a simple prototype ISSA virtual machine in about 1,000 lines of C code. During execution, it reads and parses an ASCII representation of ISSA code and then executes it using a simple interpretive engine consisting of a switch statement (with 30 case statements, one for each instruction opcode) embedded in a loop. The virtual machine is untyped; all immediate and register values are 32-bit words but may be used as integers, single-precision floats, or indexes into the *array vector*, and instruction numbers used as operands are also 32-bits. It uses an array to efficiently implement the PhiSet buffer. In addition,



(a) Factorial



(b) Fibonacci (Scalar)



(c) Fibonacci (Array)

Figure 10: performance slowdown

the the virtual machine performs dynamic bounds checking to ensure that neither invalid instruction numbers nor illegal array manipulation can violate its integrity.

Although dynamic bounds checking guarantees the virtual machine's integrity, the virtual machine does not verify other properties whose violation can only affect program correctness. In particular, CEN values on branches, `phi` instructions, and `pfe` instructions, must be used correctly in order to implement standard SSA semantics, misuse may produce programs that are not in SSA form. Similarly the virtual machine does not distinguish float, integer, and array reference types; instead all data exists as 32-bit words and each instruction uses each of those words as the type appropriate for that instruction operand.

Even though our interpreter was written using loop/switch-dispatch and prioritizing simplicity over performance, we measured its performance on a few simple benchmarks. Because there is no compiler targeting ISSA, we manually transliterated each of our benchmarks from C into ISSA, Perl and Java, and timed their execution in their respective environments[2]. The resulting execution times (in seconds) are shown in Table 1. There were three benchmarks: the first computes the first computes 12! 10,000,000 times, the second computes (in scalar variables) the first 47 elements of the Fibonacci sequence 10,000,000 times, and the third builds a dynamically allocated array containing the first 47 elements of the Fibonacci sequence, repeating this 100,000 times. These benchmarks were executed on a dual-processor 1GHz Pentium III Xeon with 256KB cache and 1GB of 133Mhz SRAM, running RedHat Linux 7.2 with a Linux 2.4.18 kernel; all I/O was suppressed, and the preprocessing times of ISSA virtual machine and Perl times were excluded[3]. The ISSA Fibonacci array benchmark was run using both destructive and non-destructive array manipulations. From this it can be seen that single-assignment semantics for arrays resulted in a $3\times$ slowdown relative to destructive array manipulation, which is perhaps less than expected considering that there were 47 array updates (which one would expect to be the most expensive operation) in every iteration of the outer loop.

Slowdowns relative to optimized C code are show in Figure 10. The ISSA vir-

[2]The prototype virtual machine and the C benchmarks were produced using gcc 2.96 with -O3 switch, and the Java benchmarks were compiled to Java Bytecode using jikes 1.15. The Perl benchmarks were executed using Perl 5.6.1 compiled by RedHat, and the Java benchmarks were executed using the Blackdown Java 2 SDK 1.3.1_02b FCS.

[3]We were unable to obtain the current userspace time consumption from within Java, so we examined the wall-clock time consumed by the computation itself, the time reported by Java's profiling feature, and the user time reported by Linux for the process's complete run, and reported the lowest of these three.

tual machine's performance (with full single-assignment semantics) varied from $122\times$ to $225\times$ slower than the optimized C code. In all cases, it was slower than Sun's JVM implementation but faster than Perl. This puts it slower than the best optimized interpreters (which have slowdowns of less than $10\times$) but as expected for a simple non-threading interpreter and faster than some production interpreters.

4 Future Work

4.1 A Faster ISSA Virtual Machine

The prototype implementation, described above, was written prioritizing code simplicity and the directness of ISSA model implementation over execution speed. We plan to rewrite the interpreter, possibly using vmGen [Ertl et al., 2002] prioritizing performance. We expect this rewrite, applying some of the state-of-the-art interpreter optimizations and hand-tuning the interpreter code, to result in as much as an order of magnitude speedup.

Much of the execution time spent by an optimized interpreter on a modern processor can be attributed to dispatching cost Ertl and Gregg [2003]. Our prototype virtual machine uses loop/switch-dispatch, which is platform independent and easy to implement but is relatively expensive. Each dispatch typically requires the execution of 3 control transfer instructions Gagnon [2002], one of which is an indirect branch that is particularly difficult for hardware to predict Ertl and Gregg [2003]. Threaded execution dispatch techniques [Bell, 1973, Dewar, 1975] can reduce this overhead. In addition, superinstructions [Proebsting, 1995, Piumarta and Riccardi, 1998] can reduce the number of dispatches required, and instruction replication can increase the effectiveness of hardware branch predictors [Ertl and Gregg, 2003]. Our rewritten interpreter will utilizes some type of threaded dispatch and may also make use of superinstructions and replication.

Portable interpreter implementations tend to implement operand stacks and virtual registers as elements of arrays in memory. In stack architectures, it is possible to use one (or more) local variable to hold the top element(s) of the stack Ertl [1995], reducing the number of memory loads. This optimization is not possible in ISSA, but the results of SSA instructions are often used soon after their creation. Thus caching the most recently generated result registers as local variables and accessing these explicitly in the subsequent instructions may result in a significant reduction in operand loads.

In addition to any design-level optimizations, the interpreter code itself could be improved significantly. For example, the interpreters state variables (e.g. IP, CEN) are not currently local variables; thus it is impossible for the compiler to place these into registers. We expect that the code can be tightened significantly.

4.2 A SafeTSA Interpreter

In parallel with the construction of an improved ISSA interpreter described above, Amme and Apel [2003] are creating an interpreter for the SafeTSA representation [Amme et al., 2001], which utilizes some of the techniques described in this paper. This interpreter is what Klint [1981] classifies as a Type III interpreter, consisting of a relatively extensive preprocessor and an interpretive engine. In the initial static preprocessor pass, the interpreter converts SafeTSA's tree structured control primitives into a flat sequence of instructions with explicit branches. In addition, this pass translates ϕ-functions into ISSA-like `phi` instructions, adding the correct CEN operands to branches and adding `pfe` instructions after the `phi` instructions in each basic block. The dynamic interpretive engine currently supports a subset of the features required to implement the SafeTSA language. Specifically, primitive data types can be manipulated and static method calls can be dispatched, but reference types and dynamic dispatch are not yet implemented.

Although reference types have not yet been implemented, two properties of SafeTSA will simplify the treatment of non-scalars compared to the handling of arrays described in this paper. First, SafeTSA's enforced type safety replaces the *array vector*, since array and object references can be statically verified and implemented with direct pointers. Second, SafeTSA's memory operations are destructive making the non-destructive array handling described here is unnecessary.

5 Related Work

As mentioned earlier, this work was motivated by the existence of SafeTSA [Amme et al., 2001] as a mobile code format. SafeTSA differs from ISSA in several ways, including the lack of annotated CFG edge numbers (CEN) and explicit phi-function end (pfe) instructions, the use of tree structured control primitives instead of unrestricted gotos, and the use of destructive heap-memory primitives. The published work on SafeTSA has concentrated on the program representation itself [Amme et al., 2001], processing it with an optimizing compiler in a Java Virtual Machine [Amme et al., 2003], and reducing the online cost of optimiza-

tions [von Ronne et al., 2001, 2002, Hartmann et al., 2003]. None of this work, however, addresses the efficient interpretation of SafeTSA; in fact, Krintz [2002] speculates that direct interpretation is impossible.

Both the Program Dependence Web (PDW) of Ballance et al. [1990] and the Static Single Information (SSI) of Ananian [1999] augment SSA Form with addition information which allows for more explicit execution semantics. To represent a program as a PDW, each of an SSA program's ϕ-functions is replaced with either a γ- or μ-function, depending on whether the operands come from forward or backwards control flow; in addition η-functions (which mark values after the termination of loops) and switches are inserted. This conversion is only possible for programs with reducible control flow graphs, but provides "all the information needed for control-driven, data-driven, or demand-driven interpretation". The interpretation envisioned, however, is not that of an efficient byte-code interpreter but rather that of a dataflow architecture simulator. Similarly, the SSI^+ variant of Static Single Information form adds ξ-functions to loops in order to enable abstract interpretation and provide event driven semantics. The conversion of programs in SSA Form to each of these representations is more involved than annotating branches with CENs and grouping phi-functions with pfe instructions as required for conversion to ISSA.

Interpreting programs in SSA form represents a departure from the traditional stack-based virtual machine; another alternative is the virtual register machine. Davis et al. [2003] report that by having less instructions (and thus reducing indirect jumps) machines with a virtual register architecture can outperform those with a stack-based architecture despite requiring extra memory loads for the explicit operands. The performance characteristics of an ISSA interpreter should be closer to that of a virtual register machine than to those with stack architectures. Both virtual register machines and ISSA reduce the number of instructions at the cost of adding explicit input-operands. The difference is that the ISSA interpreter has less operands, because the instruction result is implicit; this benefit is achieved at the cost of having one result register per instruction, which is less dense than a typical virtual register machine and may increase the size of each operand and have detrimental cache effects.

6 Conclusion

One can indeed construct an interpretable Static Single Assignment Form. Programs in standard SSA Form can be translated into this Interpretable SSA (ISSA)

Form by simply renaming operands to implicit registers, annotating edge numbers at branches, and marking the last ϕ-function in each converging basic block.

We have demonstrated how to build an ISSA interpreter for scalars using a *result register* for each instruction, a *control-flow edge number* register to select `phi` instruction operands, and a *PhiSet* buffer to simultaneously commit `phi` instruction result values. In addition, we have provided an actual implementation of the *Access* and *Update* functions from the single-assignment array model of Cytron et al.. Our prototype ISSA virtual machine is able to handle all of these constructs with the performance expected of a simple non-threading interpreter.

This demonstrates the practicality of constructing virtual machines which interpret programs represented in Static Single Assignment Form. Such SSA interpreters may prove useful in debugging SSA compilers and are a prerequisite for mixed-mode virtual machines using only SafeTSA.

7 Acknowledgements

We would like to thank Fermín Reig and the other reviewers of earlier versions of this paper; their feedback has been invaluable. This research effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Work on the SafeTSA interpreter is directed by Wolfram Amme and funded by the Deutsche Forschungsgemeinschaft under grant AM-150/1-1.

References

Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical Report SMLI TR-2000-87, Sun Microsystems, Palo Alto, CA, June 2000.

Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, 1988.

Wolfram Amme and Alexander Apel. Interpreting programs in SafeTSA.

Technical report, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Jena, Germany, 2003.

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 *Proceedings of the SIGPLAN'01 conference on Programming language design and implementation*, pages 137–147, 2001.

Wolfram Amme, Jeffery von Ronne, and Michael Franz. Using the SafeTSA representation to boost the performance of an existing java virtual machine. In *10th International Workshop on Compilers for Parallel Computers*, January 2003.

C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachussetts, September 1999. URL `http://www.cag.lcs.mit.edu/~cananian/Publications/`.

Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the conference on Programming language design and implementation*, pages 257–271, 1990.

James R. Bell. Threaded code. *Communications of the ACM*, 16:370–372, 1973.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.

Robert B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18: 330–331, June 1975.

M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN 1995 conference on Programming language design and implementation*, pages 315–327, 1995.

M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the SIGPLAN 2003 conference on Programming language design and implementation*, pages 278–288, 2003.

M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

Etienne M. Gagnon. *A Portable research framework for the execution of java bytecode*. PhD thesis, McGill University, 2002. URL `http://www.info.uqam.ca/~egagnon/gagnon-phd.pdf`.

Andreas Hartmann, Wolfram Amme, Jeffery von Ronne, and Michael Franz. Code annotation for safe and efficient dynamic object resolution. In *2nd International Workshop on Compiler Optimization Meets Compiler Verification*, April 2003.

Paul Klint. Interpretation techniques. In *Software–Practice and Experience*, pages 11:963–973, 1981.

Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 107–120, 1998.

Chandra Krintz. Improving mobile program performance through the use of a hybrid intermediate representation. In *2nd Workshop on Intermediate Representation Engineering for Virtual Machines*, June 2002.

Robert Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, Massachusetts, 1998.

Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, 1998.

Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, 1995.

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

Jeffery von Ronne, Michael Franz, Niall Dalton, and Wolfram Amme. Compile time elimination of null- and bounds-checks. In *9th Workshop on Compilers for Parallel Computers*, June 2001.

Jeffery von Ronne, Andreas Hartmann, Wolfram Amme, and Michael Franz. Efficient online optimization by utilizing offline analysis and the SafeTSA representation. In James F. Power and John T. Waldron, editors, *Recent Advances in Java Technology: Theory, Application, Implementation*, chapter 27, pages 233–241. Computer Science Press, Trinity College Dublin, 2002. ISBN 0-9544145-0-0.

A Implementation of the Interpreter Core

A.1 ssa_vm.c

```
#include "inst.h"
#include "ssa_vm.h"
#include "ssa_array.h"
#include "ssa_parser.h"
#include <stdlib.h>
#include <stdio.h>

typedef struct phi_assignment phi_assignment;

struct phi_assignment
{
  ssa_variable v;         //v = value to be transfered
  int ovi;                //ovi = output variable index
};

typedef struct
{
  inst **ia;              //ia = instruction array
  int ial;                //ial = instruction array length
  ssa_array_vector *av;   //av = array vector
  ssa_variable *oa;       //oa = output array (size = inst_length)
  phi_assignment *pq;     //pq = phi-assignment queue
  int pqt;                //pqt = phi-assignment queue top
  int ip;                 //ip = instruction pointer (index to inst_array)
  int cen;                //cen = CFG edge number
} vm_state;


void init(vm_state *s, inst* inst_array[], int inst_length);
void commit_phis (vm_state *s);
int execute(vm_state *s, inst *instruction);

inline int pq_empty (vm_state s) {return s.pq == NULL;}

inline ssa_variable decode_immediate (inst *instruction, int i) {
  ssa_variable v = ((ssa_variable*) instruction->data)[i];
  return v;
}

inline ssa_variable decode_operand (vm_state *s, inst *instruction, int i) {
  int index = ((int*) instruction->data)[i];
  return s->oa[index];
}

inline ssa_variable first_operand (vm_state *s, inst *instruction) {
  return decode_operand(s, instruction, 0);
}

inline ssa_variable second_operand (vm_state *s, inst *instruction) {
  return decode_operand(s, instruction, 1);
}
```

```
void ssa_vm (inst* inst_array[], int inst_length)
{
  vm_state s;
  inst* ci;

  init(&s, inst_array, inst_length);
  while (1)
    {
      if (s.ip < 0 || s.ip >= s.ial) abort(); // we jumped out of the method
      ci = s.ia[s.ip];
      if (execute (&s, ci) == 0) break;
    }
}


void init(vm_state *s, inst* inst_array[], int inst_length)
{
  // instructions -- should we copy these?
  s->ia = inst_array;
  s->ial = inst_length;
  // simple registers;
  s->ip = 0;
  s->cen = 0;
  // complex data
  s->av = av_init();
  s->oa = (ssa_variable*) calloc (inst_length, sizeof(ssa_variable));
  s->pq = (phi_assignment*) calloc (inst_length, sizeof(phi_assignment));
  s->pqt = 0;
}

void commit_phis(vm_state *s)
{
  int i;

  for (i = 0; i < s->pqt; i++) {
    s->oa[s->pq[i].ovi] = s->pq[i].v;
  }

  s->pqt = 0;

}

int execute(vm_state *s, inst *ci)
{
  // temporaries
  int t,n,a,i;
  ssa_variable x,y;

  switch (ci->opcode)
    {
    case CONST:
      x = decode_immediate (ci,0);
      s->oa[s->ip] = x;
      s->ip++;
      break;
    case PRINT:
```

```
  //puts("printing int");
  x = first_operand(s,ci);
  printf ("==> %d\n", x.i);
  s->ip++;
  break;
case ADD:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i + y.i;
  s->ip++;
  break;
case SUB:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i - y.i;
  s->ip++;
  break;
case DIV:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i / y.i;
  s->ip++;
  break;
case MUL:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i * y.i;
  s->ip++;
  break;
case AND:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i && y.i;
  s->ip++;
  break;
case OR:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  s->oa[s->ip].i = x.i || y.i;
  s->ip++;
  break;
case NEG:
  x = first_operand(s,ci);
  s->oa[s->ip].i = -x.i;
  s->ip++;
  break;
case BGE:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  t = decode_immediate(ci,2).t;          // branch target
  n = decode_immediate(ci,3).n;          // CFG Edge Number
  if (x.i >= y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
```

```
    }
  break;
case BGT:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  t = decode_immediate(ci,2).t;          // branch target
  n = decode_immediate(ci,3).n;          // CFG Edge Number
  if (x.i > y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BLE:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  t = decode_immediate(ci,2).t;          // branch target
  n = decode_immediate(ci,3).n;          // CFG Edge Number
  if (x.i <= y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BLT:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  t = decode_immediate(ci,2).t;          // branch target
  n = decode_immediate(ci,3).n;          // CFG Edge Number
  if (x.i < y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BNE:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
  t = decode_immediate(ci,2).t;          // branch target
  n = decode_immediate(ci,3).n;          // CFG Edge Number
  if (x.i != y.i)
    {
      s->cen = n;
      s->ip = t;
    } else {
      s->ip++;
    }
  break;
case BEQ:
  x = first_operand(s,ci);
  y = second_operand(s,ci);
```

```
    t = decode_immediate(ci,2).t;          // branch target
    n = decode_immediate(ci,3).n;          // CFG Edge Number
    if (x.i == y.i)
      {
        s->cen = n;
        s->ip = t;
      } else {
        s->ip++;
      }
    break;
  case GOTO:
    t = decode_immediate(ci,0).t;          // branch target
    n = decode_immediate(ci,1).n;          // CFG Edge Number
    s->cen = n;
    s->ip = t;
    break;
  case EXIT:
    //puts("exiting");
    exit(0);
  case RETURN:
    x = decode_operand(s,ci,0);   // thing to set the element to
    exit(x.i);
  case PHI:
    // check that the PHI is big enough for the cfg edge number
    n = ci->opdnum;

    if (s->cen >= n) abort();        // phi must have enough operands
    if (s->pqt >= s->ial) abort(); // can't overflow phi queue buffer

    // record the data in the phi-assignment queue
    s->pq[s->pqt].ovi = s->ip;
    s->pq[s->pqt].v = decode_operand(s, ci, s->cen);;
    s->pqt++;

    // next instruction
    s->ip++;
    break;
  case PFE:
    commit_phis(s);
    s->cen = 0;
    s->ip++;
    break;
  case NOOP:
    s->ip++;
    break;
  case NEWARRAY:
    x = decode_operand(s,ci,0);   // thing to set the element to
    s->oa[s->ip].a = av_newarray(s->av, x.i);
    s->ip++;
    break;
  case UPDATE:
    a = decode_operand(s,ci,0).a;  // array (index of array in array vector)
    i = decode_operand(s,ci,1).i;  // element (index into array)
    x = decode_operand(s,ci,2);    // thing to set the element to
    s->oa[s->ip].a = av_update(s->av, a, i, x);  // result is new array
    s->ip++;
    break;
```

```
    case ACCESS:
      a = decode_operand(s,ci,0).a;  // array (index of array in array vector)
      i = decode_operand(s,ci,1).i;  // element (index into array)
      s->oa[s->ip] = av_access(s->av, a, i);  // result is element
      s->ip++;
      break;
    case FADD:
      x = first_operand(s,ci);
      y = second_operand(s,ci);
      s->oa[s->ip].f = x.f + y.f;
      s->ip++;
      break;
    case FSUB:
      x = first_operand(s,ci);
      y = second_operand(s,ci);
      s->oa[s->ip].f = x.f - y.f;
      s->ip++;
      break;
    case FDIV:
      x = first_operand(s,ci);
      y = second_operand(s,ci);
      s->oa[s->ip].f = x.f / y.f;
      s->ip++;
      break;
    case FMUL:
      x = first_operand(s,ci);
      y = second_operand(s,ci);
      s->oa[s->ip].f = x.f * y.f;
      s->ip++;
      break;
    case FCONST:
      s->oa[s->ip] = decode_immediate (ci,0);
      s->ip++;
      break;
    case FPRINT:
      x = first_operand(s,ci);
      s->ip++;
      break;
    default:
      abort();
    }
  return 1;
}
```

A.2 ssa_vm.h

```
#ifndef SSA_VM_H
#define SSA_VM_H

#include "inst.h"

typedef signed s32;
typedef unsigned u32;
typedef float f32;

typedef union ssa_variable {
    s32 i;  // 32-bit signed integer integer
    f32 f;  // 32-bit floating point value
    u32 a;  // 32-bit unsigned array vector index
    u32 n;  // 32-bit unsigned array CFG Edge Number
    u32 t;  // 32-bit unsigned branch target
} ssa_variable;

void ssa_vm(inst* inst_array[], int inst_length);
#endif
```

A.3 ssa_array.h

```
#ifndef SSA_ARRAY_H
#define SSA_ARRAY_H

#include "ssa_vm.h"
#include <stdlib.h>

typedef struct
{
  unsigned l;         // array length
  ssa_variable *a;  // array of ssa_variables
} ssa_array;

typedef struct
{
  unsigned na;  // next array
  unsigned l;   // allocated length
  ssa_array *v; // array (vector) of arrays
} ssa_array_vector;

ssa_array_vector *av_init();
u32 av_newarray (ssa_array_vector *av, int size);
void av_cleanup (ssa_array_vector *av);
u32 av_update (ssa_array_vector *av, u32 av_index, u32 a_index, ssa_variable v);
ssa_variable av_access (ssa_array_vector *av, u32 av_index, u32 array_index);
u32 av_fastupdate (ssa_array_vector *av, u32 av_index, u32 array_index,
    ssa_variable v);

#endif
```

A.4 inst.h

```
#ifndef INST_H
#define INST_H

#include <stdlib.h>

#define MAX_INSTS 1024

#define BASE 257 /* the first 256 is assigned to ascii char */

typedef struct
{
  int     opcode;
  int     opdnum; /* length in words */
  char    * img;
  void    * data;
  int     data_type; /* int, bool, float */
}inst;


typedef struct
{
  char * img;
  int opdnum;
}inst_attribute;

static inst_attribute  inst_att [] = {
  {"const",1},
  {"fconst", 1},
  {"add"  ,2},
  {"sub"  ,2},
  {"div"  ,2},
  {"mul"  ,2},
  {"and"  ,2},
  {"or"   ,2},
  {"neg"  ,1},
  {"fadd",   2},
  {"fsub",   2},
  {"fdiv",   2},
  {"fmul",   2},
  {"bge"  ,4},
  {"bgt"  ,4},
  {"ble"  ,4},
  {"blt"  ,4},
  {"bne"  ,4},
  {"beq"  ,4},
  {"goto" ,2},
  {"phi"  ,-1},
  {"pfe"  ,0},
  {"update", 3},
  {"access", 2},
  {"newarray", 1},
  {"exit", 0},
  {"return", 1},
  {"print", 1},
  {"fprint", 1},
```

```
  {"null",0},
};

extern inst * insts_array[];
extern int insts_size;

/* inst.c */
inst *new_inst(int opcode);
inst *new_unary_inst(int opcode, int opd);
inst *new_unary_finst(int opcode, float opd);
inst *new_binary_inst(int opcode, int opd1, int opd2);
inst *new_tenary_inst(int opcode, int opd1, int opd2, int opd3);
inst *new_quandary_inst(int opcode, int opd1, int opd2, int opd3, int opd4);
inst *new_phi_inst(int opcode, int opdnum, int opd[]);
void print_all_insts (inst * insts[], int size);
void print_inst (inst * ist);
void delete_all_insts(inst *insts[], int size);

#endif /* no INST_H */
```

B Benchmarks

B.1 Factorials

B.1.1 factorial.ssa

```
//  Find 12!, 40,000,000

0 const_0              // zero
1 const_1              // one
2 const_12             // x
3 const_40000000       // iterations

// Outer Loop
4 phi_2 (0 12)
5 pfe

// Inner Loop
6 phi_2 (1 9)   // f
7 phi_2 (1 10)   // j
8 pfe

9  mul (6 7)
10 add (7 1)
11 ble (10 2) [6] 1

// Outer Loop Continued
12 add (4 1)
13 blt (12 3) [4] 1

// Exit
14 exit
```

B.1.2 factorial.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv){

        clock_t start, end, used = 0;

        int f,x,i,j;

        start = clock();
        f = 1;
        x = 12;

        i = 0;
        do {
                f = 1;
                j = 1;
                do {
                        f = f * j;
                        j++;
                } while (j <= x);
                i++;
        } while (i < 10000000);

        end = clock();
        fprintf(stderr,"Time used %d\n",end-start);
}
```

B.1.3 Factorial.java

```
public class Factorial {

    public static void main(String[] args){

        long start;
        long end;

        int f,x,i,j;

        start = System.currentTimeMillis();
        f = 1;
        x = 12;

        i = 0;
        do {
                f = 1;
                j = 1;
                do {
                        f = f * j;
                        j++;
                } while (j <= x);
                i++;
        } while (i < 10000000);
        end = System.currentTimeMillis();
        System.out.println("Time used : " + Long.toString(
           end-start));

    }
}
```

B.1.4 factorial.pl

```
#!/usr/bin/perl -w

require 'sys/syscall.ph';

$TIMEVAL_T = "LLLL";
$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_times, $start, -1);
$f=1;
$x=12;

$i=0;
do {
                $f = 1;
                $j = 1;
                do {
                        $f = $f * $j;
                        $j++;
                } while ($j <= $x);
                $i++;
        } while ($i < 10000000);


syscall(&SYS_times, $done, 0);

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# fix microseconds
#for ($done[1], $start[1]) { $_ /= 1_000_000 }

print"Time used : ".($done[0]-$start[0])."\n";
```

B.2 Fibbonacci Sequence (in scalars)

B.2.1 fibonacci.ssa

```
// F(0) = 0
// F(1) = 1
// F(n) = F(n-2) + F(n-1) for all n >= 2
//
// Calculate F(46), 10,000,000 times

// Block 0
0 const_0          //
1 const_1          //
2 const_46         // max = 46
3 const_2          // n = 2
4 const_10000000 // iterations

// Outer Loop
5 phi_2 (0 14)
6 pfe

// Inner Loop
7  phi_2 (0  8)          //  phi (f_0, f_{n-2})
8  phi_2 (1  11)         //  phi (f_1, f_{n-1})
9  phi_2 (3  12)         //  phi (n=2,   n+1)
10 pfe

11 add   (7  8)          //  f_{n} = f_{n-2} + f_{n-1}
12 add   (1  9)          //  n <- n+1
13 ble   (12 2) [7] 1    //  n <= max repeat loop

//Outer Loop Continued
14 add (5 1)
15 blt (14 4) [5] 1

// End
16 exit
```

B.2.2 fibonacci.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv){

        clock_t start, end, used = 0;
        int i, n, max = 46, f_n, f_n_1, f_n_2;

        start = clock();

        f_n_1 = 0;
        f_n_2 = 1;

        i = 0;
        do {
                n = 2;
                do {
                        f_n = f_n_2 + f_n_1;
                        f_n_2 = f_n_1;
                        f_n_1 = f_n;
                        n = n + 1;
                } while (n <= 46);
                i = i + 1;
        } while (i<10000000);

        end = clock();
        fprintf(stderr,"Time used %d\n",end-start);
}
```

B.2.3 Fibonacci.java

```
public class Fibonacci {

    public static void main(String[] args){

        long start, end;

        int f_n_2, f_n_1, f_n, n;
        int max = 46;

        int i;

        start = System.currentTimeMillis();

        f_n_1 = 0;
        f_n_2 = 1;

        i = 0;
        do{
                n=2;
                do{
                        f_n = f_n_2 + f_n_1;
                        f_n_2 = f_n_1;
                        f_n_1 = f_n;
                        n = n + 1;
                }while(n<=46);
                i = i + 1;
        }while(i<10000000);

        end = System.currentTimeMillis();
        System.out.println("Time used : " + Long.toString(
            end-start));

    }
}
```

B.2.4 fibonacci.pl

```
#!/usr/bin/perl -w

require 'sys/syscall.ph';

$TIMEVAL_T = "LLLL";
$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_times, $start, 0);

$f_n_1 = 0;
$f_n_2 = 1;

$i=0;
do {
                $n = 2;
                do {
                        $f_n = $f_n_2 + $f_n_1;
                        $f_n_2 = $f_n_1;
                        $f_n_1 = $f_n;
                        $n = $n + 1;
                } while ($n <= 46);
                $i = $i + 1;
        } while ($i<10000000);

syscall(&SYS_times, $done, 0);

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# fix microseconds
#for ($done[1], $start[1]) { $_ /= 1_000_000 }

print"Time used : ".($done[0]-$start[0])."\n";
```

B.3 Fibbonacci Sequence (in an array)

B.3.1 fibonacci_array.ssa

```
//  F[0] = 0
//  F[1] = 1
//  F[n] = F[n-1] + F[n-2]
//  Find F[46], 100,000 using arrays

0 const_0         // 0
1 const_1         // 1
2 const_2         // j0 = 2
3 const_46        // x = 46
4 const_100000    // iterations

// Outer Loop
5 phi_2 (0 22)     // i
6 pfe

7  add (3 1)
8  newarray 7          // F = newarray x+1
9  update (8 0) 0     // f[0] = 0
10 update (9 1) 1   // F[1] = 1

// Inner Loop
11 phi_2 (2 20)          // j
12 phi_2 (10 19)         // F
13 pfe

14 sub (11 1)            // j - 1
15 sub (11 2)            // j - 2
16 access (12 14)        // F[j-1]
17 access (12 15)        // F[j-2]
18 add (16 17)
19 update (12 11) 18   // F[j] = F[j-1] + F[j-2]
20 add (11 1)            // j = j + 1
21 ble (20 3) [11] 1


// Outer Loop Continued
```

```
22 add (5 1)
23 blt (22 4) [5] 1

// Exit
24 exit
```

B.3.2 fib_array.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv){

        clock_t start, end;

        int i, j, *f, x;


        start = clock();

        x=46;
        i = 0;

        do {
                f = (int*) malloc (sizeof(int) * (x + 1));
                f[0] = 0;
                f[1] = 1;
                j = 2;
                do {
                        f[j] = f[j-1] + f[j-2];
                        j++;
                } while (j <= x);
                free (f);
                i++;
        } while (i < 100000);

        end = clock();
        fprintf(stderr,"Time used %d\n",end-start);
}
```

B.3.3 FibArray.java

```
public class FibArray {

    public static void main(String[] args){

        long start;
        long end;

        int[] f;
        int i , j;
        int x;

        start = System.currentTimeMillis();

        x = 46;
        i = 0;
        do{
                f = new int[x+1];
                f[0]=0;
                f[1]=1;
                j=2;
                do{
                        f[j] = f[j-1] + f[j-2];
                        j++;
                }while(j<=x);
                f=null;
                i++;
        }while(i<100000);

        end = System.currentTimeMillis();
        System.out.println("Time used : " + Long.toString(
            end-start));
    }
}
```

B.3.4 fib_array.pl

```
#!/usr/bin/perl -w

require 'sys/syscall.ph';

$TIMEVAL_T = "LLLL";
$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_times, $start, 0);

$x=46;
$i=0;

do {
        @f = (0..$x);
        $j = 2;
        do {
                $f[$j] = $f[$j-1] + $f[$j-2];
                $j++;
        } while ($j <= $x);
        $i++;
} while ($i < 100000);

syscall(&SYS_times, $done, 0);

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

print"Time used : ".($done[0]-$start[0])."\n";
```