

# Interpreting Programs in Static Single Assignment Form

Jeffery von Ronne  
jronne@ics.uci.edu

Ning Wang  
wangn@ics.uci.edu

Michael Franz  
franz@uci.edu

School of Information and Computer Science  
University of California Irvine  
Irvine, CA 92697

## 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.

## 1. INTRODUCTION

Intermediate representations based on Static Single Assignment (SSA) Form [2, 22] have been used in many research and industrial optimizing compilers. Leveraging this body of work, we have previously developed SafeTSA [3], a verifiable external program representation, which decreases the effort needed for just-in-time compilation without sacrificing the safety or quality of the resulting machine code [4]. Just-in-time compilation, however, still imposes a startup delay, which may not be justified for infrequently executed methods. Java Virtual Machine implementations, such as Sun's HotSpot Performance Engine [1], typically use mixed-mode interpretation and compilation to combine interpretation's shorter startup times with compiled code's better throughput. Perhaps because of several non-imperative features of SSA, conventional high-performance interpreters have not been written for SSA representations. Consequently, Krintz has 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 [18], allowing both compilation and interpretation at the cost of supporting two program representations.

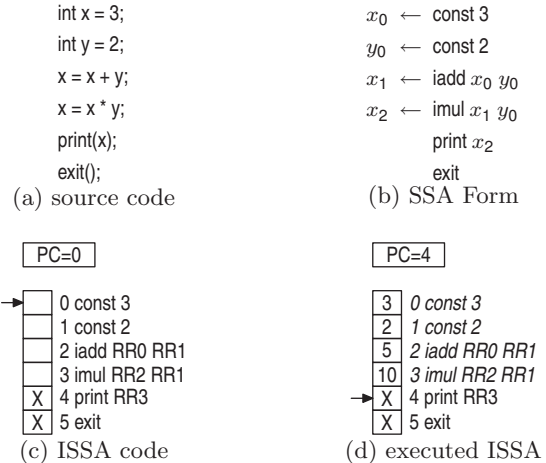


Figure 1: simple program

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 proposed [18] without the overhead of supporting two input program representations. Incidentally, the same interpreter technology could also 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 have non-obvious imperative semantics: the provision of a separate variable name for each definition, the selection of  $\phi$ -function operands, the simultaneous execution of mutually dependent  $\phi$ -functions, and the handling of non-scalar variables (such as arrays) with single-assignment semantics.

## 2. INTERPRETABLE SSA

### 2.1 Unique Naming

The principal property of Static Single Assignment Form 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), our In-

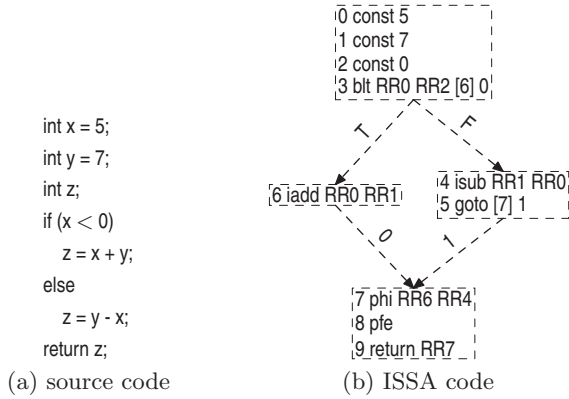


Figure 2: program with if-then-else structure

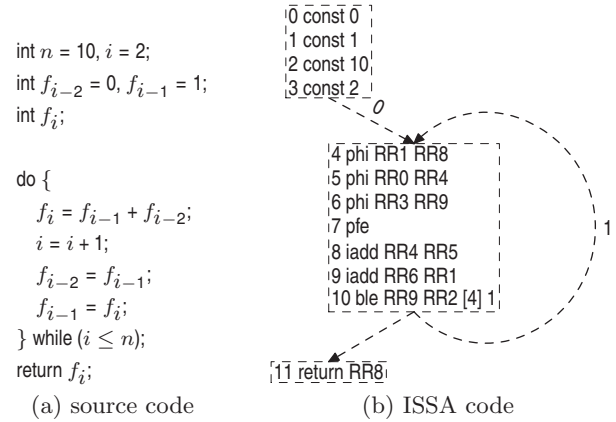


Figure 4: program finding the Fibonacci sequence

## 2.2 Choosing $\phi$ -function Operands

SSA’s  $\phi$ -functions pose the greatest obstacles to direct imperative interpretation. In standard SSA Form, each  $\phi$ -function resides in a basic block (at which more than one control flow edge converges) and selects an output from among its input operands based on which control flow edge the dynamic execution entered the basic block through.

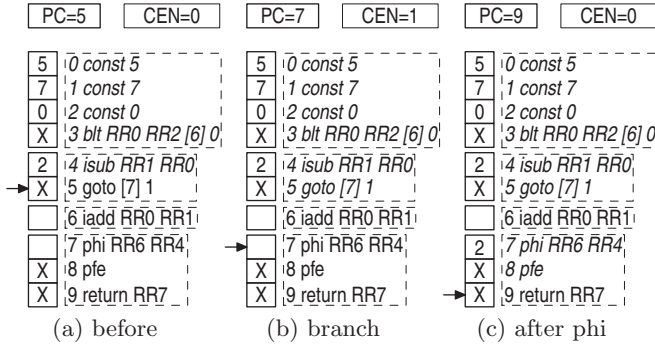


Figure 3: executing if-then-else constructs

interpretable SSA (ISSA) instantiates an abstract machine for each program containing one *result register* per instruction. Each instruction in ISSA is labeled with an instruction number. A few instruction types, such as the *const* instructions, take immediate integer values, but most have indirect operands which specify a value by referring to the result register of the value’s defining instruction.

Figure 1 shows the execution of a simple program’s abstract machine. The instructions’ result registers appear as the boxes to the left of the instructions; an auxiliary *program counter* (PC) register is used to indicate the instruction to be executed next. As each instruction executes, it retrieves its inputs from the indicated registers, performs its computation, and writes to the appropriate result register (RR) on its left. For example, as instruction 3 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.

Figure 2 shows a simple program with converging control flow translated into Interpretable SSA. The  $\phi$ -functions (which would exist in standard SSA Form) have been replaced by *phi* instructions. It is not clear how an interpreter should decide whether the *phi* instruction is to copy from RR6 or RR4, especially since the basic-block control flow graph (CFG) (which is shown as the dashed boxes and arrows in Figure 2(b)) is not explicitly represented in ISSA.

For this reason, ISSA provides an auxiliary CFG-Edge Number (CEN) register, which is set on branching instructions and is used by each *phi* instruction to select among its operands. Figure 3 shows some snapshots of this program’s execution. Consider the execution of instruction 5 (transforming the state of Figure 3(a) into that of Figure 3(b)); this corresponds to the traversal of the CFG edge labeled “1” in Figure 2(b). ISSA’s *goto* instruction takes two immediate operands, the first is the target instruction number (in this case, 7) and the edge number (in this case, 1). When instruction 5 is executed the CEN register is set to 1 and the control is transferred to instruction 7 (the *phi* instruction). Because the CEN register is 1, the second operand of the *phi* instruction is selected, and 2 is read from RR4 and placed in RR7. After this the CEN register is reset to 0; the resulting state can be seen in Figure 3(c).

## 2.3 Simultaneous Execution of $\phi$ -functions

The observant reader will have noticed that the previous section glossed over the *pfe* (Phi-Function End) instruction, which marks the end of the *phi* instructions within a basic block. The *pfe* instruction is needed<sup>1</sup> because standard SSA Form  $\phi$ -function semantics require that  $\phi$ -functions be “executed” at the beginning of the basic block in which they

<sup>1</sup>Some method of marking basic blocks or super-instructions containing multiple  $\phi$ -functions could be used instead.

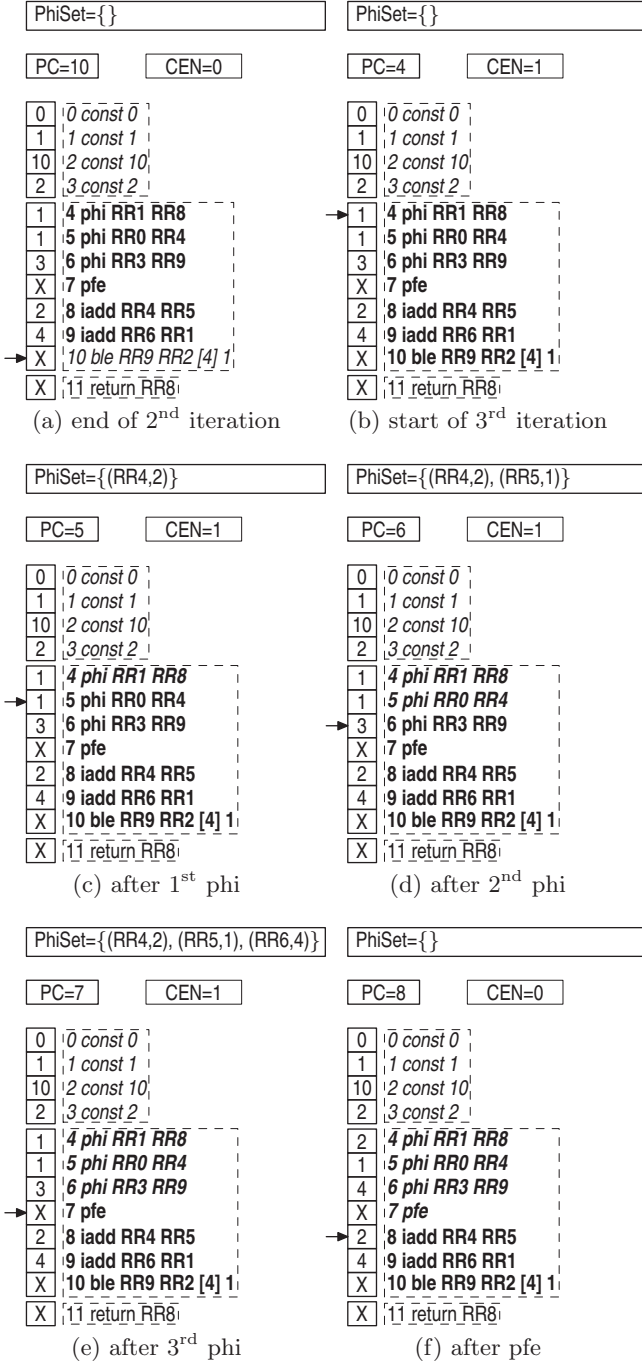


Figure 5: computing the Fibonacci sequence

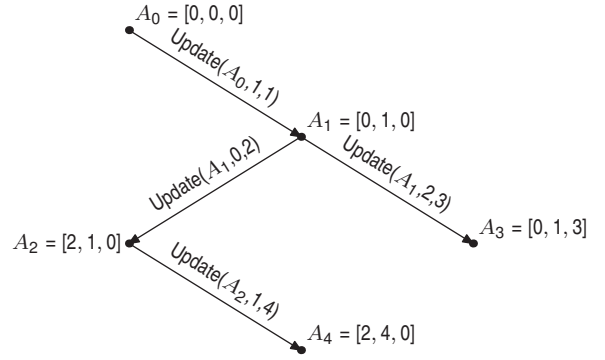


Figure 6: array model tree

reside [8]. An often overlooked consequence of this rule manifests itself when a  $\phi$ -function (in a loop) references the result value of another  $\phi$ -function within the same basic block. In this case, they must be implemented so that they behave as if they were all executed simultaneously using the previous iteration's result values [19].

A concrete example of this problem occurs during the execution of the program shown in Figure 4, which calculates the first 10 numbers of the Fibonacci sequence. This program has a  $\phi$ -function (instruction 5) that references the result (RR4) of another  $\phi$ -function from the previous iteration. (This happens, because the previous iteration's  $f_{i-1}$  becomes the new iteration's  $f_{i-2}$ ; in more complicated programs, there could be multiple mutually dependent  $\phi$ -functions.) If these  $\phi$ -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 (Figure 5(a)), RR4 and RR5 will both be 1; for the third iteration (Figure 5(f)), 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  $\phi$ -functions occur at the ISSA `phi` instructions before performing any of the writes (at the `pfe` instruction).

## 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 [17]). The simplest array semantics consistent with the single-assignment property are found in the seminal description of SSA by Cytron et al. [8]. This model treats 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 at 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  $j$  has been changed to

a = new int [3];	$a_0 \leftarrow \text{newarray } 3$
a[0] = 13;	$a_1 \leftarrow \text{update } (a_0, 0, 13)$
a[1] = 14;	$a_2 \leftarrow \text{update } (a_1, 1, 14)$
x = a[1];	$x_0 \leftarrow \text{access } (a_2, 1)$
a[1] = 15;	$a_3 \leftarrow \text{update } (a_2, 1, 15)$
y = a[1];	$y_0 \leftarrow \text{access } (a_3, 1)$
(a) source Code	(b) SSA form

$a_0 \leftarrow \text{newarray } 3$	0 const 13
$a_1 \leftarrow \text{update } (a_0, 0, 13)$	1 const 14
$a_2 \leftarrow \text{update } (a_1, 1, 14)$	2 const 15
$a_3 \leftarrow \text{update } (a_2, 1, 15)$	3 const 0
$x_0 \leftarrow \text{access } (a_2, 1)$	4 const 1
$y_0 \leftarrow \text{access } (a_3, 1)$	5 const 3
(c) after code motion	6 newarray RR5
	7 update RR6 RR3 RR0
	8 update RR7 RR4 RR1
	9 update RR8 RR4 RR2
	10 access RR8 RR4
	11 access RR9 RR4
	(d) ISSA code

Figure 7: program with arrays

the value,  $V$ . This model can be viewed as creating a tree (Figure 6) 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 *Updates* and *Accesses*.

Maintaining multiple instances of each array may seem expensive, but they are 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 (e.g., when debugging compiler output after partial redundancy elimination), an SSA interpreter supporting non-destructive array semantics must be prepared to deal with the possibility of an array access being moved below an update.<sup>2</sup>

An example program requiring non-destructive *Update* semantics is shown in Figure 7. Figure 7(b) shows the direct SSA translation of the source program in Figure 7(a). Figure 7(c) shows an SSA program, which is semantically equivalent to that shown in Figure 7(b) but which has been altered by legal code motion and, as a result, accesses an old version of an array (i.e.,  $a_2$ ) even after it has been updated (becoming  $a_3$ ). Thus,  $a_2$  and  $a_3$  have overlapping live ranges and, for this reason, cannot share the same storage space.

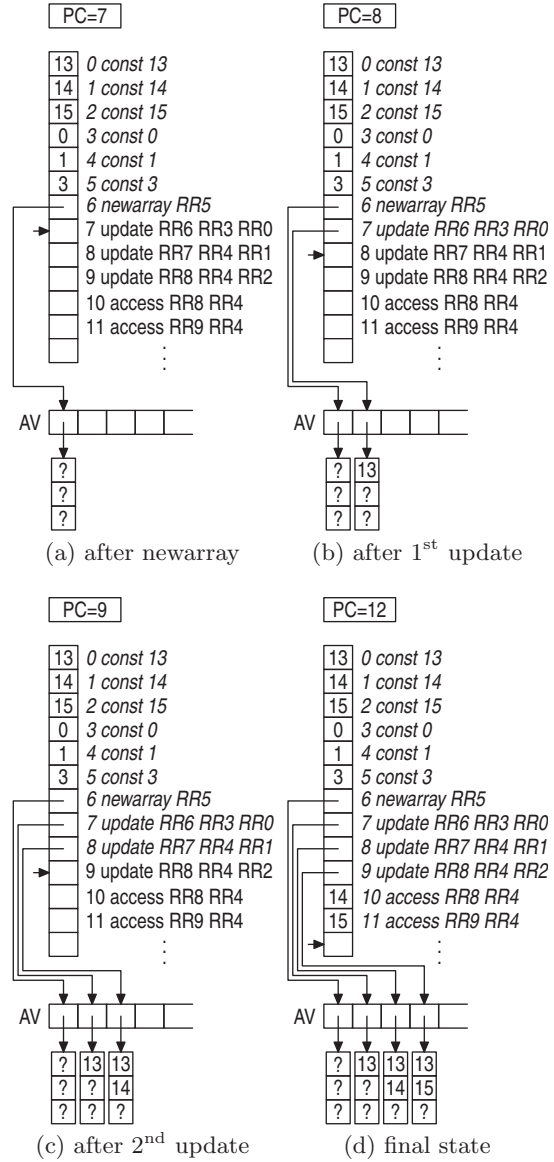


Figure 8: execution of array manipulation

<sup>2</sup>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 prior to interpretation, but these solutions go beyond single-assignment semantics.

### 2.4.2 ISSA’s Implementation of Arrays

ISSA supports array manipulation with `newarray`, `access`, and `update` instructions modeled after the *Update* and *Access* functions of the Cytron et al. model [8], which treats each entire array as a single SSA variable. Each `newarray` instruction takes as its operand the number of elements, creates a new array of that size, and places a reference to the new array in the `newarray` instruction’s result register. Every `access` takes, as operands, a reference to an array and the index into that array. It, then, fetches the appropriate element from that array and places a copy of that element’s 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 new value. After that, it copies the array and writes the new value to the element of the new array identified by the index. Finally, the `update` places a reference to the new array in its result register.

As a concrete example, we will now describe the dynamic execution of the program shown in Figure 7; several steps in this program’s execution are illustrated in Figure 8. Array references are implemented as indexes into the *array vector* (AV), a dynamic data structure, 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 (Figure 8(a), Figure 8(b), and Figure 8(c)). The `access` instructions select array instances by referencing the result register of the array instance’s defining instruction; this result register contains an index to the array vector, which in turn contains a pointer to the actual array instance. For example, the `access` of instruction 10 uses the array produced by instruction 8, which was unaffected by the `update` at instruction 9, so it retrieves the “old” value of element 1 (i.e., 14). Instruction 11, however, uses the “current” version of the array produced by the `update` at instruction 9 and retrieves the “current” value of element 1 (i.e., 15) (Figure 8(d)).

### 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. PROTOTYPE 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*. Similarly, result registers are referenced using 32-bit words encoding each definition’s instruction numbers. The result register file and the PhiSet buffer are implemented as arrays in main memory. In addition, the virtual machine performs dynamic bounds checking to ensure that neither invalid instruction numbers nor illegal array manipulation can violate its integrity.

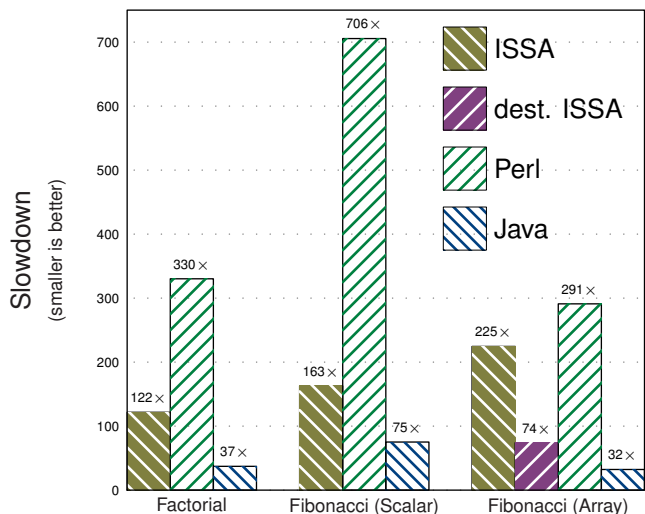


Figure 9: performance slowdown

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 individual instructions use those words as if they were of the types appropriate for those instructions.

Even though our interpreter was written using switch dispatch and prioritizing simplicity over performance, we measured its performance on a few simple benchmarks.<sup>3</sup> 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.<sup>4</sup> The resulting execution times (in seconds) are shown in Table 1. There were three benchmarks:

1. **Factorial** computes the first twelve factorials ten million times.
2. In each of ten million iterations, **Fibonacci (Scalar)** uses scalar variables to find the forty-seventh element of the Fibonacci sequence.<sup>5</sup>

<sup>3</sup>The complete source code for these benchmarks can be found in [25].

<sup>4</sup>The prototype interpreter 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.

<sup>5</sup>12! and the 47th element of the Fibonacci sequence are the largest numbers of their respective sequence that do not overflow 32-bit words.

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 $47 \times 10^7$ )	-	166.12	1.02	719.67	3.20	76.64
Fibonacci (Array)	(first $47 \times 10^5$ )	2.95	8.98	0.04	11.64	0.37	1.29

Table 1: execution times

3. **Fibonacci (Array)** builds a dynamically allocated array containing the first forty-seven elements of the Fibonacci sequence, repeating this one hundred thousand 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 the ISSA and Perl interpreters were excluded.<sup>6</sup> The ISSA Fibonacci array benchmark was run using both destructive and non-destructive array manipulations. On this benchmark, single-assignment semantics for arrays resulted in a  $3\times$  slowdown relative to destructive array manipulation, which is, perhaps, less than would be expected considering that there were forty-seven array updates (which one would expect to be the most expensive operation) in every iteration of the outer loop.

Figure 9 shows performance slowdowns relative to optimized C code. The ISSA interpreter’s performance (with full single-assignment semantics for arrays) varied from  $122\times$  to  $225\times$  slower than the optimized C code. This places its performance between that of Sun’s JVM and Perl on all three benchmarks. Thus, while the prototype is slower than the best optimized interpreters (which can have slowdowns of less than  $10\times$ ), it is faster than at least one widely used interpreter and performs reasonably well for a simple non-threading interpreter.

## 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 [13], 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 [12]. Our prototype virtual machine uses switch dispatch, which is platform independent and easy to implement but is also relatively expensive. Each dispatch typically requires the execution of 3 control transfer instructions

<sup>6</sup>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.

[14], one of which is an indirect branch that is particularly difficult for hardware to predict [12]. Threaded execution dispatch techniques [7, 10] can reduce this overhead. In addition, superinstructions [21, 20] can reduce the number of dispatches required, and instruction replication can increase the effectiveness of hardware branch predictors [12]. Our rewritten interpreter will utilize some type of threaded dispatch and may also make use of superinstructions and replication.

Portable interpreter implementations often implement their operand stacks and virtual registers as elements of arrays in memory. In interpreters of stack-oriented languages, it is possible to use one (or more) machine registers to hold the top element(s) of the stack [11], 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., PC, 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 are creating an interpreter for the SafeTSA representation [3], which utilizes some of the techniques described in this paper. This interpreter is designed as what Klint classifies as a Type III interpreter [16], 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  $\phi$ -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 more expensive non-destructive array handling described here unnecessary.

An additional challenge to interpreting SafeTSA will be the efficient handling of recursive function calls. The most obvious implementation is to copy all result registers into the stack on each function call. This naive mechanism may prove too expensive in space or time, and it may be necessary to explicitly represent the storing of live result registers on a stack.

## 5. RELATED WORK

This work was motivated by the existence of SafeTSA [3] as a mobile code format, but 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 [3], processing it with an optimizing compiler in a Java Virtual Machine [4], and reducing the online cost of optimizations [23, 24, 15]. None of this work, however, addresses the interpretation of SafeTSA.

Both the Program Dependence Web (PDW) [6] and the Static Single Information (SSI) [5] augment SSA Form with additional information which allows for more explicit execution semantics. To represent a program as a PDW, each of an SSA program’s  $\phi$ -functions is replaced with either a  $\gamma$ - or  $\mu$ -function (depending on whether the operands come from forward or backwards control flow); in addition  $\eta$ -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 bytecode interpreter but rather that of a dataflow architecture simulator. Similarly, the SSI<sup>+</sup> variant of Static Single Information form adds  $\xi$ -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. argue 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 [9]. 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) by simply renaming operands to implicit registers, annotating edge numbers at branches, and marking the last  $\phi$ -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 of Cytron et al.’s single-assignment array model. Our prototype handles all of these constructs with the performance expected of a simple non-threading interpreter.

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

## 7. ACKNOWLEDGEMENTS

We would like to thank Alex Apel for his assistance, especially in preparing the Perl and Java versions of the benchmark programs. We would also like to thank Fermín Reig, Vivek Haldar, Andreas Hartmann, Roxana Dianconescu, Niall Dalton, Peter Fröhlich, and the anonymous reviews for their suggestions, feedback, and corrections; these have been invaluable.

This effort is partially funded 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, by the National Science Foundation under grants CCR-0205712 and CCR-0105710, and by the Office of Naval Research under grant N00014-01-1-0854.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA), the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

Wolfram Amme and Alex Apel are developing the SafeTSA interpreter under grant AM-150/1-3 of the Deutsche Forschungsgemeinschaft.

## 8. REFERENCES

- [1] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. Technical Report SMLI TR-2000-87, Sun Microsystems, Palo Alto, CA, June 2000.

- [2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, 1988.
- [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 *Proceedings of the SIGPLAN'01 conference on Programming language design and implementation*, pages 137–147, 2001.
- [4] W. Amme, J. von Ronne, and M. Franz. Using the SafeTSA representation to boost the performance of an existing java virtual machine. In *10th International Workshop on Compilers for Parallel Computers*, Jan. 2003.
- [5] C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1999.
- [6] R. A. Ballance, A. B. Maccabe, and K. 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.
- [7] J. R. Bell. Threaded code. *Communications of the ACM*, 16:370–372, 1973.
- [8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. 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.
- [9] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*, pages 41–49, 2003.
- [10] R. B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18:330–331, June 1975.
- [11] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN 1995 conference on Programming language design and implementation*, pages 315–327, 1995.
- [12] M. A. Ertl and D. 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.
- [13] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.
- [14] E. M. Gagnon. *A Portable research framework for the execution of java bytecode*. PhD thesis, McGill University, 2002.
- [15] A. Hartmann, W. Amme, J. von Ronne, and M. Franz. Code annotation for safe and efficient dynamic object resolution. In *2nd International Workshop on Compiler Optimization Meets Compiler Verification*, Apr. 2003.
- [16] P. Klint. Interpretation techniques. In *Software—Practice and Experience*, pages 11:963–973, 1981.
- [17] K. Knobe and V. 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.
- [18] C. 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.
- [19] R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Woburn, Massachusetts, 1998.
- [20] I. Piumarta and F. Ricciardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, 1998.
- [21] T. 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.
- [22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.
- [23] J. von Ronne, M. Franz, N. Dalton, and W. Amme. Compile time elimination of null- and bounds-checks. In *9th Workshop on Compilers for Parallel Computers*, June 2001.
- [24] J. von Ronne, A. Hartmann, W. Amme, and M. Franz. Efficient online optimization by utilizing offline analysis and the SafeTSA representation. In J. F. Power and J. T. Waldron, editors, *Recent Advances in Java Technology: Theory, Application, Implementation*, chapter 27, pages 233–241. Computer Science Press, Trinity College Dublin, 2002.
- [25] J. von Ronne, N. Wang, A. Apel, and M. Franz. A virtual machine for interpreting programs in static single assignment form. Technical Report 03-19, Information and Computer Science, Univeristy of California, Irvine, Oct. 2003.