SSI on Demand

André Luiz Camargos Tavares, Fernando Magno Quintão Pereira,
Mariza A. S. Bigonha, Roberto S. Bigonha

Universidade Federal de Minas Gerais, Brazil

Abstract. The Static Single Information (SSI) form is a program representation that enables optimizations such as array bound checking elimination and conditional constant propagation. Transforming a program into SSI form has a non-negligible impact on compilation time; however, only a few SSI clients, that is, optimizations that use SSI, require a full conversion. In this paper we describe the SSI framework that we have implemented for the LLVM compiler, and that is now part of this compiler's standard distribution. In our design, optimizing passes inform to the compiler a list of variables of interest, which are then transformed to present, fully or partially, the SSI properties. Thus, we provide to each client only the subset of SSI that the client needs. Our implementation orchestrates the execution of clients in sequence, avoiding redundant work when two clients request the conversion of the same variable. As empirically demonstrated, in the context of an industrial strength compiler, our approach saves compilation time and keeps the program representation small, while enabling a vast array of code optimizations.

1 Introduction

The Static Single Information (SSI) form is a program representation introduced by Scott Ananian [1]. This ten years old program representation redefines some variables at *program split points*, which are basic blocks with two or more successors. SSI form enables many compiler optimizations, because it allows an analyzer to augment variables with information inferred from the result of conditional branches. A non-exhaustive list of potential SSI clients includes array bounds check elimination [3], bitwidth analysis [23], flow sensitive range interval analysis [24], conditional constant propagation [1, 25], partial redundancy elimination [16], fast liveness analysis [4, 21], and busy expression elimination [20].

The SSI representation suits the needs of many compiler optimizations – henceforth called clients; however, different clients require that different subsets of the source program be in SSI form. For instance, Bodik *et al.* [3]'s ABCD algorithm uses information from conditional branches to put bounds on the value of variables used as array indices. Thus, it requires that only integer variables used in conditionals bear SSI properties. Even less demanding is the sparse conditional constant propagation algorithm described by Ananian [1] and Singer [21], which demands that variables used in equality comparisons be in SSI form. On the other hand, the partial redundancy elimination algorithm described by Johnson

et al. [15] uses an analysis called *anticipability*. A non-iterative computation of the anticipatable variables requires that all program variables be in SSI form.

In this paper we describe an *on-demand SSI conversion* framework. Our framework saves compilation time and space in three different dimensions. First, it converts only a subset of variables in the source program to SSI form. Clients provide to our module a list of variables that must have the SSI properties, and only these variables are transformed. Second, we provide two conversion modes for each variable: *full* and *partial*. If a variable is fully converted into SSI form, then it presents the SSI properties traditionally described in the literature [1, 4, 21]. On the other hand, if a variable is partially transformed, then it presents a restricted set of properties, that we describe in this paper. The partial conversion fits the needs of many SSI clients [1, 3, 21, 23–25], and, contrary to the full conversion, it uses a non-iterative algorithm, which is faster, as we empirically demonstrate. Third, our SSI conversion algorithm is a state-full black-box. Because we allow different clients invoking our converter in sequence, we log the SSI conversions that we perform, so that subsequent requests on the same variable do not lead to redundant work being performed.

Our SSI framework is now part of the default distribution of the Low Level Virtual Machine [17] (LLVM), version 2.6. LLVM is an industrial strength compiler, used by companies like Cray [1] and Apple [2]. We have implemented two SSI clients: the ABCD algorithm of Bodik *et al.* [3], and a sparse conditional constant propagation (CCP) algorithm, similar to the one described by Singer [21, p.59]. When compiling the SPEC CPU 2000 benchmark suite, the partial transformations that ABCD and CCP request are about 15 and 24 times faster than fully converting a program to SSI. The SSI conversion is based on the insertion of special instructions – σ-functions and ϕ-functions – in the source program. ABCD and CCP generate approximately 6.5 and 10 times less special instructions than the full conversion. We emphasize that the same infrastructure is used in the three transformations that we have compared: CCP, ABCD and full; however, because we build the SSI representation on demand, we give to each client only the program properties that it requires.

In Section 2 we review the SSI form. In Section 3 we describe some compiler optimizations and analyses that benefit from SSI form, and we show how different subsets of this program representation serve these clients. In Section 4 we discuss our approach to build the SSI representation on demand. Section 5 validates the paper with a series of experiments, and Section 6 concludes this work.

2 Background on SSI

The term Static Single Information form seems to have been coined by Scott Ananian in his master thesis [1]; however, program representations with similar properties have been described before [16]. Benoit *et al.* [4] distinguish two main flavors of the Static Single Information form: *strong*, introduced by Ananian [1]

[1] `http://blogs.rapidmind.com/2009/05/27/why-we-chose-llvm/`

[2] `http://arstechnica.com/apple/news/2007/03/apple-putting-llvm-to-good-use.ars`

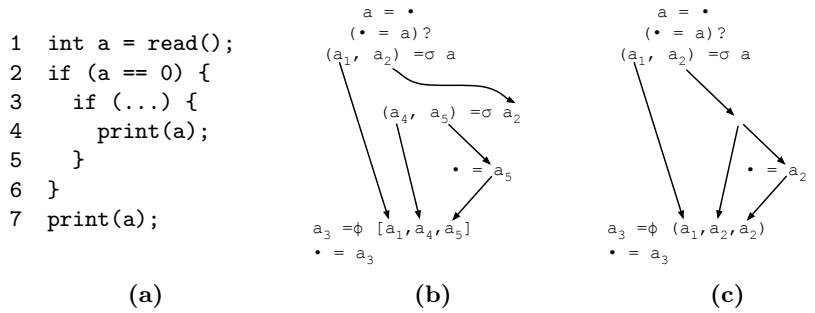

Fig. 1. (a) Source program. (b) Full SSI conversion. (c) Partial SSI conversion. We use the symbol • to indicate that the left or right side of the instruction is not important.

and *weak*, described by Singer [21]. These representations are not equivalent [4]. Any strong SSI form program is also a weak SSI form program; thus, we will be using SSI as a synonym for Strong SSI. According to Benoit *et al.*, four properties characterize strong SSI form:

- *pseudo-definition*: there exists a definition of each variable at the starting point of the program's control flow graph.
- *single reaching-definition*: each program point is reached by at most one definition of each variable.
- *pseudo-use*: there exists a use of each variable at the ending point of the program's control flow graph.
- *single upward-exposed-use*: from each program point it is possible to reach at most one use of a variable, without passing by a previous use.

Figure 1(a) shows a program written in a C like language, and Figure 1(b) gives the control flow graph of this program, in SSI form. The program in Figure 1(c) is not in SSI form, as it contains a point exposed to two different uses of a_2.

In order to convert a program into SSI form we need two special types of instructions: ϕ-functions and σ-functions. ϕ-functions are an abstraction used in the *Static Single Assignment form* (SSA) [11] to join the live ranges of variables. Any SSI form program is a SSA form program. For instance, the assignment below, at the beginning of a basic block B,

$$v = \phi(v_1, \ldots, v_n)$$

works as a multiplexer. It will assign to v the value in v_i, if the program flow reaches block B coming from the i^{th} predecessor of B.

The σ-functions are the dual of ϕ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs an assignment depending on the execution path taken. For instance, the assignment below, at the end of a basic block B,

$$(v_1, \ldots, v_n) = \sigma\ v$$

has the effect of assigning to v_i the value in v if control flows into the i^{th} successor of B. Notice that variables alive in different branches of a basic block are given different names by the σ-function that ends that basic block.

The insertion of ϕ and σ functions is a form of *live range splitting*. The live range of a variable is the set of program points where that variable is alive. Variable v is said to be alive at program point p if there is a path from p to a use of v that does not go through any definition of v. Two algorithms for converting a program into SSI form have been described in the literature: we have Ananian's [1] pessimistic algorithm, and Singer's [21] optimistic approach. We describe Singer's approach in Section 4.

There exists an interesting relationship between the live range of program variables and graphs. Chaitin *et al.* [10] have shown that the intersection graph of the live ranges of a general program can be any type of graph. In 2005 a number of researchers have shown that the intersection graphs produced from programs in SSA form are chordal [6, 8, 19, 13]. Even more recently, Brisk *et al.* [9] showed that the interference graphs of programs in SSI form are interval graphs, a subset of the family of chordal graphs. This last proof had some omissions, later fixed by Benoit and Brisk [4].

3 Examples of SSI clients

In this section we show examples of compiler optimizations that use the SSI representation, giving emphasis on the subset of SSI that each client needs. The SSI facilitates two types of program analyses. First, it helps analyses that extract information from conditional statements, such as constant propagation and array bound checks elimination. Second, it facilitates sparse backwards analyses that associate information with the uses of variables. Section 3.1 discusses examples in the former class, and Section 3.2 goes over the latter.

3.1 Information analyses

Information analyses are among the main reasons behind the design of the SSI representation. These analyses use information from conditional branches to enable compiler optimizations, such as removing redundancies inserted to ensure language safety. For instance, Figure 2(a-c) shows three common Java idioms where exceptional cases are identified by the programmer via conditional tests. However, similar tests will be implicitly created by the java compiler to enforce the strongly typed nature of the language [2]. In the figures, these tests appear in bold face. In another example, Figure 2(d) shows a Ruby program where a runtime test is used to handle integer overflows. The code in lines 3 and 4 is implicitly performed, at runtime, by the Ruby interpreter; however, given the loop boundaries, this test will never be true.

Among the examples of redundant code elimination based on information analyses we cite Bodik *et al.*'s ABCD algorithm [3] and the sparse conditional constant propagation method of Wegman and Zadeck [25]. Ananian describes

```
1  int array[];
2  void s(int i, int v) {
3   if (i < v.length) {
4     if (i >= v.length)
5       throw new ArrayIndex-
        OutOfBoundsException();
6     v[i] = v;
7   } else {
8     // handle error
9   }
10 }
```

(a)

```
1  void f(Object o) {
2    if (o instanceof V)
3      if (o.getClass() != V)
4        throw new Class-
         CastException();
5      ((V) o).m();
6    else {
7      // handle error
8    }
9  }
```

(b)

```
1  int div(int a, int b) {
2    if (b != 0) {
3      if (b == 0)
4        throw new
         ArithmeticException()
6      return a / b;
7    } else {
8      // handle error
9    }
10 }
```

(c)

```
1  sum = 0
2  (1..10).each do |i|
3    if (sum_overflow(sum, i))
4      change sum to BigInt
5    sum += i
6  end
```

(d)

Fig. 2. Examples of defensive programming idioms.

a long list of information analyses when introducing the SSI representation [1]. Furthermore, many compilers already perform simple forms of redundant check elimination. For instance, LLVM is able to eliminate simple boundary checks inserted by the GNAT front-end used in the compilation of ADA programs.

In addition to removing redundant code, information analyses are also useful to discover the range of values that variables might assume. For instance, in Figure 2(a) we know that any value of variable `v` used in the true branch of the conditional is less than `v.length`. Harrison [14] provides examples of how compilers benefit from range analyses. Members of such family of analyses include the bitwidth inference engine of Stephenson *et al.* [23], the range propagation algorithm of Su and Wagner [24], and the range analyzis used by Patterson to predict the outcome of branches [18].

Although well known in the literature, information analyses are described using different program representations; however, they all can be modeled as sets of constraints extracted from SSI form programs. For instance, in order to perform conditional constraint propagation on the program in Figure 1(b) we must solve the constraints $\langle a = \top \rangle$, $\langle a_1 \neq 0 \rangle$, $\langle a_2 = 0 \rangle$, $\langle a_4 = a_2 \rangle$ and $\langle a_5 = a_2 \rangle$. After solving the constraints we can replace the instruction $\bullet = a_5$ by $\bullet = 0$.

Different SSI form clients have different needs. For instance:

– the ABCD algorithm that removes redundant array bound checks [3], as in Figure 2(a), requires only that variables used in conditionals, and that represent either array indices or array lengths have SSI properties;
– in order to remove redundant type casts, as in Figure 2(b), a client must require that variables used as operands of the `instanceof` function be in SSI form.
– in order to remove redundant divide-by-zero tests, as in Figure 2(c), we need that numeric variables used in conditionals be in SSI form.
– the version of ABCD that we implement in this paper requires any variable used in branches to be in SSI form. Our implementation is used as a general *redundant test elimination*, that removes the test in line 3 in Figure 2(d), for instance.

Information analyses do not require that variables be fully converted into SSI form. Instead, they need a representation that restricts the *value range* of variables. The value range of a variable is the set of values that the variable may assume during program execution. For instance, variable a in line 1 of Figure 1(a) may assume any value of the integer type in the Java language; thus, its value range is $[-2^{31}, 2^{31} - 1]$. However, the conditional branch in line 2 restricts the value range of a to $[0, 0]$ when the test is true. If every variable created by a σ-function has the same value range, like $(a_4, a_5) = \sigma\ a_2$ in Figure 1(b), then this σ-function is redundant for an information analysis. As an example, the program in Figure 1(c) also allows to infer that the value of a_2 is always 0, but with fewer constraints than the program in Figure 1(b).

3.2 Backward analyses

In addition to being useful for information analyses, the SSI representation also facilitates sparse backward analyses. Singer [20] gives two examples of such analyses: very busy expressions, and the dual available expression analysis. An expression e is very busy at program point p if e is computed in any path from p to the end of the program, before any variable that is part of it is redefined. Such analysis, also called *anticipatable expressions analysis* by Johnson and Pingali [16], is useful for performing optimizations such as partial redundancy elimination. Conversely, an expression e is *available* at program point p if it is computed in any path from the beginning of the program until p, and none of the variables that are part of e are redefined thereafter.

A sparse analysis associates information to variables, instead of program points. That is, the busy expressions associated to variable v are the busy expressions at the definition point of v. Similarly, the available expressions associated to v are the expressions available at the program point where v is last used. The SSI representation allows us to perform these analyses non-iteratively [20]. As another example, Benoit *et al.* [4] have shown how SSI speeds up the computation of liveness analysis. This is a dataflow analysis that finds which are the live variables at each program point.

Contrary to the analyses described in Section 3.1, the backward dataflow analyses demand the full power of the SSI representation. That is, every variable

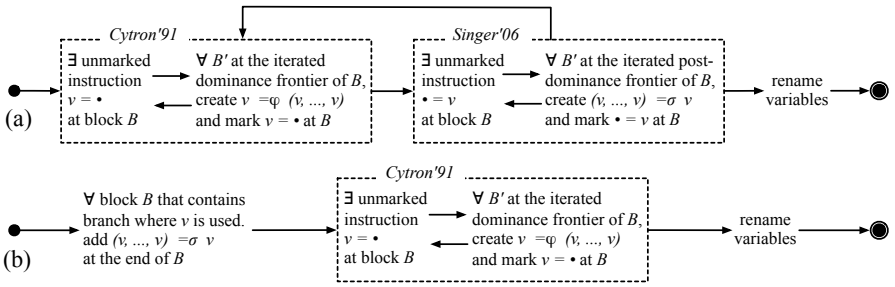

Fig. 3. (a) Singer optimistic algorithm to convert a program into SSI form (b) Our algorithm to produce partial SSI form.

of the source program must present the SSI properties enumerated in Section 2. We will show that the same infrastructure that supports information analyses also supports the backward analyses described in this section.

4 Building SSI on demand

We convert a program into SSI form on demand. This means that a client gives to our converter a list of variables of interest, and we modify only these variables. There are two modes of conversion, *partial* and *full*. We convert a variable to SSI form via live range splitting and renaming. The difference between the partial and the full conversion is the amount of live range splitting required.

Converting a program to full SSI form If a variable is fully converted to SSI form, then it meets the definition of strong SSI form. This type of conversion is useful for the backward analyses described in Section 3.2. In order to perform the full conversion, we use the algorithm designed by Singer [21, p.46]. This method – shown in Figure 3(a) – combines Cytron *et al.*'s algorithm to insert ϕ-functions [12], and Singer's algorithm to insert σ-functions [21].

The insertion of σ-functions guarantees the single upward-exposed-use property. This phase happens as follows: for each use of a variable v, Singer inserts a σ-function in each basic block in the post dominance frontier of v. A basic block B_2 post-dominates a basic block B_1 if every path, from the exit of the source program to B_1 contains B_2. If B_2 post-dominates a predecessor of basic block B_0, but does not post-dominates B_0, then B_0 is in the post-dominance frontier of B_2. The σ-functions produce new uses of v, which cause the insertion of more σ-functions. This process iterates until a fix-point is reached.

The insertion of ϕ-functions, necessary to guarantee the single reaching-definition property, is the dual of the insertion of σ-functions, and it follows Cytron's algorithm [12]. Whereas the insertion of σ-functions requires post-dominance frontiers and tracks uses of variables, the insertion of ϕ-functions

uses dominance frontiers and tracks variable definitions. Iterations between the two boxes in Figure 3(a) happen because the insertion of σ-functions creates new definitions of variables, forcing a new round of placement of ϕ-functions. Additionally, the insertion of ϕ-functions also leads to the insertion of σ-functions, because it creates new uses of variables. Once a fix-point is reached, meaning that the properties stated in Section 2 have been attained, a renaming pass finally converts the program into SSI form.

Converting a program to partial SSI form In order to be in partial SSI form, a variable must meet four properties. Three of these properties were seen in Section 2: pseudo-definition, single reaching-definition and pseudo-use. We call the fourth property the single upward-exposed-conditional. This property, which is less general than Section 2's single upward-exposed-use, is stated as follows:

- *single upward-exposed-conditional*: if v is used at a branch instruction i, then from i it is possible to reach only one use of v without crossing another use.

There exist two main events that may restrict the value range of a variable v: an assignment to v and a conditional branch that tests v. Thus, in order to partially convert a variable v to SSI form we can add σ-functions at the boundaries of basic blocks that end with a conditional branch where v is used. Returning to our first example, variable a is fully converted to SSI form in Figure 1(b), and it is partially converted to SSI form in Figure 1(c). Notice that the single-upward-exposed-conditional property holds in Figure 1(c) because the branch instruction $(\bullet = a)?$ is post-dominated by the use $(a_1, a_2) = \sigma\ a$.

The algorithm that we use to convert a program to partial SSI form is shown in Figure 3(b). This algorithm has lower complexity than Singer's, because the placement of σ-functions is simpler. In order to convert a variable v to partial SSI form, we loop over the uses of v, and for each use that is a conditional instruction, we create a σ-function in the basic block that contains that use. Once all the uses of v have been visited, we proceed to the insertion of ϕ-functions. The placement of ϕ-functions is the same as in Singer's method, but in the partial transformation this phase happens only once.

Figure 4 illustrates these concepts. Figure 4(a) shows the control flow graph of the program used in Figure 1(a). We are interested in partially converting variable a into SSI form. Variable a is used in blocks 1, 3 and 4. Only the first use is a branch, so we insert a σ-function after block 1, as seen in Figure 4(b). This σ-function defines two new instances of variable a, because block 1 has two successors. After σ-functions have been inserted, we move on to the insertion of ϕ-functions. Because we now have two definitions of variable a reaching block 4, we insert a ϕ-function in the beginning of this block, as shown in Figure 4(c). Finally, a renaming step will produce the program in Figure 4(d).

The algorithm in Figure 3(b) might insert more σ-functions than the minimal number necessary to guarantee the single-upward-exposed-conditional property. For instance, we would insert a σ-function after the conditional in Figure 4(a), even if there were no uses of variable a inside block 3. In this case, variable a already has the single upward-exposed-conditional property, but our simple

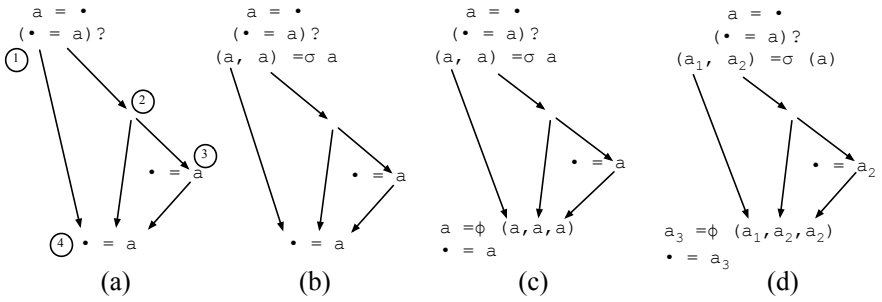

Fig. 4. Partially converting a program for information analysis.

algorithm is unable to see this fact. Tracing an analogy with the SSA conversion algorithm [7, p.7], our method produces what we would call the "maximal" partial-SSI representation. We opted to build this simple representation, instead of the pruned form because the simpler approach is faster [21]. Whereas the latter construction requires an analysis to identify which uses of variables reach branching points, the former simply inserts σ-functions after conditionals.

Complexity Analysis The complexity of converting a single variable to SSI form, using the algorithm in Figure 3(a) is computed as follows. The complexity of a round of insertion of σ-functions, or ϕ-functions, is $O(B^2)$ [12], where B is the number of basic blocks in the source program. However, as empirically demonstrated [21], this algorithm is $O(B)$ in practice. There are, indeed, true $O(B)$ algorithms for the placement of ϕ and σ-functions (see Sreedhar *et al* [22]). The maximum number of alternations between the insertion of ϕ and σ-functions is $O(B)$; therefore, the total complexity of the algorithm is $O(B^3)$.

The partial conversion has lower complexity. Inserting σ-functions is $O(U)$, where U is the number of conditional instructions using v. Notice that a variable tends to be used at most 5 times in a SSA-form assembly program [5], so this complexity is constant in practice. The complexity of inserting ϕ-functions is $O(B^2)$. As in the full conversion, it is $O(B)$ in practice. There is no alternation between the insertion of ϕ and σ-functions; thus, the total complexity of the partial conversion algorithm is $O(U) + O(B^2)$.

Orchestrating the execution of different clients A compiler might perform several passes on the same code, in order to carry out different optimizations. This includes the possibility of different clients of our SSI transformation framework running on the same program. Hence, one of the objectives of our design is to allow clients to execute in sequence, without having to perform redundant work.

Our implementation guarantees that SSI clients running in sequence will never insert redundant σ or ϕ-functions into the source program. That is, let c_1 and c_2 be two SSI clients running in sequence. Lets assume that c_1 causes the

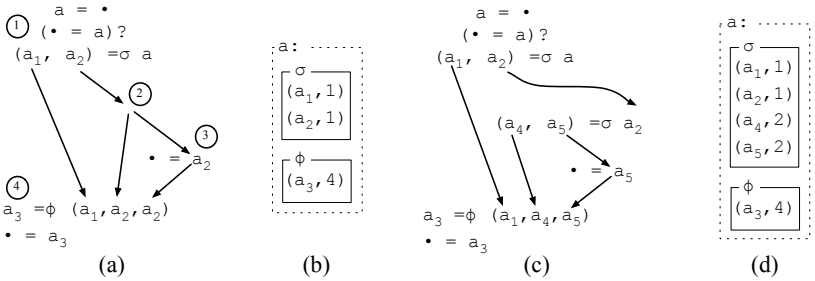

Fig. 5. Preventing clients that run in sequence from performing redundant work.

insertion of σ_1 (or ϕ_1) at program point p_1 to transform a variable v. If c_2 also requests the conversion of v, leading to the insertion of another σ instruction at p_1, then nothing will happen, because our SSI converter knows that instruction σ_1 is already breaking the live range of v. In order to avoid redundancy, our SSI implementation keeps an internal state: we map each variable to a table of pairs. Each pair consists of the identifier of either a σ or a ϕ-function, plus a program point. Figure 5 illustrates these concepts. Figure 5(b) shows the table created for variable a after some client requests the partial conversion of this variable, yielding the program in Figure 5(a). Once a second client requests the full conversion of a, we already know that no σ-function must be inserted at program point 1. However, the insertion of a σ-function at program point 2 would lead to the creation of a ϕ-function at program point 4. Again, we check a's table to avoid inserting a new ϕ-function. Upon discovering the instruction $a_3 = \phi(a_1, a_2, a_2)$, we change the two occurrences of a_2 to a_4 and a_5, as illustrated in Figure 5(c). The new table of variable a is given in Figure 5(d).

5 Experimental results

This section describes experiments that we have performed to validate our SSI framework. Our experiments were conducted on a dual core `Intel Pentium D` of 2.80GHz of clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [17], and it passes all the tests that LLVM does. The LLVM test suite consists of over 1.3 million lines of C code. In this paper we will be showing only the results of compiling SPEC CPU 2000. We will use three different clients of our SSI framework:

1. *Full*: this is the conversion algorithm shown in Figure 3(a), which converts a program to strong SSI form. This client supports the backward analyses described in Section 3.2. As an optimization, before running the full conversion we traverse the source program, identifying the variables used across basic blocks. Only these variables are touched by the converter. This optimization is similar to the "semi-pruned" SSA conversion of Briggs *et al.* [7, Fig.4].

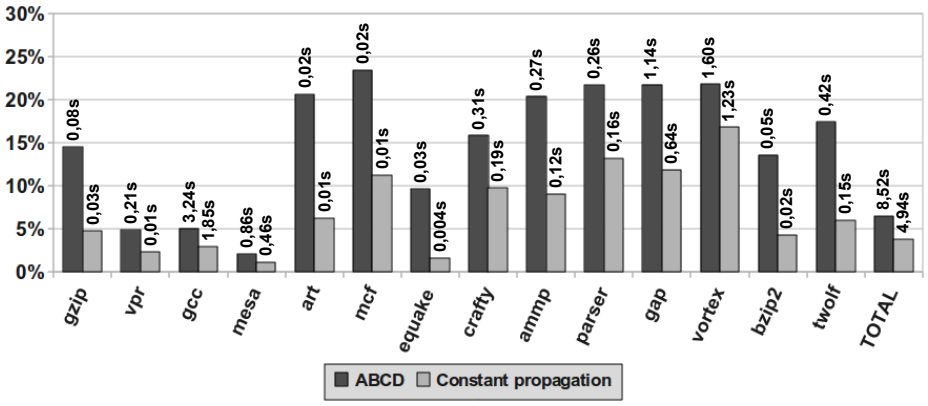

Fig. 6. Execution time of partial compared with full SSI conversion. 100% is the time of doing the full SSI transformation. The shorter the bar, the faster the partial conversion when compared to the full conversion.

2. *ABCD*: this client generalizes the ABCD algorithm for array bound checking elimination [3]. We eliminate conditional branches on numeric inequalities that we can prove redundant, such as the redundant tests in Figures 2(a) and 2(d). Our ABCD client requires that all the numeric variables used in inequalities, e.g, `<`, `=<`, `>=` and `>`, plus the equality test `==`, be converted to SSI form.
3. *CCP*: this client does conditional constant propagation, that is, it replaces the use of variables that have a value range equal to a zero length interval $[c, c]$ by the constant c. As an example, this optimization replaces the use of variable a in line 4 of Figure 1(a) by the constant 0. This client requires that only variables used in equality tests, e.g, `==`, be converted to SSI.

When reporting the time of ABCD or CCP we show the time of running the algorithm in Figure 3(b). The time of performing redundant branch elimination or conditional constant propagation is not shown. Similarly, time reports for the full conversion include only the time to run the algorithm in Figure 3(a).

The chart in Figure 6 compares the execution time of the three SSI clients. The bars are normalized to the running time of the full SSI conversion. On the average, the ABCD client runs in 6.8% and the CCP client runs in 4.1% of the time of the full conversion. The numbers on top of the bars are absolute running times. The partial conversions tends to run faster in clients with sparse control flow graphs, which present fewer conditional branches, and consequently fewer opportunities to restrict the value ranges of variables.

Figure 7 compares the running time of our partial conversion algorithm with the running time of the `opt` tool. This tool is part of the LLVM framework, and it performs target independent code optimizations. `Opt` receives a LLVM bytecode file, optimizes it, and outputs the modified file, still in LLVM bytecode

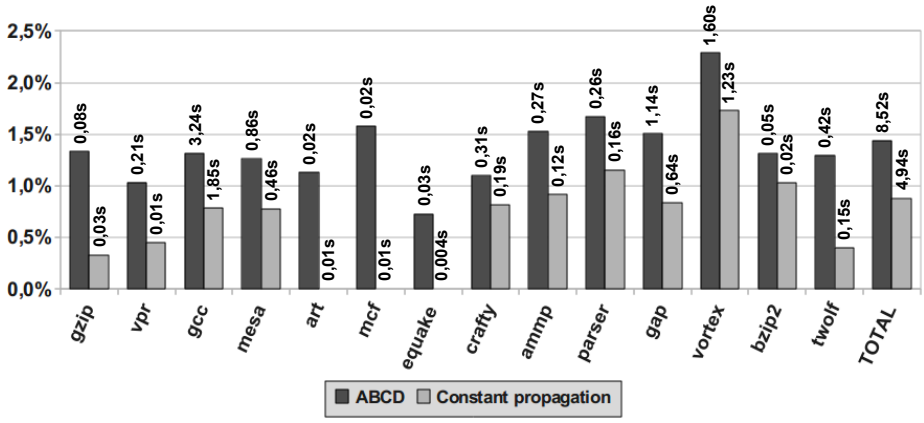

Fig. 7. Execution time of partial SSI conversion compared to the total time taken by machine independent LLVM optimization passes (`opt`). 100% is the total time taken by `opt`. The shorter the bar, the faster the partial conversion.

format. The SSI clients are `opt` passes. The bars are normalized to the `opt` time, which consists on the time taken by machine independent optimizations plus the time taken by one of the SSI clients, e.g, ABCD or CCP. Among the optimizations performed by `opt` we list partial redundancy elimination, unreachable basic block elimination and loop invariant code motion. The ABCD client takes 1.48% of `opt`'s time, and the CCP client takes 0.9%. To emphasize the speed of these passes, we notice that the bars do not include the time of doing machine dependent optimizations such as register allocation.

Figure 8 compares the number of σ and ϕ-functions inserted by the SSI clients. The bars are the sum of these instructions, as inserted by each partial conversion, divided by the number of σ and ϕ-functions inserted by the full SSI transformation. The numbers on top of the bars are the absolute quantity of σ and ϕ-functions inserted. The CCP client created 67.3K σ-functions, and 28.4K ϕ-functions. The ABCD client created 98.8K σ-functions, and 42.0K ϕ-functions. The full conversion inserted 697.6K σ-functions, and 220.6K σ-functions.

The chart in Figure 9 shows the number of σ and ϕ-functions that each SSI client inserts per variable. The figure emphasizes the difference between the partial conversion required by the two information analyses and the full SSI transformation. On the average, for each variable whose conversion is requested by either the ABCD or the CCP client, we will create 0.6 ϕ-functions, and 1.3 σ-functions. On the other hand, the full SSI conversion will insert 6.1 σ-functions and 2.7 ϕ-functions per variable.

Figure 10 shows the number of variables that have been transformed by each client. We notice that in two benchmarks, `gcc` and `vortex`, the ABCD client has transformed more variables than the full client. This fact happens because the full client transforms only variables that are alive across different basic blocks.

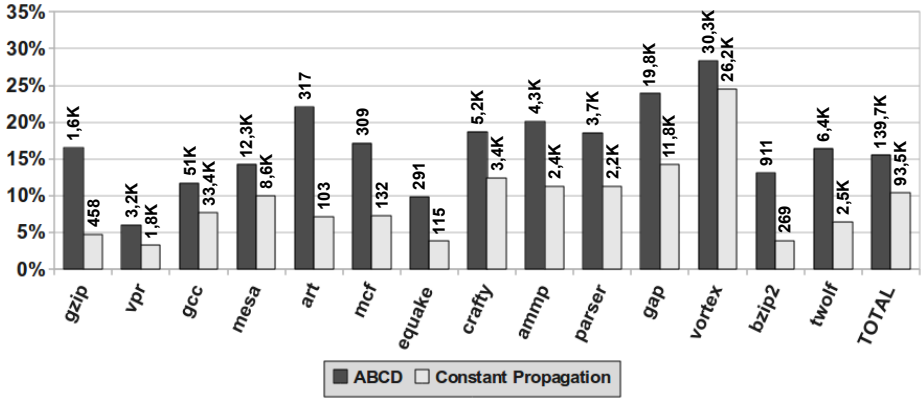

Fig. 8. Number of ϕ and σ-functions produced by the partial SSI conversion compared with the full conversion. Values on top of bars denote absolute number of instructions. 100% is the number of instructions inserted by the full conversion.

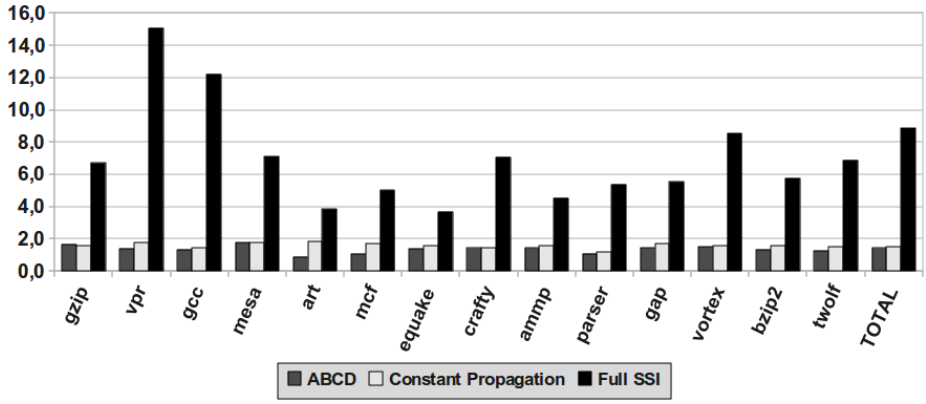

Fig. 9. Average number of ϕ and σ-functions produced per variable.

The ABCD and CCP clients, on the other hand, use the partial conversion algorithm from Figure 3(b), which converts variables used in conditionals, even when those variables are not alive outside the basic block where they are used.

The chart in Figure 11 compares the number of σ and ϕ-functions that we save by running different SSI clients in sequence. We compute the bars as follows. Let c_1 and c_2 be two SSI clients, such that c_1 inserts n_1 special instructions (ϕ or σ functions) into the source program, and c_2 inserts n_2. Let $n_{1,2}$ be the number of special instructions generated when both clients run in sequence. The bars represent the formula $1 - (n_{1,2}/(n_1 + n_2))$. This measure denotes the number of repeated instructions that are inserted by both clients running independently,

	gzip	vpr	gcc	mesa	art	mcf	equake	crafty	ammp	parser	gap	vortex	bzip2	twolf	**Total**
ABCD	968	2K	38K	7K	361	292	211	4K	3K	3K	14K	20K	686	5K	**100K**
CCP	296	1K	24K	5K	56	78	74	2K	2K	2K	7K	17K	169	2K	**62K**
Full	1K	4K	36K	12K	376	360	807	4K	5K	4K	15K	13K	1K	6K	**101K**

Fig. 10. Number of variables converted to SSI.

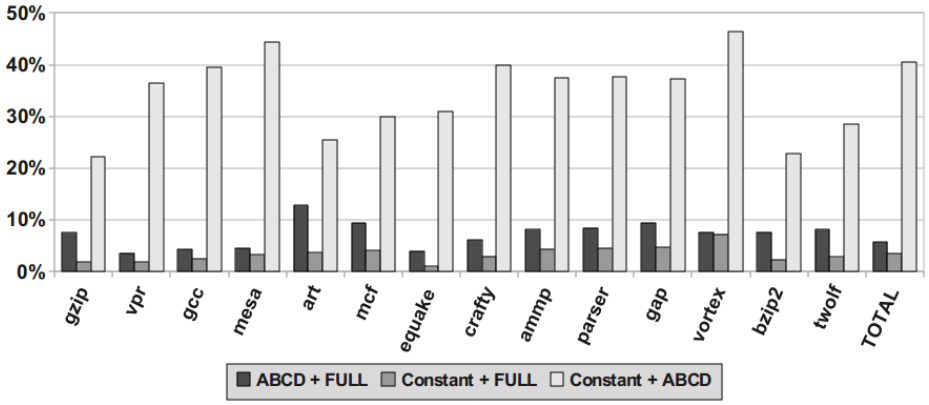

Fig. 11. Percentage of σ and ϕ-functions saved by running clients in sequence.

and that are saved when these clients run in sequence. Our framework avoids the insertion of redundant instructions by keeping a record of variables that each client transforms, as described in Section 4.

6 Conclusion

This paper has presented the SSI conversion framework that we have deployed on the LLVM compiler. Our implementation has been reviewed by members of the LLVM community, and it is now available in the default LLVM distribution. We are currently working on information analyses for Java. Our intention is to use the SSI representation to remove redundant `instanceof` checks, for instance.

References

1. Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.
2. Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition) (Java Series)*. Addison-Wesley Professional, 2005.
3. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

4. Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI properties revisited. Technical Report 00404236, LIP Research Report, 2009.
5. Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoit Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO*, pages 35–44, 2008.
6. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.
7. Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.
8. Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *TCAD*, 25(5):772–779, 2006.
9. Philip Brisk and Majid Sarrafzadeh. Interference graphs for procedures in static single information form are interval graphs. In *SCOPES*, pages 101–110. ACM Press, 2007.
10. Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
11. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.
12. 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. *TOPLAS*, 13(4):451–490, 1991.
13. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.
14. W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250, 1977.
15. Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *CC*, pages 1–16. Springer-Verlag, 2003.
16. Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89, 1993.
17. Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
18. Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.
19. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.
20. Jeremy Singer. SSI extends SSA. In *PACT (Work in Progress Session)*, pages XX–YY, 2003.
21. Jeremy Singer. *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge, 2006.
22. Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *POPL*, pages 62–73. ACM, 1995.
23. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.
24. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.
25. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.