Efficient SSI Conversion

André Luiz C. Tavares[1], **Fernando M. Q. Pereira**[1],
Mariza A. S. Bigonha[1], **Roberto S. Bigonha**[1]

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{andrelct,fpereira,mariza,bigonha}@dcc.ufmg.br

Abstract. *Static Single Information form (SSI) 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; but, only a few SSI clients, that is, optimizations that use SSI, require a full conversion. This paper describes the SSI framework we have implemented for the LLVM compiler, and that is now part of this compiler's standard distribution. In our design, optimizing passes inform the compiler a list of variables of interest, which are then transformed to present, fully or partially, the SSI properties. It is provided 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

Static Single Information (SSI) form is a program representation introduced by Scott Ananian [Ananian 1999]. This 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 [Bodik et al. 2000], bitwidth analysis [Stephenson et al. 2000], flow sensitive range interval analysis [Su and Wagner 2005], conditional constant propagation [Ananian 1999, Wegman and Zadeck 1991], partial redundancy elimination [Johnson and Pingali 1993], faster liveness analysis [Boissinot et al. 2009, Singer 2006], and busy expression elimination [Singer 2003].

Although the SSI representation suits the needs of many different compilation passes – henceforth called clients, the majority of these clients require only a subset of the SSI properties instead of a full conversion. This observation is important, because converting a program to full SSI form is a time consuming endeavor. For instance, Bodik *et al.* [Bodik et al. 2000]'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 [Ananian 1999]

and Singer [Singer 2006], 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.* [Johnson and Mycroft 2003] 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, which saves compilation time and space in three different ways. 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 [Ananian 1999, Boissinot et al. 2009, Singer 2006]. 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 [Ananian 1999, Bodik et al. 2000, Singer 2006, Stephenson et al. 2000, Su and Wagner 2005, Wegman and Zadeck 1991], 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 [Lattner and Adve 2004] (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.* [Bodik et al. 2000], and a sparse conditional constant propagation (CCP) algorithm, similar to the one described by Singer [Singer 2006, 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 representation. In Section 3 we analyze the properties that many compiler optimizations previously described in the literature require from a program representation, and we show how different subsets of SSI suit these needs. In Section 4 we discuss our approach to build the SSI representation on demand. Section 5 validates our work with a series of experiments, and Section 6 concludes this paper.

2. Background on SSI

The term Static Single Information form seems to have been coined by Scott Ananian in his master thesis [Ananian 1999]; however, program representations with similar properties have been described before [Johnson and Pingali 1993]. For instance, the SSU-form

[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`

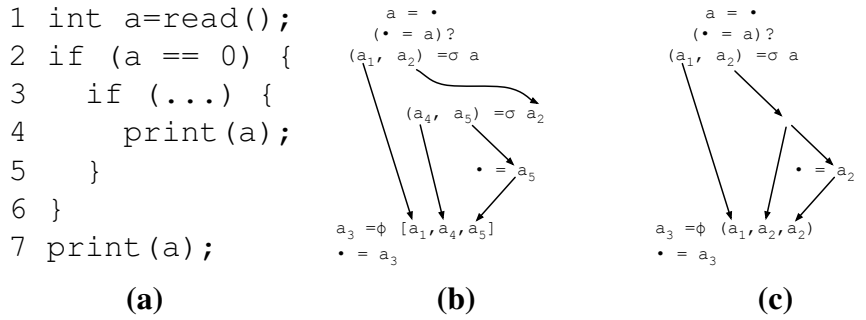

Figure 1. Example of the use of SSI on information analysis.

(Single Static Use), described by Plevyak in his Ph.D dissertation, [Plevyak 1996], seems to be equivalent to SSI, although we cannot verify this claim due to the lack of a formal specification of SSU. Boissinot *et al.* [Boissinot et al. 2009] distinguish two main flavors of the SSI form, which are not equivalent: *strong*, introduced by Ananian [Ananian 1999] and *weak*, described by Singer [Singer 2006]. 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 Boissinot *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) [Cytron et al. 1991] to join the live ranges of variables. Any SSI form program is a SSA form program. For instance, the assignment, $v = \phi(v_1, \ldots, v_n)$, at the beginning of a basic block B, 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, $(v_1, \ldots, v_n) = \sigma\ v$, at the end of a basic block B, assigns 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 [Ananian 1999] pessimistic algorithm, and Singer's [Singer 2006] optimistic approach. We use the latter, as it subsumes the former.

There exists an interesting relationship between the live range of program variables and graphs. Chaitin *et al.* [Chaitin et al. 1981] have shown the intersection graph of the live ranges of a general program can be any type of graph. In 2005, researchers have shown that the intersection graphs produced from programs in SSA form are chordal [Bouchez 2005, Pereira and Palsberg 2005]. Recently, Boissinot *et al.* [Boissinot et al. 2009] showed that the interference graphs of programs in SSI form are interval graphs, a subset of the family of chordal graphs.

3. Examples of SSI Clients

This section shows 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 [Arnold et al. 2005]. 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 3-4 is implicitly performed, at runtime, by the Ruby interpreter; but, 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 [Bodik et al. 2000] and the sparse conditional constant propagation method of Wegman and Zadeck [Wegman and Zadeck 1991]. Ananian describes a long list of information analyses when introducing the SSI representation [Ananian 1999]. 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 detect security vulnerabilities in programs [Rimsa et al. 2010], and 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`. Examples of range analyses are the bitwidth inference engine of Stephenson *et al.* [Stephenson et al. 2000], the range propagation algorithm of Su and Wagner [Su and Wagner 2005], and the range analyzis used by Patterson to predict the outcome of branches [Patterson 1995].

```
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 + i > MAX_INT)
4      change sum to BigInt
5    sum += i
6  end
```

(d)

Figure 2. Examples of defensive programming idioms.

Although well known in the literature, these analysis and heuristics are described using different program representations, whereas they all can be elegantly modeled as constraint systems built on top of some *subset* of SSI form. In this sense, different clients have different needs, and not every variable in the source program needs to be transformed to meet the SSI properties. For instance:

- ABCD algorithm that removes redundant array bound checks [Bodik et al. 2000], 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 ABCD version we implemented requires any variable used in branches to be in SSI form. So, the algorithm is used as a more general *redundant test elimination*, that allows, for instance, to remove the test in Line 3 in Figure 2(d).

In general, information analyses require that only variables used inside conditionals have the SSI properties. Furthermore, these variables do not have to show the SSI properties in all the program points where they are alive. Consider, for example, the program in Figure 1(a). It is possible to infer that the value of variable a, inside the innermost if statement, is always 0. An SSI based constant propagation analysis would produce the program in Figure 1(b), and would derive the constraints $\langle a_2 = 0 \rangle$ and $\langle a_4 = a_2 \rangle$. But the second constraint only exists because variable a_4 was created to guarantee the SSI property that no program point is reached by two different uses of the same variable. Thus, in order to avoid redundancies, we allow clients to specify the representation in Figure 1(c), which yields only the constraint $\langle a_2 = 0 \rangle$.

3.2. Backward Analyses

In addition to being useful for information analyses, the SSI representation also facilitates sparse backward analyses. Singer [Singer 2003] 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 [Johnson and Pingali 1993], 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, busy expressions associated to variable v are the busy expressions at the definition point of v. Similarly, available expressions associated to v are the expressions available at the program point where v is last used. The SSI form allows us to perform these analyses non-iteratively [Singer 2003]. As another example, Boissinot *et al.* [Boissinot et al. 2009] 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 SSI representation. That is, every program variable must present the SSI properties discussed in Section 2. We show that the same infrastructure that supports information analyses also supports the backward analyses described in this section.

4. Building Partial SSI

We convert a program into SSI form on demand. This means that a client gives to our transformation pass 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 partial and full conversion is the amount of live range splitting required.

4.1. 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. To perform the full conversion, we use the algorithm designed by Singer [Singer 2006, p.46]. This method, shown in Figure 3(a), combines Cytron *et al.*'s algorithm to insert ϕ-functions [Cytron et al. 1991], and Singer's algorithm to insert σ-functions [Singer 2006].

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 algo-

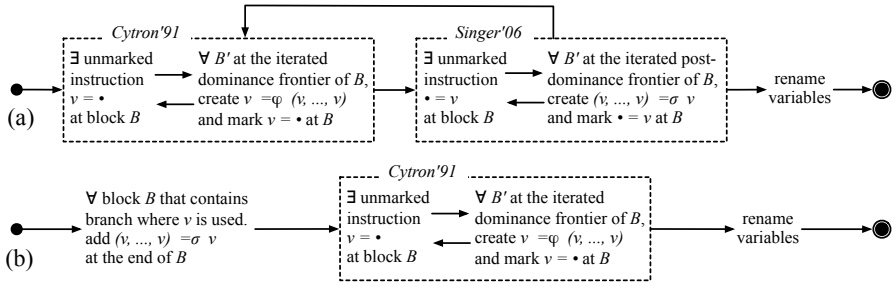

Figure 3. (a) Singer optimistic algorithm to convert a program into SSI form (b) Our algorithm to produce partial SSI form.

rithm [Cytron et al. 1991]. Whereas the insertion of σ's requires post-dominance frontiers and tracks uses of variables, the insertion of ϕ's uses dominance frontiers and tracks variable definitions. Iterations between the two boxes in Figure 3(a) happen because the insertion of σ-functions create new definitions of variables, and force 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 converts the program into SSI form.

4.2. Converting a Program to Partial SSI Form

The information analyses described in Section 3.1 do not require that variables be fully converted to 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. Thus, in Line 3 of our example program, this range is $[0, 0]$. 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.

In order to be in partial SSI form, a variable must meet four properties. Three of them 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 passing across another use.

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). On the other hand, the same variable is only partially converted to SSI form in Figure 1(c). 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

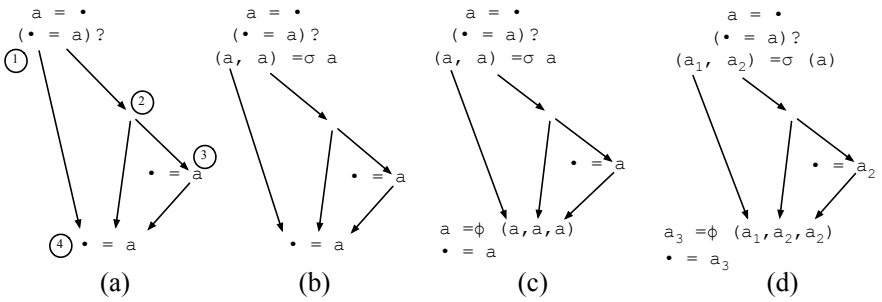

Figure 4. Partially converting a program for information analysis. Full is given in Figure 1(b).

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, (see 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. Since 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 algorithm is unable to see this fact. Tracing an analogy with the SSA conversion algorithm, our method produces what we would call the "maximal" [Briggs et al. 1998, p.7] partial-SSI representation. We opted to build this simple representation, instead of the pruned form because the simpler approach is faster [Singer 2006]. Whereas the latter construction requires an analysis to identify which uses of variables reach branching points, the former simply inserts σ-functions after conditionals.

4.3. 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 $O(B^2)$ [Cytron et al. 1991], where B is the number of basic blocks in the source program. But as empirically demonstrated [Singer 2006], 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* [Sreedhar and Gao 1995]). The maximum number of alternations between the insertion of ϕ and σ-functions is $O(B)$; so, the total complexity of the algorithm is $O(B^3)$.

The partial conversion has lower complexity. Inserting σ-functions is $O(U)$,

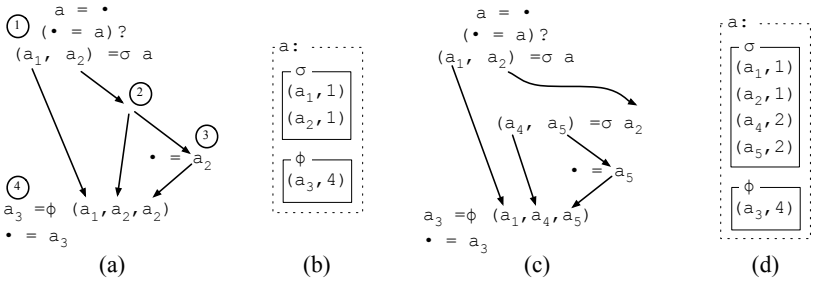

Figure 5. Preventing clients that run in sequence from performing redundant work.

where U is the number of conditional instructions using v. 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 $O(U) + O(B^2)$.

4.4. 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 separate 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 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. To avoid redundancy, our SSI implementation keeps an internal state: it maps 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 shows 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. But 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 seen in Figure 5(c). Figure 5(d) shows the new table of variable a. Notice that this data structure is not essential to avoid inserting redundant σ and ϕ instructions; we could avoid it by looking at the current state of the intermediate representation. But it speeds up redundancy checks: if not for the data structure we would have to go through the parameters of ϕ and σ functions looking for occurrences of a variable before changing it.

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

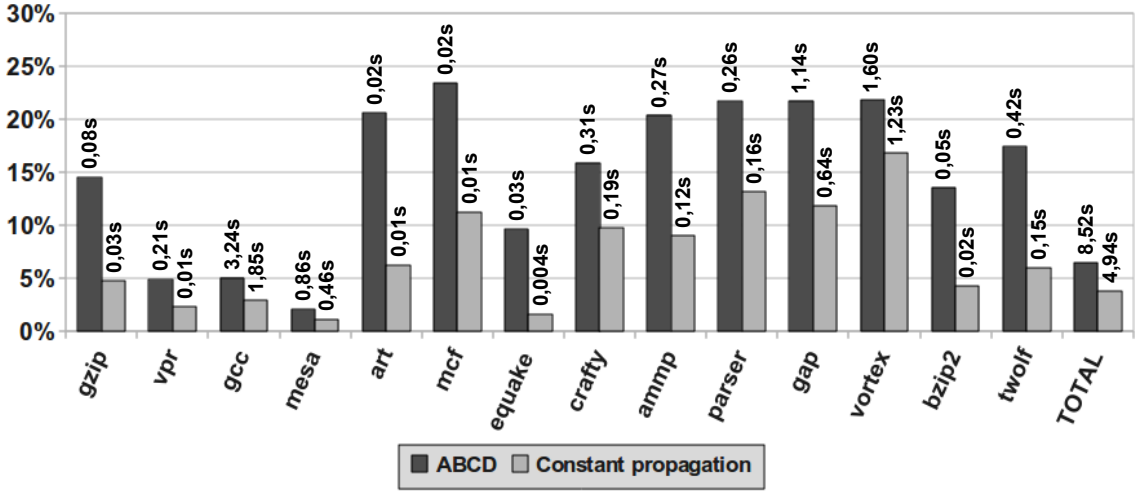

Figure 6. Execution time of partial compared with full SSI conversion. 100% is the time of doing the full SSI transformation. The shorter the bar, the faster the partial conversion when compared to the full conversion.

clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [Lattner and Adve 2004], 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*: converts a program to strong SSI form with the algorithm of Figure 3(a).
2. *ABCD*: generalizes ABCD algorithm for array bound checking elimination [Bodik et al. 2000]. We eliminate conditional branches on numeric inequalities that can prove redundant, such as the redundant tests in Figures 2(a) and 2(d).
3. *CCP*: 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 therefore 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 format. The SSI clients are `opt`

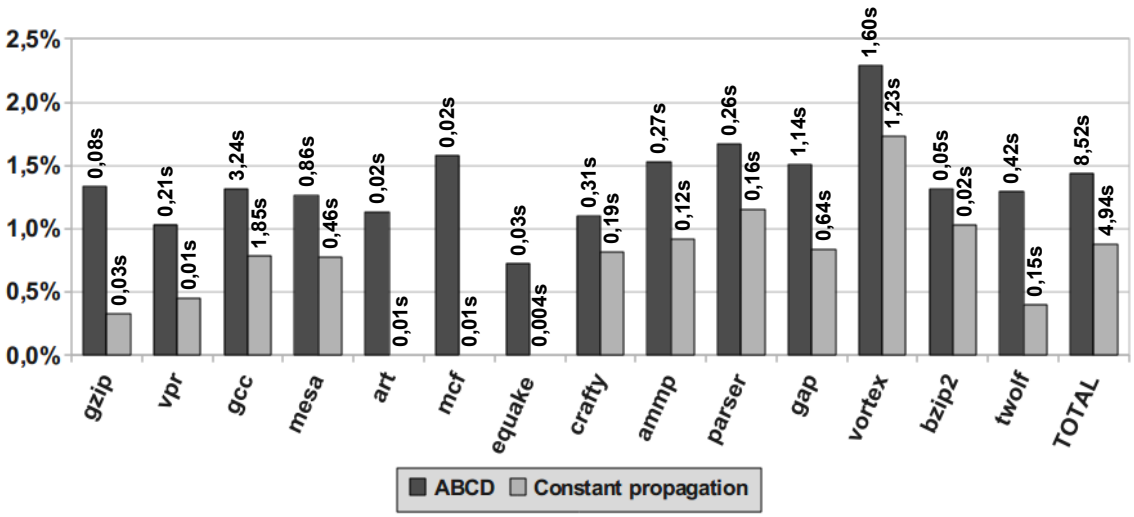

Figure 7. Execution time of partial SSI conversion compared to the total time taken by machine independent LLVM optimization passes (`opt`). 100% is the total time taken by `opt`. The shorter the bar, the faster the partial conversion.

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. There is an apparent mismatch between the number of instructions inserted and the time to do it. That is, in Figure 8 we see that about 10% to 15% of the nodes are inserted for ABCD and CCP when compared to full SSI. But in Figure 6 we see that this only takes 4% to 6% of the computation time. This fact happens because full SSI requires iterations between the insertion of σ and ϕ nodes, whereas the partial construction does not.

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. In two benchmarks, `gcc` and `vortex`, ABCD client has transformed more vari-

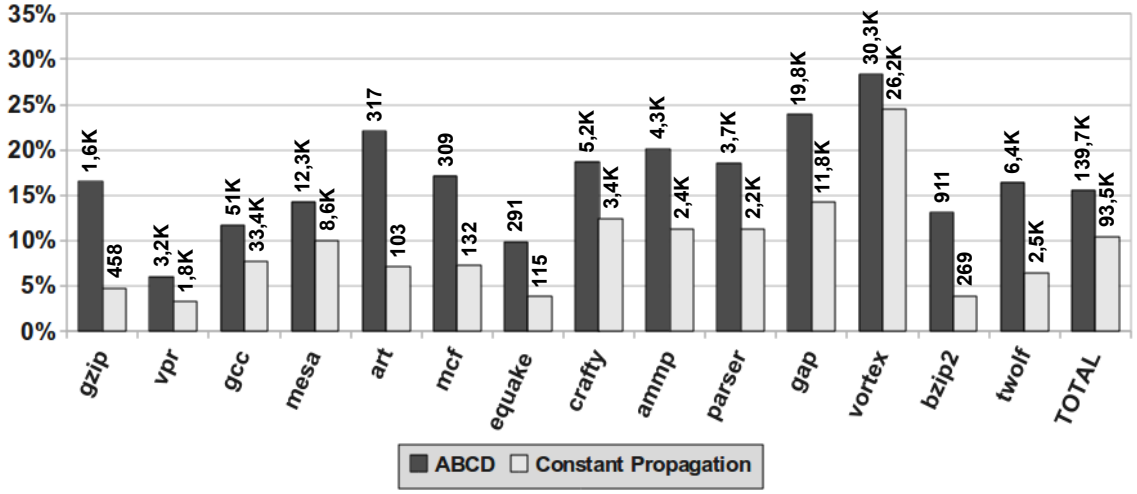

Figure 8. Number of ϕ and σ-functions produced by partial SSI conversion compared with full conversion. Values on top of bars denote absolute number of instructions. 100% is the number of instructions inserted by the full conversion.

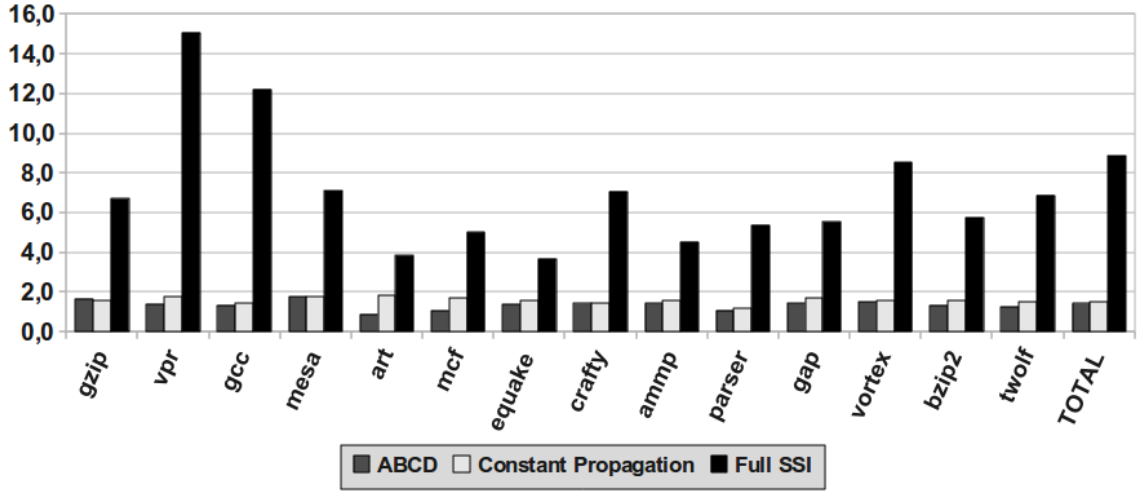

Figure 9. Average number of ϕ and σ-functions produced per variable.

	gzip	vpr	gcc	mes	art	mcf	eqk	crfty	ammp	par	gap	vor	bzp2	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**

Figure 10. Number of variables converted to SSI. We use shorter names.

ables than the full client. This fact happens because the full client transforms only variables that are alive across different basic blocks. 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. Notice that, in this case, we will not have any use of the variable after the conditional, and no σ or ϕ-functions will be inserted by the algorithm of Figure 3.

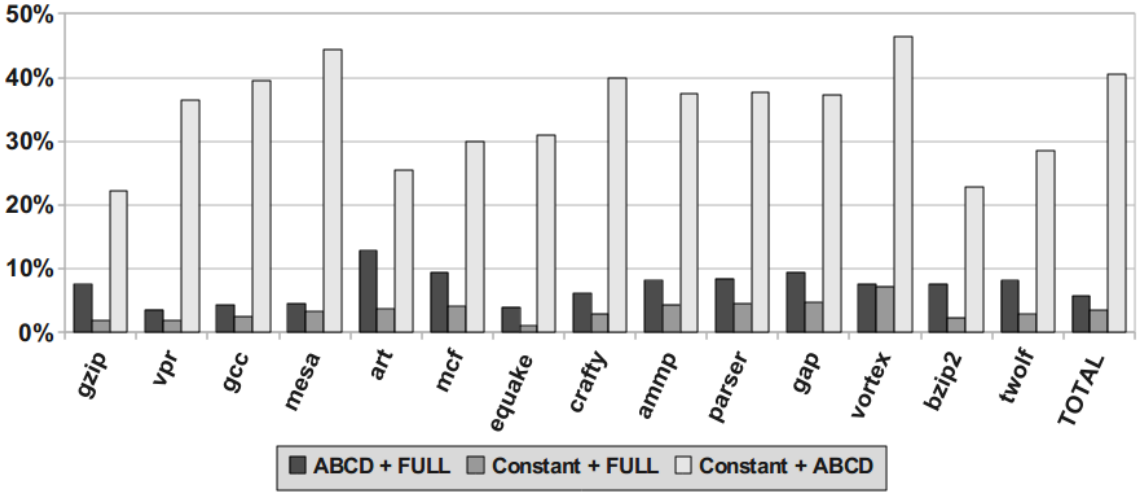

Figure 11. Percentage of σ and ϕ-functions saved by running clients in sequence.

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, 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 design and implementation of a SSI conversion framework, which is useful to several analysis and optimizations present in compiler back-ends. Our implementation differs from previous works because it allows the client to specify which variables should be converted into SSI. Furthermore, it allows a variable to be converted into SSI partially, that is, only at those program points where SSI properties are required. These two capacities of our framework allows it to be one order of magnitude faster than traditional approaches to SSI generation. Our implementation has been deployed on the LLVM compiler, and now is part of its official distribution.

Acknowledgement

This research has been supported by FAPEMIG, Edital 11/2009.

References

Ananian, S. (1999). The static single information form. Master's thesis, MIT.

Arnold, K., Gosling, J., and Holmes, D. (2005). *Java(TM) Programming Language, The (4th Edition) (Java Series)*. Addison-Wesley Professional.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Boissinot, B., Brisk, P., Darte, A., and Rastello, F. (2009). SSI properties revisited. Technical Report 00404236, LIP Research Report.

Bouchez, F. (2005). Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon.

Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, 6:47–57.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Johnson, N. and Mycroft, A. (2003). Combined code motion and register allocation using the value state dependence graph. In *CC*, pages 1–16. Springer-Verlag.

Johnson, R. and Pingali, K. (1993). Dependence-based program analysis. In *PLDI*, pages 78–89. ACM.

Lattner, C. and Adve, V. S. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM.

Pereira, F. M. Q. and Palsberg, J. (2005). Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer.

Plevyak, J. B. (1996). *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign.

Rimsa, A. A., D'Amorim, M., and Pereira, F. M. Q. (2010). Efficient static checker for tainted variable attacks. In *SBLP*. SBC.

Singer, J. (2003). SSI extends SSA. In *PACT (Work in Progress Session)*, pages XX–YY.

Singer, J. (2006). *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge.

Sreedhar, V. C. and Gao, G. R. (1995). A linear time algorithm for placing ϕ-nodes. In *POPL*, pages 62–73. ACM.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138.

Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *TOPLAS*, 13(2).