SSA Elimination after Register Allocation

Fernando Magno Quintão Pereira and Jens Palsberg

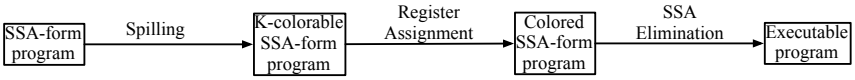
October 22, 2008

Abstract

The SSA-form uses a notational abstractions called ϕ-functions. These instructions have no analogous in actual machine instruction sets, and they must be replaced by ordinary instructions at some point of the compilation path. This process is called *SSA elimination*. Compilers usually performs SSA elimination before register allocation. But the order could as well be the opposite: our puzzle based register allocator performs SSA elimination after register allocation. SSA elimination before register allocation is straightforward and standard, while the state-of-the-art approaches to SSA elimination after register allocation have several shortcomings. In this report we present *spill-free SSA elimination*, a simple and efficient algorithm for SSA elimination after register allocation that avoids increasing the number of spilled variables. We also present three optimizations that enhance the quality of the code produced by the core algorithm. Our experiments show that spill-free SSA elimination takes less than five percent of the total compilation time of a JIT compiler. Our optimizations reduce the number of memory accesses by more than 9% and improve the program execution time by more than 1.8%.

1 Introduction

One of the main advantages of SSA based register allocation is the separation of phases between spilling and register assignment. The two-phase approach works because the number of registers needed for a program in SSA-form equals the maximum of the number of registers needed at any given program point. Thus spilling reduces to the problem of ensuring that for each program point, the needed number of registers is no more than the total number of registers. The register assignment phase can then proceed without additional spills. The next figure illustrates the phases of SSA-based register allocation.



SSA elimination *before* register allocation is easier than *after* register allocation. The reason is that after register allocation when some variables have been spilled to memory, SSA elimination may need to copy data from one memory location to another. The need for such copies is a problem for many computer architectures, including x86, that do not provide memory-to-memory copy or swap instructions. The problem is that at the point where it is necessary to transfer data from one memory location to another, all the registers may be in use! In that case, no register is available as a temporary location for performing a two-instruction sequence of a load followed by a store. One solution would be to permanently reserve a register to implement memory-to-memory transfers. We have evaluated that solution by reducing the number of available x86 integer registers from seven to six, and we observed an increase of 5.2% in the lines of spill code (load and store instructions) that LLVM [12] inserts in SPEC CPU 2000.

Brisk [4, Ch.13] has presented a flexible solution that spills a variable on demand during SSA elimination, uses the newly vacant register to implement memory transfers, and later reloads the spilled variable when a register is available. We are unaware of any implementation of Brisk's approach, but have gauged its potential quality by counting the minimal number of basic blocks where spilling would have to happen

during SSA elimination in LLVM, independent on the assignment of physical locations to variables. For x86, such a basic block contains thirteen or more ϕ-functions. We found that for SPEC CPU 2000, memory-to-memory transfers are required for all benchmarks except `181.mcf` - the smallest program in the set. We also found that the lines of spill code must increase by at least 0.2% for SPEC CPU 2000, and we speculate that an implementation of Brisk's algorithm would reveal a substantially higher number. In our view, the main problem with Brisk's approach is that its second spilling phase substantially complicates the design of a register allocator.

This report describes an algorithm that improves on these two previous techniques. We will present *spill-free SSA elimination*, a simple and efficient algorithm for SSA elimination after register allocation. Spill-free SSA elimination never needs an extra register, entirely eliminates the need for memory-to-memory transfers, and avoids increasing the number of spilled variables. The next figure summarizes the three approaches to SSA elimination.

	Accommodates optimal register assignment	Avoids spilling during SSA elimination
Spare register	No	Yes
On-demand spilling [4]	Yes	No
Spill-free SSA elimination	Yes	Yes

The starting point for our approach to SSA-based register allocation is *Conventional SSA (CSSA)-form* [18] rather than the SSA form from the original paper [7] (and text books [2]). CSSA form ensures that variables in the same ϕ-function do not interfere. We show how CSSA-form simplifies the task of replacing ϕ-functions with copy or swap instructions. We also assume that the CSSA-form program contains no *critical edges*. A critical edge is a control-flow edge from a basic block with multiple successors to a basic block with multiple predecessors. Algorithms for removing critical edges are standard [2].

This report also discusses three optimizations that are implemented on top of the core SSA elimination algorithm. We have implemented our SSA elimination framework in the puzzle-based register allocator introduced by Pereira and Palsberg [13]. We convert the source program to CSSA-form before the spilling phase. Our experiments show that our approach to SSA elimination takes less than five percent of the total compilation time of LLVM. Our optimizations reduce the number of memory accesses by more than 9% and improve the program execution time by more than 1.8%.

Our SSA elimination framework works for any SSA-based register allocator such as [10], but the implementation of ϕ-functions in SSA-based register allocators is not the only use of parallel copies in register allocation. The framework described in this report can also be used to insert the fixing code required by register allocators that follow the bin-packing model. Koes *et al.* [11], Traub *et al.* [19], Sarkar *et al.* [17] and Pereira *et al.* [13] are examples of such allocators. Bin-packing allocators allow variables to reside in different registers at different program points. A variable may move between registers due to two main factors: to avoid interferences with pre-colored registers and due to high register pressure. The price of this flexibility is the necessity of inserting fixing code at basic block boundaries. The insertion of fixing code follows the same principles that rule the implementation of ϕ-functions in SSA-based register allocators.

2 Example

We now present an example that illustrates the main difficulty of doing SSA elimination after register allocation. Figure 1 (a) contains a program that continually reads values from the input and prints these values. We built the loop using a somehow artificial arrangement of the variables a, b and t in order to show how compiler optimizations might change the SSA representation of a program in a way that complicates the elimination of ϕ-functions. Figure 1 (b) shows the control flow graph of the example program, this time converted to SSA form. The program in Figure 1 (b) presents an interesting property: variables in the same ϕ-function, such as a, a_1 and a_2 never interfere. Programs that have this property are said to be in *Conventional Static Single Assignment* (CSSA)-form; this representation is formally defined in Section 3.

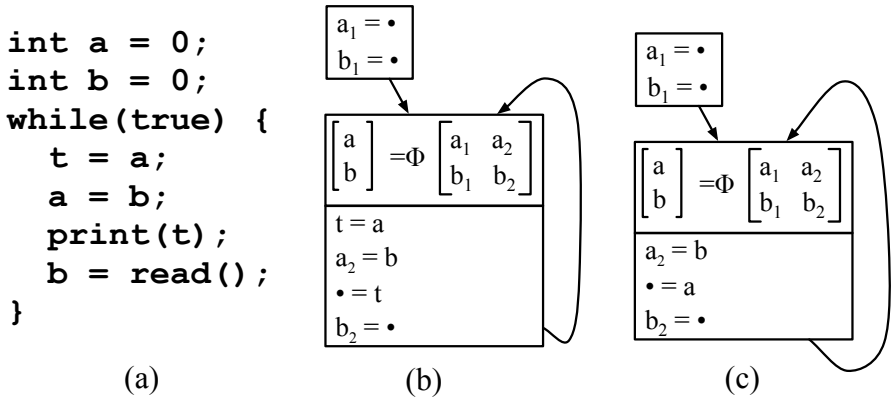


Figure 1: (a) Example program in high level language. (b) Control flow of program converted to SSA-form. (c) Program after constant propagation.

SSA elimination is very simple for programs in conventional SSA-form, as we will show in the remainder of this report, but not every SSA-form program has the conventional property. The original SSA construction algorithm proposed by Cytron *et al.* [7] always builds CSSA-form programs; however, compiler optimizations might break the conventional representation. Figure 1 (c) shows the same program after it underwent a pass of copy propagation. This optimization replaced the use of variable t with a use of variable a, and removed the copy $t = a$. After this optimization, our example program is no longer in CSSA-form, because the variables a and a_2, which are part of the same ϕ-function, interfere.

We will be performing SSA elimination after register allocation. This means that each of our program variables will be bound to a physical location that can be either a machine register or a memory address. We call a program with such bindings a *colored program*. Figure 2 (a) shows a possible colored representation of our example program, assuming a target architecture with only one register r.

As we will see in Section 3, each ϕ-matrix encodes one parallel copy per column. Thus, in order to perform SSA elimination on colored programs we must implement the parallel copies between physical locations. Figure 2 (b) shows the two parallel copies that we must implement in our running example: if control reaches block B_2 coming from block B_1, then the parallel copy $(r, m) := (r, m)$, which is a no-op, must be implemented, otherwise the parallel copy $(r, m) := (r, m_2)$ must be implemented. SSA elimination algorithms normally replace these parallel copies by inserting sequential instructions in the program points where the parallel copies are defined. Notice that this is the most natural approach to SSA elimination; however, the replacement code could be inserted anywhere inside the source program, as long as it maintains the program's semantics.

Figure 2 (c) shows our example program after SSA elimination with on-demand spilling. Notice that one of the parallel copies has been replaced with four instructions that implement a copy from m_2 to m. The need for that copy happens at a program point where the only register r is occupied by b_2. So we must first spill r to m_b, then we can copy from m_2 to m via the register r, and finally we can load m_b back into r.

Now we go on to illustrate that spill-free SSA elimination can do better. Figure 3 (a) shows the same program as in Figure 2 (a), but this time in CSSA-form. To convert the source program into CSSA-form we had to split the live range of variable a_2; this was done by inserting a new copy instruction $a_3 = a_2$, followed by renaming uses of a_2 past the new copy. Figure 3(b) shows the program after spilling and register assignment, and Figure 3(c) shows the program after spill-free SSA elimination. Notice how, in Figure 3(b), CSSA makes a difference by requiring the extra instruction that copies from a_2 to a_3. We now do register allocation and assign each of a, a_1, and a_3 the same memory location m because those variables do not interfere. In Figure 3(b), the value of a_2 arrives in memory location m_2, and is then copied to memory

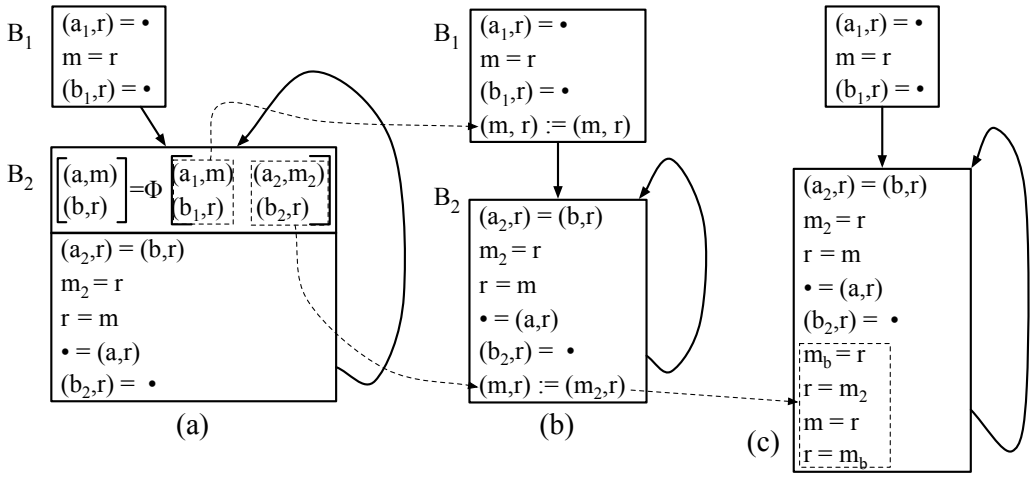


Figure 2: (a) A possible colored representation of the example program. (b) SSA elimination seen as the implementation of parallel copies. (c) SSA elimination with on-demand spilling.

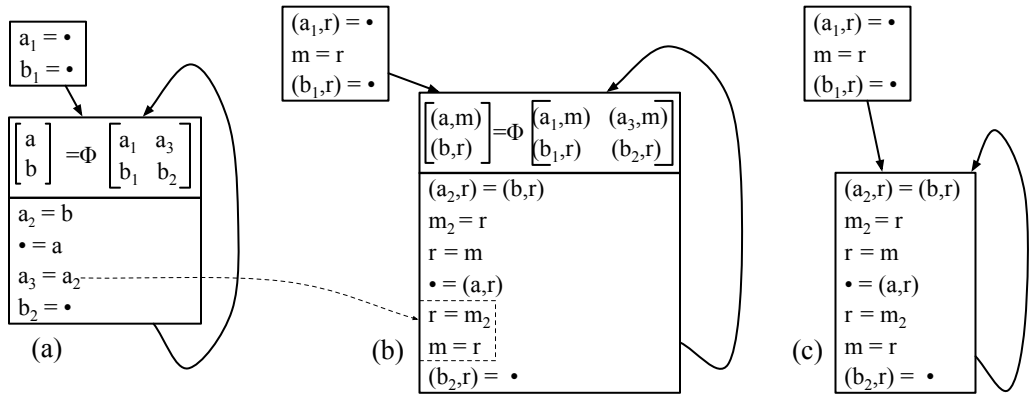


Figure 3: SSA-based register allocation and spill-free SSA elimination.

location m via the register r. The point of the copy is to let both elements of the first row of the ϕ-matrix be represented in m, just like both elements of the second row of the ϕ-matrix are represented in r. We finally arrive at Figure 3(c) without any further spills.

3 Our SSA Elimination Framework

We now show that for programs in CSSA-form, the problem of replacing each ϕ-function with copy and swap instructions is significantly simpler than for programs in SSA-form (Theorem 2). Along the way, we will define all the concepts and notations that we use.

ϕ-functions SSA form uses ϕ-functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of ϕ-functions using the matrix notation introduced by

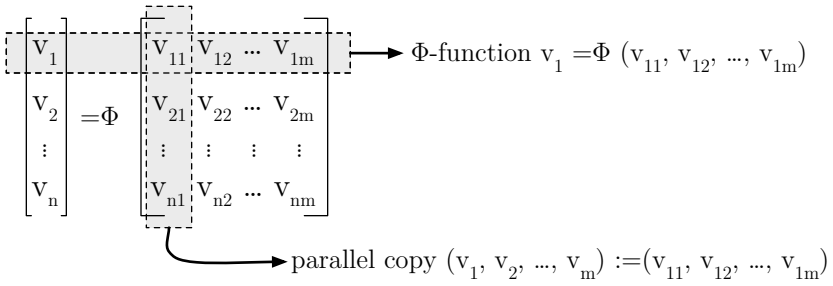


Figure 4: The ϕ-matrix.

Hack et al. [9]. An equation such as $V = \phi M$, where V is a n-dimensional vector, and M is a $n \times m$ matrix, contains n ϕ-functions and m *parallel copies*, as outlined in Figure 4. Columns in the matrix correspond to control flow paths. The ϕ symbol works as a multiplexer. It will assign to each element v_i of V an element v_{ij} of M, where j is determined by the actual path taken during the program's execution. The semantics of ϕ-functions have been nicely described in [1]. The parameters of a ϕ-function are evaluated simultaneously, at the beginning of the basic block where the ϕ-function is defined. Thus, a ϕ-equation $V = \phi M$, where M has n columns encodes n parallel copies. If the path leading to column j is taken during program execution, all the elements in that column are copied to V in parallel.

Conventional Static Single Assignment Form The CSSA representation was first described by Sreedhar et al. [18] who used it to facilitate register coalescing. In order to define CSSA-form, we first define an equivalence relation $\equiv$ over the set of variables used in a program. We define $\equiv$ to be the smallest equivalence relation such that for every set of ϕ-functions $V = \phi M$, where V is a vector of length n with entries v_i, and M is an $n \times m$ matrix with entries v_{ij}, we have

$$\text{for each } i \in 1..n : v_i \equiv v_{i1} \equiv v_{i2} \equiv \ldots \equiv v_{im}.$$

Sreedhar et al. use *ϕ-congruence classes* to denote the equivalence classes of $\equiv$.

Definition 1 *A program is in CSSA-form if and only if for every pair of variables v_1, v_2 that occur in the same ϕ-function, we have that if $v_1 \equiv v_2$, then v_1 and v_2 do not interfere.*

Figure 5(a) shows an example of a control flow graph containing three ϕ-functions, and one equivalence class of ϕ-related virtuals: $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$. The program in Figure 5(a) is not in conventional SSA-form for two reasons. First because the parameters of the ϕ-function in block 6 interfere with each other. This type of ϕ-function is created, for instance, by select instructions, used by some compilers to implement assignments such as `x += cond ? 1 : -1`. Second, v_3 and v_4 are ϕ-related, because both are related to v_1; however, they are simultaneously alive in block 2. This situation, in which ϕ-functions in the same basic block share a parameter, is due to a compiler optimization called *copy folding*, which is described extensively by Briggs et al. [3].

A SSA-form program can be converted to CSSA-form via a very simple algorithm, called the "Method I" or "naive algorithm" by Sreedhar *et al.* [18]. This algorithm splits the live ranges of each variable that is used or defined in a ϕ-function. Sreedhar *et al.* have shown that this live range splitting is sufficient to convert a SSA-form program to a program in conventional-SSA-form. Therefore, the control flow graph transformed by the naive method has the following property: *if v_1 and v_2 are two ϕ-related virtuals, then their live ranges do not overlap.* The transformed control flow graph contains one ϕ-related equivalence class for each ϕ-function, and one equivalence class for each virtual v that does not participate in any ϕ-function. Following our example, Figure 5(b) outlines the result of applying the naive method on the control flow

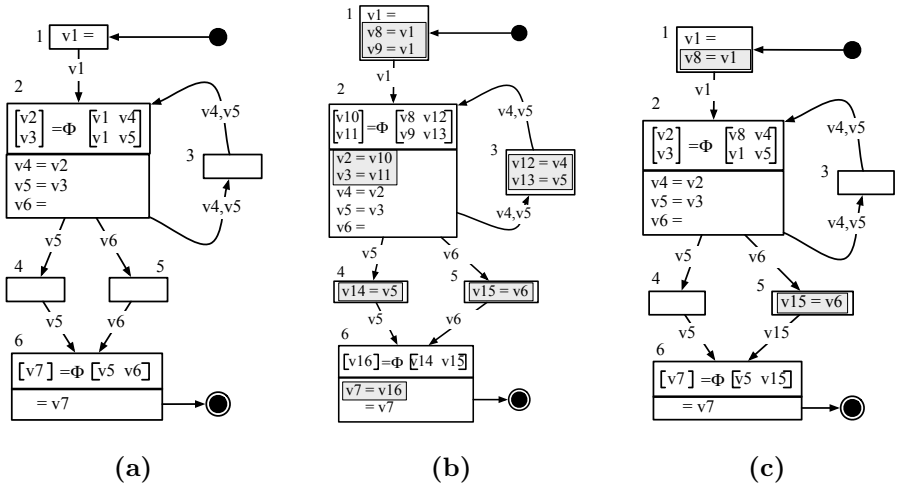


Figure 5: (a) Control flow graph (non-conventional SSA-form). (b) Program transformed by Sreedhar's "Method I" [18]. (c) Program transformed by Budimlic's coalescing technique [5].

graph given in Figure 5(a). The transformed program contains 11 equivalence classes: $\{v_1\}$, $\{v_2\}$, $\{v_3\}$, $\{v_4\}$, $\{v_5\}$, $\{v_6\}$, $\{v_7\}$, $\{v_8, v_{10}, v_{12}\}$, $\{v_9, v_{11}, v_{13}\}$ and $\{v_{14}, v_{15}, v_{16}\}$.

Although the naive algorithm produces correct programs, it is excessively conservative: two virtuals are ϕ-related if, and only if, they are used in the same ϕ-function. If a copy inserted by the naive method can be removed without creating interferences between ϕ-related variables, we call it *redundant*. Budimlic et al. [5] gave a fast algorithm to remove redundant copies. Following with the example, Figure 5(c) also illustrates a program in CSSA-form, but in this case, the program contains much fewer copy instructions: the redundant copies have been removed.

Frugal register allocators and Spartan parallel copies A register allocator for a CSSA-form program can assign the same location to all the variables $v_i, v_{i1}, \ldots, v_{im}$, for each $i \in 1..n$, because none of those variables interfere. We say that register allocation is *frugal* if it uses at most *one* memory location together with any number of registers as locations for $v_i, v_{i1}, \ldots, v_{im}$, for each $i \in 1..n$.

The problem of doing SSA-elimination consists of implementing one parallel copy for each column in each ϕ-matrix. We can implement each parallel copy independently of the others. We will use the notation

$$(l_1, \ldots, l_n) := (l'_1, \ldots, l'_n)$$

for a single parallel copy, in which $l_i, l'_i, i \in 1..n$, range over $R \cup M$, where $R = \{r_1, r_2, \ldots, r_k\}$ is a set of registers, and $M = \{m_1, m_2, \ldots\}$ is a set of memory locations. We say that a parallel copy is *well defined* if all the locations on its left side are pairwise distinct. We will use ρ to denote a *store* that maps elements of $R \cup M$ to values. If ρ is a store in which $l'_1, \ldots, l'_n$ are defined, then the meaning of a parallel copy $(l_1, \ldots, l_n) = (l'_1, \ldots, l'_n)$ is $\rho[l_1 \leftarrow \rho(l'_1), \ldots l_n \leftarrow \rho(l'_n)]$.

We say that a well-defined parallel copy $(l_1, \ldots, l_n) = (l'_1, \ldots, l'_n)$ is *spartan* if

1. for all l'_a, l'_b, if $l'_a = l'_b$, then $a = b$;
2. for all l_a, l'_b such that l_a and l'_b are memory locations, we have $l_a = l'_b$ if and only if $a = b$.

Informally, condition (1) says that the locations on the right-hand side are pairwise distinct, and condition (2) says that a memory location appears on both sides of a parallel copy if and only if it appears at the same index.

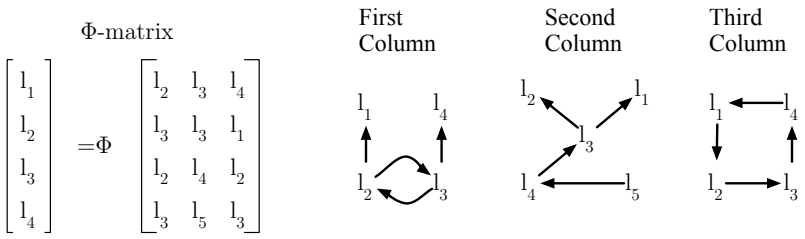


Figure 6: A ϕ-matrix and its representation as three location transfer graphs.

Theorem 2 *After frugal register allocation, the ϕ-functions used in a program in CSSA-form can be implemented using spartan parallel copies.*

Proof. We must show that the parallel copies that we derive from a CSSA-form program after frugal register allocation meet the two properties that define spartan parallel copies:

1. The CSSA-form is a subset of SSA-form; thus, every variable is defined at most once. This implies that all the parallel copies must be well defined.

2. Given a set of ϕ-functions $V = \phi M$, a frugal register allocator assigns the same memory slot to spilled variables in row i of M, and in index i of V. If a variable in row $j, j \neq i$ is spilled, it must be allocated to a memory spot different than the one reserved for variables in the i-th row, as variables in the same column interfere. The same is true for variables in V.

□

4 From windmills to cycles and paths

We now show that a spartan parallel copy can be represented using a particularly simple form of graph that we call a spartan graph (Theorem 4).

We will represent each parallel copy by a *location transfer graph*.

Definition 3 ***Location Transfer Graph***. *Given a well-defined parallel copy $(l_1, \ldots, l_n) := (l'_1, \ldots, l'_n)$, the corresponding location transfer graph $G = (V, E)$ is directed graph where $V = \{l_1, \ldots, l_n, l'_1, \ldots, l'_n\}$, and $E = \{(l'_a, l_a) \mid a \in 1..n\}$.*

Figure 6 contains a ϕ-matrix and its representation as three location transfer graphs. The location transfer graphs that represent well-defined parallel copies form a family of graphs known as *windmills* [15]. This name is due to the shape of the graphs: each connected component has a central cycle from which sprout trees, like the blades of a windmill.

The location transfer graphs that represent spartan parallel copies form a family of graphs that is significantly smaller than windmills. We say that a location transfer graph G is *spartan* if

- the connected components of G are cycles and paths;
- if a connected component of G is a cycle, then either all its nodes are in R, or it is a self loop (m, m);
- if a connected component of G is a path, then only its first and/or last nodes can be in M; and
- if (m_1, m_2) is an edge in G, then $m_1 = m_2$.

Notice that the first and second graphs in Figure 6 are not spartan because they contain nodes with out-degree 2. In contrast, the third graph in Figure 6 is spartan (if l_1, l_2, l_3, l_4 are registers): it is a cycle.

Theorem 4 *A spartan parallel copy has a spartan location transfer graph.*

Proof. It is straightforward to prove the following properties:

1. the in-degree of any node is at most 1;
2. the out-degree of any node is at most 1; and
3. if a node is a memory location m then:
   (a) the sum of its out-degree and in-degree is at most 1, or
   (b) G contains an edge (m, m).

The result is immediate from (1)–(3). □

5 SSA elimination

Our goal is to implement spartan parallel copies in the language `Seq` that contains just four types of instructions: register-to-register moves $r_1 := r_2$, loads $r := m$, stores $m := r$, and register swaps $r_1 \oplus r_2$. Notice that `Seq` does not contain instructions to swap or copy the contents of memory locations in one step. We use ι to range over instructions. A `Seq` program is a sequence I of instructions that modify a store ρ according to the following rules:

$$\frac{\langle \iota, \rho\rangle \to \rho'}{\langle \iota; I, \rho\rangle \to \langle I, \rho'\rangle}$$

$$\langle l_1 := l_2, \rho\rangle \to \rho[l_1 \leftarrow \rho(l_2)]$$

$$\langle r_1 \oplus r_2, \rho\rangle \to \rho[r_1 \leftarrow \rho(r_2), r_2 \leftarrow \rho(r_1)]$$

The problem of implementing a parallel copy can now be stated as follows.

Implementation of a Spartan Parallel Copy
Instance: a spartan parallel copy $(l_1, \ldots, l_n) = (l'_1, \ldots, l'_n)$.
Problem: find a `Seq` program I such that for all stores ρ,

$$\langle I, \rho\rangle \to^* \rho[l_1 \leftarrow \rho(l'_1), \ldots l_n \leftarrow \rho(l'_n)].$$

Our algorithm **ImplementSpartan** uses a subroutine **ImplementComponent** that works on each connected component of a spartan location transfer graph and is entirely standard.

Algorithm 1 – **ImplementComponent**: Input: G, Output: I

Require: G is a cycle or a path
Ensure: I is a `Seq` program.
1: **if** G is a path $(l_1, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, l_n)$ **then**
2: $\quad I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots; r_2 := l_1)$
3: **else if** G is a cycle $(r_1, r_2), \ldots, (r_{n-1}, r_n), (r_n, r_1)$ **then**
4: $\quad I = (l_n \oplus l_{n-1}; l_{n-1} \oplus l_{n-2}; \ldots; l_2 \oplus l_1)$
5: **end if**

Theorem 5 (Correctness) *For a spartan location transfer graph G, **ImplementSpartan**(G) is a correct implementation of G.*

Algorithm 2 – **ImplementSpartan**: Input: G, Output: program I

Require: G is a spartan location transfer graph.
Require: G has connected components $C_1, \ldots, C_m$.
Ensure: I is a `Seq` program.
1: $I =$ **ImplementComponent**$(C_1); \ldots;$ **ImplementComponent**(C_m);

Proof. See Appendix A. □

Once we have implemented each spartan parallel copy, all that remains to complete spill-free SSA elimination is to replace the ϕ-functions with the generated code. As illustrated in Figure 3, the generated code for a parallel copy must be inserted at the end of the basic block that leads to the parallel copy.

5.1 SSA Elimination and Critical Edges

Critical edges are edges that connect a basic block with multiple successors to a basic block with multiple predecessors. Briggs *et al.* [3] have shown that the existence of critical edges in the source program might lead to the production of incorrect code during the replacement of ϕ-functions by copy instructions. As demonstrated by Sreedhar et al [18], the CSSA form allows to handle the problems pointed by Briggs *et al.* without requiring the elimination of critical edges from the source program. Nonetheless, when performing SSA-elimination after register allocation, the absence of critical edges is still necessary for correctness even if the source program is in colored-CSSA form, as the example in Figure 7 shows. In this example, the elimination of the ϕ-function $(a_2, r_2) = \phi[(a_1, r_1), \ldots]$ requires the contents of register r_1 to be moved into register r_2 in the control-flow path connecting blocks 1 and 4. However, such transfer cannot be inserted at the end of block 1, or it would overwrite the value of b, nor at the beginning of block 4, or it would overwrite the value of a_3.

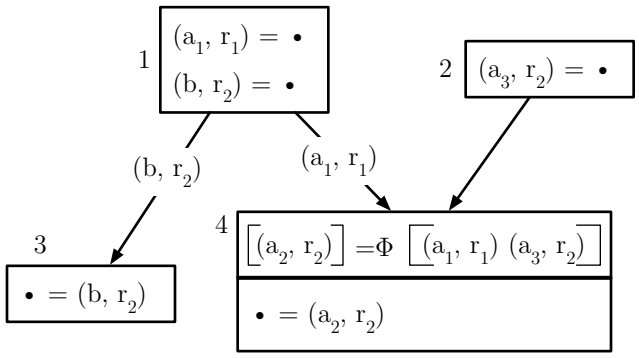


Figure 7: The presence of critical edges leads to incorrect code.

Another problem of critical edges is that they may cause an increase in the register pressure of the source program, even if ϕ-functions are eliminated before register allocation. The register pressure at any program point is the difference between the number of variables alive at that point and the number of registers available to accommodate them [8]. The total number of registers necessary to allocate all the variables in a SSA-form program P equals the maximum register pressure at any point of P [10]. For example, the program in Figure 8 (a) illustrates the *swap-problem*, pointed by Briggs *et al.* [3], and Figure 8 (b) shows the same program, converted into CSSA-form. The interference graph of the latter program has chromatic number 3, whereas the graph of the former program has chromatic number 2. Figure 8 (c) shows the same

program, after the critical edge forming the loop has been removed. The interference graph of this program has chromatic number 2.

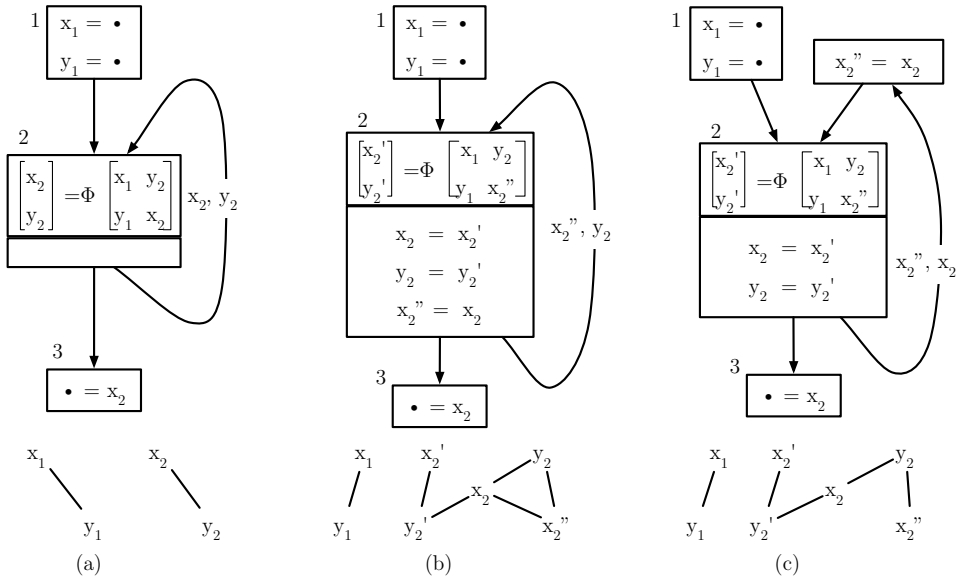


Figure 8: Example where the presence of critical edges can increase the register pressure. The interference graph is shown below each program.

On the other hand, if the source SSA-form program has no critical edges, then its register pressure is guaranteed to remain the same after the convertion into CSSA-form, as we show in Theorem 6.

Theorem 6 ***(Register Pressure)*** *Let P be a program whose control flow graph does not contain critical edges. Sreedhar's "Method I" does not increase the global register pressure in P.*

Proof. See Appendix B. □

6 Optimizations

We will present three optimizations of the **ImplementSpartan** algorithm. Each optimization (1) has little impact on compilation time, (2) has a significant positive impact on the quality of the generated code, (3) can be implemented as constant-time checks, and (4) must be accompanied by a small change to the register allocator.

6.1 Store hoisting

Each variable name is defined only once in an SSA-form program; therefore, the register allocator needs to insert only one store instruction per spilled variable. However, algorithm **ImplementSpartan** inserts a store instruction for each edge (r, m) in the location transfer graph. We can change **ImplementComponent** to avoid inserting store instructions:

1: **if** G is a path $(l_1, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, m)$ **then**
2: $I = (r_{n-1} := r_{n-2}; \ldots; r_2 := l_1)$
3: ...

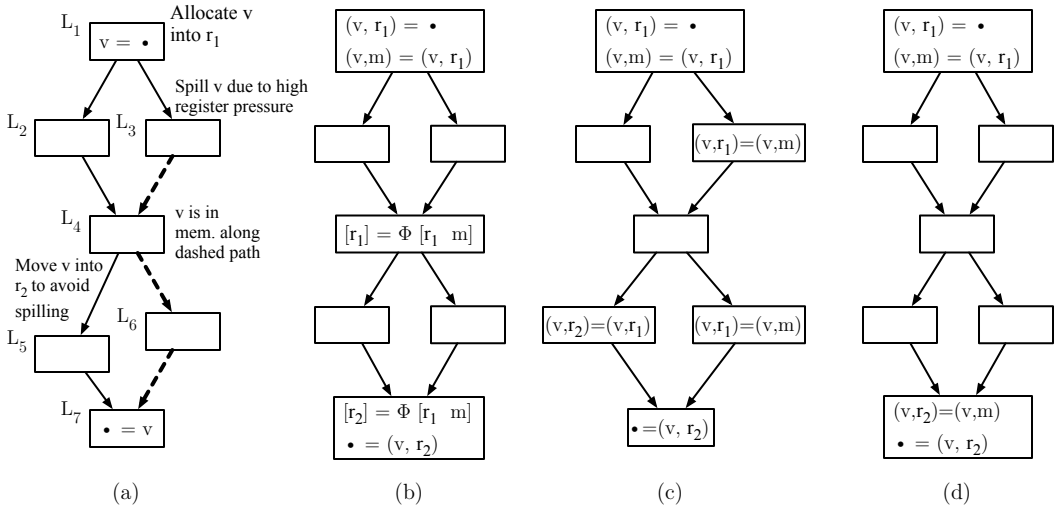


Figure 9: (a) Example program (b) Program augmented with mock ϕ-functions. (c) SSA elimination without load-lowering. (d) Load-lowering in action.

4: **end if**

For this to work, we must change the register allocator to explicitly insert a store instruction after the definition point of each spilled variable. On the average, store hoisting removes 12% of the store instructions in SPEC CPU 2000.

6.2 Load Lowering

Load lowering is the dual of store hoisting: it reduces the number of load and copy instructions inserted by the **ImplementSpartan** Algorithm. There are situations when it is advantageous to reload a variable right before it is used, instead of during the elimination of ϕ-functions. Load lowering is particularly useful in algorithms that follow the bin-packing model [11, 13, 17, 19]. These allocators allow variables to reside in different registers at different program points, but they require some fixing code at the basic block boundaries. The insertion of fixing code obeys the same principles that rule the implementation of ϕ-functions in SSA-based register allocators. In Figure 9 we simulate the different locations of variable v by inserting mock ϕ-functions at the beginning of basic blocks L_2 and L_7, as pointed in Figure 9 (b). The fixing code will be naturally inserted when these ϕ-functions are eliminated. The load lowering optimization would replace the instructions used to implement the ϕ-functions, shown in Figure 9 (c), with a single load before the use of v at basic block L_7, as outlined in Figure 9 (d).

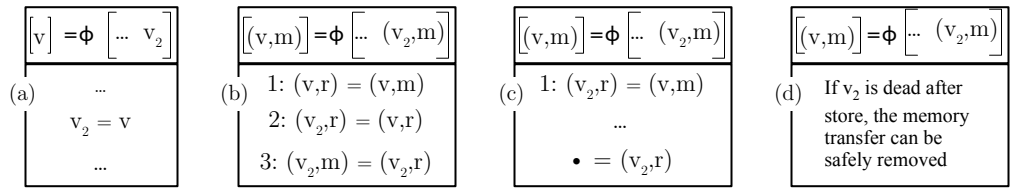
Variables can be lowered according to the nesting depth of basic blocks in loops, or the static number of instructions that could be saved. The SSA elimination algorithm must remember, for each node l in the location transfer graph, which variable is allocated into l. During register allocation we mark all the variables v that would benefit from lowering, and we avoid inserting loads for locations that have been allocated to v. Instead, the register allocator must insert reloads before each use of v. These reloads may produce redundant memory transfers, which are eliminated by the memory coalescing pass described in Section 6.3. The updated elimination algorithm is outlined below:

1: **if** G is a path $(m, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, l_n)$ **then**
2: **if** m is holding a variable marked to be lowered **then**
3: $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots; r_3 := r_2)$
4: **else**

5: $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots; r_2 := m)$
6: **end if**
7: ...
8: **end if**

6.3 Memory coalescing

A memory transfer is a sequence of instructions that copies a value from a memory location m_1 to another memory location m_2. The transfer is redundant if these locations are the same. The CSSA-form allows us to coalesce a common occurrence of redundant memory transfers. Consider, for instance, the code that the compiler would have to produce in case variables v_2 and v, in the figure below, are spilled. In order to send the value of v_2 to memory, the value of v would have to be loaded into a spare register r, and then the contents of r would have to be stored, as illustrated in figure (b). However, v and v_2 are mapped to the same memory location because they are ϕ-related. The store instruction can always be eliminated, as in figure (c). Furthermore, if the variable that is the target of the copy - v_2 in our example - is dead past the store instruction, then the whole memory transfer can be completely eliminated, as we show in figure (d) below:



7 Experimental results

The data presented in this section uses the SSA-based register allocator described by Pereira and Palsberg [13], which has the following characteristics:

- the register assignment phase occurs before the SSA-elimination phase;
- registers are assigned to variables in the order in which they are defined, as determined by a pre-order traversal of the dominator tree of the source program;
- variables related by move instructions are assigned the same register if they belong into the same ϕ-equivalence class whenever possible;
- two spilled variables are assigned the same memory address whenever they belong into the same ϕ-equivalence class;
- the allocator follows the bin-packing model, so it can change the register assigned to a variable to avoid spilling. Thus, the same variable may reach a join point in different locations. This situation is implemented via the mock ϕ-functions discussed in Section 6.2.
- SSA-elimination is performed by the Algorithm **ImplementSpartan** augmented with code to handle register aliasing, plus load-lowering, store hoisting, and elimination of redundant memory transfers.

Our register allocator is implemented in the LLVM compiler framework [12], version 1.9. LLVM is the JIT compiler used in the openGL stack of Mac OS 10.5. Our tests are executed on a 32-bit x86 Intel(R) Xeon(TM), with a 3.06GHz cpu clock, 4GB of memory and 512KB L1 cache running Red Hat Linux 3.3.3-7. Our benchmarks are the C programs from SPEC CPU 2000.

	gcc	pbk	gap	msa	vtx	twf	cfg	vpr	amp	prs	gzp	bz2	art
#ltg	72.6	40.3	22.1	15.6	15.8	6.8	7.7	4.5	4.0	5.2	.9	.73	.36
%sp	3.3	5.0	9.8	2.3	9.3	6.5	14.9	13.5	7.9	6.5	10.9	22.7	9.2
#edg	586.2	256.3	150.8	96.9	121.5	58.0	124.2	101.7	29.6	35.5	11.1	14.3	2.7
%mt	56.4	41.7	43.5	50.6	47.1	57.3	66.8	75.4	37.4	42.8	63.6	71.8	46.0

Figure 10: #ltg: number of location transfer graphs (in thousands), %sp: percentage of LTG's that are potential spills, #edg: number of edges in all the LTG's (in thousands), %mt: percentage of the edges that are memory transfers.

Impact of our SSA Elimination Method Figure 10 summarizes static data obtained from the compilation of SPEC CPU 2000. Our SSA Elimination algorithm had to implement 197,568 location transfer graphs when compiling this benchmark suite. These LTGs contain 1,601,110 edges, out of which 855,414, or 53% are memory transfers. Due to the properties of spartan location transfer graphs, edges representing memory transfers are always loops, that is, an edge from a node m pointing to itself. Because our memory transfer edges have source and target pointing to the same address, the SSA Elimination algorithm does not have to insert any instruction to implement them. Potential spills could have happened in 11,802 location transfer graphs, or 6% of the total number of graphs, implying that, if we had used a spilling on demand approach instead of our SSA elimination framework, a second spilling phase would be necessary in all the benchmark programs. We mark as potential spills the location transfer graphs that contain memory transfers, and in which the register pressure is maximum, that is, all the physical registers are used in the right side of the parallel copy.

Time Overhead of SSA-Elimination The charts in Figure 11 show the time required by our compilation passes. Register allocation accounts for 28% of the total compilation time. This time is similar to the time required by the standard linear scan register allocator, as reported in previous works [14, 16]. The passes related to SSA elimination account for about 4.8% of the total compilation time. These passes are: (i) Sreedhar's "Method I", which splits the live ranges of all the variables that are part of ϕ-functions [18, pg.199]; (ii) a pass to remove critical edges; (iii) Budimlic's copy coalescing [5], which reduces the number of copies inserted by the "Method I"; (iv) our spill-free SSA elimination pass. The amount of time taken by each of these passes is distributed as follows: (i) 0.2%, (ii) 0.5%, (iii) 1.6% and(iv) 2.5%.

Impact of the Optimizations Figure 12 shows the static reduction of load, store and copy instructions due to the optimizations described in Section 6. The criterion used to determine if a variable should be lowered or not is the number of reloads that would be inserted for that variable versus the number of uses of the variable. Before running the SSA-elimination algorithm we count the number of reloads that would be inserted for each variable. The time taken to get this measure is negligible compared to the time to perform SSA-elimination: loads can only be the last edge of a spartan location transfer graph (Theorem 4). A variable is lowered if its spilling causes the allocator to insert more reloads than the number of uses of that variable in the source program. Store hoisting (SH) alone eliminates on average about 12% of the total number of stores in the target program, which represents slightly less than 5% of the lines of spill code inserted. By plugging in the elimination of redundant memory transfers (RMTE) we remove other 2.6% lines of spill code. Finally, load lowering (LL), on top of these other two optimizations, eliminates 7.8% more lines of spill code. Load lowering also removes 5% of the copy instructions from the target programs.

The chart in the bottom part of Figure 12 shows how the optimizations influence the run time of the benchmarks. On the average, they produce a speed up of 1.9%. Not all the programs benefit from load lowering. For instance, load lowering increases the run time of `186.crafty` in almost 2.5%. This happens because, for the sake of simplicity, we do not take into consideration the loop nesting depth of basic blocks when lowering loads. We speculate that more sophisticated criteria would produce more substantial performance gains. Yet, these optimizations are being applied on top of a very efficient register allocator, and they do not incur in any measurable penalty in terms of compilation time.

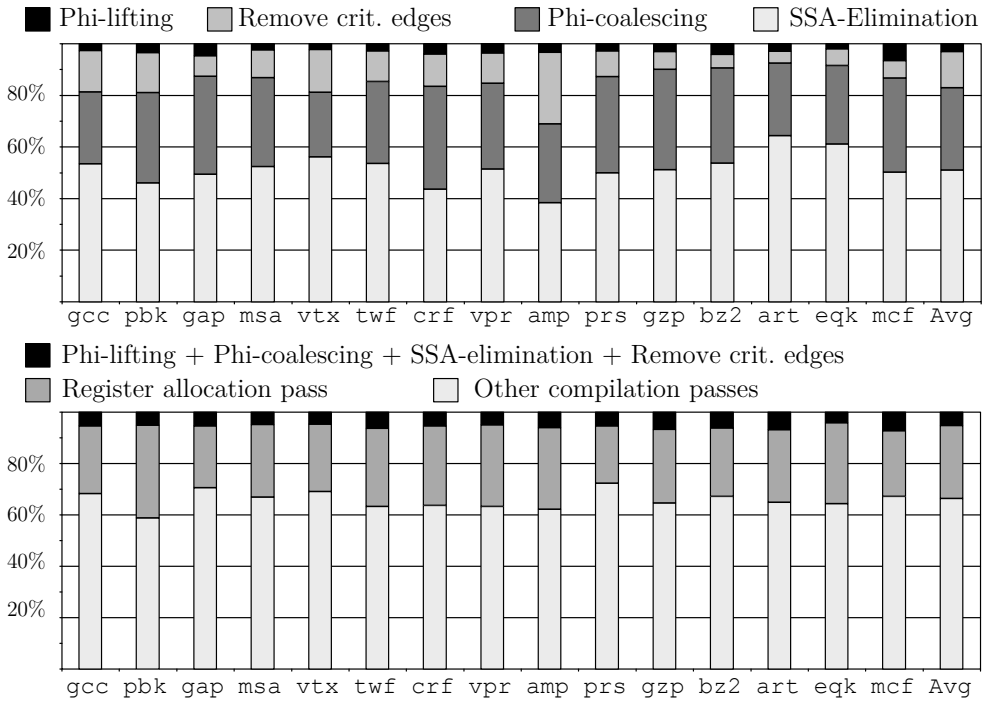


Figure 11: Execution time of different compilation passes.

8 Final Remarks

This report has presented spill-free SSA elimination, a simple and efficient algorithm for SSA elimination after register allocation that avoids increasing the number of spilled variables. Our algorithm runs in polynomial time and accounts for a small portion of the total compilation time.

Our approach relies on the ability to swap the contents of two registers. For integer registers, architectures such as x86 provide a swap instruction, while on other architectures one can implement a swap with a sequence of three `xor` instructions. In contrast, for floating point registers, most architectures provide neither a swap instruction nor a `xor` instruction, so instead compiler writers have to use one of the other approaches to SSA-elimination, e.g: separate a temporary register or perform spilling on demand.

A Proof of Theorem 5

Theorem 5 was stated as follows:

> ***(Correctness)*** *For a spartan location transfer graph G,*
> ***ImplementSpartan***(G) *is a correct implementation of G.*

By Theorem 4, G must be either a cycle or a path; thus, we divide this proof into two parts: Lemma 7 and Lemma 8. The semantics of parallel copies are defined in the obvious way:

$$\langle I, (l_1, \ldots, l_n) := (l'_1, \ldots, l'_n) \rangle \rightarrow \rho[l_1 \leftarrow \rho(l'_1), \ldots l_n \leftarrow \rho(l'_n)]. \quad (1)$$

Lemma 7 *If μ is a spartan parallel copy and its location transfer graph is a cycle, then there is a sequence of $n-1$ swaps in the language* `Seq` *that is semantically equivalent to μ.*

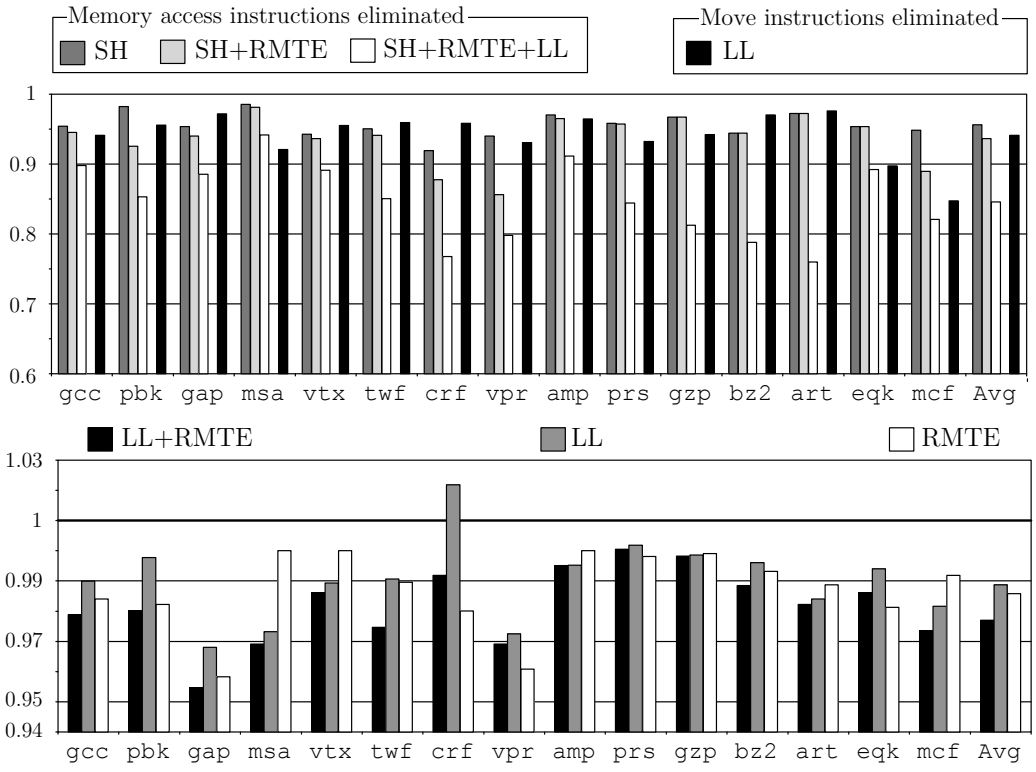


Figure 12: Impact of Load Lowering (LL) and Redundant Memory Transfer Elimination (RMTE) on the code produced after SSA-elimination. (Up) Code size. (Down) Run-time.

Proof. The proof is by induction on the length of μ. By Theorem 4 all the locations in μ are registers, because μ is a cycle by hypothesis.

Base case: if μ has lenght two, then by Equation 1 we have that $(r_1, r_2) := (r_2, r_1) \equiv r_1 \oplus r_2$.

Induction hypothesis: the theorem is true for parallel copies with up to $n-1$ variables on each side.

Induction step: we consider the parallel copy $(r_1, r_2, \ldots, r_{n-1}, r_n) := (r_2, r_3, \ldots, r_n, r_1)$ applied on the environment ρ, where $\rho(r_i) = v_i$. If we apply $r_1 \oplus r_n$ on ρ, we get the environment $\rho' = \rho[r_n \leftarrow v_1][r_n \leftarrow v_1]$. Register r_n has the location that would be assigned to it by μ. Consider now the parallel copy $\mu' = (r_1, r_2, \ldots, r_{n-2}, r_{n-1}) := (r_2, r_3, \ldots, r_{n-1}, r_1)$. The parallel copy μ' is similar to μ, except that r_1 sends its value to r_{n-1}, and r_n is no longer present. But r_1 contains now v_n, the value that should be transfered to r_{n-1}. The result follows by applying induction on μ', which has size n. □

Lemma 8 *If μ is a spartan parallel copy and its location transfer graph is a path, then there is a sequence of $n-1$ swaps in the language* `Seq` *that is semantically equivalent to μ.*

Proof. The proof is by induction on the length of μ, and it is similar to the proof of Lemma 7. □

The proof of Theorem 5 follows by combining the two previous lemmas, plus the fact that any component of a lcoation transfer graph is either a cycle or a path.

B Proof of Theorem 6

In this section we prove Theorem 6, which we re-state as follows:

> ***(Register Pressure)*** *Let P be a program whose control flow graph does not contain critical edges. The SSA-to-CSSA conversion does not increase the global register pressure in P.*

We will assume that P is in strict, pruned SSA-form. A program is in strict SSA form [5] if it is in SSA form and for each variable x, the single definition of x dominates all its uses. A program is in pruned SSA-form if none of the variables defined by a ϕ-function is a dead-definition [6]. We recall the definition of liveness analysis, as given by Appel and Palsberg [2, p.206], where l is a statement in the program, $in[l]$ is the set of variables live before l, $out[l]$ is the set of variables live after l, $def[l]$ is the set of variables defined at l, $use[l]$ is the set of variables used at l, and $succ[l]$ is the set of statements that succeed l.

$$\begin{array}{rcl} in[l] & = & use[l] \cup (out[l] - def[l]) \\ out[l] & = & \bigcup_{s \in succ[l]} in[s] \end{array} \tag{2}$$

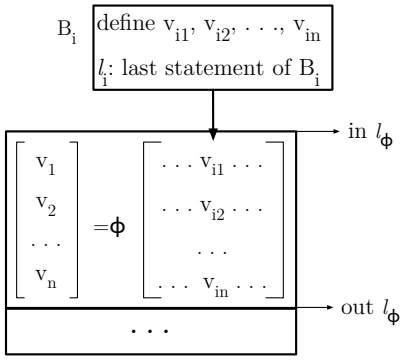


Figure 13: The parallel copy $l_\phi : (v_1, v_2, \ldots, v_n) := (v_{i1}, v_{i2}, \ldots, v_{in})$.

Lemma 9 *Let P be a strict program in pruned-CSSA-form with no critical edges. If l_i and l_ϕ are defined as in Figure 13, then $|out(l_i)| = |out(l_\phi)|$.*

Proof. In order to prove this lemma, we will use the claims listed below, where $X_i = \{v_{i1}, v_{i2}, \ldots, v_{in}\}$, and $X_\phi = \{v_1, v_2, \ldots, v_n\}$:

1. $out[l_i] = in[l_\phi]$;
2. $v_{ij} \notin out[l_\phi], 1 \leq j \leq n$;
3. $X_i \cap out[l_\phi] = \emptyset$;
4. $out[l_i] - X_i = out[l_\phi] - X_\phi$;
5. $v_{ij} \neq v_{ik}, 1 \leq j, k \leq n$ and $j \neq k$;
6. $|X_i| = |X_\phi|$;
7. $X_i \subseteq out[l_i]$;

8. $X_\phi \subseteq out[l_\phi]$;

We proof these claims as follows:

- **proof of claim 1** This follows from Equation 2, plus the fact that P has no critical edges, so $\bigcup_{s \in succ[l_i]} = \{l_\phi\}$.
- **proof of claim 2** If we assume otherwise, v_{ij} would interfere with all v_j. We have that $v_j \in out[l_\phi]$ because P is pruned. However, v_j and v_{ij} cannot interfere because P is in CSSA-form, and v_{ij} and v_j are ϕ-related.
- **proof of claim 3** Follows as a simple corollary of claim 2.
- **proof of claim 4** According to Equation 2:
  $in[l_\phi] = use[l_\phi] \cup (out[l_\phi] - def[l_\phi]) = X_i \cup (out[l_\phi] - X_\phi)$
  From claim 1:
  $out[l_i] = X_i \cup (out[l_\phi] - X_\phi)$
  From claim 3:
  $out[l_i] - X_i = out[l_\phi] - X_\phi$
- **proof of claim 5** by the definition of CSSA-form program.
- **proof of claim 6** Follows as a simple corollary of claim 5.
- **proof of claim 7** Follows from Equation 2, plus claim 1, e.g: $out[l_i] = in[l_\phi] = X_i \cup (\ldots)$.
- **proof of claim 8** This claim follows from the fact that we are dealing with a program in pruned-SSA-form. In this case, none of the variables defined by ϕ-functions are dead at the definition point.

Finally, to prove our final result, e.g $|out(l_i)| = |out(l_\phi)|$, we combine claims 4, 6, 7 and 8. □

The *global register pressure* of a program is bounded by the maximum number of variables alive at any point of the program. A program in SSA-form never requires more registers than its global register pressure. Theorem 6 shows that the conversion from SSA to CSSA-form preserves the global register pressure of the source program, that is, if P is a program in pruned-SSA-form that could be compiled with K registers before being transformed by Sreedhar's "Method I", it still can be compiled with K registers after it.

We now prove Theorem 6:

Proof. Given a ϕ-function such as $a_i : B = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m)$, Sreedhar's "Method I" changes it in two ways:

1. it splits the live range of the variable defined by the ϕ-function with an instruction $I = \langle a_i := v_i \rangle$.
2. it splits the live ranges of the variables used in the ϕ-function with m instructions like $I_j = \langle v_{ij} := a_{ij} \rangle$.

We will show that each transformation preserves the global register pressure of the source program.

1. Because P is a program in pruned-SSA-form, each variable defined by a ϕ-function is alive past its definition point. The variable v_i inserted by Sreedhar's "Method I" is alive from the ϕ-function until instruction I. Variable a_i is alive thereafter.Thus, variable v_i does not increase the register pressure in P, because v_i and a_i are never simultaneously alive.
2. From Lemma 9 we know that the register pressure at the end of a basic block that feeds a ϕ-equation $V = \phi M$ is bounded by the register pressure at program point l_ϕ, past the definition point of V, and, from the proof of (1) above, we know that the register pressure at l_ϕ remains constant after the source program is modified by Sreedhar's "Method I".

□

References

[1] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[2] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[3] 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.

[4] Philip Brisk. *Advances in Static Single Assignment Form and Register Allocation*. PhD thesis, UCLA - University of California, Los Angeles, 2006.

[5] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM Press, 2002.

[6] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66, 1991.

[7] 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.

[8] Martin Farach-colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

[9] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[10] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th Conference on Compiler Construction*, pages 247–262. Springer-Verlag, 2006.

[11] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI*, pages 204–215, 2006.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[13] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226, 2008.

[14] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[15] Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. Tilting at windmills with coq: formal verification of a compilation algorithm for parallel moves, 2008. To appear.

[16] Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software, Practice and Experience*, 33:1003–1034, 2003.

[17] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.

[18] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer-Verlag, 1999.

[19] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 142–151, 1998.