SSA Elimination after Register Allocation

Fernando Magno Quintão Pereira
Jens Palsberg

UCLA
University of California, Los Angeles

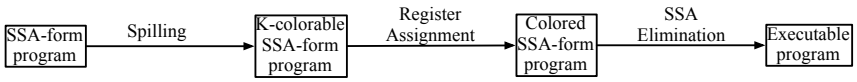
Abstract. Compilers such as gcc use static-single-assignment (SSA) form as an intermediate representation and usually perform SSA elimination before register allocation. But the order could as well be the opposite: the recent approach of SSA-based register allocation performs SSA elimination after register allocation. SSA elimination before register allocation is straightforward and standard, while previously described approaches to SSA elimination after register allocation have shortcomings; in particular, they have problems with implementing copies between memory locations. 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 of 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

Register allocation is the process of mapping a program that uses an unbounded number of *variables* to a program that uses a fixed number of *registers*, such that variables with overlapping live ranges are assigned different registers. If registers cannot accommodate all the variables that are live at some point in the program, some of these variables must be *spilled*, that is, stored in memory. Register allocation is one of the most important compiler optimizations and can improve the speed of compiled code by more than 250% [17].

Static Single Assignment (SSA) form is an intermediate representation that defines each variable at most once [9, 24] and in which φ-functions express renaming of variables. φ-functions are normally not present in the instruction sets of actual computer architectures. Thus, compilers that use SSA form must eventually do SSA elimination, replacing each φ-function with copy and swap instructions [2, 5, 8, 10, 19]. Many industrial compilers use the SSA form as an intermediate representation, including gcc 4.0 [11], Sun's HotSpot JVM [29], IBM's Java Jikes RVM [30], and LLVM [15], and they all perform SSA elimination before register allocation. But the order could as well be the opposite: the recent approach of SSA-based register allocation [3, 7, 12, 13, 21] performs SSA

elimination after register allocation. SSA-based register allocation has three main advantages: (1) the problem of finding the minimum number of registers that are needed for a program in SSA form has a polynomial-time solution, (2) a program in SSA form requires at most as many registers as the source program, and (3) register allocation can proceed in two separate phases, namely first spilling and then register assignment. The two-phase approach works because the number of registers needed for a program in SSA-form is equal to 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 this case, no register is available as a temporary location for performing a two-instruction sequence of a load followed by a store.

One solution to the memory-transfer problem would be to permanently reserve a register to implement memory-to-memory copies. 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 [15] inserts in SPEC CPU 2000. Another solution would be to force the register allocator to assign the same register to all the variables that are part of a φ-function. In this case, each φ-function would be trivially implemented as a no-op; however, this form of aggressive coalescing might lead to sub-optimal registers assignments. For instance, Pereira and Palsberg [19, Fig.3] showed an example program where aggressive coalescing produces a minimal allocation with three registers, whereas the variables of the same program in SSA-form can be allocated in two registers.

Brisk [6, 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. 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. However, in our view, the main problem with Brisk's approach is that its second spilling phase - during SSA-elimination - substantially complicates the design of a register allocator.

Our goal is to do better. 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 [6]	Yes	No
Spill-free SSA elimination	Yes	Yes

The starting point for our approach to SSA-based register allocation is *Conventional SSA (CSSA)-form* [28] rather than the SSA form from the original paper [9] (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. As explained by Sreedhar *et al.* [28, p.196], and Briggs *et al.* [5, p.873], the original algorithm that converts a program into SSA form [9] already guarantees the CSSA property; however, compiler optimizations such as copy folding might produce interferences between variables related by φ-functions and thereby lose the CSSA property. Thus, our approach to SSA elimination requires us to convert the source program back into CSSA form before register allocation starts.

In this paper we make two assumptions. First, we 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]. Second, we assume that the target architecture provides us with a way to swap the contents of two registers. If swaps are not provided, then the problem of finding the minimal number of registers required by a program is NP-complete [4, 20]. 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 direct swap instructions nor `xor` instructions, 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.

We will present both a core algorithm for spill-free SSA elimination as well as three optimizations. We have implemented our SSA elimination framework in a puzzle-based register allocator [21]. Our experiments show that our approach to SSA elimination, including the conversion of source program into CSSA-form,

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%. Our SSA elimination framework works for any SSA-based register allocator such as [13], and it can also be used to insert the fixing code required by register allocators that follow the bin-packing model [14, 21, 26, 31].

We will state three theorems with either just a proof sketch or no proof at all; the proofs can be found in Pereira's Ph.D. dissertation [18, Ch.5].

2 Example

We now present an example that assumes a target architecture with a single register r. Figure 1(a) shows a program in SSA form that contains six variables: a, a_1, a_2, b, b_1 and b_2. We use an abstract notation to represent instructions. For instance, the assignment $a_2 = b$ does not represent a move instruction, but just an instruction that defines variable a_2 and uses variable b. In the same way, $b_2 = \bullet$ is an instruction that defines b_2, and $\bullet = a$ is an instruction that uses a. Figure 1(b) shows the program after spilling and register assignment. A pair such as (b, r) indicates that variable b has been allocated to register r. Our example uses the disjoint memory addresses m, m_2 and m_b as locations for the spilled variables. Figure 1(c) shows the program after SSA elimination with on-demand spilling. Notice that in Figure 1(c), a φ-function has been replaced with four instructions that implement a copy from m_2 to m. The address m_b is used to temporarily hold the contents of r, while this register is used in the memory-to-memory transfer. 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 1(d) shows the same program as in Figure 1(a), but this time in CSSA form, Figure 1(e) shows the program after spilling and register assignment, and Figure 1(f) shows the program after spill-free SSA elimination. Notice that in Figure 1(d), top right corner, CSSA makes a difference by requiring the extra instruction that copies from a_2 to a_3. This instruction splits the live range of a_2, what is necessary because variables a and a_2 interfere. We now do register allocation and assign each of a, a_1, and a_3 to the same memory location m because those variables do not interfere. In Figure 1(e), top right corner, the value of a_2 arrives in memory location m_2, and is then copied to memory 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 1(f) without any further spills.

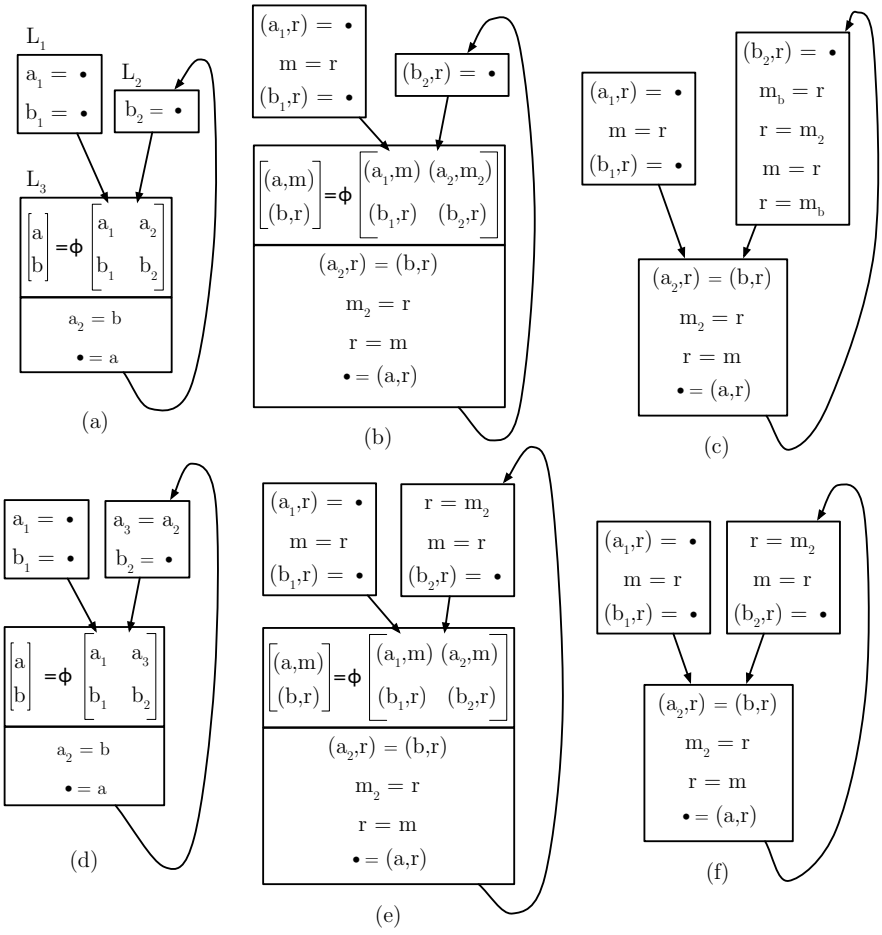


Fig. 1. Top: SSA-based register allocation and SSA elimination with on-demand spilling. Bottom: SSA-based register allocation and spill-free SSA elimination.

3 CSSA form and spartan parallel copies

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 1). Along the way, we will define all the concepts and notations that we use.

SSA form uses φ-functions to express renaming of variables. We will describe the syntax and semantics of φ-functions using the matrix notation introduced by Hack et al. [13]. Figure 1 contains examples of φ-matrices. An assignment such as $V = \varphi M$, where V is a vector of length n, and M is an $n \times m$ matrix,

represents n φ-functions and m *parallel copies* [16, 27, 32]. Each column in the φ-matrix corresponds to an incoming control-flow edge. A φ-function works as a multiplexer: it assigns to each element v_i of V an element v_{ij} of M, where j is determined by the actual control-flow edge taken during the program's execution. The parameters of a φ-function are evaluated simultaneously at the beginning of the basic block where the φ-function is defined [1]. For instance, the φ-matrix in Figure 1 (a) represents the parallel copies $(a, b) := (a_1, b_1)$ and $(a, b) := (a_2, b_2)$. The first parallel copy is executed if control reaches L_3 from L_1, while the second is executed if control reaches L_3 from L_2.

Conventional Static Single Assignment (CSSA) form was first described by Sreedhar et al. [28] who used CSSA form 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 $V = \varphi M$, 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, we have that if $v_1 \equiv v_2$, then v_1 and v_2 do not interfere.*

Budimlic et al. [8] presented a fast algorithm for converting an SSA-form program to CSSA-form. 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 $a = b$; and,
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 $a = b$.

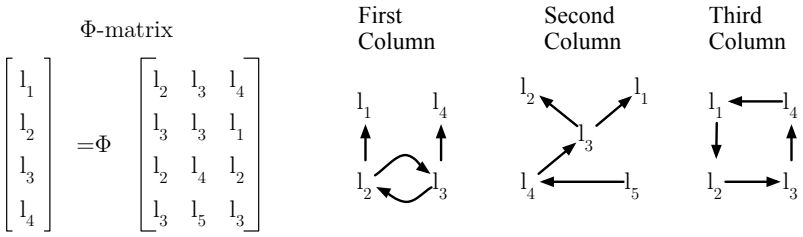


Fig. 2. A φ-matrix and its representation as three location transfer graphs.

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.

Theorem 1. *After frugal register allocation, the φ-functions used in a program in CSSA-form can be implemented using spartan parallel copies.*

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 2).

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

Definition 2. 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 a 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 2 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* [23]. 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 2 are not spartan because they contain nodes with out-degree 2. In contrast, the third graph in Figure 2 is spartan as long as l_1, l_2, l_3, l_4 are registers because the graph is a cycle.

Theorem 2. *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.

Theorem 3. *For a spartan location transfer graph G,* **ImplementSpartan**(G) *is a correct implementation of G.*

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 1, the generated code for a parallel copy must be inserted at the end of the basic block that leads to the parallel copy.

Algorithm 1 – **ImplementComponent**: Input: G, Output: program 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: $\quad I = (r_n \oplus r_{n-1}; r_{n-1} \oplus r_{n-2}; \ldots; r_2 \oplus r_1)$
5: **end if**

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);

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: $\quad I = (r_{n-1} := r_{n-2}; \ldots; r_2 := l_1)$
3: $\quad \ldots$
4: **end if**

For this to work, we must change the register allocator to 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

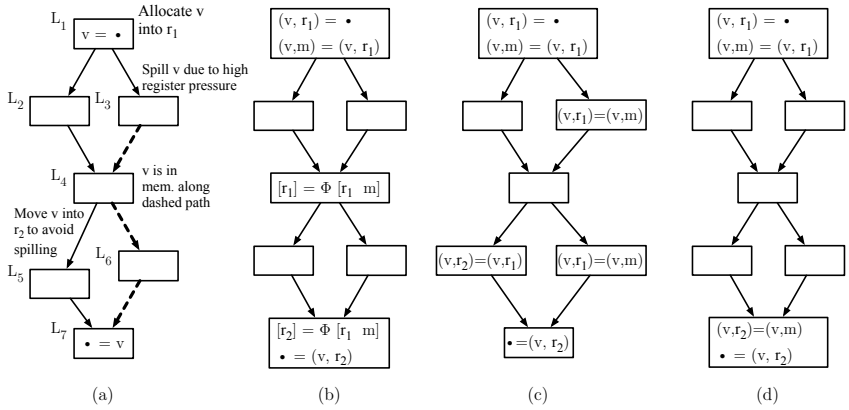


Fig. 3. (a) Example program (b) Program augmented with mock φ-functions. (c) SSA elimination without load-lowering. (d) Load-lowering in action.

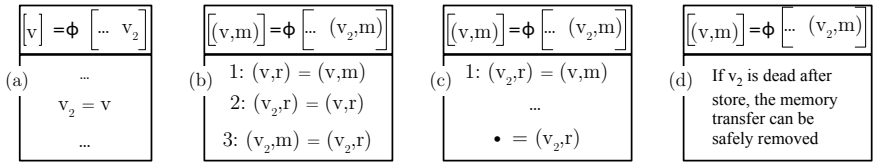
algorithms that follow the bin-packing model [14, 21, 26, 31]. These allocators allow variables to reside in different registers at different program points, but they require some fixing code at the basic block boundaries. The insertion of fixing code obeys the same principles that rule the implementation of φ-functions in SSA-based register allocators. In Figure 3 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 3 (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 3 (c), with a single load before the use of v at basic block L_7, as outlined in Figure 3 (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: $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots; r_3 := r_2; r_2 := m)$
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. Notice that (d) is not a simple case of dead-code elimination, as the pair (v_2, m) might not be dead, e.g, variable v_2 might be reloaded from m at some future program point. However, the compiler can safely eliminate this store because the value of v, which equals the value of v_2, has already been stored in m by a frugal register allocator.



7 Experimental results

The data presented in this section uses the SSA-based register allocator described by Pereira and Palsberg [21], 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.

	gcc	pbk	gap	msa	vtx	twf	cfg	vpr	amp	prs	gzp	bz2	art	eqk	mcf
#ltg	72.6	40.3	22.1	15.6	15.8	6.8	7.7	4.5	4.0	5.2	.9	.73	.36	.27	.44
%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	20.8	25.6
#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	5.8	6.1
%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	72.0	57.7

Fig. 4. #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.

Our register allocator is implemented in the LLVM compiler framework [15], 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.

Impact of our SSA Elimination Method Figure 4 summarizes static data obtained from the compilation of SPEC CPU 2000; we have ordered the benchmarks by size. 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 5 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 work [22, 25]. The passes related to SSA elimination account for about 4.8% of the total compilation time. These passes are: (i) phi-lifting, which splits the live ranges of all the variables that are part of φ-functions using "method I" due to Sreedhar *et al.* [28, pg.199]; (ii) a pass to remove critical edges; (iii) phi-coalescing, which reduces the number of copies inserted by phi-lifting using a variation of the algorithm proposed by Budimlic *et al* [8]; (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%. Our experiments show that converting a program from SSA

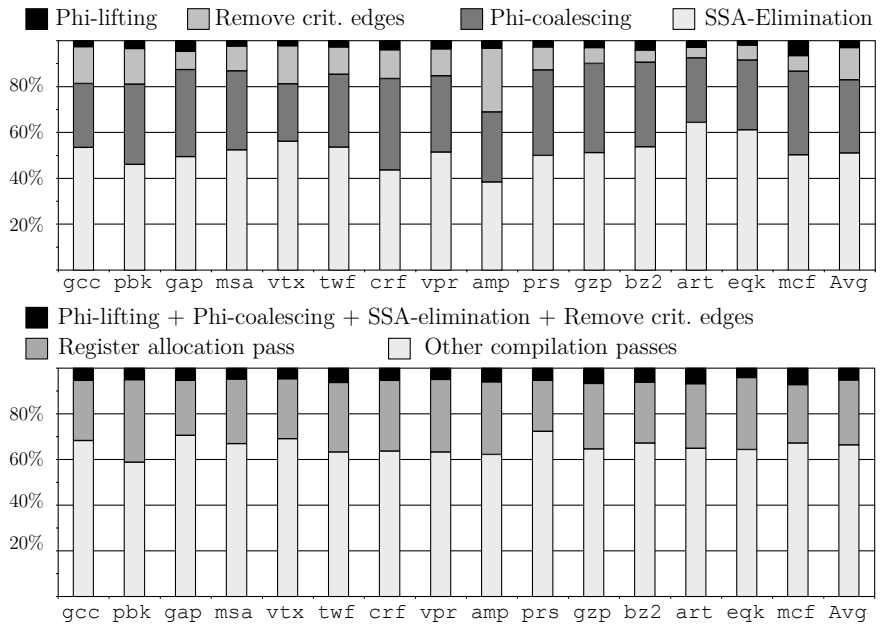


Fig. 5. Execution time of different compilation passes.

to CSSA-form is a fast process. Passes (i) and (iii) take less than 2% of the total compilation time. The conversion algorithm described by Budimlic *et al* [8] is linear space and almost linear time in the number of variables in φ-functions.

Impact of the Optimizations Figure 6 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 2). 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.

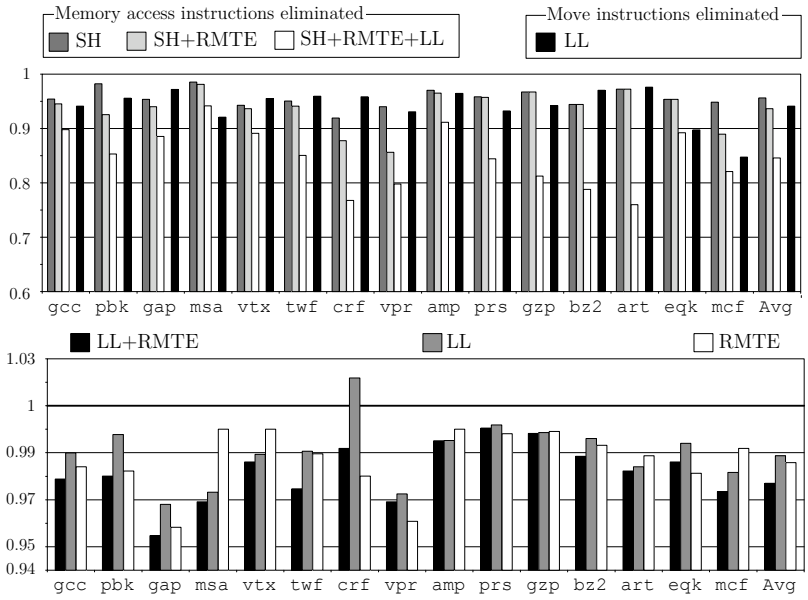


Fig. 6. Impact of Load Lowering (LL) and Redundant Memory Transfer Elimination (RMTE) on the code produced after SSA-elimination. (Up) Code size. (Down) Runtime.

The chart in the bottom part of Figure 6 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.

8 Conclusion

We have 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 to SSA elimination works for any SSA-based register allocator.

Acknowledgments. Fernando Pereira was sponsored by the Brazilian Ministry of Education under grant number 218603-9.

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. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.
4. Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In *19th International Workshop on Languages and Compilers for Parallel Computing*, pages 283–298, 2006.
5. 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.
6. Philip Brisk. *Advances in Static Single Assignment Form and Register Allocation*. PhD thesis, UCLA, University of California, Los Angeles, 2006.
7. Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):772–779, 2006.
8. 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, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–32. ACM Press, 2002.
9. 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.
10. François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. *ST Journal of Research Processor Architecture and Compilation for Embedded Systems*, 1(2):81–96, 2004.
11. Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.
12. Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *PLDI, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, 2008.
13. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC, Conference on Compiler Construction*, pages 247–262. Springer-Verlag, 2006.
14. David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–215, 2006.
15. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO, International Symposium on Code Generation and Optimization*, pages 75–88, 2004.
16. Cathy May. The parallel assignment problem redefined. *IEEE Trans. Software Eng.*, 15(6):821–824, 1989.
17. V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Proceedings of SAS, International Static Analysis Symposium*, pages 153–169, Kongens Lyngby, Denmark, August 2007.

18. Fernando Magno Quintao Pereira. *Register Allocation by Puzzle Solving*. PhD thesis, UCLA, University of California, Los Angeles, 2008.
19. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS, Asian Symposium on Programming Languages and Systems*, pages 315–329. Springer, 2005.
20. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is NP-complete. In *FOSSACS, Foundations of Software Science and Computation Structures*. Springer, 2006.
21. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2008.
22. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.
23. Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves, 2008.
24. B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27. ACM Press, 1988.
25. Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software, Practice and Experience*, 33:1003–1034, 2003.
26. Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *CC, Conference on Compiler Construction*, pages 141–155. ACM, 2007.
27. Ravi Sethi. Complete register allocation problems. In *STOC, 5th Annual ACM Symposium on Theory of Computing*, pages 182–195. ACM Press, 1973.
28. Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS, International Static Analysis Symposium*, pages 194–210. Springer-Verlag, 1999.
29. JVM Team. The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems, 2006.
30. The Jikes Team. Jikes RVM home page, 2007. http://jikesrvm.sourceforge.net/.
31. Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.
32. P. H. Welch. Parallel assignment revisited. *Software Practice and Experience*, 13(12):1175–1180, 1983.