The Designing and Implementation of A SSA-based register allocator

Fernando Magno Quintao Pereira

UCLA - University of California, Los Angeles

Abstract. The register allocation problem has an exact polynomial solution when restricted to programs in the Single Static Assignment (SSA) form. Although striking, this major theoretical accomplishment has yet to be endorsed empirically. This paper presents the first experimental results concerning a complete SSA-based register allocation algorithm. We describe novel techniques to perform register coalescing and SSA-elimination. In order to validate our allocation technique, we extensively compare different flavors of our method against a modern and heavily tuned implementation of the linear-scan register allocator. The proposed algorithm consistently produces faster code when the target architecture provides a small number of general purpose registers. For instance, we have achieved an average speed-up of 9.2% when limiting the number of registers to 7 (four general purpose, three reserved). By augmenting the algorithm with an aggressive coalescing technique, we have been able to raise the speed improvement up to 13.0%.

1 Introduction

Register allocation is the process of mapping a program that uses an unbounded number of *virtual variables* into a program that uses a fixed number of *physical registers*, in such a way that virtuals with overlapping live ranges are assigned different registers. If the number of registers is not enough to accommodate all the virtuals alive at some point of the control flow graph, some of these variables must be mapped into memory. These are called *spilled virtuals*.

The *Static Single Assignment* (SSA) form is an intermediate representation in which each variable is defined exactly once in the program code [18]. Prior work [3] has conjectured the using SSA Form could benefit register allocation due to live range splitting; however, it was only in 2005 and 2006 that several research groups [2, 5, 13] have shown that interference graphs for *regular programs* are chordal, and can therefore be efficiently colored in polynomial time. A regular program [6] is a program in SSA form with the additional property that undefined variables are never used. This result is particularly surprising, given that Chaitin et al [7] have shown that register allocation is NP-complete for general programs.

Most of the work on SSA-based register allocation remains purely theoretical. This paper presents what we believe is the first complete SSA-based register allocator. We describe the many phases that constitute the register allocation

process, from the initial assignment of physical registers to variables to the final generation of running code. We compare the proposed algorithm, and some of its variations, against a state-of-the-art implementation of the traditional linear scan used in LLVM [15], an industrial strength compiler. We have been able to compile and run a large range of benchmarks, such as `SPEC2000`, `MediaBench`, `FreeBench`, etc. In addition to the execution time of the compiled benchmarks, we use static metrics such as the number of loads and stores inserted by each register allocator in order to compare the performance of the algorithms under evaluation.

This paper also describes a novel collection of methods that we use in order to improve the quality of the code produced by our algorithm. These techniques include: (i) heuristics for performing register coalescing; (ii) a transformation of the control flow graph to facilitate the deconstruction of ϕ-functions and (iii) an aggressive analysis that can be used to reduce the number of loads and stores inserted in the final code. The objective of (iii) is to map virtuals that are related by ϕ-functions to the same memory address. The techniques (i) and (iii) can be used/removed from the algorithm in a very modular way, and, even without them, our implementation is very competitive with other register allocators.

2 Some Intuition on SSA-based Register Allocation

We illustrate the simplicity and elegance of SSA-based allocation with an example. The program in Figure 1 was taken from [16]. Figure 1 (b) is the same program, after the SSA elimination phase. Figure 1 (c) shows the sequence of assignments that would be performed by a traditional linear scan algorithm. This algorithm would traverse blocks 1, 2, 3 and finally block 4. When allocating registers in block 3, the allocator has to deal with temporaries `C` and `E` that have already been assigned machine registers in block 2: a *self imposed conflict*. The graph in Figure 1 (c) cannot be colored with two colors. Its chromatic number is 3.

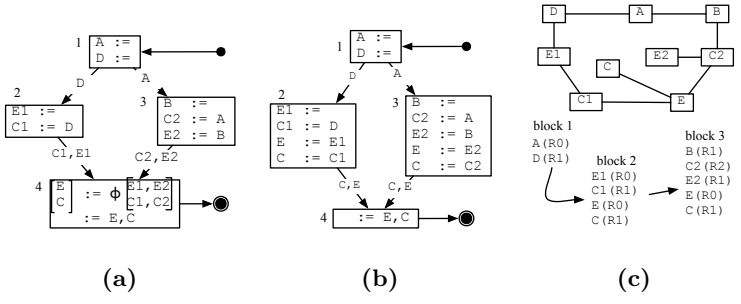

Fig. 1. (a) Example control flow graph in SSA-form [16]. (b) Program after the SSA elimination phase. (c) Interference graph and sequence of register assignments.

Because each variable in a SSA-form program is defined only once, self-imposed conflicts do not occur in SSA-based register allocation. That is, when the allocator first meets with the definition of a variable v, v cannot have been assigned a register. For a formal proof, see [14]. Figure 2 (a) outlines the dominance tree of our example program. The allocator presented in this paper traverses the dominance tree in pre-order, assigning registers to temporaries. Figure 2 (b) shows the allocation process. The interference graph of the SSA-form program can be compiled with two registers: one register less than the minimum necessary in the program after SSA-elimination! Figure 2 (c) shows the final program. Notice that we use three `xor` instructions to transfer the values of `C1` and `E1` into `C` and `E`. This example is not a coincidence: the demand of registers in the SSA-form program is never greater than the demand of registers in the original program [13]. Furthermore, a greedy allocator that traverses the dominance tree in pre-order always finds an optimal coloring, if there is no aliasing of registers and no pre-colored registers.

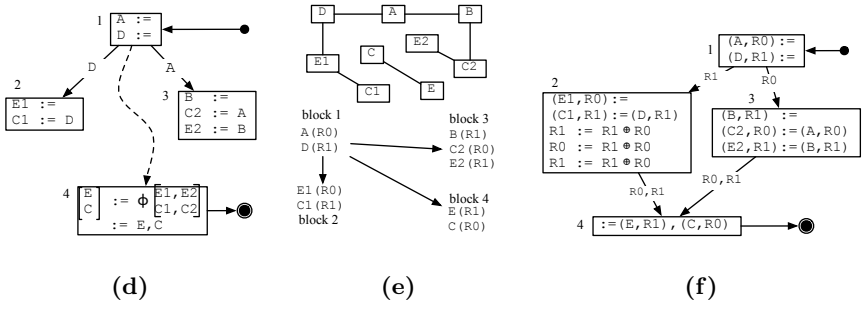

Fig. 2. (a) Dominance tree of the example program. (b) Interference graph of the SSA-form program, and assignment sequence. (c) Final program after SSA-based register allocation.

2.1 A Note on the Representation of ϕ-functions

Following the notation established by Hack et al [14], we represent the ϕ-functions existent in the beginning of a basic block as a matrix equation. Figure 3 (a) outlines the general representation of a ϕ-matrix. And Figure 3 (c) gives the intuitive semantics of the matrix shown in Figure 3 (b).

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 such as $a_i \leftarrow \phi(a_{i1}, a_{i2}, \ldots a_{im})$. Each possible execution path has a corresponding column in the ϕ-matrix, and adds one parameter to each ϕ-function. The ϕ symbol works as a multiplexer. It will assign to each element a_i of V an element a_{ij} of M, where j is determined by the actual path taken during the program's execution.

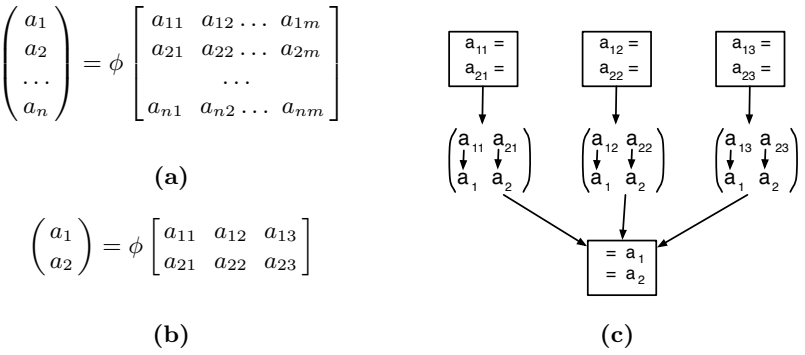

Fig. 3. (a) ϕ-functions represented as a matrix equation. (b) Matrix equation representing two ϕ-functions and three possible execution paths. (c) Control flow graph illustrating the semantics of ϕ-functions.

3 Related Work

Many industrial compilers use the SSA form as an intermediate representation: Gcc version 4.0 [12], Sun Microsystems Java HotSpot Virtual Machine [21], IBM's Java virtual machine Jikes RVM [22] and LLVM [15]. However, these compilers do not perform register allocation on SSA-form programs: instead, they require a SSA elimination pass that replaces ϕ-functions with copy instructions, and run the register allocator on the resulting code.

In 2005 Pereira and Palsberg [16] showed that 95% of the interference graphs of methods in the Java 1.5 Standard Library are chordal after the SSA elimination phase. They described heuristics for color assignment and spilling on chordal graphs, but they did not take advantage of SSA properties. The only SSA-based register allocator that has been described so far is a theoretical algorithm due to Hack et al [14]. Some differences between our algorithm and Hack et al's are (i) the way we perform register coalescing, (ii) the way we handle pre-colored registers, (iii) the spilling heuristics. However, the main difference between these algorithms is that we allow virtuals in ϕ-functions to be mapped to memory; what happens due to spilling. In Hack's case, if a parameter of a ϕ-function is spilled, and it is not possible to re-load it in the basic block from where it flows into the ϕ-function, all the other parameters and the defined variable must also be spilled. Our approach gives the compiler extra flexibility, in order to produce code of better quality, but requires a more complex ϕ-deconstruction algorithm. Once a valid register assignment has been found, neither algorithm requires further spilling during the SSA-deconstruction phase.

The name *Linear Scan* is used to designate a number of register allocation algorithms based on the work of Poletto and Sarkar [17]. The original linear scan register allocator was intended to be fast enough to be used in JIT compilers. Subsequence versions of the algorithm attempt to produce code of better quality without seriously compromising the small compilation time. Traub et al's [23]

and Wimmer et al's [24] versions handle holes in the live ranges of virtuals and split intervals to avoid spilling. Evlogimenos' [10] extensions allow to coalesce move instructions and to fold memory operands into instructions to save loads and stores. A key difference in this last work is the backtracking in face of spilling. Finally, the algorithm proposed by Sarkar et al [19] gives another polynomial time exact solution to the register allocation problem. Sarkar's algorithm does not rely on the SSA-transformation to find an optimal register assignment. Instead, it uses copy instructions and swaps, in a way very similar to the method used in our SSA-elimination phase (see Section 4.5) to split the live ranges of virtual registers. Sarkar's algorithm and SSA-based register allocation require the same number of registers, which equals the size of the maximum number of live-ranges that cross any point of the control flow graph. The main difference between all these versions of linear scan and the SSA-based register allocators is that, in the former case, register allocation occurs after ϕ-functions have been removed from the target code, whereas in the later, it happens before. Another difference is that while in a SSA-based allocator basic blocks are visited in a pre-order traversal of the dominance tree, this ordering is not required in linear scan.

The algorithm described by Cytron et al [9] produces programs in *Conventional Static Assignment Form* (CSSA). In this flavor of SSA-form, the live range of virtuals that are part of the same ϕ-function do not overlap, because they describe the same variable. The static analysis framework proposed by Sreedhar et al [20] to convert a program from non-convetional to conventional SSA-form is very similar to the transformation techniques described in Sections 4.1 and 4.3. In both cases copy instructions are used to split the live range of variables. The main difference is that Sreedhar et al's method incrementally inserts non-redundant copy instructions in the control flow graph, whereas we start with a completely partitioned program, and then proceed to the removal of redundant copy instructions until no redundancy remains. Both analysis have the same complexity and lead to similar results.

4 The Proposed Algorithm

Figure 4 gives an overview of our register allocation algorithm. We identify six main phases in the algorithm, and each of them is further detailed in this section. The grey boxes outline phases that are optional. They may improve the quality of the code produced, but may cause a degradation in the compilation time of the algorithm. Figure 4 shows the SSA deconstruction phase preceding the insertion of spill code; however, the inverse ordering is also possible.

4.1 ϕ-Lifting: Pre-transformation of the control flow graph

Our algorithm allows the partial spilling of parameters of ϕ-functions, that is, some parameters of a ϕ-function can be mapped to memory, whereas others are located in registers. In order to guarantee that no further spilling will happen

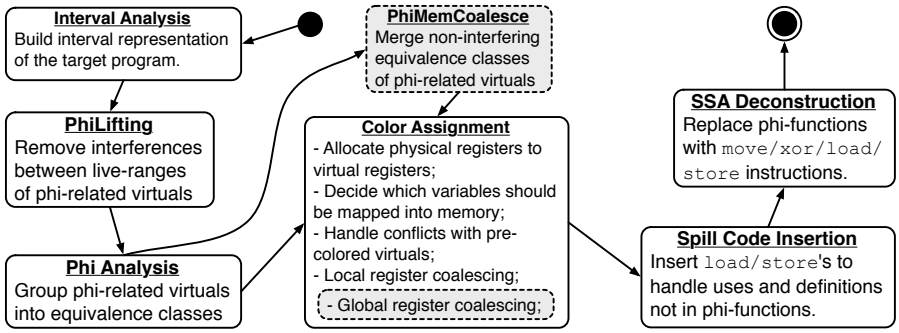

Fig. 4. Overview of the proposed algorithm.

during the SSA-deconstruction phase, we ensure that, if the parameter and the definition of a ϕ-function are located in memory, they are assigned the same memory address.

In order to facilitate our exposition, we define the equivalence class of *ϕ-related virtuals* [1] as follows: (i - reflexivity) any virtual v is ϕ-related to itself; (ii - symmetry) if virtual v is ϕ-related to virtual u, then u is ϕ-related to v; (iii - transitivity) if v_1 is ϕ-related to v_2, and v_2 is ϕ-related to v_3, then v_1 is ϕ-related to v_3. (iv) given a ϕ-function such as $v_i = \phi(v_{i1}, v_{i2}, \ldots, v_{im})$, the virtuals $v_i, v_{i1}, v_{i2}, \ldots, v_{im}$ are *ϕ-related*; Notice that the set of all the ϕ-related equivalence classes completely partition the set of virtuals in a SSA-form program. For instance, Figure 5 (a) shows an example control flow graph containing 6 ϕ-function, and three equivalence classes of ϕ-related virtuals: $\{v_1, v_3, v_4, v_5, v_6, v_7, v_9\}$, $\{v_2, v_8\}$ and $\{v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\}$.

Ideally, we would like to assign the same memory address to all the ϕ-related virtuals that have been spilled. However, this may compromise the correctness of the target program, because the live ranges of ϕ-related virtuals might overlap. Interferences between the live ranges of ϕ-related virtuals are mostly introduced by compiler optimizations such as global code motion [20] and copy folding during SSA construction [4]. For instance, assume that virtuals v_3 and v_6 have been spilled in the program show in Figure 5 (a). Although they are ϕ-related, it is not correct to store them in the same memory cell because their live ranges overlap in block 7. On the other hand, this problem does not exist if v_2 and v_8 are spilled, because they are never simultaneously alive in the target program. In order to ensure that the live ranges of ϕ-related variables do not overlap, we resort to a simple transformation of the control flow graph, which is described by the **PhiLifting** [2] algorithm given below:

[1] ϕ-related equivalence classes are called ϕ-congruence classes in [20]

[2] occasionally we will denote ϕ-functions as $a_i = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m)$, meaning that variable a_{i1} comes from block B_1.

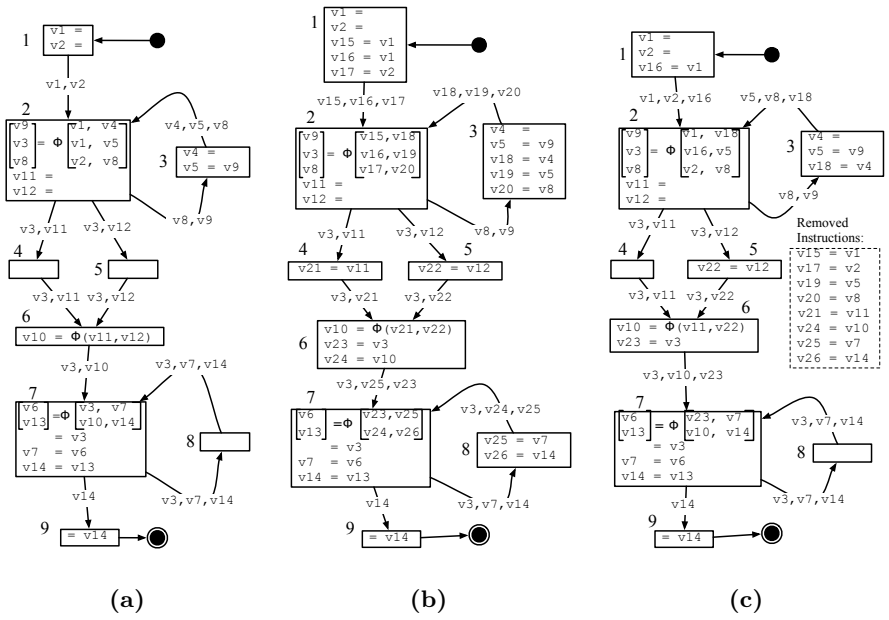

Fig. 5. (a) Control flow graph before the splitting of the live ranges of ϕ related virtuals. (b) Program transformed by **PhiLifting**. (c) Program transformed by **PhiMemCoalesce**.

PhiLifting$(a_i = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m))$
Create new virtual v_i
Add copy instruction $I = \langle a_i := v_i \rangle$ at the end of block B_j
Replace a_i by v_i in ϕ
For each virtual $a_{ij} \in \phi$,
Create new virtual v_{ij}
Add copy instruction $I = \langle v_{ij} := a_{ij} \rangle$ at the end of block B_j
Replace a_{ij} by v_{ij} in ϕ

Notice that if v_{ij} is a virtual created by the **PhiLifting** algorithm, then v_{ij} is alive only at the end of the block B_j that feeds it to its ϕ-function. Because each parameter of a ϕ-function comes from a different basic block, the control flow graph transformed by the **PhiLifting** algorithm 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 with our example, Figure 5 (b) outlines the result of applying **PhiLifting** on the control flow graph given in Figure 5 (b). The transformed program contains 14 equivalence classes: $\{v_1\}$, $\{v_2\}$,

$\{v_4\}$, $\{v_5\}$, $\{v_7\}$, $\{v_{11}\}$, $\{v_{12}\}$, $\{v_{14}\}$, $\{v_9, v_{15}, v_{18}\}$, $\{v_3, v_{16}, v_{19}\}$, $\{v_8, v_{17}, v_{20}\}$, $\{v_{10}, v_{21}, v_{22}\}$, $\{v_6, v_{23}, v_{25}\}$ and $\{v_{13}, v_{24}, v_{26}\}$.

Lemma 1. *Consider the ϕ-equation $V = \phi M$, where $V = (a_1, \ldots, a_n)$. Let the j^{th} column of M be $V_j = (a_{1j}, \ldots, a_{nj})$. We let B be the basic block where V is defined, and we let B_j be the basic block that feeds the j^{th} column of M. The register pressure at the end of B_j can never exceed the register pressure after the definition of the variables in V.*

Proof. Let $out(B_j)$ be the set of variables alive at the end of B_j. Let $S_j = \{a_{1j}, \ldots, a_{nj}\}$ be the set constituted by the elements of V_j, and let $T_j = \{t_1, \ldots, t_m\}$ be the set of variables that are alive after the definition of V, but that are not in V. We see that the set of variables alive at the end of block B_j is $out(B_j) = S_j \cup T_j$. Let $S = \{a_1, \ldots, a_n\}$. $|S| \geq |S_j|$ because $|V| = |V_j|$, and all the positions of V are filled with distinct elements, what is not necessarily the case in V_j. The set of variables alive past the definition of V is $A = S \cup T$, and by the definition of T, we have that $S \cap T = \emptyset$. We conclude that $A \geq out(B_j)$.

Theorem 1. *Let P be a program whose control flow graph does not contain critical edges.* **PhiLifting** *does not increase the register pressure in P.*

Proof. From Lemma 1 we know that the register pressure at the end of a basic block that feeds a ϕ-equation $V = \phi M$ is never greater than the register pressure after the definition of the virtuals in V. Because our target control flow graphs have no critical edges, **PhiLifting** only changes the register pressure at the end of basic blocks that feed ϕ-functions, which will be bounded by the register pressure past the definition point of those ϕ-functions.

4.2 ϕ-Analysis

The ϕ-analysis groups into equivalence classes the virtuals that are related by some ϕ-function. Because of the transformation performed by the **PhiLifiting** algorithm, the ϕ-analysis has a very efficient implementation: each ϕ-function $v = \phi(v_1, \ldots, v_m)$ already determines an equivalence class, e.g $\{v, v_1, \ldots, v_m\}$.

4.3 Aggressive Coalescing of ϕ-related virtuals

Although the **PhiLifting** algorithm produces correct programs, it is too conservative, because all the ϕ-related virtuals are used in the same ϕ-function. We now describe a coalescing technique that reduces the number of ϕ-related equivalence classes to the minimum number that do not compromise the correctness of the target program. In the algorithm **PhiMemCoalesce** below, S_I is the set of instructions created by the procedure **PhiLifting**, and S_Q is the set of equivalence classes of the program transformed by **PhiLifting**.

PhiMemCoalesce($S_I = \{I_1, I_2, \ldots, I_q\}$, $S_Q = \{Q_1, Q_2, \ldots, Q_r\}$)
For each instruction $I = \langle v_{ij} := a_{ij} \rangle \in S_I$,

$S_I := S_I \setminus I$
Let Q_v be the equivalence class of v_{ij}
Let Q_a be the equivalence class of a_{ij}
If $Q_v \cap Q_v = \emptyset$
    $S_Q := S_Q \setminus Q_v$
    $S_Q := S_Q \setminus Q_v$
    $S_Q := S_Q \cup \{Q_a \cup Q_v\}$

In order to perform operations such as $Q_i \cap Q_j$ efficiently, we rely on the interval representation of live ranges commonly used in versions of the linear scan algorithm. Each virtual is represented as a collection of ordered intervals on the linearized control flow graph. Thus, a set Q of virtuals is a set of *ordered integer* intervals. In this way, the intersection of two ϕ-equivalence classes Q_i and Q_j can be found in time linear on the number of disjoint intervals in both sets. Because a ϕ-equivalence class can have $O(V)$ disjoint intervals, the final complexity of algorithm **PhiMemCoalesce** is $|L_i| \times V$.

Figure 5 (c) illustrates the application of algorithm **PhiMemCoalesce** on the program shown in Figure 5 (b). The procedure **PhiLifting** inserts 12 copy instruction into the target control flow graph. **PhiMemCoalesce** can remove all but four of these instructions: $v_{16} = v_1$, $v_{22} = v_{12}$, $v_{18} = v_4$ and $v_{23} = v_3$. If, for example, v_1 and v_{16} were coalesced, the interfering variables v_3 and v_9 would be placed in the same ϕ-equivalence class.

4.4 The Color Assignment Phase

Live ranges of variables in a SSA-form program are contiguous along any path on the dominance tree. For instance, Figure 6 (a) shows the dominance tree of the program given in Figure 5 (c). Figure 6 (b) outlines the live range of v_3 across the path formed by the blocks 2, 6 and 7, and Figure 6 (c) depicts the live range of v_{12} along the blocks 2 and 5. This continuity allows a very simple algorithm to be used to assign physical registers to virtuals: we traverse the dominance tree of the target program allocating colors to live ranges in the order in which they appear. In the absence of pre-colored registers and aliasing this assignment is optimal [11].

The interval representation ensures that ϕ-functions are treated as parallel copies. Given a ϕ-equation $V = \phi M$, all the virtuals in the vector V are marked as alive in the beginning of the basic block where they are defined. Let S be the set of virtuals from a column of the matrix M, and assume that these virtuals are feed into the ϕ-matrix from block B. All the virtuals in S are marked as alive at the end of B, but not beyond. For example, Figure 6 (d) shows the live ranges of variables alive in blocks 1 and 2 of the program in Figure 5 (c).

Register coalescing If two virtuals v_d and v_u are related by a move instruction such as $v_d = v_u$, it is desirable that the same physical register be allocated to both v_d and v_u. In this case, the copy instruction can be removed without

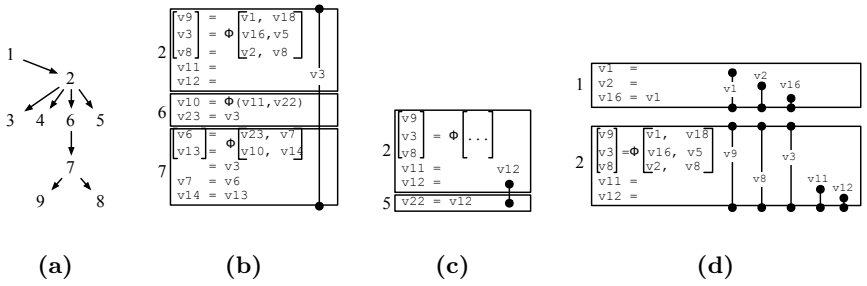

Fig. 6. (a) Dominance tree of program in Figure 5 (c). (b) Live range of v_3 along blocks 2, 6, 7. (c) Live range of v_{12} along blocks 2 and 5. (d) Live ranges of variables in blocks 1 and 2 of program in Figure 5 (c).

compromising the semantics of the program. We perform this assignment if v_u is not alive past the definition point of v_d. The coalescing in this case is always safe, given Lemma 2. Moreover, because there is no holes in the joint live range of v_u and v_d, it does not compromise the optimality of the color assignment algorithm. Although simple, this optimization is very important, because it eliminates most of the redundant instructions added by the **PhiLifting** algorithm from Section 4.1, if the **PhiMemCoalesce** pass is not used.

Lemma 2. *Two virtuals v_1 and v_2 of an strict SSA-form program interfere if, and only if, either v_1 is alive at the definition point of v_2, or vice-versa.*

Proof. See [13]

It is also interesting that the parameters and the definition of ϕ-functions be assigned the same physical register. Such assignment reduces the number of instructions necessary to eliminate ϕ-functions during the SSA-deconstruction phase. We say that a virtual v_{ij} is a *fixed point* in a ϕ-function such as $v_i = \phi(v_{i1}, \ldots, v_{ij}, \ldots, v_{im})$ if v_i and v_{ij} are assigned the same physical register during the coloring phase. It has been proved that the problem of maximizing the number of fixed points in a SSA-form program is NP-complete [14]. Therefore, we use heuristics in order to increase the number of fixed points in the final register allocated code.

For each virtual that participates in a ϕ-function, we maintain a *preference list* of physical registers. This data structure is used when necessary to assign a color to the virtual. In this case, the preference list is traversed in decreasing order, and the first available physical register is chosen. The preference list for a virtual can be computed according to the algorithm **ComputePreference**, given below. We let $l(v)$ be the location of virtual v, which may be a register (R), a memory location m, or an undefined value $\perp$ if the virtual has be been assigned a location by the register allocator.

ComputePreference v

```
Foreach physical register R
    p[r] := 0
If v is defined by a φ-function v = φ(v_1, ..., v_p),
   For each v_i ∈ φ
        if l(v_i) = R
            p[R] := p[R] + 1
For each φ-function v_d = φ(v_1, ..., v, ..., v_q)
   if l(v_d) = R
        p[R] := p[R] + 1
return p sorted in decreasing order.
```

The preference list attempts to maximize the number of fixed points once a physical register is assigned to a variable, given the information known until that moment. The worst case complexity of building the preference list is $O(V \times |\phi|)$, where $|\phi|$ is the number of ϕ-functions in the target program and V is the number of variables. In practice this complexity is $O(V)$, because each virtual tend to appear in a constant number of ϕ functions. The complexity of performing the ordering is $O(R \times \log R)$, where R denotes the number of physical registers in the target program. Given that the maximum size of each cell in the preference list is V, it can also be ordered in $O(V)$ using the bucket sort method [8].

The Spilling Heuristics If the number of live ranges in any point of the dominance tree is bigger than the number of available physical registers, then some virtual must be spilled. Most of the register allocation problems related to spilling are NP-complete, even when restricted to SSA-form program. For instance, it is NP-complete the problem of minimizing the number of registers sent to memory in an SSA-form program [25]. Again, we use heuristics to decide which variables should be mapped into memory.

We define the *weight* of a variable v as the likelihood that v will not be spilled, that is, the smaller the weight of a variable, the bigger the chance that it will be spilled. The weight of all the variables can be computed statically, according to the formula given below, where $L(inst)$ is the loop nesting depth of the basic block that contains instruction $inst$, and s is the size of v's live range, which is given by the number of instructions and control flow edges that it crosses. The weight of a variable v_i used in a ϕ-function depends on the basic block B_i that feeds it into the ϕ-function. The weight of a variable defined by a ϕ-function is the summation of the weigh of all its parameters.

```
ComputeWeigth v, s
    w := 0
    Let inst be the definition site of v
        If inst ≠ φ
           w := w + 1 + 10^L(inst)
        Else let inst = ⟨v := φ(v_1 : B_1, ..., v_p : B_p)⟩
             ∀i, 1 ≤ i ≤ p
                 If v ≠ v_i
```

$w := w + 1 + 10^{L(B_i)}$
$\forall$ *inst* $\in$ use sites of v
If $inst \neq \phi$
$w := w + 1 + 10^{L(inst)}$
Else let $inst = \langle v_{aux} := \phi(v_1 : B_1, \ldots, v : B \ldots, v_q : B_q)\rangle$
If $v_{aux} \neq v$
$w := w + 1 + 10^{L(B)}$
return w/s

The previous formula can be used to statically compute the weight of all the variables before register allocation starts. For greater accuracy, we can update the weight dynamically. As we show in Section 4.5, if the definition and parameter of a ϕ-function is in memory, no further instruction must be added to the final code during the SSA-elimination phase. Thus, it is interesting to deduct from the weight of a variable defined by a ϕ-function the contribution of the parameters that have been already spilled. To illustrate the static formula, we compute the spilling weight of each variable in Figure 5 (c). The table below outlines the nested depth of each basic block:

B_i	1	2	3	4	5	6	7	8	9
$L(B_i)$	0	1	1	0	0	0	1	1	0

The weight of each virtual is given in the following table. Notice that ϕ-functions do not contribute to the size of live-ranges. For instance, we consider that v_{10} is used immediately after it is defined. As an example, we show how the weight of v_8 is computed. The size of the live range of this virtual is 6, because it crosses the definition points of v_{11}, v_{12}, v_4 and v_5. Also, it is alive in the two edges between blocks 2 and 3. The virtual is defined by a ϕ-function; however, it is also the second parameter of the same ϕ-function. Thus, case v_8 is spilled, nothing will have to be done for the second entry of that ϕ-function. The contribution of v_2 is $1 + 10^{L(B_1)} = 2$. The final number is $2/6 = 0.3$.

v	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{16}	v_{22}	v_{23}
$\Sigma(L(B_i))$	6	4	26	22	22	24	22	2	24	6	13	2	24	24	4	4	4
s	3	2	14	2	1	1	3	6	4	2	3	1	2	3	1	1	1
$w(v)$	2	2	1.86	11	22	24	7.3	0.3	6	3	4.3	2	12	8	4	4	4

Handling Pre-colored Virtual Registers The calling conventions adopted by the compiler, and constraints imposed by the computer architecture may require that specific registers be allocated to some virtuals. These virtuals are called *pre-colored*. The polynomial solution to SSA-based register allocation does not support pre-colored registers. Indeed, a general pre-coloring of a chordal graph cannot be extended to a full coloring in polynomial time, unless $P = NP$ [1][3].

[3] This problem is polynomial if each color appears at most once in the pre-coloring [1]

In order to keep the polynomial time exact solution, it is possible to further partition the control flow graph by surrounding instructions that contain pre-colored virtuals with single parameter ϕ-functions [13]. A similar approach is adopted by [19]. We did not adopt this approach because it would require us to change the flavor of SSA representation used in our compiler framework. With the objective of keeping our implementation simple, we avoid assigning to a virtual v any physical register r if the live range of v overlaps the live range of any other virtual pre-colored with r. We build a small interference graph in order to check for conflicts between virtuals and physical registers. This graph is one order of magnitude smaller than the complete interference graph. Table 1 compares these sizes for some programs of SPEC2000 tested with our algorithm.

	`gzip`	`vpr`	`mcf`	`parser`	`bzip2`	`twolf`	`crafty`	`art`	`ammp`	`equake`
(a)	25716	198816	6508	138304	20978	206267	213653	15285	176899	15742
(b)	339771	3538532	67353	1224197	499925	5243067	2445314	191638	1453721	301670
b/a	13.21	17.80	10.35	8.85	23.83	25.42	11.44	12.54	8.22	19.16

Table 1. Comparison between the sizes of (a) the graph of interferences between virtuals and physical-registers and (b) the complete interference graph.

4.5 The SSA-deconstruction Phase

After the color assignment phase, every virtual register in the original program has been assigned one physical location, which is either a physical register, or a memory address, case the virtual has been spilled. Given an equation such as $V = \phi M$, we let $(a_1, a_2, \ldots, a_n) = \phi(a_{1j}, a_{2j}, \ldots, a_{nj})$ denote the transference of the values from the j^{th} column of the matrix. We let B_j denote the block that feeds these values. The SSA-elimination phase consists in inserting instruction in the target program to transfer the value stored in $l(a_{ij})$ to $l(a_i)$.

The algorithm used to eliminate ϕ-functions can be divided into two steps. In the first step we use copies, stores or loads to transfer values whenever it is safe to do it. In the second step we use sequences of swap instructions to transfer the coloring of the remaining variables. For the purposes of this discussion we assume a control flow graph without critical edges. The *safe operations* used in the first part of the algorithm are given in the table below. We let Q_j be the set $\{a_{1j}, a_{2j}, \ldots, a_{nj}\}$ of parameters coming from block B_j, and we let Q be the set $\{a_1, a_2, \ldots, a_n\}$ of variables defined by the ϕ-equation. After inserting a safe operation to resolve the transference $a_i \leftarrow a_{ij}$, we remove a_i from Q and we remove a_{ij} from Q_j. The safe operations are used until no longer possible [4]:

[4] The 'do nothing' operation can be seen as an instance of register coalescing.

	$l(a_i)$	$l(a_{ij})$	pre-condition	Operations
1	R	m		`store R in m`
2	m	R	$\nexists v \in Q_j, l(v) = R$	`load R from m`
3	R_x	R_y	$R_x = R_y$	do nothing
4	R_x	R_y	$\nexists v \in Q_j, l(v) = R_x$	`Rx := Ry`
5	m	m		do nothing

Lemma 3. *If no safe operation applies, then there is no virtual a_i such that $l(a_i) = m$. .*

Proof. Immediate. □

In the final step of the algorithm, swap instructions are used to transfer the values stored in registers. if $l(a_i) \neq l(a_{ij})$, we say that virtuals a_i and a_{ij} are *miscolored*. An interesting fact is that, if no safe operations applies, then the set of registers used in the miscolored shorts constitute a perfect permutation. This is proved in Lemma 4. Therefore, the coloring can be fixed with at most $n - 1$ swaps, where n is the number of miscolored variables.

Lemma 4. *If no safe operation applies in the partition point between cliques Q_j and Q, and if $a_i \leftarrow a_{ij}$ is a pair of miscolored variable, then there exists $a_x \in Q$ and $a_{yz} \in Q_j$ such that $l(a_x) = l(a_{ij})$ and $l(a_{yz}) = l(a_i)$.*

Proof. If there were $a_i \leftarrow a_{ij}$ such that $l(a_i)$ is not used by any virtual of Q_j, then either safe operation 2 or safe operation 4 would apply. From Lemma 3 we have that $\forall a_x \in Q, l(a_x) \neq m$. If $l(a_{ij})$ is not used by any virtual in Q, then there exists at least one register R used by some virtual from Q that is not used by any virtual from Q_j, and either safe operation 2 or 4 applies. □

Lemma 5. *If Q and Q_j share n pairs of miscolored variables, the coloring can be transfered with at most n operations.*

Proof. The proof is by induction on n.

Basis: $n = 1$. In this case, there exists a virtual a_i in Q such that $l(a_i) = R_i$ and there exists a virtual a_{ij} such that $l(a_{ij}) = R_j$. Because the only miscolored variable in Q_j is a_{ij}, the register R_i cannot be in use by any other register of Q_j. Therefore, we can transfer the coloring with either safe operation 2 or 4.

Induction Hypothesis: the lemma is true for up to n miscolored registers.

Inductive Step: assume that there exists $n+1$ miscolored pairs of variables between Q and Q_j. Let $a_i \leftarrow a_{ij}$ be one of these pairs, so that $l(a_i) = R_i$ and $l(a_{ij}) = R_j$. If R_i is not in use by any miscolored variable of Q_j, we can perform the transfer via either a copy or a load instruction. Otherwise R_i is miscoloring another virtual a_{xy}, and we can transfer the coloring by means of a swap sequence. After the transfer of a_{ij}'s color, this virtual can be removed from the set of miscolored variables, and the result follows by the induction hypothesis. □

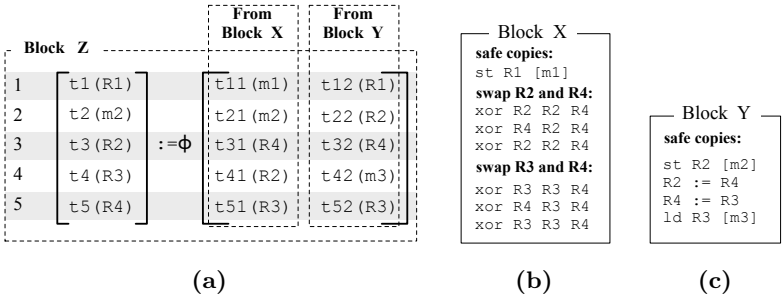

Fig. 7. (a) Five ϕ-functions joining two basic blocks. (b) Tail of block X, after ϕ-elimination. (c) Tail of block y, after ϕ-elimination.

We have implemented swaps of integer registers as sequences of three `xor` instructions. A backup register must be used to swap floating point values, as proposed by Briggs et al [4]. Figure 7 illustrates our SSA-elimination algorithm. Notice that there is a permutation of the physical registers `R2, R3, R4` between the definition vector and the column feed by block X. Such permutation cannot use safe operations, and must be eliminated by a sequence of register swaps. The transferring of values between block Y and block Z can be completely implemented via safe operations.

4.6 Spill Code Placement

After a valid color assignment has been found, it is necessary to insert load and store instructions in order to transfer values to and from memory. Because of the single assignment property, only one store is necessary to preserve the definition of a spilled register. We attempt to recycle loads among different uses of the same spilled register whenever possible. This optimization is only applicable inside a basic block. This phase only handles spilled virtuals used or defined in instructions other than ϕ-functions. Further loads and stores are inserted during the SSA-elimination phase in order to deal with ϕ-related virtuals that have been spilled. Notice that this phase is completely independent of the SSA-elimination step, and these two stages can be executed in any order.

We do not spare registers for the insertion of spill code, e.g loads and stores. Yet during the coloring phase, if a variable v is spilled, each use of v is replaced by a fresh virtual v', which is mapped to the same memory location as v. The register allocated to v' will be the target of the load instruction.

5 Step-by-step Example

We assume a hypothetical machine with only two registers X and Y and use the program in Figure 5 to illustrate the several phases of our algorithm. Figure 8

(a) outlines the sequence of events that would take place during the color assignment phase. Basic blocks are visited in a depth-first traversal of the dominance tree. The order in which blocks are visited, in our example, coincides with their numeric labels. For completeness we point out the sites where variables were spilled. For instance, when block 1 is visited, virtual v_1 is assigned register X, virtual v_2 is given Y and is spilled so that its register can be allocated to v_{16}. Notice that v_1 and v_2 have the same spilling weight (see table in Section 4.4); the choice of v_2 was arbitrary in this case.

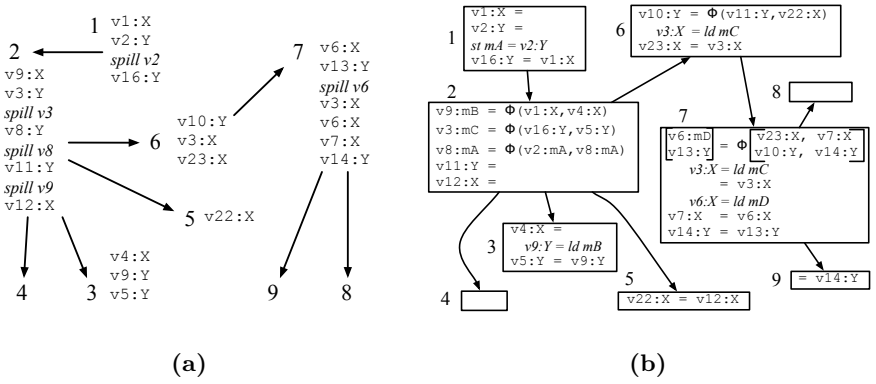

Fig. 8. (a) The color assignment phase. (b) Program after placement of spill code.

Figure 8 (b) shows the program after the coloring phase and the placement of code to handle spills. Notice that the register assigned to the definition of a ϕ-function that has been spilled becomes irrelevant on this phase. For instance, v_9 had been originally allocated the register X; however, we mark its location as a memory address in order to guide the SSA-deconstruction algorithm.

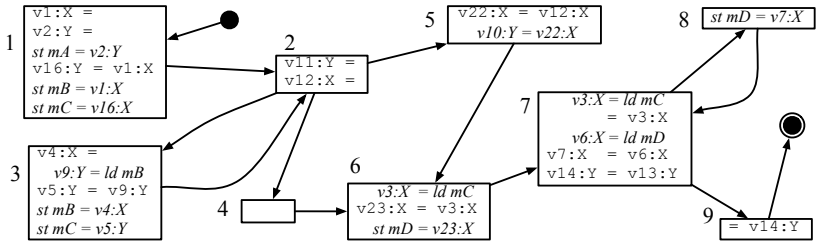

Fig. 9. The code produced after the SSA-deconstruction phase.

Finally, we proceed to the SSA-deconstruction phase, whose result is shown in Figure 9. Stores have been inserted at the end of blocks 1, 3 and 6 to handle spilled virtuals defined by ϕ-functions. A single copy instruction was inserted at the end of block 5. Notice that the final program does not use the names of virtuals. We show them in Figure 9 to facilitate the understanding of the ϕ-deconstruction process. We point also that the definitions of v_8 and v_{13} could be completely removed due to register and memory coalescin. The task of removing redundant copies such as $v5 : Y = v9 : Y$ in block 3 is left for the code generator.

6 Experimental Results

We have implemented the proposed algorithm in C++, using the LLVM [15] platform. Our implementation does not use LLVM's data structures directly. Instead, we dump the compilers' intermediate representation in our framework, process it, and map the results back into LLVM's back end. We opted for this strategy in order to facilitate the development and debugging of our allocator. Our implementation has about 4,000 lines of code (about 2,600 lines of uncommented code), including code responsible for implementing liveness analysis, phi-deconstruction, removal of critical edges and auxiliary data structures (lists, trees, etc). We have used two collection of benchmarks in our experiments. The first set of programs come from SPEC-CPU 2000 (`gzip, vpr, parser, crafty, mcf, twolf, bzip2, art, ammp`, and `equake`). The last three applications use floating point operations. The second set is the LLVM's test suite, a collection of 214 applications that include benchmarks such as `Fhourstones, FreeBench, MallocBench, Prolangs-C, ptrdist, mediabench, CoyoteBench`, etc. The LLVM test suite has provided us with 1,172 .c/.cpp files and 590 .h files, comprising 641,708 lines of C code. The hardware used for the tests is a Dual 1.25 GHz powerPC G4 with 2MB L3 cache and 1.25 GB DDR SDRAM running Mac OS X version 10.4.8. We have compiled and run our benchmarks using seven different register allocators:

- `(Simple)`: the naive algorithm, that spills every temporary.
- `(Local)`: performs local allocation, that is, it attempts to keep values in registers along the same basic block, and spills the variables alive at the block boundaries.
- `(LLVM)`: the LLVM default algorithm. This is a modern version of linear scan. Before the register assignment phase, it executes an aggressive coalescing pass. Holes in the live ranges of variables are filled with other virtuals whenever possible. One particular characteristic of this algorithm is that is backtracks in the presence of spills: if a variable v is spilled, the allocation restarts from the beginning of v's interval. This optimization avoids reserving registers to place memory transfer code. As a further optimization, the algorithm tries to use spilled values directly from memory whenever it is possible, in order to minimize the number of loads/stores inserted due to spilling. For a detailed description of the algorithm, see [10].

– `Ch -O1` the SSA-based register allocator without the coalescing heuristics described in Section 4.4.
– `Ch -O2` SSA-based register allocation with register coalescing.
– `Ch -O2` SSA-based allocator with register coalescing and coalescing of ϕ-related virtuals in memory, as described in Section 4.3.
– `GCC -O2` The gcc compiler (Notice that this is a different back-end then the one used in the other algorithms).

Figure 10 gives the running time of the compiled benchmarks. The upper part of Figure 10 compares the proposed algorithm against LLVM's register allocator when producing code for a machine with few registers. In this case we have limited the number of integer general purpose registers in the PowerPC register bank to only four. The bars give the absolute running time of all the 214 compiled applications. The next set of bars compare the seven register allocator when producing code for PowerPC with all the 32 integer registers available. On the bottom of the figure we have separated the applications whose running time, when compiled with our algorithm, was greater than 5 seconds. This happened for 33 applications, which count for about 94% of the total execution time. We also display the five largest absolute running times that we found in the collection of benchmarks, in order to further compare the proposed algorithm against LLVM's register allocator.

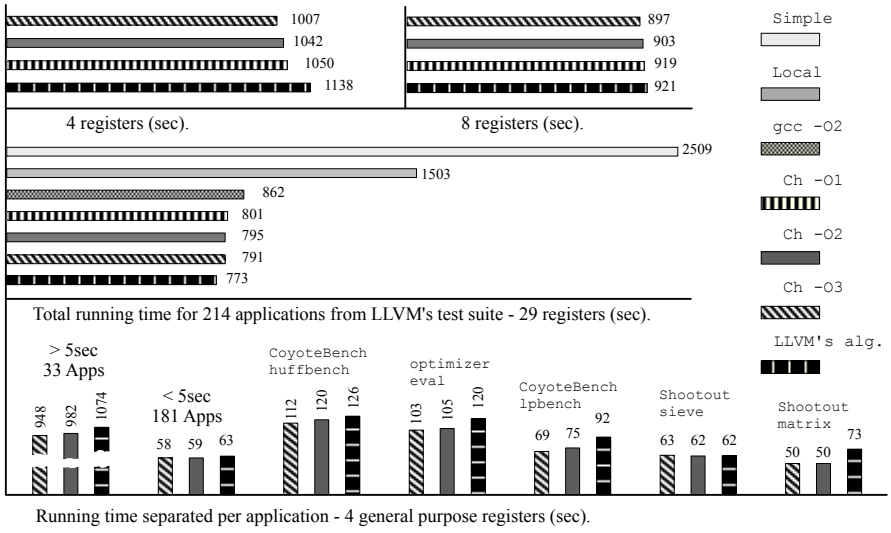

Fig. 10. Total execution time for 214 applications compiled with different register allcators.

Table 2 shows the results obtained during the compilation of programs from the SPEC-CPU 2000 set of benchmarks. We present static data and the running times of the compiled programs. The static data consists of the number of stores, loads, moves and xors present in the final assembly code. Notice that some of these instructions have not been inserted by the register allocator, but are part of the program itself. The results in Table 2 have been obtained after limiting the number of integer registers available for allocation to only four [5]. We compare `Ch -O1`, `Ch -O2` and `Ch -O3` against LLVM's implementation. The final assembly code produced by LLVM's back end was compiled with gcc (without further optimizations) in order to be executed in our target machine. The first table (lines 1-6) was produced by `Ch -O1`, thus the bigger number of copies, and, most notably, xor instructions in the final code. The second table (lines 7-12) was generated by `Ch -O2`. Lines 13-18 have been obtained with `Ch -O3`. The last table (lines 19-23) was found via LLVM's register allocator. The `Ratio` rows contain the execution time of the code produced with the LLVM's algorithm divided by the execution time produced by the different versions of our algorithm. Table 4 compares compilation results for the same set of SPEC-CPU 2000 programs, but this time with the 32 registers available.

SSA-based register allocation is effective for architectures with few registers : Our academic implementation was able to outperform the industrial quality implementation of LLVM in every case when the number of registers in the target architecture was reduced to only four. In the tests performed on the LLVM set of benchmarks, the gain in performance was approximately 9.5%, whereas the experiments in the SPEC2000 programs produced performed improvements as high as 44%, as in `bzip2`.

In our experiments, 12 registers seem to be the performance threshold between register allocation before and after SSA-elimination : the SSA-based algorithm trades copy instructions for a smaller number of spilled variables. Due to the splitting of live ranges, SSA-based register allocation tends to insert more copy instructions into the target code than the traditional linear scan algorithm. Nevertheless, if the register pressure is high, the SSA-based allocator tends to spill less. Because copy instructions are cheaper than memory accesses, this tradeoff is beneficial when the number of spills is large. However, in an architecture plenty of physical registers, spills barely happen, yet the extra move/xor instructions remain as a burden in the SSA-based allocator. The chart shown in Figure 11 compares our algorithm and the LLVM's implementation with four different sizes of register banks: 4, 8, 12 and 16. Around 12 registers the fewer number of spills stop compensating for the large number of `move` and `xor` instructions that the SSA-based algorithm requires to keep the variables from been spilled.

[5] The only exception been `crafty`, whose compilation required a minimum of eight registers in our system

		gzip	vpr	mcf	parser	bzip2	twolf	crafty	art	ammp	equake
1	Store	1516	8060	547	4812	1263	13571	6857	748	3370	555
2	Load	2077	16277	899	9085	1883	23688	15142	1050	6749	1068
3	Move	684	7563	221	3754	566	6088	7024	427	3687	814
4	Xor	221	849	55	474	125	1219	1518	13	314	27
5	Time	135.574	520.678	538.992	48.022	679.478	22.034	6.518	402.654	19.948	19.768
6	Ratio	1.08	1.08	1.02	1.02	1.39	1.24	1.14	1.04	0.99	1.12
7	Store	1516	8467	548	4939	1264	13592	6862	749	3373	556
8	Load	2078	16586	900	9171	1885	23701	15150	1053	6753	1070
9	Move	652	6862	203	3571	512	5446	6120	318	3437	457
10	Xor	86	492	13	174	50	514	951	4	106	4
11	Time	133.476	519.426	538.374	48.914	673.588	21.373	6.442	400.966	19.202	19.655
12	Ratio	1.10	1.09	1.02	1.00	1.41	1.28	1.15	1.05	1.02	1.12
13	Store	1254	6982	482	4420	1095	10812	6213	624	3186	546
14	Load	1836	15196	827	8690	1734	22028	14578	927	6535	1057
15	Move	630	6704	193	3478	493	4996	5997	310	3366	440
16	Xor	62	468	19	114	35	433	771	18	115	4
17	Time	129.784	471.407	502.60	47.845	655.875	20.963	6.409	375.760	19.156	19.102
18	Ratio	1.13	1.19	1.09	1.02	1.44	1.30	1.16	1.12	1.03	1.16
19	Store	1664	9093	576	5346	1380	13820	8673	789	3771	738
20	Load	2260	16592	911	9318	2064	24314	15996	1099	6583	1278
21	Move	201	3597	77	1631	197	2554	2190	176	1810	218
22	Xor	23	363	7	75	29	325	621	1	82	1
23	Time	146.708	564.297	549.378	48.817	947.494	27.266	7.428	421.587	19.712	22.186

Table 2. Static data collected for SPEC2000 with four general purpose registers. (1-6): `Ch -O1`. (7-12): `Ch -O2`. (13-18): `Ch -O3`. (19-23): `LLVM's alg`.

7 Conclusion

This paper has presented a SSA-based register allocator. This algorithm produces code of good quality, and clearly outperform the traditional linear-scan implementation if the number of available registers is not large. We have also described coalescing heuristics and a static analysis that can be used to improve the quality of the assembly code produced by our allocator. Further details about this project, and the implementation of the whole algorithm are available at `http://compilers.cs.ucla.edu/fernando/projects/soc/`.

References

1. M Biró, M Hujter, and Zs Tuza. Precoloring extension. i:interval graphs. In *Discrete Mathematics*, pages 267 – 279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".
2. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.

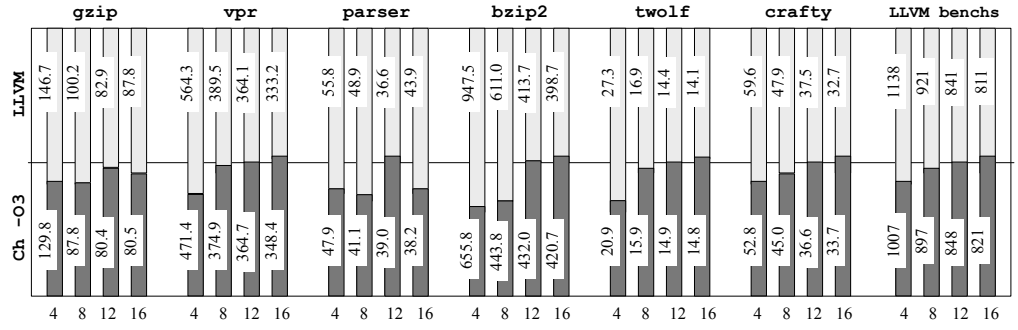

Fig. 11. Comparison between `Ch -O3` and LLVM's algorithm for architectures with different number of registers.

3. Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.
4. 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.
5. Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Transactions on Computer-Aided Design, Special Issue on IWLS*, 2006.
6. 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.
7. Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
8. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.
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. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
10. Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.
11. Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46 – 56, 1974.
12. Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.
13. Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.
14. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.
15. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Symposium on Code Generation and Optimization*, pages 75–88, 2004.

16. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.
17. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.
18. B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.
19. Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages X–X. ACM, 2007.
20. 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.
21. JVM Team. The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems, 2006.
22. The Jikes Team. Jikes RVM home page, 2007. http://jikesrvm.sourceforge.net/.
23. Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Conference on Prgramming Language Design and Implementation (PLDI)*, pages 142–151, 1998.
24. Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.
25. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133 – 137, 1987.

		gzip	vpr	mcf	parser	bzip2	twolf	crafty	art	ammp	equake
1	Spill	247	1657	93	750	288	3326	3222	138	637	150
2	stack	84	394	100	96	202	385	421	104	74	169
3	Store	780	5274	364	2970	699	13479	6914	437	2252	362
4	Load	1173	12126	617	6363	1171	22333	15265	738	4746	782
5	Move	916	8118	257	4627	685	7158	6534	380	4046	482
6	Xor	356	1170	94	666	245	2245	2211	189	634	124
7	Time	89.93	375.38	541.87	41.79	453.12	16.27	44.46	330.97	18.20	17.51
8	Ratio	1.15	1.03	0.98	1.16	1.35	1.07	1.10	0.97	1.00	0.98
9	Spill	247	1657	93	750	288	3326	3223	139	636	150
10	stack	84	392	100	92	202	384	421	105	74	169
11	Store	780	5275	364	2970	699	13489	6911	438	2255	362
12	Load	1174	12130	617	6365	1172	22332	15224	739	4811	782
13	Move	823	7269	216	4302	599	6048	2190	388	3693	467
14	Xor	71	474	25	147	53	814	621	18	124	1
15	Time	88.83	378.22	536.43	41.41	448.78	16.28	44.41	317.99	18.13	17.32
16	Ratio	1.16	1.03	0.99	1.18	1.36	1.06	1.10	1.01	1.00	0.99
17	Spill	192	1509	75	673	247	2231	2664	125	613	146
18	stack	66	316	82	77	164	285	308	95	67	165
19	Store	699	4559	344	2876	637	7179	6213	401	2241	359
20	Load	1092	11413	597	6274	1109	15985	14578	701	4797	779
21	Move	807	7246	201	4232	602	5896	5997	352	3623	461
22	Xor	53	423	25	99	38	472	771	18	109	1
23	Time	88.78	374.87	537.94	41.12	443.81	16.04	45.06	305.77	18.04	17.02
24	Ratio	1.16	1.04	0.99	1.18	1.38	1.08	1.08	1.05	1.01	1.01
25	Spill	301	2241	120	990	369	3643	5260	197	717	252
26	stack	107	421	127	177	245	405	526	159	102	271
27	Store	855	5368	401	3315	802	8511	8673	507	2503	494
28	Load	1253	11146	622	6264	1312	16610	15996	733	4511	970
29	Move	219	3326	90	1575	180	2688	2190	164	1836	160
30	Xor	23	363	7	75	29	325	621	18	82	1
31	Time	103.13	388.05	533.27	48.71	611.15	17.36	48.83	322.28	18.15	17.18

Table 3. Static data collected for SPEC2000 with eigth general purpose registers. (lines 1-8): `Ch -O1`. (lines 9-16): `Ch -O2`. (lines 17-24): `Ch -O3`. (lines 24-31): `LLVM's alg`. Time is given in seconds.

		gzip	vpr	mcf	parser	bzip2	twolf	crafty	art	ammp	equake
1	Store	517	3037	231	2113	379	4618	3416	291	1778	219
2	Load	740	7989	406	4407	521	9678	12342	508	3630	548
3	Move	1001	9790	331	5599	756	8233	8352	532	4489	503
4	Xor	395	1365	115	579	365	1026	2862	129	529	184
5	Time	78.576	341.041	378.432	38.191	437.874	14.529	31.599	298.511	17.822	16.983
6	Ratio	0.896	0.96	1.01	1.11	0.92	0.95	0.88	1.00	0.98	0.95
7	Store	517	3049	231	2120	379	4698	3381	291	1779	218
8	Load	740	8004	406	4398	511	9732	12318	508	3631	540
9	Move	1001	8491	270	5171	693	7828	7431	448	4076	506
10	Xor	395	471	31	129	65	763	852	18	118	1
11	Time	77.232	350.267	367.978	38.409	436.557	14.570	29.886	298.793	17.763	16.390
12	Ratio	0.911	0.93	1.03	1.11	0.92	0.95	0.93	1.00	0.99	0.99
13	Store	517	3037	231	2111	372	4887	3391	289	1778	219
14	Load	739	7991	406	4403	514	10404	12315	506	3629	548
15	Move	896	8353	259	5097	669	7769	7316	415	3979	490
16	Xor	53	435	34	120	65	667	708	21	118	1
17	Time	76.945	346.275	368.010	37.782	436.126	14.544	29.534	292.089	17.720	16.414
18	Ratio	0.915	0.94	1.04	1.12	0.92	0.95	0.94	1.02	0.99	0.99
19	Store	472	3062	231	2118	368	4553	3380	292	1780	228
20	Load	726	7480	405	4365	491	9849	12187	473	3671	552
21	Move	282	3707	140	2056	277	3864	2939	224	4076	210
22	Xor	23	363	7	75	29	325	621	18	82	1
23	Time	70.419	326.552	381.434	42.584	402.785	13.880	27.751	298.707	17.494	16.204

Table 4. Static data collected for SPEC2000 with 29 general purpose registers. (1-6): `Ch -O1`. (7-12): `Ch -O2`. (13-18): `Ch -O3`. (19-23): `LLVM's alg`.