Decoupled Graph-Coloring Register Allocation with Hierarchical Aliasing

André L. C. Tavares
DCC - ICEx/UFMG
andrelct@dcc.ufmg.br

Quentin Colombet
ENS Lyon
quentin.colombet@ens-lyon.fr

Mariza A. S. Bigonha
DCC - ICEx/UFMG
mariza@dcc.ufmg.br

Christophe Guillon
STMicroelectronics
christophe.guillon@st.com

Fernando M. Q. Pereira
DCC - ICEx/UFMG
fpereira@dcc.ufmg.br

Fabrice Rastello
ENS Lyon
fabrice.rastello@ens-lyon.fr

ABSTRACT

Recent results have shown how to do graph-coloring-based register allocation in a way that decouples spilling from register assignment. This decoupled approach has the main advantage of simplifying the implementation of register allocators. However, the decoupled model, as described in previous works, faces many problems when dealing with register aliasing, a phenomenon typical in architectures usually seen in embedded systems, such as ARM. In this paper we introduce the semi-elementary form, a program representation that brings decoupled register allocation to architectures with register aliasing. The semi-elementary form is much smaller than program representations used by previous decoupled solutions; thus, leading to register allocators that perform better in terms of time and space. Furthermore, this representation reduces the number of copies that traditional allocators insert into assembly programs. We have empirically validated our results by showing how our representation improves two well known graph coloring based allocators, namely the Iterated Register Coalescer (IRC), and Bouchez *et al.*'s brute force (BF) method, both augmented with Smith *et al.* extensions to handle aliasing. Running our techniques on SPEC CPU 2000, we have reduced the number of nodes in the interference graphs by a factor of 4 to 5; hence, speeding-up allocation time by a factor of 3 to 5. Additionally the semi-elementary form reduces by 8% the number of copies that IRC leaves uncoalesced.

Categories and Subject Descriptors

H.4 [**Programming Language Applications**]: Register Allocation—*complexity measures, performance measures*

General Terms

Algorithms, Experimentations

1. INTRODUCTION

Register allocation is the problem of finding storage locations to the values manipulated by a program. Traditional computer architectures provide two storage alternatives: memory and registers. Registers are much faster, yet, they come in very small number. For instance, the ARM processor contains only twelve general purpose registers. Therefore, compilers should use registers judiciously. If the allocator is unable to find a register for a variable, then this variable is stored in memory – an event normally called *spilling*.

Many recent register allocation algorithms follow a *decoupled approach* that separates spilling from register assignment [1, 12, 13, 19, 20, 23, 24, 27]. This model has important advantages. First, the separation between these two phases yields simpler and more modular implementations: different spilling heuristics can easily be combined with different register assignment and coalescing methods. Second, the fact that the local register pressure is easy to infer in decoupled designs, simplifies compiler optimizations that might change register pressure leading to further spilling, such as redundancy elimination, code motion and pre-pass scheduling. The local register pressure is the number of registers necessary to allocate the variables alive at some program point. Key to decoupled models is the concept of *live range splitting*, which allows allocating a variable to different registers along distinct parts of its live range. This very notion of live range splitting makes it difficult to extend decoupled algorithms to architectures with aliased register banks.

Quoting Smith *et al.* [26], "two register names alias when an assignment to one register name can affect the value of the other". Aliasing is present in four general purpose x86 registers: AX, BX, CX and DX. Each of these registers has two aliases, e.g., the 16-bit register AX is divided into two eight-bit registers: AH and AL. Aliasing is also found in floating point registers of many architectures typical of the embedded world, such as ARM, where single precision registers combine to make double precision ones. Architectures like ARM Neon go further, allowing the combination of two doubles into a quad-precision register. There exist also more irregular architectures, such as the Carmel model, used in digital signal processors, showing overlapping registers of 16, 32 and 40 bits [25].

In order to apply decoupled register allocation onto architectures with aliasing, it is necessary to perform live range splitting. Previous solution would split live ranges between each pair of consecutive instructions [20], creating a program representation called *Elementary Form*. However, this level of live range splitting makes traditional register allocators, like those based on PBQP [25], ILP [15] or graph coloring [8], impractical, because the number of program variables increases too much.

In this paper we solve this problem introducing a program representation that we call *Semi-Elementary Form*. Programs in this format provide the essential property required by a decoupled register allocator: if the local register pressure is lower than the number of available registers at every program point, then spill-free register allocation is possible for the whole program without requiring any additional live-range splitting. Because the semi-elementary form does much less live range splitting than the original elementary form, it fosters decoupled allocators that are faster, require a smaller memory footprint and, as a side effect, yield better register coalescing when submitted to traditional coalescing heuristics. We also introduce a way to merge the live ranges of variables – the *local merging test* – which reduces even more the size of the program's interference graphs, and speeds-up allocation time considerably. Finally, we provide as a bonus an improved spilling test, that might produce less spilling than the simplification heuristics traditionally used in graph-coloring based register allocation.

The semi-elementary form speeds up register allocation; however, it is not a new register allocation algorithm. Hence, it is not meant to increase the performance of the assembly code produced in any substantial way. Although it has the side effect of reducing the number of copies in the final assembly code, this reduction is too small to provide performance gains. Nevertheless, it considerably simplifies register allocation, and we believe that this is the best way to handle register aliasing in decoupled allocators. To substantiate this claim, we have adapted two different graph coloring-based register allocators to run in a decoupled fashion: George and Appel's Iterated Register Coalescer [11] and Bouchez *et al.*'s Brute Force Coalescer [6]. We show, via experiments, that building semi-elementary form programs is fast. Furthermore, allocators working on semi-elementary form programs consume much less memory than elementary-form based approaches, and are much faster. In our experiments we compile the SPEC CPU 2000 benchmarks to miniIR assembly, using 8, 16 and 32 aliased registers.

The rest of this paper is organized as follows: Section 2 explains in more details decoupled register allocation. Section 3 introduces our contributions, and Section 4 provides experimental data supporting our techniques. Finally, Section 5 concludes this paper.

2. REGISTER ALLOCATION VIA GRAPH COLORING

Register allocation is a problem with many different solutions; however, the most popular approach seen in the literature is based on graph coloring. This model was first introduced by Gregory Chaitin in the early eighties [8]. In order to do register allocation we color the *interference graph* of

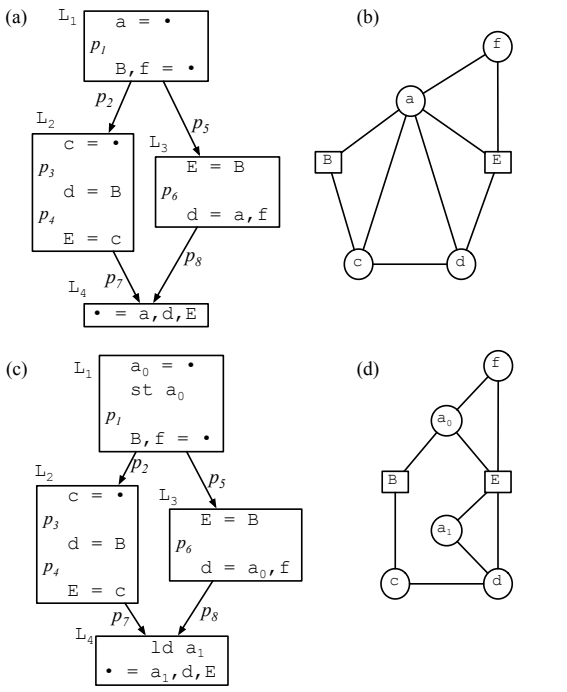

Figure 1: Traditional graph-coloring-based register allocation. (a) Example program. (b) Program's interference graph; square nodes plus upper case letters denote double precision values. (c) Program after spilling variable a. (d) New interference graph.

the source program. The interference graph contains a vertex for each variable in the source program, and two vertices are adjacent if, and only if, their corresponding variables are alive at the same program point. Here, a program point is the region between two consecutive instructions. A variable v is alive at a program point p if the control flow graph of the program contains a path from p to an instruction that uses v, which does not cross a place where v is redefined. Figure 1(a) shows an example program, and Figure 1(b) outlines its interference graph. In this example, we assume that lower-case names denote 32-bit floating-point variables, while upper-case names denote 64-bit doubles. If we assume an architecture with two 64-bit registers, each having two 32-bit aliases, then the graph in Figure 1(b) is not *colorable*. That is, no register assignment keeps all the variables simultaneously alive in registers. The register allocator normally solves this problem via *spilling*. In Figure 1(c) we have sent variable `a` to memory; thus, creating two new variables, $\mathtt{a}_0$, at the definition point of `a`, and $\mathtt{a}_1$ at its use point. The new interference graph, given in Figure 1(d) is now colorable.

Graph coloring-based register allocators tend to be iterative. In case a variable is spilled, parts of its live-range are bound to new variables, the interference graph is rebuilt, and the allocation process re-starts. As an example, Figure 2 depicts the Iterated Register Coalescing algorithm [11]. The iterative approach has two negative effects: first, it complicates

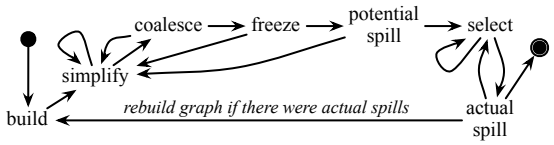

Figure 2: Iterated register coalescing, as taken from Appel and Palsberg [2].

the design of the algorithm, as Figure 2 clearly illustrates. Second, it decreases the speed of the algorithms, which must re-construct, or at least update, the interference graph, and re-do allocation steps that are, in many cases, redundant.

2.1 Decoupled Register Allocation

A decoupled register allocator separates the spilling and the register assignment phases. Hence, once we are done with the spilling phase, the existence of a register assignment that keeps all the live variables in registers is guaranteed. In order to provide this guarantee, a decoupled algorithm relies on the following property:

PROPERTY 1. *The maximum* local register pressure *at any program point equals the global register pressure.*

Property 1 is not present in every program; however, some intermediate representations, such as SSA-form [9] provide it in the absence of aliasing. Due to this property, spilling - the lowering of register pressure - can be done directly on the code, without the need of a data structure that gives a global program view, such as an interference graph. Even more, provided that all the registers have the same size, the local register pressure at a given point is the number of variables alive at that point; hence, the spill test is reduced to counting the number of variables simultaneously alive. A decoupled algorithm in general follows these four steps:

1. lower the register pressure at each program point, using any heuristics for variable spilling, i.e [1], until the variables alive at each point can be colored;
2. use live range splitting to guarantee Property 1. In general copies with a parallel semantic are used to split live-ranges [3, 13]. For an example, see Figure 3(a);
3. assign variables to registers [3, 12]. The register allocator must be able to find a way to assign registers to variables without causing further spills;
4. get rid of ϕ-functions and parallel copies [4, 21].

Notice that, during the spilling phase it is not necessary to do live range splitting. Instead, for each instruction, we try to color the interference graph formed by the variables alive in, out and through that instruction, spilling variables until this coloring becomes possible.

The advantages of decoupled register allocation: The first advantage is simplicity. Register allocators tend to be very complicated. As an example, about 20% of the lines of code of LLVM's [16] machine independent code generator are exclusively related to register allocation. Thus, from an engineering point of view, it is interesting to design register allocators that are modular. The decoupled approach fills the role very well, because it separates spilling from register assignment; hence, different heuristics and implementations can be easily combined and independently maintained. From an algorithmic point of view, decoupling simplifies and makes more effective both spilling and coalescing, because these steps have impact on the register pressure, and naturally benefit from an easier way to infer this quantity.

The second advantage is a better integration with other compiler optimizations, due to Property 1. Many compiler optimizations, such as pre-pass scheduling, if-conversion, code motion, loop-unroll and jam and partial redundancy elimination, may change the program's register pressure. Thus, these optimizations have a simpler and faster implementation if its possible to determine changes in register pressure accurately via local tests. Traditionally, the calculation of the register pressure at a certain program point needs a global view of the program; however, live range splitting allows it to be computed locally.

Decoupled register allocation and aliasing Lee *et al.* [17] have proved that register allocation with two-level aliasing is NP-complete even for SSA form programs made of a single basic block. Thus, in face of aliasing, the SSA form conversion is not extensive enough to guarantee Property 1; instead, *elementary form* can be used. We convert a program to elementary form via the insertion of parallel copies between each pair of consecutive instructions. Figure 3(a) shows our running example in elementary form. The interference graph of the new program, conveniently called an elementary graph, is given in Figure 3(b). The dotted lines denote *affinity edges*: it is interesting to assign nodes linked by such edges to the same color, because every time we fail to do it, a copy instruction will make its way into the final assembly program. Elementary programs have very simple structure: their interference graphs are a collection of many graphs that consist of two cliques only. Many problems that are NP-complete in general have polynomial time solution for such graphs. Thus, determining the local register pressure has a polynomial time solution, even when registers are allowed to have single, double or quad precision. The variables in the program given in Figure 3(a) can be allocated into our register bank made of two 64-bit registers and four aliased 32-bit registers; an improvement on the original program seen in Figure 1(a). This result is not a coincidence: any program can be transformed into the elementary form, and the elementary form program never requires more registers than the original code.

A heavy price incurred by the conversion into elementary form is the growth in the program size. For instance, the interference graph in Figure 1(b) has six nodes, but the corresponding elementary graph seen in Figure 3(b) has 26. This explosion is observed in actual benchmarks. Figure 4 compares the size of program functions taken from SPEC CPU 2000 before and after the conversion into elementary form. This transformation tends to increase quadratically the number of variables in the intermediate representation.

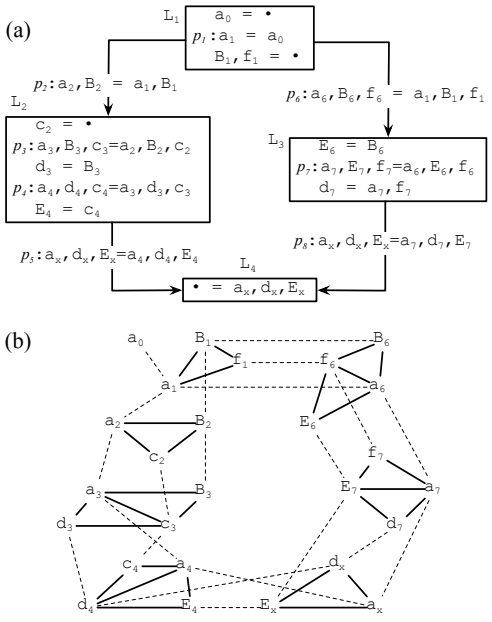

Figure 3: (a) The program from Figure 1 in elementary form. (b) The interference graph of the elementary program.

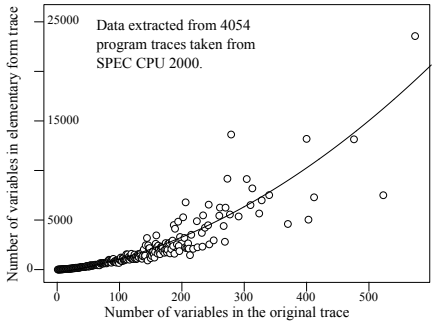

Figure 4: The growth in the number of program variables due to the conversion to elementary-form.

3. OUR CONTRIBUTIONS

In order to explain our ideas, we have adapted two different graph coloring based register allocators to run in a decoupled fashion in face of register aliasing. The first is the Iterated Register Coalescer of George and Appel [11], and the other is the Brute Force Coalescer of Bouchez *et al* [6]. Figure 5 shows our version of these algorithms. Comparing Figure 5(a) and Figure 2 it is easy to notice that the decoupled version has less iterations between its phases. Both these algorithms use the extensions of Smith, Ramsey and Holloway [26] to deal with aliasing, which we re-introduce later. Most of the phases that constitute each algorithm, i.e, simplify, coalesce, freeze and select have been thoroughly

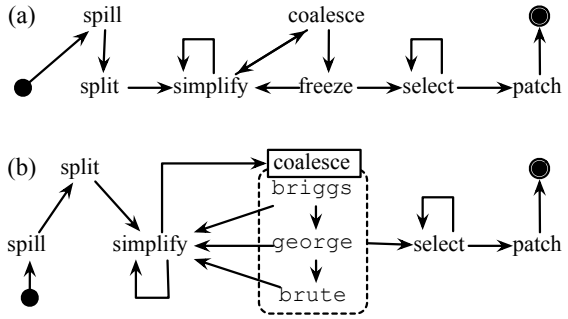

Figure 5: (a) A decoupled re-implementation of the Iterated Register Coalescer. (b) A decoupled re-implementation of Bouchez's Brute Force coalescer that handles aliasing.

described in previous works [11, 6]. Decoupled register allocators in general also use a phase called *patch*, related to the implementation of parallel copies. After register allocation, the compiler must implement these parallel copies, using the instructions present in the target architecture. Parallel copy patching has been thoroughly described before [4, 21].

The brute-force algorithm (BF), outlined in Figure 5(b), has a more modular design than the iterated register coalescer. After spilling is performed, BF orders the copies in the source program according to their *profitability*, and try to coalesce them following this ordering. The profitability of a copy is a measure of how much improvement its elimination can bring to the target code. Copies inside deeply nested loops tend to be more profitable than copies outside loops. We say that the coalescing of vertices a and b is *conservative* if the interference graph that we obtain after collapsing these nodes into a single node ab can still be allocated with the available registers. Brute Force uses one of the following three tests, in order, to guarantee that the coalescing of copy $a = b$ is conservative:

1. `Briggs`(a, b) [7]: the merging of a and b will create a node ab with fewer than K neighbors with squeeze greater than K.
2. `George`(a, b) [11]: assuming that a is a pre-allocated variable, then every neighbor of a already interferes with b, or has squeeze less than K. Notice that we must also try `George`(b, a), as this rule is asymmetric.
3. `Brute`(a, b) [6]: if we merge a and b, the new graph can be colored with K colors. We can perform this check in polynomial time via the simplification heuristics.

3.1 Decoupling Spilling from Register Assignment in Face of Aliasing

Decoupled register allocation is interesting as long as it does not cause more spilling than traditional graph-based register allocators do. The elementary form is an easy way to provide this guarantee. Given that the conversion to elementary form divides the source program in regions that

are very small and simple, the problem of determining the local register pressure for each region has polynomial time solution, at least for architectures with quad, double and single registers, such as x86, ARM, PowerPC and SPARC. The polynomial time solution still holds in face of pre-allocation, a phenomenon caused by architectural constraints that force variables to be assigned to particular registers [20].

Checking colorability via Smith's simplification test Graph-based algorithms normally rely on Kempe's technique [14] to remove nodes with degree less than K – the number of registers – until either the graph is empty, or all the nodes have higher degree. If a graph can be completely simplified via Kempe's method, then the graph is called *greedy K colorable* [5]. In the case of both the Iterated Register Coalescer, and the Brute Force coalescer, the spilling phase must guarantee that the program it passes forward to the other phases of the register allocator has an interference graph that is greedy K colorable. In the presence of aliasing, the simple test based on the node degree is not enough to check for greedy K colorability. A correct test has been devised by Smith *et al.* [26], using Fabri's idea of squeeze factor [10]. In Smith *et al.*'s framework, the computer architecture provides a number of register classes, which might alias in several ways. Each variable must be assigned to registers in a specific register class. The squeeze of a variable is the maximum number of registers, in its class, that could be denied to it, given a worst case allocation of its neighbors. Thus, a node v can be simplified if the worst case allocation of all neighbors of v is less than v's squeeze factor. Figure 6 illustrates this idea, assuming an architecture with double (R) and single (r) precision register classes. Figure 6(a) shows a subgraph of the graph given in Figure 3(b). Each vertex has been augmented with the squeeze factor of the variable that it represents, as determined by Smith *et al.*'s simplification criterion. For instance, variable $\mathtt{B}_6$ needs a double precision register, and has two neighbors, which could be assigned to aliases of different double-precision registers; thus, its squeeze factor is 2. We use the suffix R in $\mathtt{B}_6$'s squeeze factor to indicate its register class. The squeeze of a variable is bounded by the number of registers in this variable's class; hence, the squeeze of $\mathtt{a}_6$ or $\mathtt{f}_6$ is 4, although the worst case allocation, assuming an unbounded number of registers in class r would be 5 for any variable. Notice that the interference graph of the variables alive between two consecutive instructions is very simple: it consists of two cliques only. Thus, we can compute the squeeze factor of each variable simply counting variables simultaneously alive.

A correct spilling test that handles aliasing and pre-coloring: A fundamental question that concerns a decoupled register allocator is "which spill test should we use to ensure that after spilling we will be able to color the program's interference graph using the algorithm's graph coloring technique?" To answer this question one must be aware that after spilling and live range splitting no more spilling must be necessary. The graph coloring technique of choice is an important player in this game because a given heuristic may fail to color a graph that is actually colorable, after all, graph coloring is a NP-complete problem. Many allocators use Kempe's simplification test as the coloring heuristics. As we have discussed before, we are no exception.

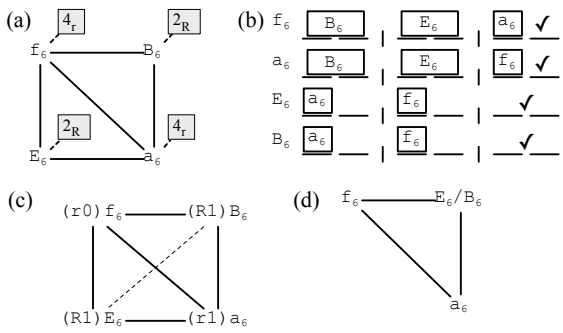

Figure 6: Smith *et al.* Simplification test. (a) A connected component of the graph in Figure 3(b). The nodes are labelled with their squeeze factors, e.g., the worst case allocation of E_6's neighbors takes off two registers of class R. (b) Worst case allocation for each variable. (c) A tight allocation produced by a puzzle solver [20]. (d) Variable merging guided by the puzzle solver. Architectural definition: $\mathbf{r} = \{r_0, r_1, r_2, r_3\}$, $\mathbf{R} = \{R_0, R_1\}$. Aliases: $\{(r_0, R_0), (r_1, R_0), (r_2, R_1), (r_3, R_1)\}$.

The notion of greedy K colorability, based on Kempe's test, is an over-approximation of colorability; however, this approximation is tight if we do not have to handle register aliasing. That is, in the absence of aliasing, if G is an elementary graph, then G is greedy K colorable if, and only if, G is K colorable. The proof of this statement follows from Bouchez's result for SSA-form programs without aliasing [5]. Therefore, without aliasing, answering the initial question is very simple: the spilling test is a simple as counting the number of variables alive at each program point.

In the presence of aliasing, greedy K colorability is different than colorability, as the example in Figure 6(a) shows. Furthermore, a combination of pre-coloring and aliasing may lead to situations in which every connected part of an elementary graph is greedy K colorable, but the global graph is not, as the example in Figure 7 illustrates. We call the interference graph formed by the live ranges live in, out and across an instruction *local*. Pre-colored nodes bind many local graphs together. Thus, the global squeeze factor of pre-colored nodes may be larger than their squeeze factor taken into consideration at each local graph. The consequence of this observation is that for a decoupled approach that performs the coalescing/coloring steps via Smith *et al.*'s method to be correct in the presence of aliasing and pre-coloring, we need to perform the spill test carefully. In other words, we must start the simplification process from the uncolored nodes, leaving the pre-colored nodes to the end. Theorem 1 proves the correctness of this procedure.

THEOREM 1. *If every connected component of an elementary graph is greedy K colorable starting the simplification process from the uncolored nodes, then then whole graph is greedy K colorable.*

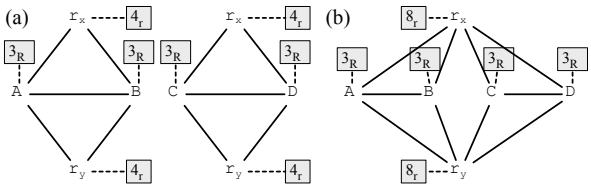

Figure 7: (a) Two greedy K colorable elementary graphs. (b) The whole graph is non-greedy K colorable.

PROOF. Any non-pre-colored node interferes only with nodes in its connected component, even taking the whole graph into consideration. Hence, the squeeze factor of these nodes is the same in the local and global interference graph. After these nodes are simplified, we are left with pre-colored nodes only. These nodes must be simplifiable, because they represent the registers in the actual architecture. □

Improving Smith's test with live range merging Assuming only two double-precision registers, the squeeze-based simplification test would fail to simplify any node in Figure 6(a), and some variable would have to be spilled. On the other hand, there exists a register assignment that accommodates all the variables, as Figure 6(c) shows. In order to improve Smith *et al.*'s simplification test, we do live range merging whenever we are unable to simplify any variable. We use the following algorithm:

- Let P be the set of variables alive in (I), out (O), and across (A) a given program instruction ι.
- While $P \neq \emptyset$
  - if $\exists v \in P : v$ is simplifiable
    - simplify(v)
    - $P = P \setminus \{v\}$
  - else if $\exists\ o \in O$ and $i \in I$: size(o) = size(i)
    - Let i, o be the largest pieces that fullfilled the condition
    - a = merge(i, o)
    - $P = P \setminus \{i, o\} \cup \{a\}$
  - else
    - Let $v \in A$, such that v is not used nor defined in instruction ι:
    - spill(v)
    - $P = P \setminus \{v\}$

When merging variables, we start with pairs of variables in register classes with the largest size, because this strategy reduces more drastically the squeeze factor of the other variables alive in that program point. Another important detail of our algorithm is the fact that we use live range merging with discretion. If we are stuck in the simplification process, then we choose only one pair of pieces, merge them, and re-try the simplification test. We proceed in this careful fashion because merged variables will be assigned the same register. This restriction might have the undesirable side effect of constraining too much the register coalescer that will run after spilling takes place.

```
byte f(int A, byte b) {
  while (true) {
    int C;
    byte e;
    C, e = div (A, b);
    A = C + e;
  }
  ret b;
}
```

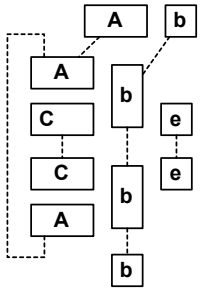

Figure 8: Example showing the deficiencies of traditional coalescing techniques.

We do no apply live range merging at program points that contain pre-allocated variables. Pre-allocation might prohibit the merging of live ranges, and, in face of this phenomenon we fall back to Smith *et al.*'s simplification test.

3.2 Semi-Elementary Form

Traditional coalescing tests, such as George's [11] or Briggs's [7] have a number of disadvantages if used on elementary graphs. The first disadvantage is in terms of runtime. Each of these tests would have to be invoked once for each affinity edge in the elementary graph. The second disadvantage concerns the quality of the code produced. In the presence of aliasing, the traditional coalescing techniques may fail to eliminate copies, even though they are not necessary. For instance, Figure 8 shows a program in elementary form, in which every copy could be completely coalesced away. However, neither George nor Briggs rules would be able to coalesce the inner copies. This limitations happen because these rules are applied *sequentially*. Coalescing would be possible if all the affinity edges were analyzed in parallel.

The rational behind the elementary form is to reduce the amount of spilling during register allocation. With such purpose, the conversion to elementary form splits the live ranges of the variables at every program point. However, most of these splits are unnecessary. We have developed two techniques to reduce the size of the program's interference graph. The first technique, that we call *the critical node test*, is based on a criterion that avoids splitting live ranges whenever possible. We call the program representation that results from this method *semi-elementary form*. The second technique merges variables, whenever it is conservative to do so. In order to perform this merging we rely on a method that we call the *local merging test*. We explain these two strategies in the rest of this section.

A criterion to avoid live range splitting An elementary graph if formed by many unconnected components, which represent the live ranges of variables at some particular pro-

gram point. Therefore, we expect a lot of redundancies between graphs formed from consecutive instructions. Given two instructions, the guider and the follower, all the vertices that correspond to variables live-in at the follower are connected through affinity edges to the vertices in the guider. The only vertices in the follower's graph which have no affinities for vertices in the guider's are those nodes that represent variables defined in the follower instruction. We call them *critical nodes*. Normally an instruction defines at most one variable; hence, we expect to find at most one critical node in the follower, and this test can be performed quickly. There exist, of course, architectures which contains instructions that write into more than one register. For instance, in x86 we have `mul`, `div`, `lodsb`, etc. In this case, we must perform the critical node test once for each variable defined in the instruction. In light of these observations, our criterion to avoid live range splitting is as follows:

> **The critical node test:** if every critical vertex in the follower's graph has a squeeze factor less than K, then it is not necessary to insert a parallel copy between guider and follower to achieve Property 1.

THEOREM 2. *Let G_g and G_f be the interference graph at the guider and the follower, as previously defined. If G_g is greedy K colorable, then the graph that results from merging G_g and G_f via the critical node test is greedy K colorable.*

PROOF. The proof is straightforward: if the merging is done, the resulting graph is formed by all the nodes from G_g plus the critical nodes in the follower. Because of our criterion we know that every critical node can be simplified. Once they are simplified, we fall back into G_g, which, by hypothesis, is greedy K colorable. □

Figure 9 illustrates our method when applied to the sequence of instructions from program point p_1 to p_5 in Figure 3(a). We have augmented the graphs in Figure 9(a) with the squeeze factor of each node, and we have highlighted the squeeze factor of each critical node in the next figures. Considering two double precision registers available, we can avoid all the parallel copies but the last, because the squeeze of $\mathtt{E}_4$ is 4. On the other hand, if applied on the program in Figure 8, the critical node test would avoid every live range splitting. In this case, the semi-elementary form program equals the original program converted to SSA form.

The critical node test avoids splitting live ranges unnecessarily. As the experiments from Section 4 show, the interference graphs of semi-elementary form programs that we found are about eight times smaller than the corresponding graph of elementary-form programs; however, the former graphs are still approximately twice as big as the interference graphs of the original programs. In order to avoid this growth, we can go even further, merging live ranges of non-affinity related variables whenever it is conservative to do so. We call this type of preprocessing the *local merging* of live ranges, and explain it in the rest of this section.

The local merging of live ranges To further reduce

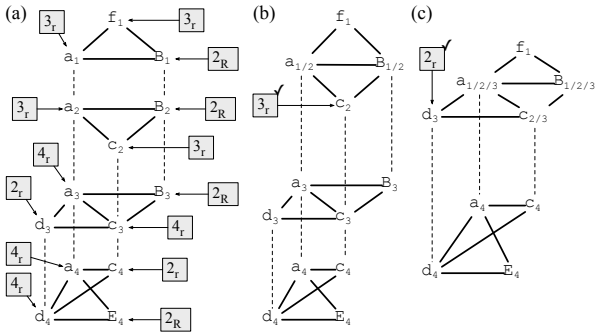

Figure 9: Construction of semi-elementary form. (a) a subgraph of the graph in Figure 3. (b) and (c) Subgraphs that result from avoiding the insertion of two parallel copies. We cannot avoid the last parallel copy, otherwise we would build a graph that is non greedy K colorable.

the size of the interference graph, we can merge some non-affinity related variables, using a technique based on punctual coalescing [22]. Punctual coalescing is a strategy used in conjunction with puzzle-based register allocation to remove copies in the target code. The punctual coalescer traverses the dominator tree of the source program, analyzing one instruction at a time. The algorithm processes the interference graph formed by variables alive around this instruction, remembering the allocation of the previous instruction. It is a locally optimal approach; that is, given only the knowledge of the variables alive across two consecutive instructions, it finds the largest number of matches between variables that do not compromise Property 1. We use the results that we get from the punctual coalescer to design a local live range merging method. Our live range merging technique based on punctual coalescing is given below:

- For each pair of consecutive instructions, *guider* and *follower* inside a basic block, let G_g and G_f be the local interference graphs that denote the register allocation problem for each instruction.
- We let the punctual coalescer [22] place in the same registers the vertices that have affinities. The punctual coalescer tends to maximize the number of matches between two consecutive instructions.
- For each pair of same-size variables $u \in G_g$, and $v \in G_f$, that have been assigned the same register r:
  1. If the vertex uv that results from merging u and v does not interfere with any vertex w that has been assigned r or an alias of r by the punctual coalescer, then replace u and v by uv. This type of interference might happen if u and v have non-contiguous live ranges, and w is alive between the kill site of u and the definition site of v.
- For each s denoting a variable defined in the *follower*:

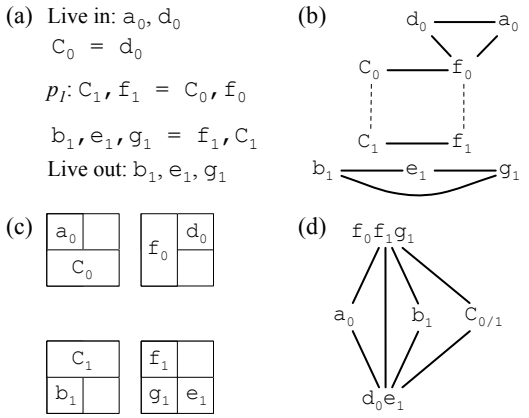

Figure 10: A constructed example showing punctual merging. (a) The elementary-form program. (b) The interference graph. (c) The solution of punctual coalescing. (d) The solution of punctual merging.

1. If s has a squeeze factor greater than the number of registers in the register class of s, then undo every merging of the previous step.

We only merge live ranges inside the same basic block, because, by merging non-affinity related variables, we may eliminate coalescing opportunities. As we show in Section 4, punctual merging decreases the capacity of both, the Iterated Register Coalescer and the Brute Force coalescer to eliminate copies in the final assembly code. Figure 10 illustrates punctual merging. We have used a different example this time, because our running example from Figure 3 is not complex enough to exercise the interesting aspects of punctual merging. Notice that the critical node test, when applied on Figure 10(b) would only merge the variables `C`'s and `f`'s. However, assuming a solution of punctual coalescing that places variables $\mathtt{f}_0$, $\mathtt{f}_1$ and $\mathtt{g}_1$ into the same column, we can also merge these pieces. The same happen with – non-contiguous – variables $\mathtt{d}_0$ and $\mathtt{e}_1$. On the other hand, we cannot merge variables $\mathtt{a}_0$ and $\mathtt{b}_1$, because $\mathtt{C}_0$ and $\mathtt{C}_1$ have been allocated to aliased of the registers assigned to $\mathtt{a}_0$ and $\mathtt{b}_1$. If we merged $\mathtt{a}_0$ and $\mathtt{b}_1$, then the resulting variable would interfere with both $\mathtt{C}_0$ and $\mathtt{C}_1$.

4. EXPERIMENTS

Testing environment: The algorithms were implemented in Python, producing code to a prototype architecture called MiniIR [1], which is based on the YAML [2] serialization format. YAML is used by STMicroelectronics *Inc* to quickly prototype hardware. MiniIR provides a minimalist textual machine level intermediate representation to be used for experimental tools. We report numbers for the x86 architecture, which we described in miniIR (Figures 13, 14 and 15),

[1] `http://www.assembla.com/wiki/show/bE6Ve4RQir36HFeJe5cbLr`

[2] `http://www.yaml.org/`

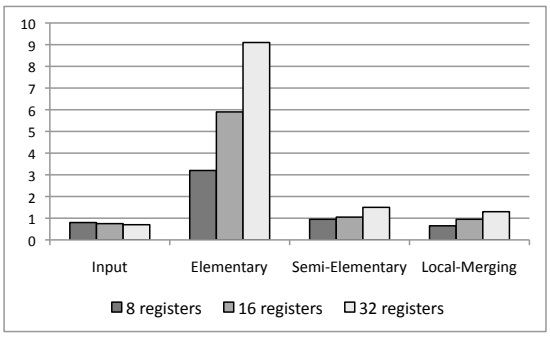

Figure 11: Number of nodes, in millions, of the interference graphs of different program representations. *Input:* input graph passed to the original iterated register coalescing algorithm – the number of nodes is the number of variables in the source program.

and for an artificial architecture with 8, 16 and 32 registers, also described via miniIR (Figure 12). In this case, each register has 32 bits, and is divided into two 16-bit aliases. We have checked the validity of each register allocation using the type-system of Nandivada *et al.* [18]. We chose to run our experiments on SPEC CPU 2000, which we have compiled into MiniIR using LLVM 2.7 [16].

The spilling approach: All the numbers in this section refer to greedy K colorable interference graphs. That is, because we do decoupled register allocation, we perform spilling before doing register assignment. We are using the simple "spill everywhere" approach: whenever the register pressure is too high at some program point, we choose the variable that has the furthest use in a linearization of the control flow graph, and spill it to memory. We check the register pressure via the improved Smith test that we have described in Section 3.1. To spill we replace the definition of the variable by a store, and each use by a load. We do not try to re-use loads.

Size of interference graphs: The chart in Figure 11 illustrates the effectiveness of the conversion into semi-elementary form in order to reduce the size of the interference graphs. The interference graphs of elementary-form programs, on average, are 800% larger than the interference graphs of the original programs. This difference falls down to 200% if we use semi-elementary form instead. If we do local merging, then we obtain interference graphs that are even smaller than the graphs produced for the original program, when we use eight registers only. The graphs tend to become larger as more registers are taken into consideration, because the amount of spilling decreases, but the proportion of variables having different sizes remains the same. Thus, although we spill less when more registers are available, we still must perform live range merging to avoid spilling, as we do in Figure 1. This fact also explains why the input graphs are smaller when we have more registers available: in this case, less spilling will happen, and fewer variables will be created to hold the source/destination of stores/loads.

Allocation time: The size of the interference graph has

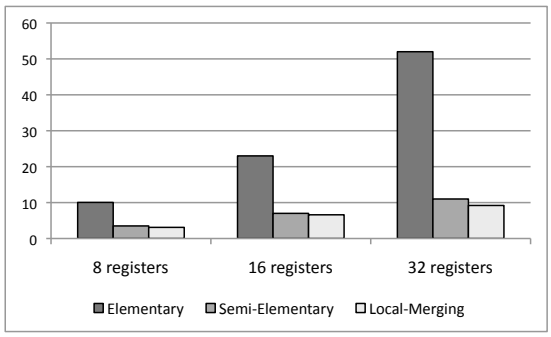

Figure 12: Allocation time, in hundreds of seconds, for different kinds of interference graphs.

a direct impact on the allocation time, as Figure 12 shows. Considering eight registers only, allocation in semi-elementary form is 3.1 times faster than in elementary form. This different increases to 3.3 times if we perform local live range merging. With 32 registers the difference is even larger. Semi-elementary form speeds-up register allocation by a factor of 4.7x, and local merging moves this factor to 5.5x.

To put these results in perspective, let's consider the largest function that we found in our benchmark, assuming eight registers. This function, in SSA-form, has 10,163 variables. The interference graph of the elementary program contains 99,364 nodes. On this graph, IRC takes 4,544 seconds. This is 49.40x slower than the punctual coalescer [22], which does not build an interference graph. The semi-elementary form program has an interference graph with 19,024 nodes. In this case, IRC is 5,30 times slower than the punctual coalescer. After local live range merging we have a graph with 13,764 nodes, in which IRC is 1.82x slower than punctual coalescing. Thus, IRC is 27.2x faster on a graph after local live range merging than on an elementary graph.

Measuring the effectiveness of live range merging on the register coalescers The semi-elementary form improves the effectiveness of copy coalescing, as we show in Figures 13, 14 and 15. Figure 13 shows only the result of IRC using Smith *et al.*'s [26] extensions, implemented according to Figure 5(a). The algorithm executing over elementary graphs left 3418 copies on the SPEC CPU 2000 programs. On semi-elementary form this number falls down to 3221 copies. If we perform local live range merging on the source programs, then IRC leaves 4133 copies on the assembly code. In these experiments we count only copies inserted into the program to do live range splitting; that is, we do not count the copy instructions that were part of the source program, even though many of them were coalesced too. In this way, we focus on the ability of the allocators to handle aliasing. The local live range merging decreases the coalescing power of IRC, when compared to the results that we obtain by using semi-elementary form. We speculate that this fact happens because the punctual merging constraints too much the interference graph, creating nodes with larger squeeze factors.

Figure 14 shows the effectiveness of the brute force coalescer implemented using the modifications from Section 3,

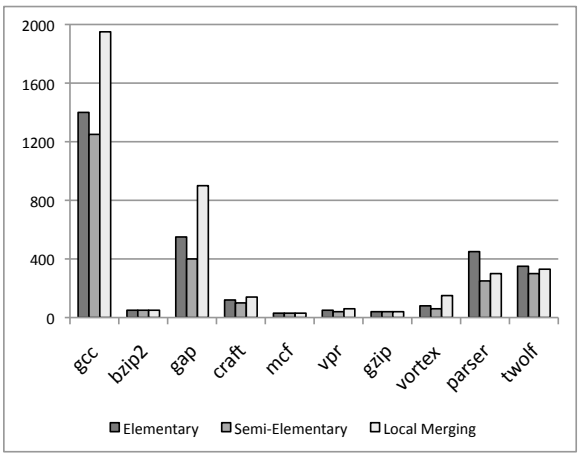

Figure 13: Number of copy instructions left by the iterated register coalescer (IRC) when running on different program representations.

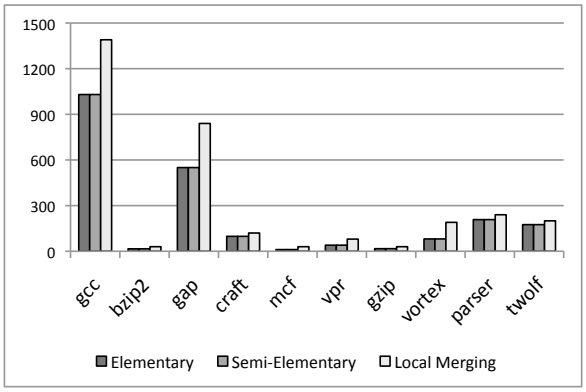

Figure 14: Number of copy instructions left by the brute force coalescer (BF) when running on different program representations.

following Figure 5(b). Running the algorithm directly on elementary-form or semi-elementary form programs leaves 2803 copies on the final assembly code produced by the algorithm. Local live range merging degrades the ability of the coalescer to eliminate copies. In this case, BF leaves 3503 copies on the assembly programs.

Figure 15 compares the three algorithms: brute force (BF), iterated register coalescing (IRC) and punctual coalescing [22] in terms of the number of copies that each algorithm eliminates via coalescing. We have chosen the best configuration for each algorithm: IRC and BF run after simple live range merging, while punctual coalescing can only run on elementary-form programs. Confirming previous results [6, 22], the brute force coalescer is the most effective algorithm, followed by IRC. Confirming previous results [20], the punctual coalescer increases the final assembly programs by 6.8% on average. This relatively bad result of the punctual coalescer is due to the fact that it is a local approach, which does not attempt to eliminate copies between basic blocks.

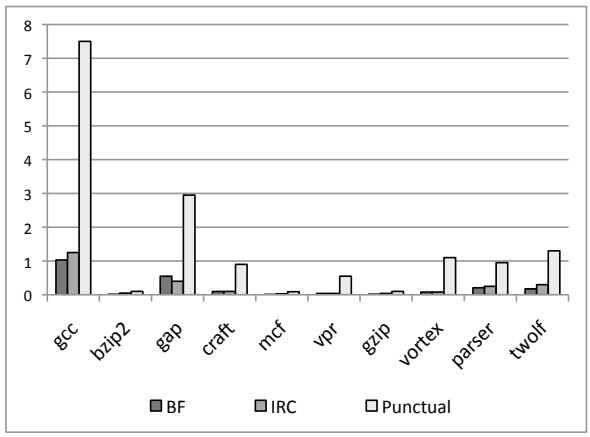

Figure 15: Number of copy instructions left by three different register coalescers, in thousands. IRC and BF run on semi-elementary form, and the punctual coalescer runs in elementary form programs.

5. CONCLUSION

This paper has introduced a number of techniques that make graph coloring-based register allocation more practical and effective in the presence of live range splitting. Live range splitting helps to decrease the number of variables spilled during register allocation. However, in order to produce code to architectures with aliased register banks, previous register allocators use a very aggressive form of live range splitting – the elementary format – which would increase too much the size of the program's interference graph, in addition of potentially causing the insertion of extra copies into the final assembly code. Our new techniques allows the register allocators to use all the power of the elementary format, while at the same time avoiding the size explosion, and decreasing the amount of copies into the assembly program.

Acknowledgments: This project has been made possible by the cooperation FAPEMIG-INRIA, grant 11/2009.

6. REFERENCES

[1] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM, 2001.

[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, Quentin Colombet, Alain Darte, Fabrice Rastello, and Christophe Guillon. Parallel copy motion. In *SCOPES*, pages 1–10. ACM, 2010.

[5] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO*, pages 102 – 104. IEEE, 2007.

[6] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES*, pages 147 – 156. ACM, 2008.

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *TOPLAS*, 16(3):428–455, 1994.

[8] 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.

[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] Janet Fabri. Automatic storage optimization. In *CC*, pages 83–91. ACM, 1979.

[11] Lal George and Andrew W. Appel. Iterated register coalescing. *TOPLAS*, 18(3):300–324, 1996.

[12] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *PLDI*, pages 227–237. ACM, 2008.

[13] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.

[14] A. B. Kempe. On the geographical problem of the four colors. *American Journal of Mathematics*, 2(1):193–200, 1879.

[15] Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.

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

[17] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation. In *ICALP*, 2007.

[18] V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *SAS*, pages 153–169. Springer, Kongens Lyngby, Denmark, August 2007.

[19] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.

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

[21] Fernando Magno Quintao Pereira and Jens Palsberg. SSA elimination after register allocation. In *CC*, pages 158 – 173, 2009.

[22] Fernando Magno Quintão Pereira and Jens Palsberg. Punctual coalescing. In *CC*, pages 165–184, 2010.

[23] Hongbo Rong. Tree register allocation. In *MICRO*, pages 67–77. ACM, 2009.

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

[25] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.

[26] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI*, pages 277–288. ACM, 2004.

[27] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *CGO*, pages 170–179. ACM, 2010.