Punctual Coalescing

Fernando Magno Quintão Pereira[1]
Jens Palsberg[2]

[1] Universidade Federal de Minas Gerais, Belo Horizonte
[2] University of California, Los Angeles

Abstract. Compilers use register coalescing to avoid generating code for copy instructions. For architectures with register aliasing such as x86, Smith, Ramsey, and Holloway (2004) presented a polynomial-time approach, while Scholz and Eckstein (2002) presented an optimal, exponential-time approach together with a near-optimal, quadratic-time heuristic. Both methods scale poorly after aggressive live range splitting, especially for programs in elementary form where live ranges are split at every program point. In contrast, we mentioned in a previous paper (2008), without giving details, that we have a scalable, linear-time heuristic for programs in elementary form. In an effort to formalize that heuristic, we discovered an even better algorithm, called Punctual Coalescing, which we present here. Punctual Coalescing is scalable, linear time, locally optimal in general, close to globally optimal for straight-line code, and proven correct with the Twelf theorem prover. We define global optimality with an ILP-formulation and we show via experiments that Punctual Coalescing compares well to this and two other approaches.

1 Introduction

Register allocation is the problem of mapping program variables to physical locations, which are either registers or memory. Compared to mapping all variables to memory, a good register allocator can improve the speed of the generated code on a RISC architecture by 250% [22]. We will focus on a combination of three important challenges for register allocation, namely *live-range splitting*, *coalescing* and *aliasing*, which we recall next.

To keep more variables in registers, compiler writers use live-range splitting [2, 6, 17, 26, 31]: split the live range of a variable y by (1) introducing a fresh variable name x, (2) inserting the copy instruction $x = y$ somewhere in y's live range, and (3) using the name x instead y after that copy instruction. After the split, the register allocator has the opportunity to map x to a register and y to memory, or vice versa. Coalescing [9, 11, 12, 15–17, 23, 30] is the dual of live-range splitting: eliminate copy instructions of the form $x = y$ by mapping both x and y to the same register. Intuitively, the more we do live-range splitting, the more we need coalescing to eliminate unnecessary copy instructions. The third challenge, aliasing, is a property of architectures such as ARM, PowerPC, Sparc v8/v9, and x86: quoting Smith *et al.*, "two registers alias when assigning a value

to one may change the value of the other" [33]. Open until now is the problem of designing a scalable, high-quality, and provably correct register allocator that after aggressive live-range splitting does coalescing for an architecture with aliasing. Let us briefly summarize the most closely related previous works.

There exist register allocation algorithms that deal with aliasing. An example is the integer linear programming (ILP) approach of Kong and Wilken [19]. Scholz and Eckstein (2002) [32] have addressed aliasing with partitioned boolean quadratic programming (PBQP). They presented an optimal, exponential-time approach together with a near-optimal, quadratic-time heuristic. Smith, Ramsey, and Holloway [33] have generalized graph coloring register allocation to incorporate aliased registers. Also based on graph coloring allocation, Minwook *et al.* [1] have described an optimistic coalescing algorithm that is competitive with Smith *et al.*'s iterative approach. These methods scale poorly after aggressive live range splitting. Intuitively, aggressive live-range splitting enables a high number of variables to be mapped to registers, but it also overwhelms the register allocator with copy instructions. We will show how to deal with the high number of copy instructions by adopting a particular program representation and then developing a new coalescing algorithm.

We will work with Appel and George's idea from 2001 [2] of "ultimate" live-range splitting that splits every live-range at every program point, that is, between every pair of consecutive instructions. The result is a program in what we call elementary form. A compiler can convert any program to elementary form in polynomial time, and the elementary program requires at most as many registers as its original version. We use the notion of elementary form because it allows us to avoid a difficult problem. The problem of finding the minimal number of registers that is needed to compile straight-line code to an architecture with aliasing is NP-complete [21], while for a program in elementary form, the problem can be solved in linear time by a puzzle solver [26]. Our goal is to add coalescing to the linear-time puzzle solver without changing the time complexity.

In a previous paper [26] we mentioned, without giving details, that we have a scalable puzzle solver that embodies a heuristic for coalescing. In other words, that unpublished heuristic goes a long way toward solving the open problem. In an effort to formalize that heuristic, we discovered an even better algorithm, called *Punctual Coalescing*, which we present here.

Punctual Coalescing is scalable, runs in linear time, and is a form of biased coloring [9] that uses only local information. The puzzle solver with Punctual Coalescing traverses the dominator tree of the source program finding at each program point a register assignment that minimizes the number of variables sent to memory. The assignment is guided by the assignment found at the most-recently visited program point. Punctual coalescing is well suited for just-in-time compilers such as TraceMonkey [14], and tree-scan-based allocators such as Braun and Hack's [8]. We have proved the correctness of Punctual Coalescing, and in particular we have proved the main lemma with the Twelf theorem prover [28].

In general, punctual coalescing is locally optimal for straight-line code, and close to globally optimal. Our experiments with compiling SPEC CPU 2000 to

x86 show that punctual coalescing finds a locally optimal solution for 89% of the program points in our benchmarks. We define global optimality with an ILP-formulation that combines ideas from papers by Kong and Wilken [19], who showed how to handle aliasing, and by Grund and Hack [16], who showed how to handle coalescing. During the compilation of the SPEC CPU 2000 benchmark suite to x86, only one copy was inserted per 14 instructions in the original program. These copies were typically used to insert fixing code between basic blocks, and to avoid conflicts with pre-allocated registers, as we discuss in Section 6.

We have done an experimental comparison of four register assignment approaches: register allocation via coloring of chordal graphs [25], the heuristics used in the original puzzle solver [26], the punctual coalescing algorithm and the ILP formulation – the last two algorithms are introduced in this paper. To overcome scalability issues with the ILP approach we derived long program traces from SPEC CPU 2000, that is, long sequences of code that were executed in order. For those program traces our experiments show that Punctual Coalescing is considerably better than the other approaches and close to globally optimal.

In the next section we briefly review register allocation by puzzle solving, and illustrate the coalescing problem with an example. In Section 3 we describe Punctual Coalescing, in Section 4 we describe our ILP-formulation of global optimality, in Section 5 we show experimental results, in Section 6 we discuss limitations of punctual coalescing and in Section 7 we conclude the paper.

2 Background

A *program point* is any point in between two consecutive instructions, or in between two consecutive basic blocks. The program in Figure 1 has five program points, numbered 2 to 6. A variable v is alive at program point p if there is a path from p to an instruction that uses v that does not cross a definition of v. For instance, in Figure 1, variable a is alive at program points 2, 3, 4, 5 and 6. The program points where variable v is alive form v's live range. We can split the live range of a variable inserting a copy instruction at some program point in the live range, and doing variable renaming. Many register allocators use live range splitting to keep more variables in registers [2, 17, 26, 31, 34]. The *elementary form* is a program representation introduced by Appel and George [2] in which the live ranges of variables are split at each program point. If P is a program with V variables and I instructions, and P' is P converted to elementary form, then P' contains $O(I \times V)$ variables. Many register allocators are at least $O(V^2)$ – in particular, aliasing aware methods such as Scholz and Eckstein [32]'s PBQP approach and Smith *et al.*'s [33] extensions for Chaitin style algorithms. Hence, these algorithms run in at least $O(I^2 \times V^2)$ when applied to elementary programs.

Register allocation by puzzle solving: Register allocation by puzzle solving [26] relies on elementary form to minimize register usage. In this paradigm, registers are modeled as a puzzle board, and the live ranges of the variables as puzzle pieces. There is one puzzle per program instruction, and the challenge is to

Fig. 1. An example of register allocation by puzzle solving.

arrange the pieces on the board, so that no piece will be left out. We illustrate this method with the example given in Figure 1. The program on the left side of the figure has six instructions and five variables, a, B, c, d and E. The live ranges of the variables are shown in the middle of the figure. We assume a target architecture with two registers, each one with two aliases. Such architectures are called *T1*, for type 1 puzzle. The type of a puzzle is determined by the number of columns in each board area: a puzzle Tn has 2^n columns per area. Lower case letters denote single precision values, whereas upper case letters denote double precision values. We can store two single precision values or one double precision value in one register. The opcode of each instruction is not relevant to our explanations, so we use •'s for "don't care's". The right side of Figure 1 shows a solution to this instance of the register allocation problem.

In this paper we provide coalescing algorithms for T1 puzzles. These puzzles model registers that have two independent aliases, such as the general purpose registers found in x86 (`AX, BX, CX` and `DX`), and the floating point registers found in ARM and PowerPC. It subsumes T0 puzzles, which we find in integer registers of PowerPC and ARM. T1 puzzles have three types of pieces: X, Y and Z. X pieces, such as a, d and E in puzzle six of Figure 1 can only be placed on the upper half of a board area. On the other hand, Z pieces such as B in puzzle two are only placed on the lower half of an area. Y pieces such as a and B in puzzle three occupy the upper and lower part of an area. A T1 puzzle piece may have width one or two. Size one pieces such as a, c and d in Figure 1 fit in one column of an area; they represent eight bit variables in x86, or single precision floating point values in ARM and PowerPC. Size two pieces, such as B and E span two columns. They represent 16 or 32 bit values in x86, or double precision numbers in ARM and PowerPC. We will be working with *padded* puzzles, that is, our puzzle solver expects that the area of the pieces will equal the area freely

available on the board. We pad a puzzle by adding to its original set of pieces as many size one X and Z pieces as needed. A puzzle has solution if, and only if, the padded version does [26, Lemma 26].

Register coalescing: The register assignment in Figure 1 is optimal in two senses. First, it uses the minimal number of registers – it is not possible to compile this program with only one register divided into two aliases. Second, it uses the minimal number of copies to split the live ranges of variables. In order to obtain the minimal register assignment, we had to move variable a from register R_0 to register R_3. This split is performed by a register move inserted at program point five. This solution is globally optimal – the minimal register assignment requires the insertion of one copy instruction into the source code. In general, inserting copies to avoid mapping variables to memory leads to faster programs [26]; however, ideally we would like to minimize the number of copies inserted into the final program – an optimization known as coalescing. We distinguish two variations of coalescing: global and punctual, which we define below:

- Global Coalescing
  Instance: a program P in elementary form that can be compiled with K registers.
  Problem: find a register assignment for P, using K registers, that minimizes the number of instructions between puzzles.

- Punctual Coalescing
  Instance: two consecutive puzzles p_1 and p_2, such that p_1 is already solved.
  Problem: find a solution of p_2 that minimizes the number of copies inserted between p_1 and p_2. We call puzzle p_1 the *guider*, and puzzle p_2 the *follower*.

In the definition of global coalescing we assume that the input program is *greedy K-colorable* [6, p.18], that is, it is possible to find an allocation of variables to registers using at most K registers. K colorability ensures that spilling plays no role in the coalescing problem. This is the principle behind many register allocators based on live range splitting [2, 17, 18, 25, 26, 31]. These algorithms are divided into two phases [6]. Initially a spilling phase removes variables, mapping them to memory, in order to ensure K colorability. Subsequently, a coloring phase finds a valid mapping of variables to registers using the available registers.

Global coalescing has a natural description as a graph coloring problem. The *interference graph* of a program is the interference graph of the live-ranges of the variables in the program. That is, given a program P, if G is its interference graph, then G has one vertex for each variable in P, and two nodes are adjacent if, and only if, they correspond to variables with overlapping live ranges. In the global coalescing problem we consider a second type of edge called *affinity edge*. There exists an affinity edge between two nodes v_1 and v_2 if P contains a copy instruction $v_1 = v_2$. The global coalescing problem asks for a coloring of G with at most K colors that maximizes the number of affinity related nodes that get the same color. In the presence of register aliasing, we consider each color as an

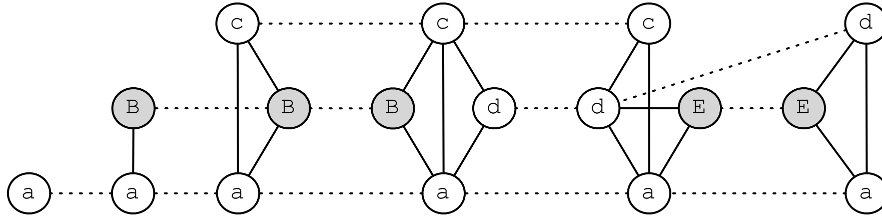


Fig. 2. A graph coloring representation for the global coalescing problem in Figure 1.

integer number, so that some nodes must receive two consecutive colors. Figure 2 shows the graph coloring representation for the coalescing problem in Figure 1. Dashed lines represent affinity edges, grey nodes represent variables that fit into a full register, and white nodes represent variables that fit in half a register.

Global coalescing is the version traditionally studied in the compiler literature. This problem is NP-complete [6]; thus, it is normally solved by heuristics, such as Chaitin's aggressive algorithm [12], or Brigg's conservative algorithm [9]. In Section 4 we give an optimal solution to this problem, in a T1 architecture, via integer linear programming. When restricted to program traces, global coalescing has polynomial time solution for T0 register banks, and is NP-complete for T1 register banks [21]. The problem is NP-complete if some variables are forced to be in particular registers [4], even restricted to program traces in T0 settings. Punctual coalescing has polynomial time solution for T0 and T1 architectures, as we show in Section 3. The complexity of this problem in the context of higher order register banks, or when pre-coloring is allowed is left open.

A sequence of optimal solutions to the punctual coalescing problem might produce a solution to its global counterpart, as in Figure 1. However, that is not always the case, as we show in Figure 3. The figure contains three instances of the punctual coalescing problem, one for each point between two consecutive puzzles. Each of these instances is optimally solved, and a copy is inserted between puzzles two and three. However, there is a register assignment that does not require copies between instructions, shown in the right column of the figure.

3 An Efficient Punctual Coalescing Algorithm

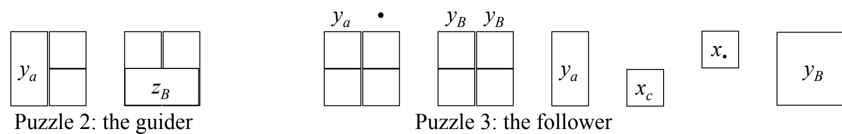
In this section we describe a strategy for solving the punctual coalescing problem. Our strategy is optimal for settings with initially empty follower boards. By optimal we mean that, if an instance of punctual coalescing has a solution with at most n copies inserted, then our algorithm will find it.

If a piece v fills the bottom of area a in the guider's board, we say that a is the *preferred area* for v in the follower's board. For instance, in Figure 1, R_1 is the preferred area for piece c in puzzle four, because the bottom part of R_1 is holding c in puzzle three. Also, R_2/R_3 are the preferred areas of piece B in

Fig. 3. An example where a sequence of optimal punctual coalescings is worse than global coalescing.

puzzle three, for these areas are the location of B in puzzle two. In general, X and Y pieces have preferred areas, whereas Z pieces never have it.

We extend the notation introduced in [26] to include preferences between pieces and board areas. If a piece has no preference, we call it anonymous, in contrast with labeled pieces, which have preference for some area. Anonymous pieces are marked with the symbol •, and labeled pieces are given the name of the variable that they represent. Each column of the board area now has a label, which is the name of the piece with a preference for that column. There are eight ways, up to symmetry, to label a T1 area. These patterns are shown in Figure 4. The shaded areas are not part of the pattern; they only illustrate where the preferred pieces should stay. Each area of the follower board has one of these patterns. Going back to the running example from Figure 1, area R_0/R_1 of puzzle two has pattern (h). However, the same area in puzzle five has pattern (g), with a preference for pieces a and c, as the registers R_0 and R_1 contain these pieces in puzzle four. As another example, we illustrate puzzle three below:



Our puzzle solving algorithm is given in Figure 5. This algorithm, written in a visual language, solves puzzles by pattern matching. It has eight *statements*, one for each possible pattern of preferences that can be found in an area. Each statement is composed by one or more *rules*, which specify how an area must be filled with pieces. Syntactically, a rule is a two-by-two diagram formed by a *pattern* and a *strategy*. The pattern is one of the eight configurations given in Figure 4. A strategy is a description of how to complete the area, including

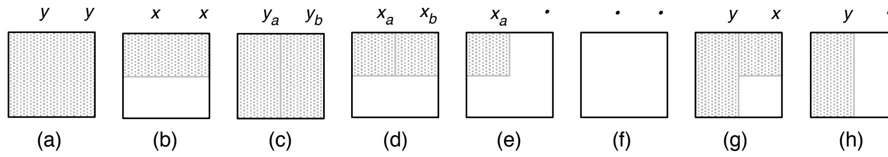


Fig. 4. Patterns of preferences. The shaded areas are not part of the notation; they only emphasize where the preferred pieces should stay.

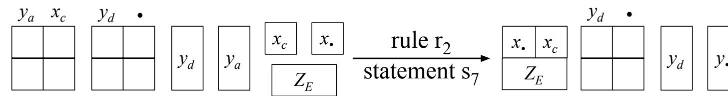
which pieces to use and where to put them. We say that the pattern of a rule *matches* an area a if the pattern contains the same sequence of preferences as a. For a rule r and an area a where the pattern of r matches a:

– the application of r to a *succeeds* if the pieces needed by the strategy of r are available; the result is that these pieces are placed in a;
– the application of r to a *fails* otherwise.

The complexity of solving a puzzle with A areas is $O(A)$. The rules of a statement are tried in order. If one of them succeeds, then the statement succeeds. If no rule succeeds, then the statement fails. The solution of a puzzle is found by successive applications of statements on empty board areas, as follows:

For each i from 1 to 8:
  For each empty area a such that the pattern of s_i matches a:
    – apply s_i to a
    – if the application of s_i to a fails,
      then terminate the entire execution and report failure.

If the preferred area of a piece v is filled with a piece other than v, and v is still available to fill other areas, we remove the name of v and mark it as an anonymous piece. We illustrate this step in the figure below, which uses puzzle five from Figure 1 as an example:



The piece $x_\bullet$ was added to pad the puzzle. This example shows the application of the second rule of statement seven of our solving algorithm. After the application, the piece y_a can no longer be allocated into its preferred spot, so we relabel it to an anonymous piece $y_\bullet$.

The algorithm in Figure 5 determines an order in which areas must be filled with pieces. Part of the ordering that we chose is arbitrary, e.g, any ordering between statements one to seven would preserve the optimality of the solution.

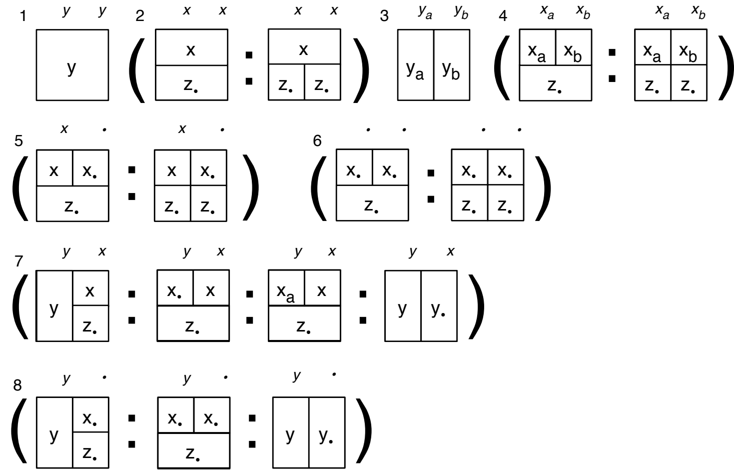
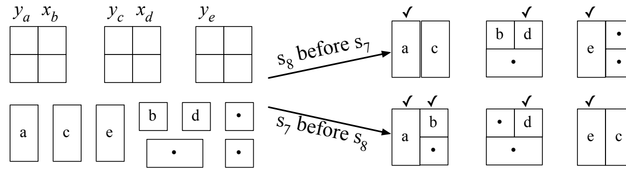
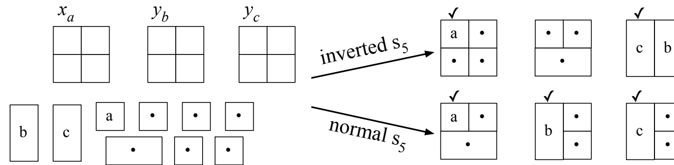


Fig. 5. Program P_c that solves punctual coalescing for empty follower boards.

However, some choices are essential to guarantee the optimal solution of punctual coalescing. For instance, the figure below illustrates a case in which we get more copies if we switch the precedence between statements seven and eight:



Similarly, the figure below illustrates a case in which we get worse results if we invert the order of rules inside statement five:



Correctness We have proven that the algorithm in Figure 5 solves a puzzle with an initially empty board, if, and only if, that puzzle has solution [24, ch.4]. This result comes from the combination of two lemmas: progress (Lemma 1) and preservation (Lemma 2). In particular, we give a mechanical proof of Lemma 2 using the Twelf Meta Theorem prover [28].

Lemma 1. ***(Progress)*** *If P is a solvable puzzle, then there is a rule r in the algorithm from Figure 5 that applies to P.*

Lemma 2. ***(Preservation)*** *If P is a solvable puzzle, and the algorithm from Figure 5 applies rule r to P to produce P', then P' is solvable.*

We also show the optimality of our solution, which we state as Theorem 1 below. To state optimality we need to define the number of displaced pieces. The number n of pieces displaced in a solution of a type-1 puzzle, as found by the algorithm in Figure 5, is determined uniquely by the types of patterns and the number of size 2 Z pieces in the puzzle. There are eight different patterns, shown in Figure 4. We let Z_2 be the number of size 2 Z pieces, and we let $P_i, i \in \{a, \ldots, h\}$ be the number of patterns i in the puzzle board. The algorithm to compute n is given below:

- let $n_d = Z_2 - (P_b + P_d + P_e + P_f)$
- if $n_d \leq 0$
  - then $n \leftarrow 0$
  - else if $P_h \geq n_d$
    - then $n \leftarrow n_d$
    - else $n \leftarrow P_h + 2 \times (n_d - P_h)$

Theorem 1. ***(Optimality)*** *If P is solvable with n displaced pieces, and rule r is applied on P producing P' and causing k displaced pieces, then P' is solvable with at most $n - k$ displaced pieces.*

The proofs of progress, preservation and optimality are given in [24, ch.4].

4 ILP Formulation

We use a 0/1 integer linear programming (ILP) formulation to find a solution to the global coalescing problem. Our ILP model uses three sets: puzzle areas R, puzzle pieces V and a set N of puzzles with one element for each split point in the source program. The set R contains $3m$ elements, where m is the number of columns in the puzzle board. We assume that, for all $i, 0 \leq i \leq m$, areas $2i$ and $2i+1$ alias area $i + 2m$. Figure 6 gives an example. In this case we have two puzzle areas, labeled four and five. Area four is divided into columns zero and one, and area five is divided into columns two and three. We define binary variables p_{nvr} ranging on these three sets. Each p_{nvr} is 1 if piece p has been allocated to the area r of the puzzle n, and is 0 otherwise. Notice that p_{nvr} only exists if the piece v has the same width as area r. For instance, in Figure 6, piece a of puzzle five produces the variables $p_{5a0}, p_{5a1}, p_{5a2}$ and p_{5a3}, but not p_{5a4}, because piece a has width one, and area four has width two.

Following Grund *et al.* [16], we define *affinity variables*. The affinity variable a_{ijvr} is 0 if the puzzle pieces p_{ivr} and p_{jvr} have the same value. This happens when the pieces representing variable v have been assigned to the same puzzle area r across two consecutive puzzles i and j. Affinity variables model the control

flow graph of the source program. Thus, due to affinity edges, our ILP model finds an optimal solution to register coalescing for the whole program, and not only for a single program block. The objective function consists in minimizing the sum of the affinity variables:

$$\min\ f = \sum_{i,j,v,r} a_{ijvr}$$

Our formulation uses three basic types of constraints:

1. Each puzzle piece must be allocated to just one area. That is, given piece v at puzzle n, for each area r with the same width as v we have that:

$$\sum_{r} p_{nvr} = 1$$

2. Each puzzle area must contain at most one piece. That is, given an area, we define four inequalities, one for each region where a piece can be placed. For all p_{nvr} that can be placed on the same region, and all $0 \leq i \leq m$, we have the equations below, where the double summation is due to the double aliasing of T1 puzzles:

$$\sum_{v} p_{nv(2i)} + \sum_{v} p_{nv(2m+i)} \leq 1 \text{ and also } \sum_{v} p_{nv(2i+1)} + \sum_{v} p_{nv(2m+i)} \leq 1$$

3. Each affinity edge a_{ijvr} must be greater than or equal the absolute value of $p_{ivr} - p_{jvr}$.

4.1 Example

As an example, we model the constraints that are produced by the puzzle in Figure 6, i.e, puzzle five from Figure 1. We have numbered the puzzle areas using roman numerals to help our explanation. Also, we have added indices to the variable names, to distinguish those that are part of puzzle five from those that are part of other puzzles. For each of the four quadrants of an area we have a constraint that forces the piece stored in that location to be unique. These constraints are given in Figure 7. Notice that the constraint that refers to an area uses only the variables that may be allocated in that area. In this way, the constraint of area i mentions only pieces a_5, c_5 and d_5.

Figure 8 shows the constraints used to guarantee that each piece will receive a puzzle area. There are four such constraints, one for each variable.

Finally, the affinity edges add 44 equations to our model. These equations are described by the expressions below:

$$\forall(r \in \{0,1,2,3\}, v \in \{a,c,d\}), f_{45vr} \geq p_{4vr} - p_{5vr} \text{ and also } f_{45vr} \geq p_{5vr} - p_{4vr}$$

$$\forall(r \in \{0,1,2,3\}, v \in \{a,d\}), f_{56vr} \geq p_{5vr} - p_{6vr} \text{ and also } f_{56vr} \geq p_{6vr} - p_{5vr}$$

$$\forall(r \in \{4,5\}), f_{56Er} \geq p_{5Er} - p_{6Er} \text{ and also } f_{56Er} \geq p_{6Er} - p_{5Er}$$

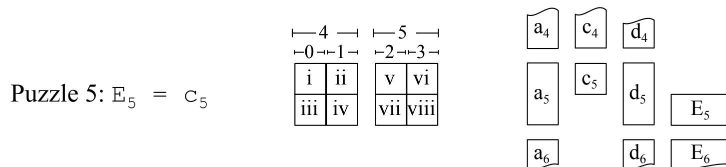


Fig. 6. Puzzle five from Figure 1.

i	$p_{5a0} + p_{5c0} + p_{5d0} \leq 1$	v	$p_{5a2} + p_{5c2} + p_{5d2} \leq 1$
ii	$p_{5a1} + p_{5c1} + p_{5d1} \leq 1$	vi	$p_{5a3} + p_{5c3} + p_{5d3} \leq 1$
iii	$p_{5a0} + p_{5d0} + p_{5E4} \leq 1$	vii	$p_{5a2} + p_{5d2} + p_{5E5} \leq 1$
iv	$p_{5a1} + p_{5d1} + p_{5E4} \leq 1$	viii	$p_{5a2} + p_{5d2} + p_{5E5} \leq 1$

Fig. 7. Constraints asserting that a puzzle area can contain only one piece.

a	$p_{5a0} + p_{5a1} + p_{5a2} + p_{5a3} = 1$	d	$p_{5d0} + p_{5d1} + p_{5d2} + p_{5d3} = 1$
c	$p_{5c0} + p_{5c1} + p_{5c2} + p_{5c3} = 1$	E	$p_{5E4} + p_{5E5} = 1$

Fig. 8. Constraints asserting that a piece must be placed on only one area.

5 Experimental Results

This section empirically validates our punctual coalescing approach. In order to ensure reproducibility, the material used in these experiments is available at `http://homepages.dcc.ufmg.br/~fpereira/projects/puzzles/punctual/`.

Punctual Coalescing in x86 We have implemented our punctual coalescing algorithm on top of the original puzzle solver [26], running on LLVM 2.2 [20]. When compiling SPEC CPU 2000, our implementation is 4% slower than LLVM's default register allocator, an extended version of linear scan [29]. We emphasize that our implementation is a research artifact, whereas LLVM's is an industrial quality software that does not convert the input program into elementary form.

In terms of number of copies, results are very good: no copy was required between two consecutive puzzles in which the follower had an empty puzzle board during the compilation of SPEC CPU 2000. These puzzles account for 89% of the instructions in the source programs. The puzzle solver inserted approximately one copy per each group of 14 puzzles; however, these copies were used to implement fixing code between basic blocks (63% of copies), and to avoid conflicts between program variables and pre-allocated registers (37% of copies); we discuss these issues in Section 6. These results mean that we have not found a pattern such as that in Figure 1 in our benchmarks. However, x86 is an "easy" target for punctual coalescing, because it contains only four aliased registers (`AX, BX, CX`

and DX). Moreover 67% of the puzzles that we found contain only pieces of the same size, in which case it is possible to find a solution for punctual coalescing requiring zero copies [24, ch.4]. Thus, to verify the behavior of our algorithm in a larger puzzle board and with more diverse inputs, we tested it in an artificial architecture, as we describe in the next section.

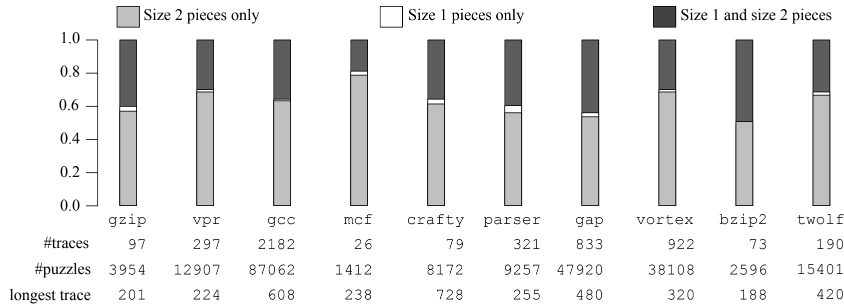
Punctual versus Global Coalescing We have seen, in Section 2, that a sequence of optimal solutions to punctual coalescing may be worse than an optimal solution to global coalescing, even for straight line programs. The objective of this section is to measure this difference and to compare the punctual coalescer with other polynomial-time algorithms. In these experiments, we use LLVM [20] to compile SPEC CPU 2000 to an artificial architecture. LLVM uses a typed intermediate representation, in which integer values have a well known bit width: 1, 8, 16 or 32 bits. We assume a T1 architecture with 32-bit registers, each of them divided into two 16-bit aliases. A register may contain one 32-bit value, or two 1, 8, or 16-bit values. LLVM's IR does not use any form of pre-allocated registers; thus, all the puzzle instances produced have an empty register board.

The nature of the data produced. We use program traces in these experiments, as they are small enough for our ILP solver to handle. A trace is a set of instructions that are executed in sequence. We build traces by concatenating successive basic blocks. For each function in SPEC CPU 2000, we compile the longest trace that we obtain given a depth first traversal of the function's control flow graph. Our longest trace, taken from `186.crafty`, contains 728 puzzles. For each trace, we assume that our target architecture contains exactly the minimal number of registers necessary to compile all its puzzles. This number, called *T1 register pressure*, has a simple formula for puzzles with initially empty boards. In the formula below, Y is the number of size two Y pieces, and y is the number of size one Y pieces; similar notation applies to Z, z, X and x:

$$\text{T1 register pressure} = \lceil (2Y + y + \max((2X + x), (2Z + z)))/2 \rceil \qquad (1)$$

By equaling available registers and register pressure, we ensure that an optimal allocator can find a register assignment without causing spills. Spilling plays no role in the experiments, because the four register assignment algorithms that we compare fit the model explained in Section 2, which decouples register assignment from register spilling [6].

Given the scenario previously described, we obtained the puzzle distribution detailed in Figure 9. We have produced 5,020 traces from the ten integer SPEC CPU 2000 programs that LLVM is able to compile in our system. Together, these traces contain 226,789 puzzles. We distinguish three groups of puzzles: (i) those with all the pieces having size two, (ii) those with all the pieces having size one and (iii) those having pieces of both sizes. We notice that size one pieces are rare: puzzles of group (ii) correspond to less than 2% of all the puzzles, and over 60% of our puzzles are in group (i), thus containing only size two pieces. This discrepancy is due to most C programmers seldom using the `char` and `short` data types, recurring instead to `int`, even to represent boolean variables.



	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
#traces	97	297	2182	26	79	321	833	922	73	190
#puzzles	3954	12907	87062	1412	8172	9257	47920	38108	2596	15401
longest trace	201	224	608	238	728	255	480	320	188	420

Fig. 9. Puzzle distribution obtained from LLVM's intermediate representation. **#traces**: the total number of traces produced. **#puzzles**: the total number of puzzles produced. **longest trace**: size of longest trace, in number of puzzles.

The Competing Coalescers We compare four register assignment algorithms. Two of them are the punctual coalescer of Section 3 and the ILP formulation of Section 4. The other two algorithms are polynomial-time register assignment heuristics: a coalescing oblivious allocator based on the coloring of chordal graphs [25], and the register assignment heuristics used in the original puzzle based allocator [26]. The ILP algorithm uses CPLEX, the two punctual approaches – the optimal and the heuristic – are implemented in C++, and the chordal based allocator is written in Java.

Register allocation via coloring of chordal graphs follows from the fact that programs in *static single assignment* (SSA) [13] form have chordal interference graphs, and thus, can be optimally colored in polynomial time [5, 10, 18]. This property also applies to elementary programs, which are in SSA form [26]. In this experiments, we use the register allocator introduced by Pereira and Palsberg [25]. This chordal allocator is not guaranteed to deliver optimal results in the presence of aliasing. If we fail to find an allocation with n registers, where n is the $T1$ register pressure of the input program, then we re-run the algorithm with $n + 1$ registers. None of our traces has caused such an iteration.

We have included the chordal based approach in these experiments to show how bad a coalescing oblivious algorithm can do compared to an optimal allocator. There exists effective coalescing heuristics for chordal based allocators. Good examples are given by Bouchez *et al.* [7] and Hack *et al.* [17]. However, we do not use these sophisticated coalescing methods. Instead, after color assignment is performed, we use a very simple coalescing heuristics. If we let $G = (V, E, A)$ be an interference graph with a set V of vertices, a set E of interference edges, and a set A of affinity edges, our heuristics is:

$\forall$ affinity edge $(u, v) \in A$ such that $(u, v) \notin E$
    if $\exists$ color c such that c is not assigned to any neighbor of u or v,
        assign c to u and v

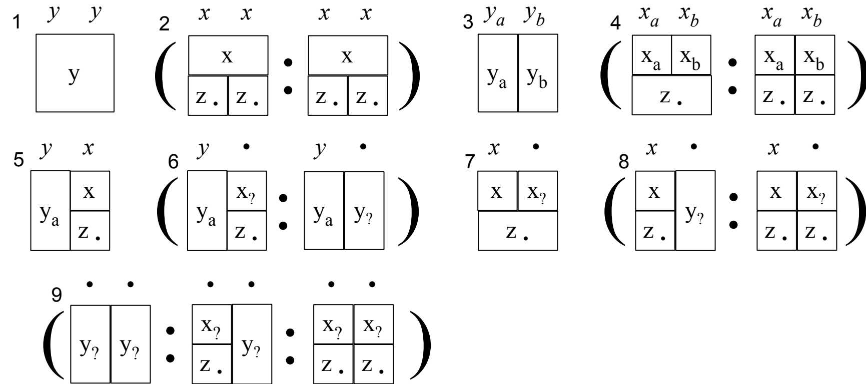


Fig. 10. The puzzle solving program for empty boards used by Pereira and Palsberg [26].

The original puzzle solving heuristics [26] was the inspiration for the punctual algorithm described in Section 3. The original placement rules are shown in Figure 10. The main difference between this program and the program shown in Figure 5 is the arbitrary choice of pieces for areas without preferences. In Figure 5 we use $p_\bullet$ to denote a piece that has no preference for any area, and we use p_a to denote a piece that has preference for a given area a. In Figure 10 we write $p_?$ to indicate that we do not take the preference of piece p into consideration when choosing an area to place it.

Results for SPEC CPU 2000 traces Figure 11 compares the number of copies inserted by the coalescing algorithms. The ILP solver did not finish running on four traces, given a two hours time limit. In total we run the CPLEX solver for 5+ days in order to find solutions to all the traces. In contrast the punctual coalescer, implemented in C++, took 33 seconds to find a register allocation for all the traces, and the original heuristics took 30 seconds. The chordal based algorithm runs for 6+ hours; however, we point that this is a Java program implemented with no concern for fast running time. For any practical purposes, the ILP and the two punctual approaches generate a very small number of copies, hence causing negligible increase in code size. Furthermore, for straight line programs, the optimal punctual approach delivers results that are very close to the ILP method. For instance, our punctual coalescing algorithm required 17 copies to solve the 87,000 puzzles of `gcc`. This is less than one copy per 5,000 puzzles! Only the punctual algorithms – optimal and heuristic – are implemented in LLVM, an there is no runtime performance difference between them. Based on the results of Hack and Goos [17], we speculate that there will be no measurable differences among the four algorithms when targeting x86.

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
Chordal	13,471	47,677	329,783	4,757	46,182	27,082	174,633	199,355	10,581	101,816
Original	13	32	241	1	21	19	135	79	12	44
Punctual	0	10	17	0	1	5	33	1	0	0
ILP	0	2	4	0	0	0	3	0	0	0

Fig. 11. Number of copies inserted by: (chordal) the coalescing oblivious register allocator via coloring of chordal graphs. (original) the coalescing heuristics used in the original puzzle solver [26], (punctual) the algorithm from Section 3, (ILP) the ILP formulation from section 4.

The influence of variable widths on the performance of punctual coalescing. We have observed that the width of the variables found in the traces plays an important role on the quality of the solution produced by punctual coalescing. The width of a variable determines if it fits in half a register, or if it demands a full register. In order to support this observation, we define two types of register pressures: T0 and T1. The T1 register pressure is computed by Equation 1. The T0 register pressure is the register pressure computed assuming a register bank without aliasing, and it is calculated by Equation 2, where X, Y, Z, x, y and z are defined as in Equation 1.

$$\text{T1 register pressure} = Y + y + \max((X + x), (Z + z)) \tag{2}$$

For instance, in the example of Figure 1, the average T1 register pressure is 1.83, and the maximum T1 register pressure is 2. On the other hand, the average T0 register pressure is 2.5, and the maximum T0 register pressure is 3. Figure 12 gives a histogram in which the traces produced from SPEC CPU 2000 are grouped according to the T0 and T1 register pressures. Both numbers are very similar in our benchmarks. On the average, each of our puzzles could be solved with 7.13 registers, assuming no aliasing, and with 7.08, given T1 aliasing. Furthermore, 95.2% of all the traces could be compiled with 16 registers of type T0, whereas 95.4% of the functions could be compiled assuming a T1 target architecture. These numbers are similar because programmers tend to use 32 bit types such as `int` instead of smaller types.

Punctual coalescing tends to produce better results when the T1 pressure is close to the T0 pressure. The intuition behind this fact is simple: for T0 puzzles, if the number of registers is greater than or equal to the maximum register pressure in the trace, then there is a register assignment that requires no copy, and the punctual coalescing strategy discussed in Section 3 trivially finds it. As an illustration, we have inverted the proportion of size one and size two variables presented in Figure 9, obtaining the histograms in Figure 13. In this artificial setting, we have more size one than size two variables, resulting in a conspicuous difference between the T1 and T2 register pressures. The results of global and punctual coalescing in this new context are given in Figure 14. Our punctual technique inserts 20 times more copies than before; however, this number is still negligible given the amount of puzzles solved: one copy per each 160 puzzles.

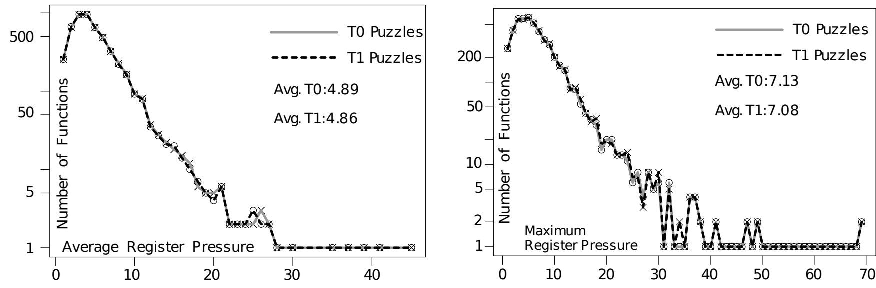


Fig. 12. (Left) Histogram of average register pressures. (Right) Histogram of maximum register pressures.

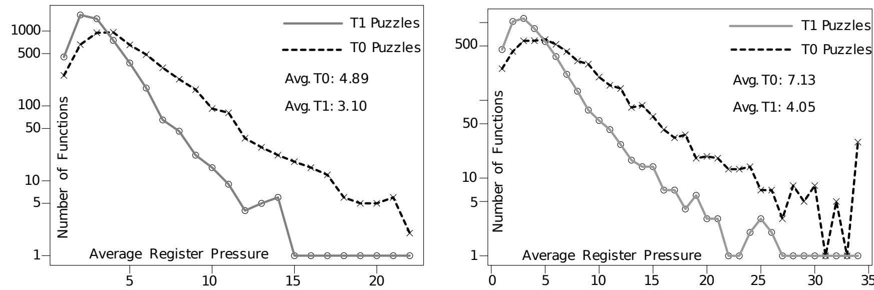


Fig. 13. Histograms obtained by inverting the proportion of size one and size two pieces in our benchmarks. (Left) average register pressures. (Right) maximum register pressures.

Benchmark	`gzip`	`vpr`	`gcc`	`mcf`	`crafty`	`parser`	`gap`	`vortex`	`bzip2`	`twolf`
Chordal	15,160	48,878	337,608	4,746	55,468	26,894	176,097	204,325	10,581	101,747
Original	282	1,025	5,554	91	683	486	2,128	3,686	182	1,456
Punctual	25	108	516	20	29	47	251	288	13	111
ILP	0	2	39	0	1	9	21	0	2	8

Fig. 14. Number of copies inserted by different allocators compiling the traces from Figure 13.

6 Limitations of Punctual Coalescing

The punctual coalescing algorithm of Section 3 may not give optimal results in two situations: settings with two or more guiding puzzles, and settings with non-empty follower boards.

Two or more guiding puzzles stems from a merge in the control-flow graph of the input program. Figure 15 shows an example. The program in Figure 15(a)

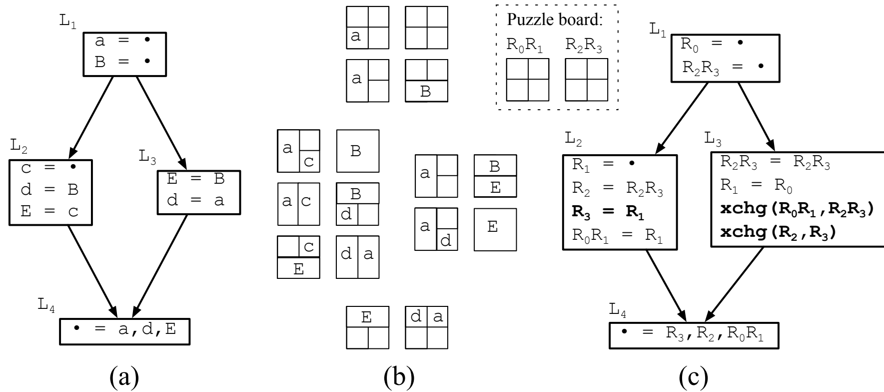


Fig. 15. The complete example, from puzzle solving to code generation.

contains four basic blocks. Three of these blocks – L_1, L_2 and L_4 – form the program trace in Figure 1. If our punctual coalescer traverses this trace first, then it will produce one copy instruction, moving variable a from register R_0 into register R_3, as seen in Figure 1, and shown again in Figure 15(b). However, when performing register assignment in the trace formed by basic block L_3, our coalescer will not take into consideration the mapping of variables to registers in block L_4, previously visited. Thus, it may be necessary to insert fixing code between basic blocks L_3 and L_4. The insertion of this code is analogous to SSA elimination after register allocation, and there are standard algorithms to perform it [27]. Figure 15(c) shows the final assembly program produced; fixing code is shown in bold face. We borrowed the `xchg` instruction, that swaps the contents of two registers, from the x86 lexicon.

The problem of maximizing coalescing in a setting with two or more guiding puzzles is NP-complete. The reduction is from the Global Pinning problem, defined by Rastello *et al.* [30]. However, we have observed that in practice, at least in the x86 architecture, the punctual coalescer produces good results: SSA elimination after register allocation adds approximately 5% more instructions to the final assembly program, and has negligible impact on the run time of compiled programs [26]. As a future work, we will couple register assignment with the static branch prediction technique of Ball and Larus [3] to increase the likelihood that our puzzle solver will traverse hot program paths first.

Non-empty follower boards stem from constraints in the target architecture's instruction set. For instance, x86's `div` instruction always produces a result in register `AX`. Thus, the puzzle board created for a `div` instruction contains the area that corresponds to `AX` initially taken. Punctual coalescing is not guaranteed to deliver optimal results if the follower board contains pre-allocated pieces. Pre-assignment may take away the preferred spot of Y and X pieces. When faced with

pre-allocation we use the original puzzle solving algorithm [26] to eliminate areas containing pre-assigned pieces, and then apply the punctual coalescing program from Figure 5 on the remaining areas. In the x86 experiments, move and swap instructions due to pre-coloring increased the final assembly program in about 2%. Optimal punctual coalescing in face of pre-assignment is an open-problem.

7 Conclusion

This paper has presented punctual coalescing, a technique for reducing the number of copy instructions inserted by tree-scan register allocators that rely on live range splitting to lower register pressure. In addition, this paper gave an optimal solution to global coalescing in register banks with aliasing. A comparison between these two techniques showed that the linear time punctual approach is very close to the exponential time global algorithm for straight line programs. We are currently adapting our punctual algorithm to run on a trace compiler[14].

References

1. Minwook Ahn, Jooyeon Lee, and Yunheung Paek. Optimistic coalescing for heterogeneous register architectures. *SIGPLAN Notices*, 42(7):93–102, 2007.
2. Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM, 2001.
3. Thomas Ball and James R. Larus. Branch prediction for free. In *PLDI*, pages 300–313. ACM, 1993.
4. M. Biró, Mihály Hujter, and Zsolt Tuza. Precoloring extension. I. interval graphs. *Discrete Mathematics*, 100(1-3):267–279, 1992.
5. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.
6. Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, ENS Lyon, 2008.
7. Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES*, pages 147 – 156. ACM, 2008.
8. Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for SSA-form programs. In *CC*, pages 174–189, 2009.
9. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *TOPLAS*, 16(3):428–455, 1994.
10. Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *TCAD*, 25(5):772–779, 2006.
11. G. J. Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.
12. 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.
13. 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.

14. Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Jess Ruderman, Edwin Smith, Rick Reitmair, Mohammad R. Haghighat, Michael Bebenita, Mason Change, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478 . ACM, 2009.
15. Lal George and Andrew W. Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.
16. Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction*, volume 4420, pages 111–115. Springer, 2007.
17. Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *PLDI*, pages 227–237. ACM, 2008.
18. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.
19. Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.
20. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
21. Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation. *Theoretical Computer Science*, 407(1-3):258–273, 2008.
22. 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.
23. Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *IEEE PACT*, pages 196–204, 1998.
24. Fernando Magno Quintao Pereira. *Register Allocation by Puzzle Solving.* PhD thesis, University of California, Los Angeles, 2008.
25. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.
26. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM, 2008.
27. Fernando Magno Quintao Pereira and Jens Palsberg. SSA elimination after register allocation. In *CC*, pages 158 – 173, 2009.
28. Frank Pfenning and Carsten Schürmann. Twelf - a meta-logical framework for deductive systems. In *CADE*, pages 202–206. Springer, 1999.
29. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.
30. Fabrice Rastello, François de Ferriére, and Christophe Guillon. Optimizing translation out of SSA using renaming constraints. Technical Report 03-35, École Normale Supérieure de Lyon, 2003.
31. Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.
32. Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.
33. Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI*, pages 277–288. ACM, 2004.
34. Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI*, pages 142–151. ACM, 1998.