Register Allocation by Puzzle Solving

Fernando Magno Quintão Pereira   Jens Palsberg

UCLA Computer Science Department
University of California, Los Angeles
{fernando,palsberg}@cs.ucla.edu

Abstract

We show that register allocation can be viewed as solving a collection of puzzles. We model the register file as a puzzle board and the program variables as puzzle pieces; pre-coloring and register aliasing fit in naturally. For architectures such as PowerPC, x86, and StrongARM, we can solve the puzzles in polynomial time, and we have augmented the puzzle solver with a simple heuristic for spilling. For SPEC CPU2000, the compilation time of our implementation is as fast as that of the extended version of linear scan used by LLVM, which is the JIT compiler in the openGL stack of Mac OS 10.5. Our implementation produces x86 code that is of similar quality to the code produced by the slower, state-of-the-art iterated register coalescing of George and Appel with the extensions proposed by Smith, Ramsey, and Holloway in 2004.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation

General Terms Algorithms, Theory

Keywords Register allocation, puzzle solving, register aliasing

1. Introduction

Researchers and compiler writers have used a variety of abstractions to model register allocation, including graph coloring [13, 20, 42], integer linear programming [2, 23], partitioned Boolean quadratic optimization [26, 41], and multi-commodity network flow [30]. These abstractions represent different tradeoffs between compilation speed and quality of the produced code. For example, linear scan [39, 44] is a simple algorithm based on the coloring of interval graphs that produces code of reasonable quality with fast compilation time; iterated register coalescing [20] is a more complicated algorithm that, although slower, tends to produce code of better quality than linear scan. Finally, the Appel-George algorithm [2] achieves optimal spilling, with respect to a cost model, in worst-case exponential time via integer linear programming.

In this paper we introduce a new abstraction: register allocation by puzzle solving. We model the register file as a puzzle board and the program variables as puzzle pieces. The result is a collection of puzzles with one puzzle per instruction in the intermediate representation of the source program. We will show that puzzles are easy to use, that we can solve them efficiently, and that they produce code that is competitive with the code produced by state-of-the-art algorithms. Specifically, we will show how for architectures such as PowerPC, x86, and StrongARM we can solve each puzzle in linear time in the number of registers, how we can extend the puzzle solver with a simple heuristic for spilling, and how *pre-coloring* and *register aliasing* fit in naturally. Pre-colored variables are variables that have been assigned to particular registers before register allocation begins; two register names alias [42] when an assignment to one register name can affect the value of the other.

We have implemented a puzzle-based register allocator. Our register allocator has four steps:

1. transform the program into an *elementary program* by augmenting it with special instructions called φ-functions [15], π-functions [5], and parallel copies (using the technique described in Section 2.2);
2. transform the elementary program into a collection of puzzles (using the technique described in Section 2.2);
3. do puzzle solving, spilling, and coalescing (using the techniques described in Sections 3 and 4); and finally
4. transform the elementary program and the register allocation result into assembly code (by implementing φ-functions, π-functions, and parallel copies using the permutations described by Hack *et al.* [25]).

For SPEC CPU2000, our implementation is as fast as the extended version of linear scan used by LLVM, which is the JIT compiler in the openGL stack of Mac OS 10.5. We compare the x86 code produced by gcc, our puzzle solver, the version of linear scan used by LLVM [16], the iterated register coalescing algorithm of George and Appel [20] with the extensions proposed by Smith, Ramsey, and Holloway [42], and the partitioned Boolean quadratic optimization algorithm [26]. The puzzle solver produces code that is, on average, faster than the code produced by extended linear scan, and of similar quality to the code produced by iterated register coalescing.

The key insight of the puzzles approach lies in the use of elementary programs, which are described in Section 2.2. In an elementary program, all live ranges are small and that enables us to define and solve one puzzle for each instruction in the program.

In the following section we define our puzzles and in Section 3 we show how to solve them. In Section 4 we present our approach to spilling and coalescing, and in Section 5 we discuss some optimizations in the puzzle solver. We give our experimental results in Section 6, and we discuss related work in Section 7. Finally, Section 8 concludes the paper. The appendices contain proofs of the theorems stated in the paper.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06…$5.00

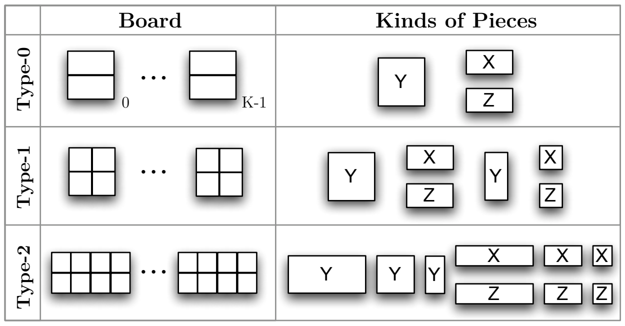


Figure 1. Three types of puzzles.

2. Puzzles

A puzzle consists of a *board* and a set of *pieces*. Pieces cannot overlap on the board, and a subset of the pieces are already placed on the board. The *challenge* is to fit the remaining pieces on the board.

We will now explain how to map a register file to a puzzle board and how to map program variables to puzzle pieces. Every resulting puzzle will be of one of the three types illustrated in Figure 1 or a hybrid.

2.1 From Register File to Puzzle Board

The bank of registers in the target architecture determines the shape of the puzzle board. Every puzzle board has a number of separate *areas*, where each area is divided into two rows of *squares*. We will explain in Section 2.2 why an area has exactly *two* rows. The register file may support aliasing, which determines the number of columns in each area, the valid shapes of the pieces, and the rules for placing the pieces on the board. We distinguish three types of puzzles: type-0, type-1 and type-2, where each area of a type-n puzzle has 2^n columns.

Type-0 puzzles. The bank of registers used in PowerPC and the bank of integer registers used in ARM are simple cases because they do not support register aliasing. Figure 2(a) shows the puzzle board for PowerPC. Every area has just one column that corresponds to one of the 32 registers. Both PowerPC and ARM give a type-0 puzzle for which the pieces are of the three kinds shown in Figure 1. We can place an X-piece on any square in the upper row, we can place a Z-piece on any square in the lower row, and we can place a Y-piece on any column. It is straightforward to see that we can solve a type-0 puzzle in linear time in the number of areas by first placing all the Y-pieces on the board and then placing all the X-pieces and Z-pieces on the board.

Type-1 puzzles. Figure 2(b) shows the puzzle board for the floating point registers used in the ARM architecture. This register bank has 32 single precision registers that can be combined into 16 pairs of double precision registers. Thus, every area of this puzzle board has two columns, which correspond to the two registers that can be paired. For example, the 32-bit registers S0 and S1 are in the same area because they can be combined into the 64-bit register D0. Similarly, because S1 and S2 cannot be combined into a double register, they denote columns in different areas. ARM gives a type-1 puzzle for which the pieces are of the six kinds shown in Figure 1. We define the *size* of a piece as the number of squares that it occupies on the board. We can place a size-1 X-piece on any square in the upper row, a size-2 X-piece on the two upper squares of any area, a size-1 Z-piece on any square in the lower row, a size-2 Z-piece on the two lower squares of any area, a size-2 Y-piece on any column, and a size-4 Y-piece on any area. Section 3 explains how to solve a type-1 puzzle in linear time in the number of areas.

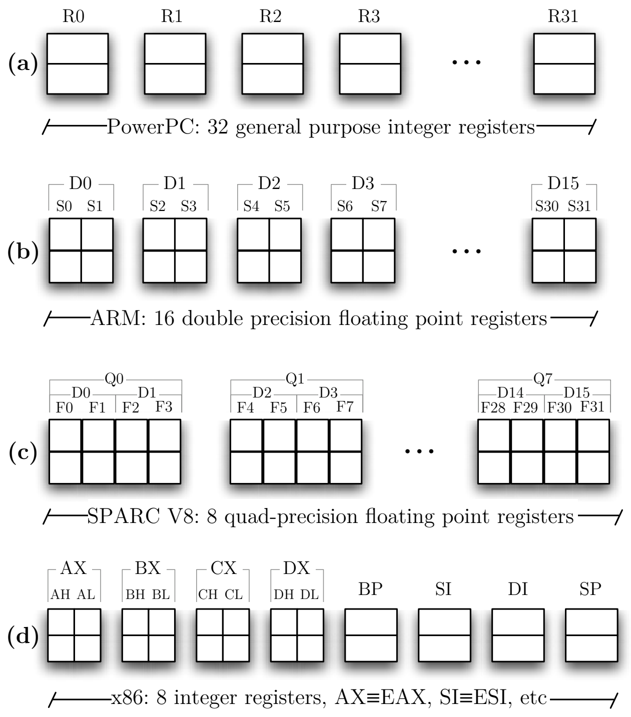

R0
R1
R2
R3
R31
(a)
PowerPC: 32 general purpose integer registers
D0
D1
D2
D3
D15
S0 S1
S2 S3
S4 S5
S6 S7
S30 S31
(b)
ARM: 16 double precision floating point registers
Q0
Q1
Q7
D0 D1
D2 D3
D14 D15
F0 F1 F2 F3
F4 F5 F6 F7
F28 F29 F30 F31
(c)
SPARC V8: 8 quad-precision floating point registers
AX
BX
CX
DX
AH AL
BH BL
CH CL
DH DL
BP
SI
DI
SP
(d)
x86: 8 integer registers, AX≡EAX, SI≡ESI, etc

Figure 2. Examples of register banks mapped into puzzle boards.

Type-2 puzzles. SPARC V8 [27, pp 33] supports two levels of register aliasing: first, two 32-bit floating-point registers can be combined to hold a single 64-bit value; then, two of these 64-bit registers can be combined yet again to hold a 128-bit value. Figure 2(c) shows the puzzle board for the floating point registers of SPARC V8. Every area has four columns corresponding to four registers that can be combined. This architecture gives a type-2 puzzle for which the pieces are of the nine kinds shown in Figure 1. The rules for placing the pieces on the board are a straightforward extension of the rules for type-1 puzzles. Importantly, we can place a size-2 X-piece on either the first two squares in the upper row of an area, or on the last two squares in the upper row of an area. A similar rule applies to size-2 Z-pieces. Solving type-2 puzzles remains an open problem.

Hybrid puzzles. The x86 gives a hybrid of type-0 and type-1 puzzles. Figure 3 shows the integer-register file of the x86, and Figure 2(d) shows the corresponding puzzle board. The registers `AX`, `BX`, `CX`, `DX` give a type-1 puzzle, while the registers `EBP`, `ESI`, `EDI`, `ESP` give a type-0 puzzle. We treat the `EAX`, `EBX`, `ECX`, `EDX` registers as special cases of the `AX`, `BX`, `CX`, `DX` registers; values in `EAX`, `EBX`, `ECX`, `EDX` take up to 32 bits rather than 16 bits. Notice that x86 does not give a type-2 puzzle because even though we can fit four 8-bit values into a 32-bit register, x86 does not provide register names for the upper 16-bit portion of that register. For a hybrid of type-1 and type-0 puzzles, we first solve the type-0 puzzles and then the type-1 puzzles.

The floating point registers of SPARC V9 [45, pp 36-40] give a hybrid of a type-2 and a type-1 puzzle because half the registers can be combined into quad precision registers.

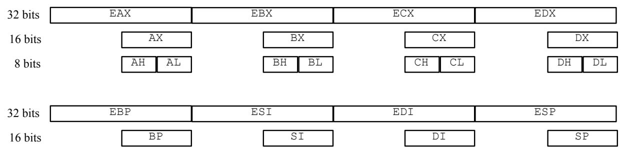


Figure 3. General purpose registers of the x86 architecture

2.2 From Program Variables to Puzzle Pieces

We map program variables to puzzle pieces in a two-step process: first we convert a source program into an *elementary program* and then we map the elementary program into puzzle pieces.

From a source program to an elementary program. We can convert an ordinary program into an *elementary program* in three steps. First, we transform the source program into static single assignment (SSA) form [15] by renaming variables and adding φ-functions at the beginning of each basic block. Second, we transform the SSA-form program into static single information (SSI) form [1]. In our flavor of SSI form, every basic block ends with a π-function that renames the variables that are live going out of the basic block. (The name π-assignment was coined by Bodik *et al.* [5]. It was originally called σ-function in [1], and *switch operators* in [28].) Finally, we transform the SSI-form program into an elementary program by inserting a parallel copy between each pair of consecutive instructions in a basic block, and renaming the variables alive at that point. Appel and George used the idea of inserting parallel copies everywhere in their ILP-based approach to register allocation with optimal spilling [2]. In summary, in an elementary program, every basic block begins with a φ-function, has a parallel copy between each consecutive pair of instructions, and ends with a π-function. Figure 4(a) shows a program, and Figure 4(b) gives the corresponding elementary program. As an optimization, we have removed useless φ-functions from the beginning of blocks with a single predecessor. In this paper we adopt the convention that lower case letters denote variables that can be stored into a single register, and upper case letters denote variables that must be stored into a pair of registers. Names in typewriter font, e.g., `AL`, denote pre-colored registers. We use $x = y$ to denote an instruction that uses y and defines x; it is not a simple copy.

Cytron *et al.* [15] gave a polynomial time algorithm to convert a program into SSA form, and Ananian [1] gave a polynomial time algorithm to convert a program into SSI form. We can perform the step of inserting parallel copies in polynomial time as well.

From an elementary program to puzzle pieces. A *program point* [2] is a point between any pair of consecutive instructions. For example, the program points in Figure 4(b) are $p_0, \ldots, p_{11}$. The collection of program points where a variable v is alive constitutes its *live range*. The live ranges of programs in elementary form contain at most two program points. A variable v is said to be *live-in* at instruction i if its live range contains a program point that precedes i; v is *live-out* at i if v's live range contains a program point that succeeds i. For each instruction i in an elementary program we create a puzzle that has one piece for each variable that is live in or live out at i (or both). The live ranges that end at i become X-pieces; the live ranges that begin at i become Z-pieces; and the live ranges that cross i become Y-pieces. Figure 5 gives an example of a program fragment that uses six variables, and it shows their live ranges and the resulting puzzle pieces.

We can now explain why each area of a puzzle board has exactly two rows. We can assign a register both to one live range that ends in the middle and to one live range that begins in the middle. We model that by placing an X-piece in the upper row and a Z-piece right below in the lower row. However, if we assign a register to a long live range, then we cannot assign that register to any other live range. We model that by placing a Y-piece, which spans *both* rows.

The sizes of the pieces are given by the types of the variables. For example, for x86, an 8-bit variable with a live range that ends in the middle becomes a size-1 X-piece, while a 16 or 32-bit variable with a live range that ends in the middle becomes a size-2 X-piece. Similarly, an 8-bit variable with a live range that begins in the middle becomes a size-1 Z-piece, while a 16 or 32-bit variable with a live range that ends in the middle becomes a size-2 Z-piece. An 8-bit variable with a long live range becomes a size-2 Y-piece, while a 16-bit variable with a long live range becomes a size-4 Y-piece. Figure 9(a) shows the puzzles produced for the program in Figure 4(b).

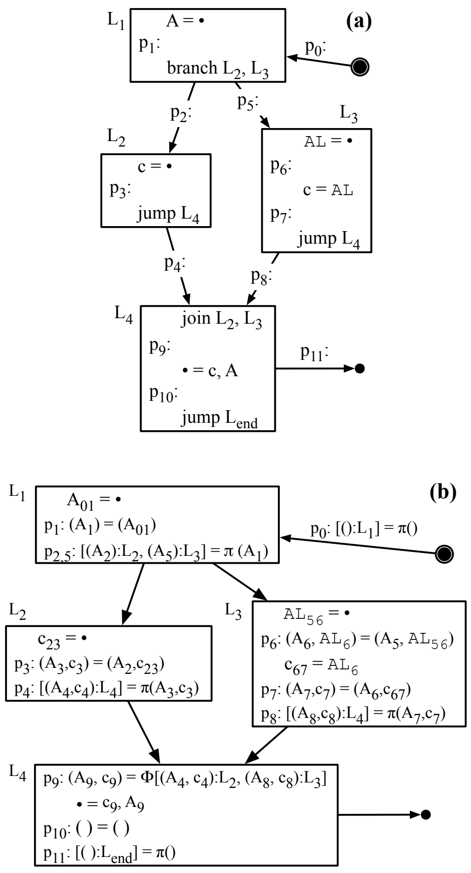


Figure 4. (a) Original program. (b) Elementary program.

2.3 Register Allocation and Puzzle Solving are Equivalent

The core register allocation problem, also known as *spill-free register allocation*, is: given a program P and a number K of available registers, can each of the variables of P be mapped to one of the K registers such that variables with interfering live ranges are assigned to different registers?

In case some of the variables are pre-colored, we call the problem *spill-free register allocation with pre-coloring*.

THEOREM 1. **(Equivalence)** *Spill-free register allocation with pre-coloring for an elementary program is equivalent to solving a collection of puzzles.*

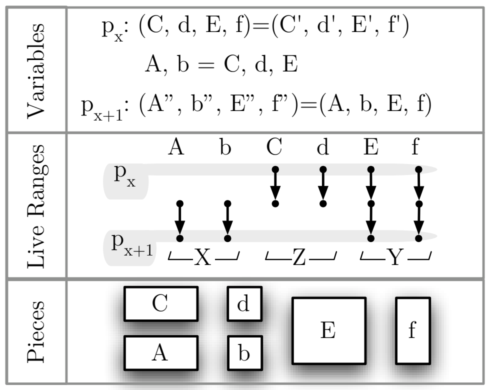


Figure 5. Mapping program variables into puzzle pieces.

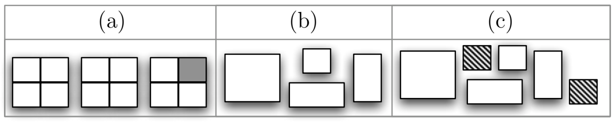


Figure 6. Padding: (a) puzzle board, (b) pieces before padding, (c) pieces after padding. The new pieces are marked with stripes.

Proof. See Appendix A. □

3. Solving Type-1 Puzzles

Figure 8 shows our algorithm for solving type-1 puzzles. Our algorithmic notation is visual rather than textual. The goal of this section is to explain how the algorithm works and to point out several subtleties. We will do that in two steps. First we will define a visual language of puzzle solving programs that includes the program in Figure 8. After explaining the semantics of the whole language, we then focus on the program in Figure 8 and explain how seemingly innocent changes to the program would make it incorrect.

We will study puzzle-solving programs that work by completing *one area at a time*. To enable that approach, we may have to *pad* a puzzle before the solution process begins. If a puzzle has a set of pieces with a total area that is less than the total area of the puzzle board, then a strategy that completes one area at a time may get stuck unnecessarily because of a lack of pieces. So, we pad such puzzles by adding size-1 X-pieces and size-1 Z-pieces, until these two properties are met: (i) the total area of the X-pieces equals the total area of the Z-pieces; (ii) the total area of all the pieces is $4K$, where K is the number of areas on the board. Note that *total area* includes also pre-colored squares. Figure 6 illustrates padding. Lemma 16 in the Appendix shows that a puzzle is solvable if and only if its padded version is solvable.

3.1 A Visual Language of Puzzle Solving Programs

We say that an area is *complete* when all four of its squares are covered by pieces; dually, an area is *empty* when none of its four squares are covered by pieces.

The grammar in Figure 7 defines a visual language for programming type-1 puzzle solvers: a program is a sequence of statements, and a statement is either a rule r or a conditional statement $r : s$.

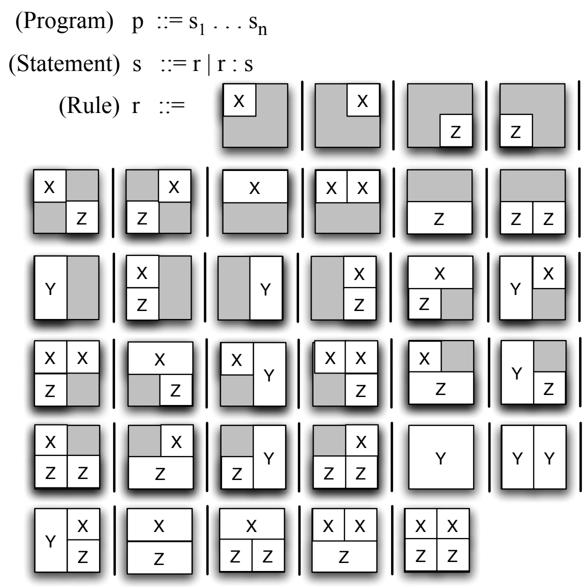


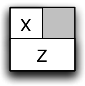
Figure 7. A visual language for programming puzzle solvers.

We now informally explain the meaning of rules, statements, and programs.

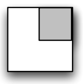
Rules. A rule explains how to complete an area. We write a rule as a two-by-two diagram with two facets: a *pattern*, that is, dark areas which show the squares (if any) that have to be filled in already for the rule to apply; and a *strategy*, that is, a description of how to complete the area, including which pieces to use and where to put them. We say that the pattern of a rule *matches* an area a if the pattern is the same as the already-filled-in squares of 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 the pieces needed by the strategy of r are placed in a;
- the application of r to a *fails* otherwise.

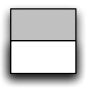
For example, the rule



has a pattern consisting of just one square—namely, the square in the top-right corner, and a strategy consisting of taking one size-1 X-piece and one size-2 Z-piece and placing the X-piece in the top-left corner and placing the Z-piece in the bottom row. If we apply the rule to the area



and one size-1 X-piece and one size-2 Z-piece are available, then the result is that the two pieces are placed in the area, and the rule succeeds. Otherwise, if one or both of the two needed pieces are not available, then the rule fails. We cannot apply the rule to the area



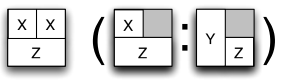
because the pattern of the rule does not match this area.

Statements. For a statement that is simply a rule r, we have explained above how to apply r to an area a where the pattern of r matches a. For a conditional statement $r : s$, we require all the rules in $r : s$ to have the *same* pattern, which we call the pattern of $r : s$. For a conditional statement $r : s$ and an area a where the pattern of $r : s$ matches a, the application of $r : s$ to a proceeds by first applying r to a; if that application succeeds, then $r : s$ succeeds (and s is ignored); otherwise the result of $r : s$ is the application of the statement s to a.

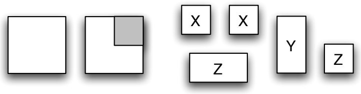
Programs. The execution of a program $s_1 \ldots s_n$ on a puzzle $\mathcal{P}$ proceeds as follows:

- For each i from 1 to n:
  - For each area a of $\mathcal{P}$ such that the pattern of s_i matches a:
    - apply s_i to a
    - if the application of s_i to a failed, then terminate the entire execution and report failure

Example. Let us consider in detail the execution of the program

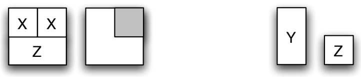


on the puzzle



.

The first statement has a pattern which matches only the first area of the puzzle. So, we apply the first statement to the first area, which succeeds and results in the following puzzle.



.

The second statement has a pattern which matches only the second area of the puzzle. So, we apply the second statement to the second area. The second statement is a conditional statement, so we first apply the first rule of the second statement. That rule fails because the pieces needed by the strategy of that rule are not available. We then move on to apply the second rule of the second statement. That rule succeeds and completes the puzzle.

Time Complexity. It is straightforward to implement the application of a rule to an area in constant time. A program executes $O(1)$ rules on each area of a board. So, the execution of a program on a board with K areas takes $O(K)$ time.

3.2 Our Puzzle Solving Program

Figure 8 shows our puzzle solving program, which has 15 numbered statements. Notice that the 15 statements have pairwise different patterns; each statement completes the areas with a particular pattern. While our program may appear simple and straightforward, the ordering of the statements and the ordering of the rules in conditional statements are in several cases crucial for correctness. In general our program tries to fill the most constrained patterns first. For example, statements 1–8 can only be filled in one way, while the other statements admit two or more solutions. We will discuss four such subtleties.

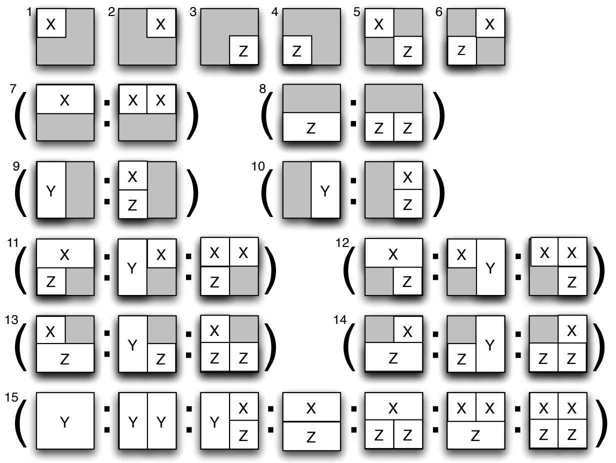
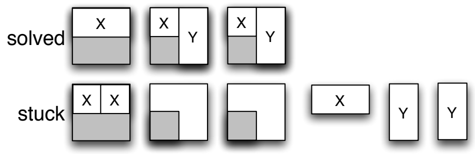


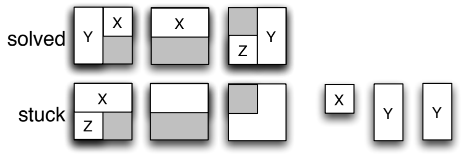
Figure 8. Our puzzle solving program

First, it is imperative that in statement 7 our program prefers a size-2 X-piece over two size-1 X-pieces. Suppose we replace statement 7 with a statement 7′ which swaps the order of the two rules in statement 7. The application of statement 7′ can take us from a solvable puzzle to an unsolvable puzzle, for example:



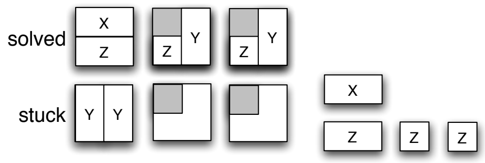
Because statement 7 prefers a size-2 X-piece over two size-1 X-pieces, the example is impossible. Notice that our program also prefers the size-2 pieces over the size-1 pieces in statements 8–15 for reasons similar to our analysis of statement 7.

Second, it is critical that statements 7–10 come before statements 11–14. Suppose we swap the order of the two subsequences of statements. The application of rule 11 can now take us from a solvable puzzle to an unsolvable puzzle, for example:



Notice that the example uses an area in which two squares are filled in. Because statements 7–10 come before statements 11–14, the example is impossible.

Third, it is crucial that statements 11–14 come before statement 15. Suppose we swap the order such that statement 15 comes before statements 11–14. The application of rule 15 can now take us from a solvable puzzle to an unsolvable puzzle, for example:



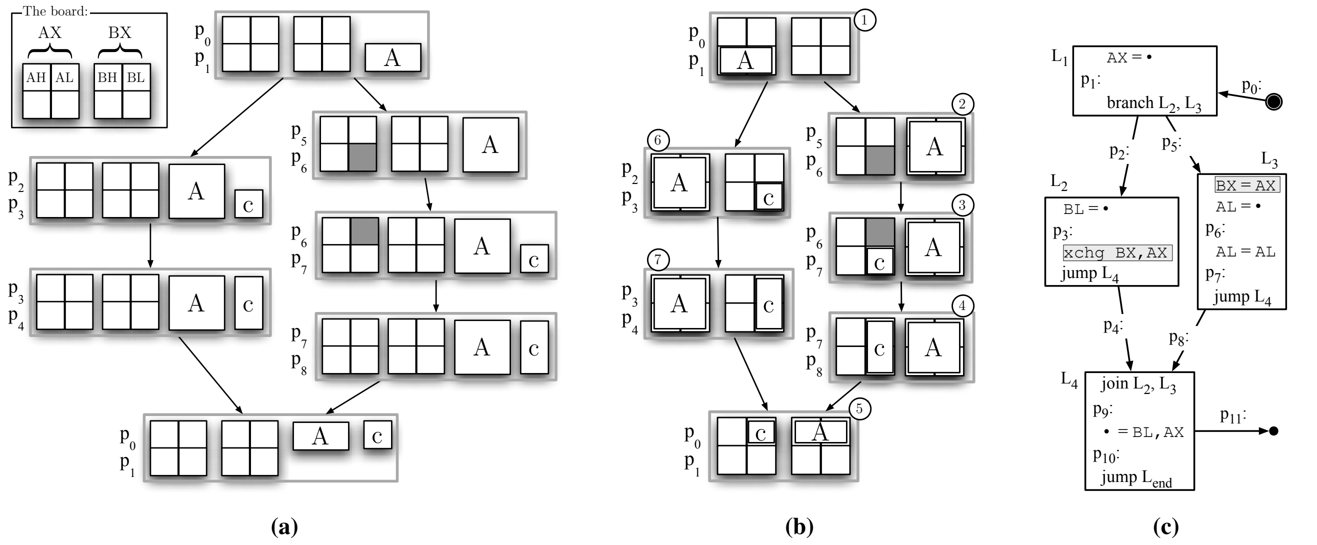
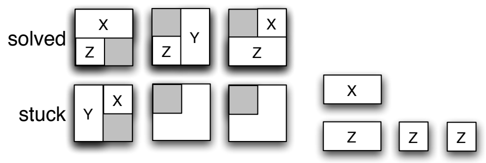


Figure 9. (a) The puzzles produced for the program given in Figure 4(b). (b) An example solution. (c) The final program.

Notice that the example uses an area in which one square is filled in. Because statements 11–14 come before statement 15, the example is impossible.

Fourth, it is essential that in statement 11, the rules come in exactly the order given in our program. Suppose we replace statement 11 with a statement 11′ which swaps the order of the first two rules of statement 11. The application of statement 11′ can take us from a solvable puzzle to an unsolvable puzzle. For example:



When we use the statement 11 given in our program, this situation cannot occur. Notice that our program makes a similar choice in statements 12–14; all for reasons similar to our analysis of statement 11.

THEOREM 2. **(Correctness)** *A type-1 puzzle is solvable if and only if our program succeeds on the puzzle.*

Proof. See Appendix B. □

For an elementary program P, we generate $|P|$ puzzles, each of which we can solve in linear time in the number of registers. So, we have Corollary 3.

COROLLARY 3. **(Complexity)** *Spill-free register allocation with pre-coloring for an elementary program P and $2K$ registers is solvable in $O(|P| \times K)$ time.*

A solution for the collection of puzzles in Figure 9(a) is shown in Figure 9(b). For simplicity, the puzzles in Figure 9 are not padded.

4. Spilling and Coalescing

We now present our approach to spilling and coalescing. Figure 10 shows the combined step of puzzle solving, spilling, and coalescing.

Spilling. If the polynomial-time algorithm of Theorem 3 succeeds, then all the variables in the program from which the puzzles were generated can be placed in registers. However, the algorithm may fail, implying that the need for registers exceeds the number of available registers. In that situation, the register allocator faces the task of choosing which variables will be placed in registers and which variables will be *spilled*, that is, placed in memory. The goal is to spill as few variables as possible.

We use a simple spilling heuristic. The heuristic is based on the observation that when we convert a program P into elementary form, each of P's variables is represented by a *family of variables* in the elementary program. For example, the variable c in Figure 4(a) is represented by the family of variables $\{c_{23}, c_3, c_4, c_{67}, c_7, c_8, c_9\}$ in Figure 4(b). When we spill a variable in an elementary program, we choose to simultaneously spill *all* the variables in its family and thereby reduce the number of pieces in many puzzles at the same time. The problem of *register allocation with pre-coloring and spilling of families of variables* is to perform register allocation with pre-coloring while spilling as few families of variables as possible.

THEOREM 4. **(Hardness)** *Register allocation with pre-coloring and spilling of families of variables for an elementary program is NP-complete.*

Proof. See Appendix C. □

Theorem 4 justifies our use of a spilling heuristic rather than an algorithm that solves the problem optimally. Figure 10 contains a while-loop that implements the heuristic; a more detailed version of this code is given in [38]. It is straightforward to see that the heuristic visits each puzzle once, that it always terminates, and that when it terminates, all puzzles have been solved.

In order to avoid separating registers to reload spilled variables only certain pieces can be removed from an unsolved puzzle. These pieces represent variables that are neither used nor defined in the instruction that gave origin to the puzzle. For instance, only the Y piece f can be removed from the puzzle in Figure 5. When choosing a piece to be removed from a puzzle, we use the "furthest-first" strategy of Belady [3] that was later used by Poletto and Sarkar [39] in linear-scan register allocation. The furthest-first strategy spills a

```
• $S$ = empty
• For each puzzle $p$, in a preorder traversal of the dominator tree
  of the program:
    ▪ while $p$ is not solvable:
        – choose and remove a piece $s$ from $p$, and for every
          subsequent puzzle $p'$ that contains a variable $s'$ in the
          family of $s$, remove $s'$ from $p'$.
    ▪ $S'$ = a solution of $p$, guided by $S$
    ▪ $S$ = $S'$
```

Figure 10. Register allocation with spilling and local coalescing

family of variables whose live ranges extend the furthest, according to a linearization determined by a depth first traversal of the dominator tree of the source program. We do not give preference to any path. Giving preference to a path would be particularly worthwhile when profiling information is available.

The total number of puzzles that will be solved during a run of our heuristic is bounded by $|P| + |\mathcal{F}|$, where $|P|$ denotes the number of puzzles and $|\mathcal{F}|$ denotes the number of families of variables, that is, the number of variables in the source program.

Coalescing. Traditionally, the task of register coalescing is to assign the same register to the variables x and y in a copy statement $x = y$, thereby avoiding the generation of code for that statement. An elementary program contains many parallel copy statements and therefore many opportunities for a form of register coalescing. We use an approach that we call *local coalescing*. The goal of local coalescing is to allocate variables in the same family to the same register, as much as possible. Local coalescing traverses the dominator tree of the elementary program in preorder and solves each puzzle guided by the solution to the *previous puzzle*, as shown in Figure 10. In Figure 9(b), the numbers next to each puzzle denote the order in which the puzzles were solved.

The pre-ordering has the good property that every time a puzzle corresponding to statement i is solved, all the families of variables that are defined at program points that dominate i have already been given at least one location. The puzzle solver can then try to assign to the piece that represents variable v the same register that was assigned to other variables in v's family. For instance, in Figure 4(b), when solving the puzzle between p_2 and p_3, the puzzle solver tries to match the registers assigned to A_2 and A_3. This optimization is possible because A_2 is defined at a program point that dominates the definition site of A_3, and thus is visited before.

During the traversal of the dominator tree, the physical location of each live variable is kept in a vector. If a spilled variable is reloaded when solving a puzzle, it stays in a register until another puzzle, possibly many instructions after the reloading point, forces it to be evicted again. Our approach to handling reloaded variables is somewhat similar to the second-chance allocation described by Traub *et al.* [44].

Figure 9(c) shows the assembly code produced by the puzzle solver for our running example. We have highlighted the instructions used to implement parallel copies. The x86 instruction `xchg` swaps the contents of two registers.

5. Optimizations

In this section we introduce three optimizations that we have found useful in our implementation of register allocation by puzzle solving for x86. We call these optimizations: first-chance global coalescing, load lowering and Redundant Memory Transfer Elimination (RMTE). Before delving into these optimizations, we describe some implementation details that help to understand our optimizations.

Size of the intermediate representation. An elementary program has many more variable names than an ordinary program; fortunately, we do not have to keep any of these extra names. Our solver uses only one puzzle board at any time: given an instruction i, variables alive before and after i are renamed when the solver builds the puzzle that represents i. Once the puzzle is solved, we use its solution to rewrite i and we discard the extra names. The parallel copy between two consecutive instructions i_1 and i_2 in the same basic block can be implemented right after the puzzle representing i_2 is solved.

Critical Edges and Conventional SSA-form. Before solving puzzles, our algorithm performs two transformations in the target control flow graph that, although not essential to the correctness of our allocator, greatly simplify the elimination of φ-functions and π-functions. The first transformation, commonly described in compiler text books, removes critical edges from the control flow graph. These are edges between a basic block with multiple successors and a basic block with multiple predecessors [9]. The second transformation converts the target program into a variation of SSA-form called *Conventional SSA-form* (CSSA) [43]. Programs in this form have the following property: if two variables v_1 and v_2 are related by a parallel copy, e.g.: $(\ldots, v_1, \ldots) = (\ldots, v_2, \ldots)$, then the live ranges of v_1 and v_2 do not overlap. Hence, if these variables are spilled, the register allocator can assign them to the same memory slot. A fast algorithm to perform the SSA-to-CSSA conversion is given in [12]. These two transformations are enough to handle the 'swap' and 'lost-copy' problems pointed out by Briggs *et al.* [9].

Implementing φ-functions and π-functions. The allocator maintains a table with the solution of the first and last puzzles solved in each basic block. These solutions are used to guide the elimination of φ-functions and π-functions. During the implementation of parallel copies, the ability to swap register values is necessary to preserve the register pressure found during the register assignment phase [7, 37]. Some architectures, such as x86, provide instructions to swap the values in registers. In systems where this is not the case, swaps can be performed using xor instructions.

5.1 First-chance Coalescing

We will say that two variables are φ-related if they are syntactically related by a common φ-function. For instance, an instruction such as $v = \varphi(\ldots, u, \ldots, w, \ldots)$ would cause the variables u, v and w to be φ-related. We will call *Global Coalescing* the optimization that attempts to minimize the number of physical registers assigned to families of φ-related variables. In our case it is always safe to assign the same physical register to φ-related variables, because the CSSA form guarantees that these variables cannot interfere.

Our instance of the global coalescing problem is NP-complete, as proved by Bouchez *et al.* [8]. Thus, in order to preserve our fast compilation time, we use a simple heuristics to perform limited global coalescing. In our representation, every variable is part of an equivalence class of φ-related names. If a variable is never defined nor used by any φ-function, then its equivalence class is a singleton. Upon deciding on which part of the puzzle board a newly defined variable v should fit, we check if its equivalence class has a *preferred slot*. If that is not the case, then we choose any available position p and allocate p to v; we then make p the preferred slot of the equivalence class that contains v. Otherwise, we check if the preferred position is vacant, and if possible, we assign it to v, otherwise we let the puzzle solver pick a place for v according to the algorithm described in Section 2.2. As an optimization, we allow two preferred registers per equivalence class, one with high priority, and another with low priority. Our algorithm always attempts to pick the high priority slot first, and if that fails, it tries the low priority position.

5.2 Elimination of Redundant Memory Copies

We use a linear time pass after register assignment to remove redundant memory transfers. A memory transfer is a sequence of instructions that copies a value from a memory location m_1 to another memory location m_2. The transfer is said to be redundant if these locations are the same. In our puzzle solver, redundant memory transfers arise due to copy instructions between two φ-related variables, plus the fact that our allocator assigns the same memory location to φ-related variables. For instance, consider a copy $a = b$, where a and b are φ-related. If the puzzle solver reaches this instruction in a state where variable b is not in any register, but in memory location m, then it will add a load $r = m$ before that instruction, where r is the register allocated to variable a. Later on, if a is spilled, the register allocator will insert a store $m = r$ after the instruction. This load/store sequence, e.g $r = m; m = r$ is redundant and can safely be eliminated.

5.3 Load Lowering

Load lowering is an optimization that aims at reducing the total number of loads in the target program. This optimization fits in allocators that follow the second-chance bin-packing model, e.g [30, 40, 44]. A common characteristics of these algorithms is the fact that, once a value is reloaded, it stays in registers until it is either no longer necessary, or it has to be spilled again. Although the second chance bin-packing approach tends to avoid successive reloads of spilled variables, allocators must be carefully implemented to avoid inserting useless reload instructions in the target code.

In our case, we postpone the insertion of spill code until all the variables have been assigned to registers. We store the result of the first and last puzzle solved for each basic block. If a variable v is not on the puzzle board of a last puzzle p_l, but it is on the board of a first puzzle p_f, and p_f is the puzzle that succeeds p_l, we must load v into the register assigned to it in p_f. This case happens when a value reaches a join point in memory through a branch and in register trough another branch. Situations like this may occur many times, causing the insertion of many loads in the final program to preserve the value of variables that are used a few times. Figure 11 illustrates an example. The numbers labeling each basic block denote the order in which those basic blocks are visited by the puzzle solver. The puzzle solver is always able to find a board slot for variable v along the path from basic block 1 to 2 and from basic block 2 to 3. However, when visiting basic block 4, the solver has to spill v, in order to find a vacant slot to variable u. This spilling causes variable v to be on memory along the dashed line. During the elimination of π and φ-functions, we would have to insert loads as shown in Figure 11 (b), and clearly the first load is redundant.

In order to avoid inserting redundant loads without having to recur to expensive liveness analysis based algorithms, we perform load lowering. If the register allocator needs to insert more reloads for a variable than its number of uses, then we replace all the reloads for loads immediately before each use. As we show in Section 6, this simple heuristics is able to reduce in about 10% the number of loads inserted into the final program. Load lowering also reduces the number of copies in the target program. This happens because, once a variable is reloaded immediately before its use, all the copies that have been inserted before, to keep the variable in registers while it had not yet been spilled become redundant and can be removed. Load lowering is a linear pass, and it can be performed along the removal of redundant memory transfers. Its time is negligible compared to the time taken by register assignment.

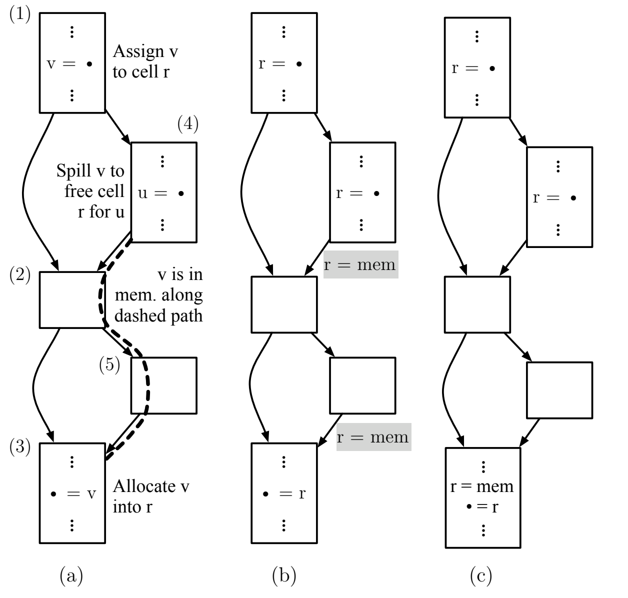


Figure 11. (a) Example program (b) SSA elimination without load-lowering. (c) Load-lowering in action.

6. Experimental Results

Experimental platform. We have implemented our register allocator in the LLVM compiler framework [31], version 1.9. LLVM is the JIT compiler in the openGL stack of Mac OS 10.5. Our tests are executed on a 32-bit x86 Intel(R) Xeon(TM), with a 3.06GHz cpu clock, 3GB of free memory (as shown by the linux command `free`) and 512KB L1 cache running Red Hat Linux 3.3.3-7.

Benchmark characteristics. The LLVM distribution provides a broad variety of benchmarks: our implementation has compiled and run over 1.3 million lines of C code. LLVM 1.9 and our puzzle solver pass the same suite of benchmarks. In this section we will present measurements based on the SPEC CPU2000 benchmarks. Some characteristics of these benchmarks are given in Figure 12. All the figures use short names for the benchmarks; the full names are given in Figure 12. We order these benchmarks by the number of non-empty puzzles that they produce, which is given in Figure 14.

Puzzle characteristics. Figure 13 counts the types of puzzles generated from SPEC CPU2000. A total of 3.45% of the puzzles have pieces of different sizes plus pre-colored areas so they exercise all aspects of the puzzle solver. Most of the puzzles are simpler: 5.18% of them are empty, *i.e.*, have no pieces; 58.16% have only pieces of the same size, and 83.66% have an empty board with no pre-colored areas. Just 226 puzzles contained only short pieces with precolored areas and we omit them from the chart.

As we show in Figure 14, 94.6% of the nonempty puzzles in SPEC CPU2000 can be solved in the first try. When this is not the case, our spilling heuristic allows for solving a puzzle multiple times with a decreasing number of pieces until a solution is found. Figure 14 reports the average number of times that the puzzle solver had to be called per nonempty puzzle. On average, we solve each nonempty puzzle 1.05 times.

Number of moves/swaps inserted by the puzzle solver. Figure 15 shows the number of copy and swap instructions inserted by the puzzle solver in each of the compiled benchmarks. Local copies denote instructions used by the puzzle solver to implement parallel copies between two consecutive puzzles inside the same basic

	Benchmark	LoC	asm	btcode
gcc	176.gcc	224,099	12,868,208	2,195,700
plk	253.perlbmk	85,814	7,010,809	1,268,148
gap	254.gap	71,461	4,256,317	702,843
msa	177.mesa	59,394	3,820,633	547,825
vtx	255.vortex	67,262	2,714,588	451,516
twf	300.twolf	20,499	1,625,861	324,346
crf	186.crafty	21,197	1,573,423	288,488
vpr	175.vpr	17,760	1,081,883	173,475
amp	188.ammp	13,515	875,786	149,245
prs	197.parser	11,421	904,924	163,025
gzp	164.gzip	8,643	202,640	46,188
bz2	256.bzip2	4,675	162,270	35,548
art	179.art	1,297	91,078	40,762
eqk	183.equake	1,540	91,018	45,241
mcf	181.mcf	2.451	60,225	34,021

Figure 12. Benchmark characteristics. `LoC`: number of lines of C code. `asm`: size of x86 assembly programs produced by LLVM with our algorithm (bytes). `btcode`: program size in LLVM's intermediate representation (bytes).

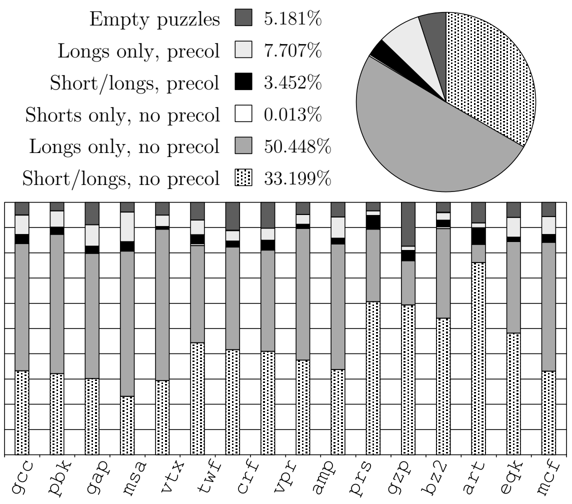


Figure 13. The distribution of the 1,486,301 puzzles generated from SPEC CPU2000.

block. Global copies denote instructions inserted into the final program during the SSA-elimination phase in order to implement φ-functions and π-functions. Target programs contains one copy or swap per each 14.7 puzzles in the source program, that is, on average, the puzzle solver has inserted 0.025 local and 0.043 global copies per puzzle.

Three other register allocators. We compare our puzzle solver with three other register allocators, all implemented in LLVM 1.9 and all compiling and running the same benchmark suite of 1.3 million lines of C code. The first is LLVM's default algorithm, which is an industrial-strength version of linear scan that uses extensions by Wimmer *et al.* [46] and Evlogimenos [16]. The algorithm does aggressive coalescing before register allocation and handles holes in live ranges by filling them with other variables whenever possible. We use ELS (Extended Linear Scan) to denote this register allocator.

The second register allocator is the iterated register coalescing of George and Appel [20] with extensions by Smith, Ramsey, and Holloway [42] for handling register aliasing. We use EIRC (Extended Iterated Register Coalescing) to denote this register allocator.

Benchmark	#puzzles	avg	max	once
gcc	476,649	1.03	4	457,572
perlbmk	265,905	1.03	4	253,563
gap	158,757	1.05	4	153,394
mesa	139,537	1.08	9	125,169
vortex	116,496	1.02	4	113,880
twolf	60,969	1.09	9	52,443
crafty	59,504	1.06	4	53,384
vpr	36,561	1.10	10	35,167
ammp	33,381	1.07	8	31,853
parser	31,668	1.04	4	30,209
gzip	7,550	1.06	3	6,360
bzip2	5,495	1.09	3	4,656
art	3,552	1.08	4	3,174
equake	3,365	1.11	8	2,788
mcf	2,404	1.05	3	2,120
	1,401,793	1.05	10	1,325,732

Figure 14. Number of calls to the puzzle solver per nonempty puzzle. #puzzles: number of nonempty puzzles. `avg` and `max`: average and maximum number of times the puzzle solver was used per puzzle. `once`: number of puzzles for which the puzzle solver was used only once.

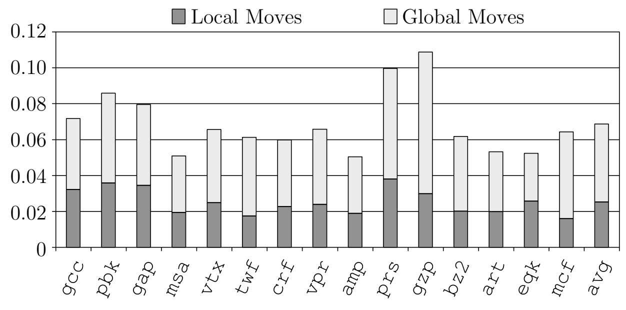


Figure 15. Number of copy and swap instructions inserted per puzzle.

The third register allocator is based on partitioned Boolean quadratic programming (PBQP) [41]. The algorithm runs in $O(|V|K^3)$, where $|V|$ is the number of variables in the source program, and K is the number of registers. PBQP can find an optimal register allocation with regard to a cost model in a great number of cases [26]. We use this algorithm to gauge the potential for how good a register allocator can be. Lang Hames and Bernhard Scholz produced the implementations of EIRC and PBQP that we are using.

Stack size comparison. The top half of Figure 16 compares the maximum amount of space that each assembly program reserves on its call stack. The stack size gives an estimate of how many different variables are being spilled by each allocator. The puzzle solver and extended linear scan (LLVM's default) tend to spill more variables than the other two algorithms.

Spill-code comparison. The bottom half of Figure 16 compares the number of load/store instructions in the assembly code. The puzzle solver is the algorithm that inserts the fewest amount of spill code into the target code. It inserts 9% fewer memory access instructions than PBQP, 11% fewer memory access instructions than EIRC, and 20% fewer memory access instructions than extended linear scan (LLVM's default). Note that although the puzzle solver spills more variables than the other allocators, it removes only part

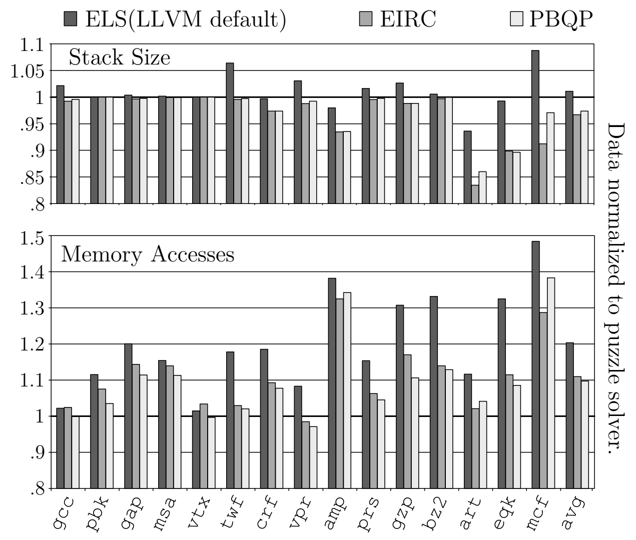


Figure 16. In both charts, the bars are relative to the puzzle solver; shorter bars are better for the other algorithms. **Stack size:** Comparison of the maximum amount of bytes reserved on the stack. **Number of memory accesses:** Comparison of the total static number of load and store instructions inserted by each register allocator.

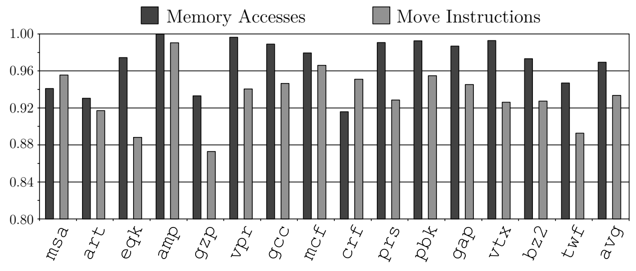


Figure 17. Static improvement due to First-Chance Coalescing. The bars are normalized to register allocation without global coalescing.

of the live range of a spilled variable. Also, the optimizations described in Section 5 play a significant role in reducing the amount of memory accesses introduced by the puzzle solver.

The effect of First-Chance Global Coalescing. Figure 17 outlines static improvements on the code produced by the puzzle solver due to the global coalescing heuristics described in Section 5.1. On average, first-chance coalescing reduces the number of memory accesses, mostly store instructions, by 3.4%. It reduces the number of move instructions by 7.1% on average. Notice that this form of coalescing does not reduce the number of spilled variables. The reduction in spill instructions happens during the elimination of π and φ-functions, because there are more variables in the same physical location across basic blocks.

The effects of Load Lowering and RMTE The optimizations described in Sections 5.2 and 5.3 reduce substantially the amount of spill code that the puzzle solver inserts into the target program, as shown in Figure 18. The load lowering optimization alone removes 7% of the load instructions in the final code produced by the puzzle solver. The elimination of redundant memory transfers accounts for

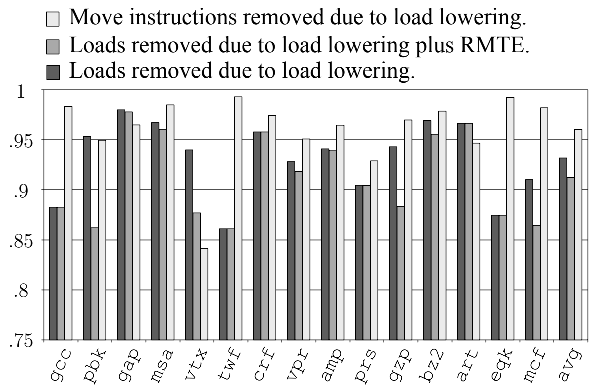


Figure 18. Code reduction due to Load Lowering and Redundant Memory Transfer Elimination (RMTE). The results are normalized to the puzzle solver with no optimization.

another 2% of instructions removed. An exceptional case is `vpr`. The `main` function of this benchmark contains many conditional statements that lead the puzzle solver to produce a large number of redundant memory transfers. In this benchmark, the elimination of redundant memory operations removes almost 10% of the total number of memory accesses. Load lowering also accounts, on the average, for the elimination of about 4% of the copy instructions present in the benchmarks.

Run-time comparison. Figure 19 compares the run time of the code produced by each allocator. Each bar shows the average of five runs of each benchmark; smaller is better. The base line is the run time of the code when compiled with gcc -O3 version 3.3.3. Note that the four allocators that we use (the puzzle solver, extended linear scan (LLVM's default), EIRC and PBQP) are implemented in LLVM, while we use gcc, an entirely different compiler, only for reference purposes. Considering all the benchmarks, the four allocators produce faster code than gcc; the fractions are: puzzle solver 0.944, extended linear scan (LLVM's default) 0.991, EIRC 0.954 and PBQP 0.929. If we remove the floating point benchmarks, *i.e.*, `msa, amp, art, eqk`, then gcc -O3 is faster. The fractions are: puzzle Solver 1.015, extended linear scan (LLVM's default) 1.059, EIRC 1.025 and PBQP 1.008. We conclude that the puzzle solver produces faster code than the other polynomial-time allocators, but slower code than the exponential-time allocator.

We have found that the puzzle solver does particularly well on sparse control-flow graphs. We can easily find examples of basic blocks where the puzzle solver outperforms even PBQP, which is a slower algorithm. For instance, with two register pairs (`AL, AH, BL, BH`) available, the puzzle solver allocates the program in Figure 20 without spilling, while the other register allocators (ELS, EIRC and PBQP) spill at least one variable. In this example, the puzzle solver inserts one copy between instructions four and five to split the live range of variable a.

Compile-time comparison. Figure 21 compares the register allocation time and the total compilation time of the puzzle solver and extended linear scan (LLVM's default). On average, extended linear scan (LLVM's default) is less than 1% faster than the puzzle solver. The total compilation time of LLVM with the default allocator is less than 3% faster than the total compilation time of LLVM with the puzzle solver. We note that LLVM is industrial-strength and highly tuned software, in contrast to our puzzle solver.

We omit the compilation times of EIRC and PBQP because the implementations that we have are research artifacts that have not been optimized to run fast. Instead, we gauge the relative compi-

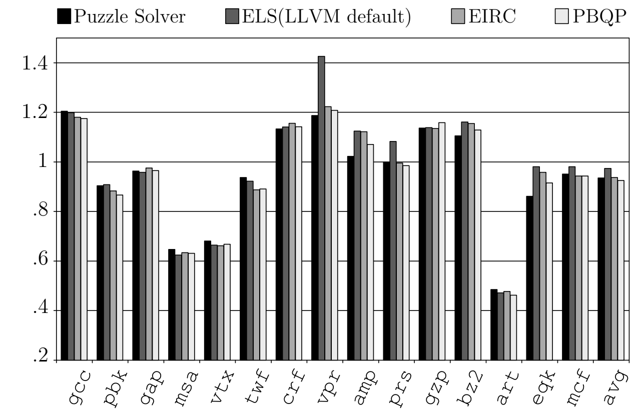


Figure 19. Comparison of the running time of the code produced with our algorithm and other allocators. The bars are relative to gcc -O3; shorter bars are better.

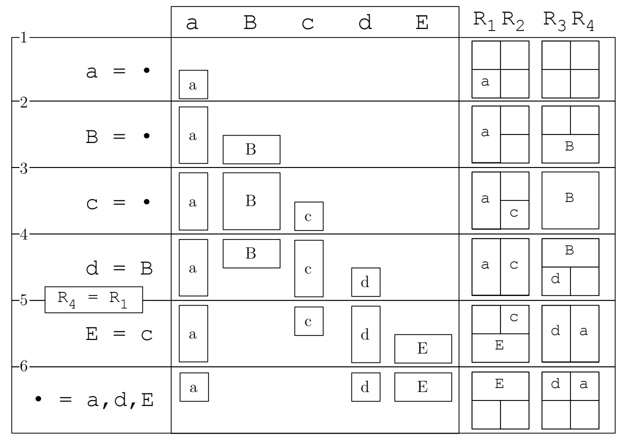


Figure 20. (left) Example program. (center) Puzzle pieces. (right) Register assignment.

lation speeds from statements in previous papers. The experiments shown in [26] suggest that the compilation time of PBQP is between two and four times the compilation time of extended iterated register coalescing. The extensions proposed by Smith *et al.* [42] can be implemented in a way that adds less than 5% to the compilation time of a graph-coloring allocator. Timing comparisons between graph coloring and linear scan (the core of LLVM's algorithm) span a wide spectrum. The original linear scan paper [39] suggests that graph coloring is about twice as slow as linear scan, while Traub *et al.* [44] gives an slowdown of up to 3.5x for large programs, and Sarkar and Barik [40] suggests a 20x slowdown. From these observations we conclude that extended linear scan (LLVM's default) and our puzzle solver are significantly faster than the other allocators.

7. Related Work

We now discuss work on relating programs to graphs and on complexity results for variations of graph coloring. Figure 24 summarizes most of the results.

Register allocation and graphs. The intersection graph of the live ranges of a program is called an *interference graph*. Figure 22

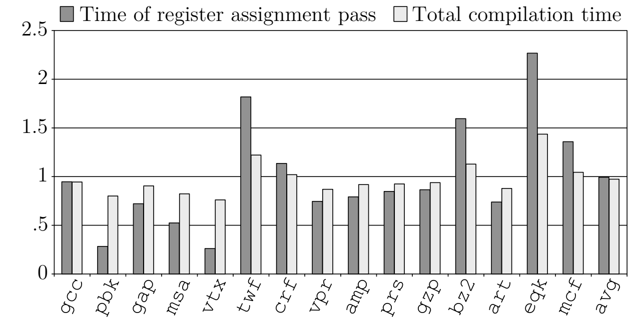


Figure 21. Comparison between compilation time of the puzzle solver and extended linear scan (LLVM's default algorithm). The bars are relative to the puzzle solver; shorter bars are better for extended linear scan.

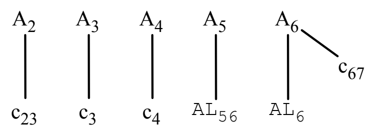


Figure 22. Interference graph of the program in Figure 4(b).

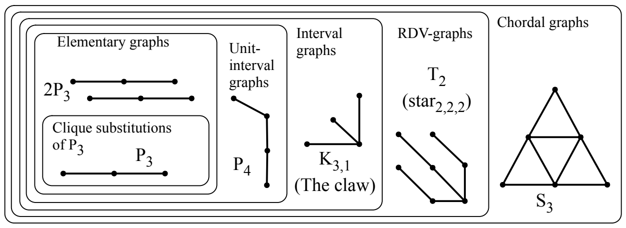


Figure 23. Elementary graphs and other intersection graphs. RDV-graphs are intersection graphs of directed lines on a tree [35].

shows the interference graph of the elementary program in Figure 4(b). Any graph can be the interference graph of a general program [13]. SSA-form programs have chordal interference graphs [6, 10, 25, 36], and the interference graphs of SSI-form programs are interval graphs [11]. We call the interference graph of an elementary program an *elementary graph* [38]. Each connected component of an elementary graph is a clique substitution of P_3, the simple path with three nodes. We construct a clique substitution of P_3 by replacing each node of P_3 by a clique, and connecting all the nodes of adjacent cliques.

Elementary graphs are a proper subset of interval graphs, which are contained in the class of chordal graphs. Figure 23 illustrates these inclusions. Elementary graphs are also *Trivially Perfect Graphs* [21], as we show in [38]. In a trivially perfect graph, the size of the maximal independent set equals the size of the number of maximal cliques.

Spill-free Register Allocation. Spill-free register allocation is NP-complete for general programs [13] because coloring general graphs is NP-complete. However, this problem has a polynomial time solution for SSA-form programs [6, 10, 25] because chordal

Program	Class of graphs general	 SSA-form	 SSI-form	 elementary
Problem	general	chordal	interval	elementary
ALIGNED 1-2-COLORING EXTENSION	NP-cpt [29]	NP-cpt [4]	NP-cpt [4]	linear [TP]
ALIGNED 1-2-COLORING	NP-cpt [29]	NP-cpt [32]	NP-cpt [32]	linear [TP]
COLORING EXTENSION	NP-cpt [29]	NP-cpt [4]	NP-cpt [4]	linear [TP]
COLORING	NP-cpt [29]	linear [18]	linear [18]	linear [18]

Figure 24. Algorithms and hardness results for graph coloring. NP-cpt = NP-complete; TP = this paper.

graphs can be colored in polynomial time [4]. This result assumes an architecture in which all the registers have the same size.

Aligned 1-2-Coloring. Register allocation for architectures with type-1 aliasing is modeled by the *aligned 1-2-coloring* problem. In this case, we are given a graph in which vertices are assigned a weight of either 1 or 2. Colors are represented by numbers, e.g.: $0, 1, \ldots, 2K - 1$, and we say that the two numbers $2i, 2i + 1$ are *aligned*. We define an *aligned 1-2-coloring* to be a coloring that assigns each weight-two vertex two aligned colors. The problem of finding an optimal 1-2-aligned coloring is NP-complete even for interval graphs [32].

Pre-coloring Extension. Register allocation with pre-coloring is equivalent to the *pre-coloring extension problem* for graphs. In this problem we are given a graph G, an integer K and a partial function φ that associates some vertices of G to colors. The challenge is to extend φ to a total function φ' such that (1) φ' is a proper coloring of G and (2) φ' uses less than K colors. Pre-coloring extension is NP-complete for interval graphs [4] and even for unit interval graphs [34].

Aligned 1-2-coloring Extension. The combination of 1-2-aligned coloring and pre-coloring extension is called *aligned 1-2-coloring extension*. We show in [38] that this problem, when restricted to elementary graphs, is equivalent to solving type-1 puzzles; thus, it has a polynomial time solution.

Register allocation and spilling. When spills happen, loads and stores are inserted into the source program to transfer values to and from memory. If we assume that each load and store has a cost, then the problem of minimizing the total cost added by spill instructions is NP-complete, even for basic blocks in SSA-form, as shown by Farach *et al.* [17]. If the cost of loads and stores is not taken into consideration, then a simplified version of the spilling problem is to determine the minimum number of variables that must be removed from the source program so that the program can be allocated with K registers. This problem is equivalent to determining if a graph G has a K-colorable induced subgraph, which is NP-complete for chordal graphs, but has polynomial time solution for interval graphs, as demonstrated by Yannakakis and Gavril [47].

8. Conclusion

In this paper we have introduced register allocation by puzzle solving. We have shown that our puzzle-based allocator runs as fast as the algorithm used in an industrial-strength JIT compiler and that it produces code that is competitive with state-of-the-art algorithms. A compiler writer can model a register file as a puzzle board, and straightforwardly transform a source program into elementary form and then into puzzle pieces. For a compiler that already uses SSA-form as an intermediate representation, the extra step to elementary form is small. Our puzzle solver works for architectures such as x86, ARM, and PowerPC. Puzzle solving for SPARC V8 and V9 (type-2 puzzles) remains an open problem. Our puzzle solver produces competitive code even though we use simple approaches to spilling and coalescing. We speculate that if compiler writers implement a puzzle solver with advanced approaches to spilling and coalescing, then the produced code will be even better.

Acknowledgments

Fernando Pereira was sponsored by the Brazilian Ministry of Education under grant number 218603-9. We thank Lang Hames and Bernhard Scholz for providing us with their implementations of EIRC and PBQP. We thank João Dias, Glenn Holloway, Ayee Kannan Goundan, Stephen Kou, Jonathan Lee, Todd Millstein, Norman Ramsey, Ben Titzer, and the PLDI reviewers for helpful comments on a draft of the paper.

References

[1] Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.

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

[3] L. Belady. A study of the replacement of algorithms of a virtual storage computer. *IBM System Journal*, 5:78–101, 1966.

[4] M Biró, M Hujter, and Zs Tuza. Precoloring extension. I interval graphs. In *Discrete Mathematics*, pages 267 – 279. ACM Press, 1992.

[5] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM Press, 2000.

[6] Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.

[7] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin *et al.* really prove? Or revisiting register allocation: Why and how. In *LCPC*, pages 283–298. Springer, 2006.

[8] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO*, pages 102–114, 2007.

[9] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *SPE*, 28(8):859–881, 1998.

[10] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *IWLS*. ACM Press, 2005.

[11] Philip Brisk and Majid Sarrafzadeh. Interference graphs for procedures in static single information form are interval graphs. In *SCOPES*, pages 101–110. ACM Press, 2007.

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

[13] 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.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[15] 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.

[16] Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.

[17] Martin Farach and Vincenzo Liberatore. On local register allocation. In *Symposium on Discrete Algorithms*, pages 564–573. ACM Press, 1998.

[18] Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47 – 56, 1974.

[19] Fanica Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discrete Mathematics*, 13:237 – 249, 1975.

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

[21] Martin Charles Golumbic. Trivially perfect graphs. *Discrete Mathematics*, 24:105 – 107, 1978.

[22] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 1st edition, 2004.

[23] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *CC*, volume 4420, pages 111–115. Springer, 2007.

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

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

[26] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC*, pages 346–361. Springer, 2006.

[27] Corporate SPARC International Inc. *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1st edition, 1992.

[28] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89. ACM Press, 1993.

[29] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

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

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

[32] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation for straight-line programs is NP-complete. In *ICALP*. Springer, 2007.

[33] Daniel Marx. Parameterized coloring problems on chordal graphs. *Theoretical Computer Science*, 351(3):407–424, 2006.

[34] Daniel Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995 – 1002, 2006.

[35] Clyde L. Monma and V. K. Wei. Intersection graphs of paths in a tree. *Journal of Combinatorial Theory, Series B*, 41(2):141 – 181, 1986.

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

[37] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is NP-complete. In *FOSSACS*. Springer, 2006.

[38] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving, 2007. http://compilers.cs.ucla.edu/fernando/projects/puzzles/.

[39] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.

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

[41] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. *SIGPLAN Notices*, 37(7):139–148, 2002.

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

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

[44] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI*, pages 142–151. ACM Press, 1998.

[45] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1st edition, 1994.

[46] Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM Press, 2005.

[47] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133 – 137, 1987.

A. Proof of Theorem 1

We will prove Theorem 1 for register banks that give type-1 puzzles. Theorem 1 states:

> **(Equivalence)** *Spill-free register allocation with pre-coloring for an elementary program is equivalent to solving a collection of puzzles.*

In Section A.1 we define three key concepts that we use in the proof, namely aligned 1-2-coloring extension, clique substitution of P_3, and elementary graph. In Section A.2 we state four key lemmas and show that they imply Theorem 1. Finally, in four separate subsections, we prove the four lemmas.

A.1 Definitions

We first state again a graph-coloring problem that we mentioned in Section 7, namely aligned 1-2-coloring extension.

> ALIGNED 1-2-COLORING EXTENSION
> **Instance**: a number of colors $2K$, a weighted graph G, and a partial aligned 1-2-coloring φ of G. **Problem**: Extend φ to an aligned 1-2-coloring of G.

We use the notation $(2K, G, \varphi)$ to denote an instance of the aligned 1-2-coloring extension problem. For a vertex v of G, if $v \in dom(\varphi)$, then we say that v is *pre-colored*.

Next we define the notion of a *clique substitution of* P_3. Let H_0 be a graph with n vertices $v_1, v_2, \ldots, v_n$ and let $H_1, H_2, \ldots, H_n$ be n disjoint graphs. The *composition graph* [22] $H = H_0[H_1, H_2, \ldots, H_n]$ is formed as follows: for all $1 \le i, j \le n$, replace vertex v_i in H_0 with the graph H_i and make each vertex of H_i adjacent to each vertex of H_j whenever v_i is adjacent to v_j in H_0. Figure 25 shows an example of composition graph.

P_3 is the path with three nodes, e.g., $(\{x, y, z\}, \{xy, yz\})$. We define a clique substitution of P_3 as $P_{X,Y,Z} = P_3[K_X, K_Y, K_Z]$, where each K_S is a complete graph with $|S|$ nodes.

DEFINITION 5. *A graph G is an elementary graph if and only if every connected component of G is a clique substitution of P_3.*

A.2 Structure of the Proof

We will prove the following four lemmas.

- Lemma 6: Spill-free register allocation with pre-coloring for an elementary program P is equivalent to the aligned 1-2-coloring extension problem for the interference graph of P.
- Lemma 13: An elementary program has an elementary interference graph.
- Lemma 15: An elementary graph is the interference graph of an elementary program.

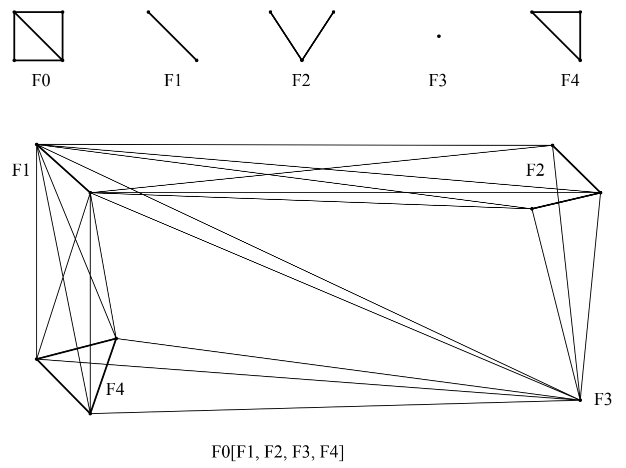


Figure 25. Example of a composition graph (taken from [22]).

- Lemma 17: Aligned 1-2-coloring extension for a clique substitution of P_3 is equivalent to puzzle solving.

We can now prove Theorem 1:

Proof. From Lemmas 6, 13, and 15 we have that spill-free register allocation with pre-coloring for an elementary program is equivalent to aligned 1-2-coloring extension for elementary graphs. From Lemma 17 we have that aligned 1-2-coloring extension for elementary graphs is equivalent to solving a collection of puzzles. $\square$

A.3 From register allocation to coloring

LEMMA 6. *Spill-free register allocation with pre-coloring for an elementary program P is equivalent to the aligned 1-2-coloring extension problem for the interference graph of P.*

Proof. Chaitin *et al.* [13] have shown that spill-free register allocation for a program P is equivalent to coloring the interference graph of P, where each color represents one physical register. To extend the spill-free register allocation to an architecture with a type-1 register bank, we assign weights to each variable in the interference graph, so that variables that fit in one register are assigned weight 1, and variables that fit in a register-pair are assigned weight 2. To include pre-coloring, we define $\varphi(v) = r$, if vertex v represents a pre-colored variable, and color r represents the register assigned to this variable. Otherwise, we let $\varphi(v)$ be undefined. $\square$

A.4 Elementary programs and graphs

We will show in three steps that an elementary program has an elementary interference graph. We first give a characterization of clique substitutions of P_3 (Lemma 8). Then we show that a graph G is an elementary graph if and only if G has an *elementary interval representation* (Lemma 10). Finally we show that the interference graph of an elementary program has an elementary interval representation and therefore is an elementary graph (Lemma 13).

A.4.1 A Characterization of Clique Substitutions of P_3

We will give a characterization of a clique substitution of P_3 in terms of forbidden induced subgraphs. Given a graph $G = (V, E)$, we say that $H = (V', E')$ is an induced subgraph of G if $V' \subseteq V$ and, given two vertices v and u in V', $uv \in E'$ if, and only if, $uv \in E$. Given a graph F, we say that G is F-free if none of its induced subgraphs is isomorphic to F. In this case we say that F is a forbidden subgraph of G. Some classes of graphs can be characterized in terms of forbidden subgraphs, that is, a set of graphs that cannot be induced in any of the graphs in that class. In this section we show that any graph $P_{X,Y,Z}$ has three forbidden subgraphs: (i) P_4, the simple path with four nodes; (ii) C_4, the cycle with four nodes, and (iii) $3K_1$, the graph formed by three unconnected nodes. These graphs are illustrated in Figure 26, along with the bipartite graph $K_{3,1}$, known as *the claw*. The claw is important because it is used to characterize many classes of graphs. For example, the interval graphs that do not contain any induced copy of the claw constitute the class of the unit interval graphs [22, p. 187]. A key step of our proof of Lemma 10 shows that elementary graphs are claw-free.

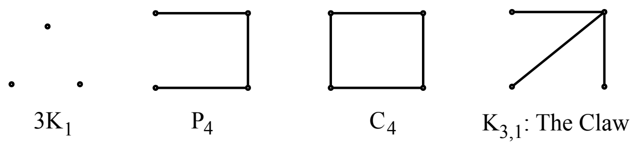


Figure 26. Some special graphs.

We start our characterization by describing the class of the *Trivially Perfect Graphs* [21]. In a trivially perfect graph, the size of the maximal independent set equals the size of the number of maximal cliques.

THEOREM 7. **(Golumbic [21])** *A graph G is trivially perfect if and only if G contains no induced subgraph isomorphic to C_4 or P_4.*

The next lemma characterizes $P_{X,Y,Z}$ in terms of forbidden subgraphs.

LEMMA 8. *A graph G is a clique substitution of P_3 if and only if G contains no induced subgraph isomorphic to C_4, P_4, or $3K_1$.*

Proof. ($\Rightarrow$) Let G be a clique substitution of P_3, and let G be of the form $P_{X,Y,Z}$. Let us first show that G is trivially perfect. Note that G contains either one or two maximal cliques. If G contains one maximal clique, we have that G is of the form $P_{\emptyset,Y,\emptyset}$, and the maximal independent set has size 1. If G contains two maximal cliques, those cliques must be $X \cup Y$ and $X \cup Z$. In this case, the maximal independent set has two vertices, namely an element of $X - Y$ and an element of $Z - Y$. So, G is trivially perfect, hence, by Theorem 7, G does not contain either C_4 nor P_4 as induced subgraphs. Moreover, the maximum independent set of G has size one or two; therefore, G cannot contain an induced $3K_1$.

($\Leftarrow$) If G is C_4-free and P_4-free, then G is trivially perfect, by Theorem 7. Because G is $3K_1$-free, its maximal independent set has either one or two nodes. If G is unconnected, we have that G consists of two unconnected cliques; thus, $G = P_{X,\emptyset,Y}$. If G is connected, it can have either one or two maximal cliques. In the first case, we have that $G = P_{\emptyset,Y,\emptyset}$. In the second, let these maximal cliques be C_1 and C_2. We have that $G = P_{C_1 - C_2, C_1 \cap C_2, C_2 - C_1}$. $\square$

A.4.2 A Characterization of Elementary Graphs

We recall the definitions of an *intersection* graph and an *interval* graph [22, p.9]. Let $\mathcal{S}$ be a family of nonempty sets. The intersection graph of $\mathcal{S}$ is obtained by representing each set in $\mathcal{S}$ by a vertex and connecting two vertices by an edge if and only if their corresponding sets intersect. An *interval* graph is an intersection graph of a family of subintervals of an interval of the real numbers.

A *rooted tree* is a directed tree with exactly one node of in-degree zero; this node is called *root*. Notice that there is a path from the root to any other vertex of a rooted tree. The intersection graph of a family of directed vertex paths in a rooted tree is called *a rooted directed vertex path graph*, or *RDV* [35]. A polynomial time algorithm for recognizing RDV graphs was described in [19]. The family of RDV graphs includes the interval graphs, and is included in the class of chordal graphs. An example of RDV graph is given in Figure 27.

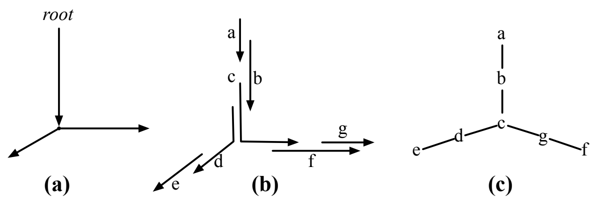


Figure 27. (a) Directed tree T. (b) Paths on T. (c) Corresponding RDV graph.

Following the notation in [19], we let $L = \{\overline{v_1}, \ldots, \overline{v_n}\}$ denote a set of n directed paths in a rooted tree T. The RDV graph that corresponds to L is $G = (\{v_1, \ldots, v_n\}, E)$, where $v_iv_j \in E$ if and only if $\overline{v_i} \cap \overline{v_j} \neq \emptyset$. We call L the *path representation* of G. Because T is a rooted tree, each interval $\overline{v}$ has a well-defined start point $begin(\overline{v})$, and a well-defined end point $end(\overline{v})$: $begin(\overline{v})$ is the point of $\overline{v}$ closest to the root of T, and $end(\overline{v})$ is the point of $\overline{v}$ farthest from the root.

Given a connected graph $G = (V, E)$, the *distance* between two vertices $\{u, v\} \subseteq V$ is the number of edges in the shortest path connecting u to v. The *diameter* of G is the maximal distance between any two pairs of vertices of G. A key step in the proof of Lemma 10 below (Claim 3) shows that the diameter of any connected component of an elementary graph is at most 2.

We define *elementary interval representation* as follows:

DEFINITION 9. *A graph G has an* elementary interval representation *if:*

1. G is a RDV graph.
2. If $uv \in E$, then $begin(\overline{u}) = begin(\overline{v})$, or $end(\overline{u}) = end(\overline{v})$.
3. If $uv \in E$, then $\overline{u} \subseteq \overline{v}$ or $\overline{v} \subseteq \overline{u}$.

Lemma 10 shows that any elementary graph has an elementary interval representation.

LEMMA 10. *A graph G is an elementary graph if, and only if, G has an elementary interval representation.*

Proof. ($\Leftarrow$) We first prove six properties of G:

- Claim 1: If $a, b, c \in V$, $ab \in E$, $bc \in E$ and $ac \notin E$, then we have $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$ in any path representation of G.
- Claim 2: G is P_4-free.
- Claim 3: Let $C = (V_C, E_C)$ be a connected component of G. Given $a, b \in V_C$ such that $ab \notin E_C$, then $\exists v$ such that $av \in E_C$ and $bv \in E_C$.
- Claim 4: G is claw-free.
- Claim 5: Every connected component of G is $3K_1$-free.
- Claim 6: G is C_4-free.

Proof of Claim 1. Let us first show that $\overline{b} \not\subseteq \overline{a}$. If $\overline{b} \subseteq \overline{a}$, then, from $ac \notin E$ we would have $bc \notin E$, which is a contradiction. Given that $ab \in E$ and $\overline{b} \not\subseteq \overline{a}$ we have that $\overline{a} \subseteq \overline{b}$. By symmetry, we have that $\overline{c} \subseteq \overline{b}$. We conclude that $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$.

Proof of Claim 2. Assume G contains four vertices x, y, z and w that induce the path $\{xy, yz, zw\}$ in G. From Claim 1 we have $(\overline{x} \cup \overline{z}) \subseteq \overline{y}$; in particular, $\overline{z} \subseteq \overline{y}$. Similarly we have $(\overline{y} \cup \overline{w}) \subseteq \overline{z}$; in particular, $\overline{y} \subseteq \overline{z}$. So, $\overline{y} = \overline{z}$. From $zw \in E$ and $\overline{y} = \overline{z}$, we have $yw \in E$, contradicting that the set $\{x, y, z, w\}$ induces a path in G.

Proof of Claim 3. From Claim 2 we have that G is P_4-free, so any minimal-length path between two connected vertices contains either one or two edges. We have $a, b \in V_C$ so a, b are connected, and we have $ab \notin E_C$, so we must have a minimal-length path $\{av, vb\}$ for some vertex v.

Proof of Claim 4. Let L be G's directed path representation. Suppose G contains four vertices x, y, z, w that induce the claw $\{xy, xz, xw\}$. Without loss of generality, we assume $begin(\overline{x}) = begin(\overline{y})$. Because G is an RDV-graph, we must have $end(\overline{x}) = end(\overline{z})$. However, x and w interfere, yet, w cannot share the starting point with x, or it would interfere with y, nor can w share its end point with x, or it would interfere with z. So, the claw is impossible.

Proof of Claim 5. Let $C = (V_C, E_C)$ be a connected component of G. Assume, for the sake of contradiction, that there are three vertices $\{a, b, c\} \in V_C$ such that $ab \notin E_C$, $ac \notin E_C$ and $bc \notin E_C$. From Claim 3 we have that there exists a vertex v_{ab} that is adjacent to a and b. Likewise, we know that there exists a vertex v_{bc} that is adjacent to b and c. From Claim 1 we have that in any path representation of G, $(\overline{a} \cup \overline{b}) \subseteq \overline{v_{ab}}$. We also know that $(\overline{b} \cup \overline{c}) \subseteq \overline{v_{bc}}$. Therefore, $\overline{b} \subseteq (\overline{v_{ab}} \cap \overline{v_{bc}})$, so $v_{ab}v_{bc} \in E_C$, hence either $\overline{v_{ab}} \subseteq \overline{v_{bc}}$ or $\overline{v_{bc}} \subseteq \overline{v_{ab}}$. If the first case holds, $\{a, b, c, v_{bc}\}$ induces a claw in G, which is impossible, given Claim 4. In the second case, $\{a, b, c, v_{ab}\}$ induces a claw.

Proof of Claim 6. By definition, RDV graphs are chordal graphs, which are C_4 free.

Finally, we prove that every connected component of G is a clique substitution of P_3. By Lemma 8, a minimal characterization of clique substitutions of P_3 in terms of forbidden subgraphs consists of C_4, P_4, and $3K_1$. G is C_4-free, from Claim 6, and G is P_4-free, from Claim 2. Any connected component of G is $3K_1$-free, from Claim 5.

($\Rightarrow$) Let G be a graph with K connected components, each of which is a clique substitution of P_3. Let $P_{X,Y,Z}$ be one of G's connected components. We first prove that $P_{X,Y,Z}$ has an elementary interval representation. Let T be a rooted tree isomorphic to $P_4 = (\{a, b, c, d\}, \{ab, bc, cd\})$, and let a be its root. We build an elementary graph G_P, isomorphic to $P_{X,Y,Z}$ using intervals on T. We let $\overrightarrow{v_1v_2 \ldots v_n}$ denote the directed path that starts at node v_1 and ends at node v_n. We build an elementary interval representation of $P_{X,Y,Z}$ as follows: for any $x \in X$, we let $\overline{x} = \overrightarrow{ab}$. For any $y \in Y$, we let $\overline{y} = \overrightarrow{abcd}$. And for any $z \in Z$, we let $\overline{z} = \overrightarrow{cd}$. It is straightforward to show that the interval representation meets the requirements of Definition 9.

Let us then show that G has an elementary interval representation. For each connected component $C_i, 1 \leq i \leq K$ of G, let T_i be the rooted tree that underlies its directed path representation, and let $root_i$ be its root. Build a rooted tree T as $root \cup T_i, 1 \leq i \leq K$, where $root$ is a new node not in any T_i, and let $root$ be adjacent to each $root_i \in T_i$. The directed paths on each branch of T meet the requirements in Lemma 10 and thus constitute an elementary interval representation. $\square$

Lemma 10 has a straightforward corollary that justifies one of the inclusions in Figure 23.

COROLLARY 11. *An elementary graph is a unit interval graph.*

Proof. Let us first show that a clique substitution of P_3 is a unit interval graph. Let i be an integer. Given $P_{X,Y,Z}$, we define a unit

$$
\begin{array}{rcl}
P & ::= & S\ (L\ \ \varphi(m,n)\ i^*\ \ \pi(p,q))^*\ E \\
L & ::= & L_{start}, L_1, L_2, \ldots, L_{end} \\
v & ::= & v_1, v_2, \ldots \\
r & ::= & \mathtt{AX}, \mathtt{AH}, \mathtt{AL}, \mathtt{BX}, \ldots \\
o & ::= & \bullet \\
 & | & v \\
 & | & r \\
S & ::= & L_{start} :\ \pi(p,q) \\
E & ::= & L_{end} :\ halt \\
i & ::= & o = o \\
 & | & V(n) = V(n) \\
\pi(p,q) & ::= & M(p,q) = \pi V(q) \\
\varphi(n,m) & ::= & V(n) = \varphi M(m,n) \\
V(n) & ::= & (o_1, \ldots, o_n) \\
M(m,n) & ::= & V_1(n) : L_1, .., V_m(n) : L_m
\end{array}
$$

Figure 28. The grammar of elementary programs.

interval graph I in the following way. For any $x \in X - Y$, let $\overline{x} = [i, i+3]$; for any $y \in (Y - (X \cup Z))$, let $\overline{y} = [i+2, i+5]$; and for any $z \in Z - Y$, let $\overline{z} = [i+4, i+7]$. Those intervals represent $P_{X,Y,Z}$ and constitute a unit interval graph.

By the definition of elementary graphs we have that every connected component of G is a clique substitution of P_3. From each connected component G we can build a unit interval graph and then assemble them all into one unit interval graph that represents G. $\square$

A.4.3 An elementary program has an elementary interference graph

Elementary programs were first introduced in Section 2.2. In that section we described how elementary programs could be obtained from ordinary programs via live range splitting and renaming of variables; we now give a formal definition of elementary programs.

Program points and live ranges have been defined in Section 2.2. We denote the live range of a variable v by $LR(v)$, and we let $def(v)$ be the instruction that defines v. A program P is strict [12] if every path in the control-flow graph of P from the start node to a use of a variable v passes through one of the definitions of v. A program P is *simple* if P is a strict program in SSA-form and for any variable v of P, $LR(v)$ contains at most one program point outside the basic block that contains $def(v)$. For a variable v defined in a basic block B in a simple program, we define $kill(v)$ to be either the unique instruction outside B that uses B, or, if v is used only in B, the last instruction in B that uses v. Notice that because P is simple, $LR(v)$ consists of the program points on the unique path from $def(v)$ to $kill(v)$. Elementary programs are defined as follows:

DEFINITION 12. *A program produced by the grammar in Figure 28 is in elementary form if, and only if, it has the following properties:*

1. *P_e is a simple program;*
2. *if two variables u, v of P_e interfere, then either $def(u) = def(v)$, or $kill(u) = kill(v)$; and*
3. *if two variables u, v of P_e interfere, then either $LR(u) \subseteq LR(v)$, or $LR(v) \subseteq LR(u)$.*

We can produce an elementary program from a strict program:

- insert φ-functions at the beginning of basic blocks with multiple predecessors;
- insert π-functions at the end of basic blocks with multiple successors;
- insert parallel copies between consecutive instruction in the same basic block; and
- rename variables at every opportunity given by the φ-functions, π-functions, and parallel copies.

An elementary program P generated by the grammar 28 is a sequence of basic blocks. A basic block, which is named by a label L, is a sequence of instructions, starting with a φ-function and ending with a π-function. We assume that a program P has two special basic blocks: L_{start} and L_{end}, which are, respectively, the first and last basic blocks to be visited during P's execution. Ordinary instructions either define, or use, one operand, as in $r_1 = v_1$. An instruction such as $v_1 = \bullet$ defines one variable but does not use a variable or register. Parallel copies are represented as $(v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$.

In order to split the live range of variables, elementary programs use φ-functions and π-functions. φ-functions are an abstraction used in SSA-form to join the live ranges of variables. An assignment such as:

$$(v_1, \ldots, v_n) = \varphi[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m]$$

contains n φ-functions such as $v_i \leftarrow \varphi(v_{i1} : L_1, \ldots, v_{im} : L_m)$. The φ symbol works as a multiplexer. It will assign to each v_i the value in v_{ij}, where j is determined by L_j, the basic block last visited before reaching the φ assignment. Notice that these assignments happen in parallel, that is, all the variables $v_{1i}, \ldots, v_{ni}$ are simultaneously copied into the variables $v_1, \ldots, v_n$.

The π-functions were introduced in [28] with the name of *swicth nodes*. The name π-node was established in [5]. The π-nodes, or π-functions, as we will call them, are the dual of φ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs a parallel assignment depending on the execution path taken. Consider, for instance, the assignment below:

$$[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m] = \pi(v_1, \ldots, v_n)$$

which represents m π-nodes such as $(v_{i1} : L_1, \ldots, v_{im} : L_m) \leftarrow \pi(v_i)$. This instruction has the effect of assigning to each variable $v_{ij} : L_j$ the value in v_i if control flows into block L_j. Notice that variables alive in different branches of a basic block are given different names by the π-function that ends that basic block.

LEMMA 13. *An elementary program has an elementary interference graph.*

Proof. Let P be an elementary program, let $G = (V, E)$ be P's interference graph, and let T_P be P's dominator tree. We first prove that for any variable v, $LR(v)$ determines a directed path in T_P. Recall that $LR(v)$ consists of the vertices on the unique path from $def(v)$ to $kill(v)$. Those vertices are all in the same basic block, possibly except $kill(v)$. So every vertex on that path dominates the later vertices on the path, hence $LR(v)$ determines a directed path in T_P. So, G is an RDV-graph. Given a variable v, we let $begin(LR(v)) = def(v)$, and we let $end(LR(v)) = kill(v)$. The second and third requirements in Lemma 10 follow immediately from the second and third requirements in Definition 12. $\square$

A.5 An elementary graph is the interference graph of an elementary program

In this section we show in two steps that any elementary graph is the interference graph of some elementary program.

LEMMA 14. *A clique substitution of P_3 is the interference graph of an instruction sequence.*

Proof. Let $G = P_{X,Y,Z}$ be a clique substitution of P_3. Let $m = |X|$, $n = |Y|$ and $p = |Z|$. We build a sequence of $2(m+n+p)$

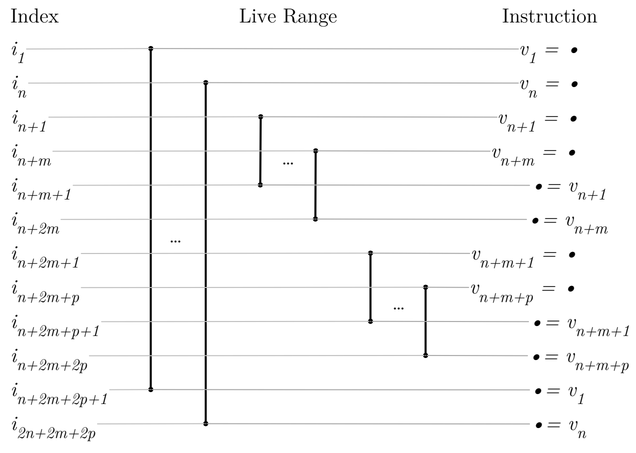


Figure 29. An elementary program representing a clique substitution of P_3.

instructions $i_1, \dots i_{2(m+n+p)}$ that use $m + n + p$ variables, such that each instruction either defines or uses one variable:

$$\begin{array}{lrcll}
i_j & v_j & = & \bullet & \text{for } j \in 1..n \\
i_{n+j} & v_{n+j} & = & \bullet & \text{for } j \in 1..m \\
i_{n+m+j} & \bullet & = & v_{n+j} & \text{for } j \in 1..m \\
i_{n+2m+j} & v_{n+m+j} & = & \bullet & \text{for } j \in 1..p \\
i_{n+2m+p+j} & \bullet & = & v_{n+m+j} & \text{for } j \in 1..p \\
i_{n+2m+2p+j} & \bullet & = & v_j & \text{for } j \in 1..n
\end{array}$$

Figure 29 illustrates the instructions. It is straightforward to show that $P_{X,Y,Z}$ is the interference graph of the instruction sequence. □

LEMMA 15. *An elementary graph is the interference graph of an elementary program.*

Proof. Let G be an elementary graph and let $C_1, \dots, C_n$ be the connected components of G. Each C_i is a clique substitution of P_3 so from Lemma 14 we have that each C_i is the interference graph of an instruction sequence s_i. We build an elementary program P with $n + 2$ basic blocks: $B_{start}, B_1, \dots, B_n, B_{end}$, such that B_{start} contains a single jump to B_1, each B_i consists of s_i followed by a single jump to B_{i+1}, for $1 \leq i \leq n - 1$, and B_n consists of s_n followed by a single jump to B_{end}. The interference graph of the constructed program is G. □

A.6 From Aligned 1-2-coloring to Puzzle Solving

We now show that aligned 1-2-coloring extension for clique substitutions of P_3 and puzzle solving are equivalent under linear-time reductions. Our proof is in two steps: first we show how to simplify the aligned 1-2-coloring extension problem by *padding* a graph, and then we show how to map a graph to a puzzle.

Padding of puzzles has been defined in Section 3. A similar concept applies to clique substitutions of P_3. We say that a graph $P_{X,Y,Z}$ is $2K$*-balanced*, if (1) the weight of X equals the weight of Z, and (2) the weight $X \cup Y$ is $2K$. We pad $P_{X,Y,Z}$ by letting X', Z' be sets of fresh vertices of weight one such that the padded graph $P_{(X \cup X'),Y,(Z \cup Z')}$ is $2K$-balanced. It is straightforward to see that padding executes in linear time. Figure 30 shows an example of padding. The original graph has two maximal cliques: $K_X \cup K_Y$ with weight 5 and $K_Y \cup K_Z$ with weight 4. We use

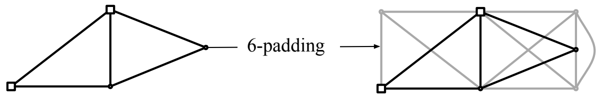


Figure 30. Example of padding. Square nodes represent vertices of weight two, and the other nodes represent vertices of weight one.

square nodes to denote vertices of weight two. After the padding, each maximal clique of the resulting graph has weight 6.

LEMMA 16. *For any partial aligned 1-2-coloring φ whose domain is a subset of $X \cup Y \cup Z$, we have that $(2K, P_{X,Y,Z}, \varphi)$ is solvable if and only if $(2K, P_{(X \cup X'),Y,(Z \cup Z')}, \varphi)$ is solvable.*

Proof.
($\Rightarrow$) Let $G = P_{X,Y,Z}$, and let $G_p = P_{(X \cup X'),Y,(Z \cup Z')}$. Let c be a function that associates vertices of G with colors, and let c_p be a function that associates vertices of G_p with colors. Let v be a vertex of G, and let v_p be the vertex of G_p that corresponds to v. Finally, let s be a short vertex of G_p created due to padding. Given c, we let $c_p(v_p) = c(v)$. After coloring vertices v_p, we greedily color the vertices s. Let M_1 and M_2 be the maximal cliques of G. Because M_1 is a comparability graph, the coloring of M_1 uses a number of colors equal to its weight w_1 [22, pp.133]. The same holds true for M_2. Let Q_1 be the maximal clique of G_p that corresponds to M_1, let s_1 be the short vertices of Q_1 created due to padding and let v_1 be the vertices of Q_1 which have corresponding vertices in M_1. We define w_2, Q_2, v_2 and s_2 in analogous way. The number of s_1 vertices in Q_1 equals $2K - w_1$, by the definition of padding, and the coloring of vertices v_1 requires exactly w_1 colors. We color the s_1 vertices with the remaining $2K - w_1$ colors. The number of s_2 vertices in Q_2 equals $2K - w_2$, and the coloring of vertices v_2 requires exactly w_2 colors. We color the s_2 vertices with the remaining $2K - w_2$ colors. The colors greedily assigned to vertices s_1 and s_2 constitute a valid coloring, because s_1 vertices are not adjacent to s_2 vertices.

($\Leftarrow$) Given c_p, we build c as $c(v) = c_p(v_p)$. □ □

We now define a bijection $\mathcal{F}$ from the aligned 1-2-coloring extension problem for $2K$-balanced clique substitutions of P_3 to puzzle solving. We will view a board with K areas as a 2-dimensional $2 \times 2K$ table, in which the i'th area consists of the squares with indices $(1, 2i)$, $(1, 2i + 1)$, $(2, 2i)$ and $(2, 2i + 1)$.

Let $(2K, G, \varphi)$ be an instance of the aligned 1-2-coloring extension problem, where G is a $2K$-balanced clique substitution of P_3. We define a puzzle $\mathcal{F}(2K, G, \varphi)$ with K areas and the following pieces:

- $\forall v \in X$, weight of v is one: a size-1 X-piece. If $\varphi(v)$ is defined and $\varphi(v) = i$, then the piece is placed on the square $(1, i)$, otherwise the piece is off the board.
- $\forall v \in X$, weight of v is two: a size-2 X-piece. If $\varphi(v)$ is defined and $\varphi(v) = \{2i, 2i + 1\}$, then the piece is placed on the upper row of area i, otherwise the piece is off the board.
- $\forall v \in Y$, weight of v is one: a size-2 Y-piece. If $\varphi(v)$ is defined and $\varphi(v) = i$, then the piece is placed on the squares $(1, i)$ and $(2, i)$, otherwise the piece is off the board.
- $\forall v \in Y$, weight of v is two: a size-4 Y-piece. If $\varphi(v)$ is defined and $\varphi(v) = \{2i, 2i + 1\}$, then the piece is placed on area i. otherwise the piece is off the board.

- $\forall v \in Z$, weight of v is one: a size-1 Z-piece. If $\varphi(v)$ is defined and $\varphi(v) = i$, then the piece is placed on the square $(2, i)$, otherwise the piece is off the board.
- $\forall v \in Z$, weight of v is two: a size-2 Z-piece. If $\varphi(v)$ is defined and $\varphi(v) = \{2i, 2i + 1\}$, then the piece is placed on the lower row of area i, otherwise the piece is off the board.

Given that φ is a partial aligned 1-2-coloring of G, we have that the pieces on the board don't overlap. Given that G is $2K$-balanced, we have that the pieces have a total size of $4K$ and that the total size of the X-pieces is equal to the total size of the Z-pieces.

It is straightforward to see that $\mathcal{F}$ is injective and surjective, so $\mathcal{F}$ is a bijection. It is also straightforward to see that $\mathcal{F}$ and $\mathcal{F}^{-1}$ both execute in $O(K)$ time.

LEMMA 17. *Aligned 1-2-coloring extension for a clique substitution of P_3 is equivalent to puzzle solving.*

Proof. First we reduce aligned 1-2-coloring extension to puzzle solving. Let $(2K, G, \varphi)$ be an instance of the aligned 1-2-coloring extension problem where G is a clique substitution of P_3. Via the linear-time operation of padding, we can assume that G is $2K$-balanced. Use the linear-time reduction $\mathcal{F}$ to construct a puzzle $\mathcal{F}(2K, G, \varphi)$. Suppose $(2K, G, \varphi)$ has a solution. The solution extends φ to an aligned 1-2-coloring of G, and we can then use $\mathcal{F}$ to place all the pieces on the board. Conversely, suppose $\mathcal{F}(2K, G, \varphi)$ has a solution. The solution places the remaining pieces on the board, and we can then use $\mathcal{F}^{-1}$ to define an aligned 1-2-coloring of G which extends φ.

Second we reduce puzzle solving to aligned 1-2-coloring. Let $\mathcal{P}$ be a puzzle and use the linear-time reduction $\mathcal{F}^{-1}$ to construct an instance of the aligned 1-2-coloring extension problem $\mathcal{F}^{-1}(\mathcal{P}) = (2K, G, \varphi)$, where G is a clique substitution of P_3. Suppose $\mathcal{P}$ has a solution. The solution places all pieces on the board, and we can then use $\mathcal{F}^{-1}$ to define an aligned 1-2-coloring of G which extends φ. Conversely suppose $\mathcal{F}^{-1}(\mathcal{P})$ has a solution. The solution extends φ to an aligned 1-2-coloring of G, and we can then use $\mathcal{F}$ to place all the pieces on the board. □

B. Proof of Theorem 2

Theorem 2 states:

> **(Correctness)** *A type-1 puzzle is solvable if and only if our program succeeds on the puzzle.*

We first show that an application of a rule from the algorithm given in Figure 8 preserves solvability of a puzzles.

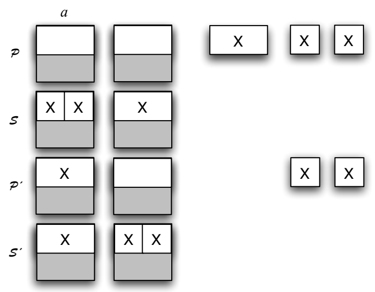
LEMMA 18. **(Preservation)** *Let $\mathcal{P}$ be a puzzle and let $i \in \{1, \ldots, 15\}$ be the number of a statement in our program. For $i \in \{11, 12, 13, 14\}$, suppose every area of $\mathcal{P}$ is either complete, empty, or has just one square already filled in. For $i = 15$, suppose every area of $\mathcal{P}$ is either complete or empty. Let a be an area of $\mathcal{P}$ such that the pattern of statement i matches a. If $\mathcal{P}$ is solvable, then the application of statement i to a succeeds and results in a solvable puzzle.*

Proof. We begin by outlining the proof technique that we will use for each $i \in \{1, \ldots, 15\}$. Notice that statement i contains a rule for each possible strategy that can be used to complete a. Let $\mathcal{S}$ be a solution of $\mathcal{P}$. Given that $\mathcal{S}$ completes a, it is straightforward to see that the application of statement i to a succeeds, although possibly using a different strategy than $\mathcal{S}$. Let $\mathcal{P}'$ be the result of the application of statement i to a. To see that $\mathcal{P}'$ is a solvable puzzle, we do a case analysis on (1) the strategy used by $\mathcal{S}$ to complete a and (2) the strategy used by statement i to complete a. For each case of (1), we analyze the possible cases of (2), and we show that one can rearrange $\mathcal{S}$ into $\mathcal{S}'$ such that $\mathcal{S}'$ is a solution of $\mathcal{P}'$. Let us now do the case analysis itself. If statement i is a conditional statement, then we will use $i.n$ to denote the n^{th} rule used in statement i.

$i = 1$. The area a can be completed in just one way. So, $\mathcal{S}$ uses the same strategy as statement 1 to complete a, hence $\mathcal{S}$ is a solution of $\mathcal{P}'$.

$i \in \{2, 3, 4, 5\}$. The proof is similar to the proof for $i = 1$, we omit the details.

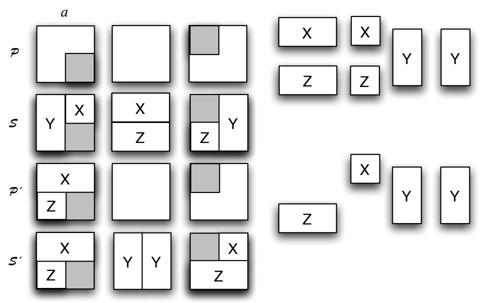
$i = 7$. The area a can be completed in two ways. If $\mathcal{S}$ uses the strategy of rule 7.1 to complete a, then statement 7 uses that strategy, too, hence $\mathcal{S}$ is a solution of the resulting puzzle. If $\mathcal{S}$ uses the strategy of rule 7.2 to complete a, we have two cases. Either statement 7 uses the strategy of rule 7.2, too, in which case $\mathcal{S}$ is a solution of $\mathcal{P}'$. Otherwise, statement 7 uses the strategy of rule 7.1, in which case we can create $\mathcal{S}'$ from $\mathcal{S}$ in the following way. We swap the two size-2 X-pieces used by $\mathcal{S}$ to complete a, with the size-2 X-piece used by statement 7 to complete a. To illustrate the swap, here are excerpts of $\mathcal{P}$, $\mathcal{S}$, $\mathcal{P}'$, $\mathcal{S}'$ for a representative $\mathcal{P}$.



It is straightforward to see that $\mathcal{S}'$ is a solution of $\mathcal{P}'$.

$i \in \{8, 9, 10\}$. The proof is similar to the proof for $i = 7$, we omit the details.

$i = 11$. The area a can be completed in three ways. If $\mathcal{S}$ uses the strategy of rule 11.1 or of rule 11.3 to complete a, the proof proceeds in a manner similar to the proof for $i = 7$, we omit the details. If $\mathcal{S}$ uses the strategy of rule 11.2 to complete a, we have two cases. Either statement 11 uses the strategy of rule 11.2, too, in which case $\mathcal{S}$ is a solution of $\mathcal{P}'$. Otherwise, statement 11 uses the strategy of rule 11.1, and now we have several of subcases of $\mathcal{S}$. Because of the assumption that all areas of $\mathcal{P}$ are either complete, empty, or has just one square already filled in, the following subcase is the most difficult; the other subcases are easier and omitted. Here are excerpts of $\mathcal{P}$, $\mathcal{S}$, $\mathcal{P}'$, $\mathcal{S}'$.



It is straightforward to see that $\mathcal{S}'$ is a solution of $\mathcal{P}'$.

$i \in \{12, 13, 14\}$. The proof is similar to the proof for $i = 11$, we omit the details.

$i = 15$. The proof is similar to the proof for $i = 11$, with a total of 28 subcases. All the subcases turn out to be easy because of the assumption that all areas of $\mathcal{P}$ are either complete or empty. We omit the details. □

We can now prove Theorem 2 (**Correctness**).

Proof. Suppose first that $\mathcal{P}$ is a solvable puzzle. We must show that our program succeeds on $\mathcal{P}$, that is, all the 15 statements succeed. From Lemma 18 and induction on the statement number we have that indeed all 15 statements succeed.

Conversely, suppose $\mathcal{P}$ is a puzzle and that our program succeeds on $\mathcal{P}$. Statements 1–4 complete all areas with three squares already filled in. Statements 5–10 complete all areas with two squares already filled in. Statements 11–14 complete all areas with one square already filled in. Statement 15 completes all areas with no squares already filled in. So, when our program succeeds on $\mathcal{P}$, the result is a solution to the puzzle. □

As a collary we get the following complexity result.

LEMMA 19. *The aligned 1-2-coloring extension problem for an elementary graph G is solvable in $O(C \times K)$, where C is the number of connected components of G, and $2K$ is the number of colors.*

Proof. Let $(2K, G, \varphi)$ be an instance of the aligned 1-2-coloring extension problem for which G is an elementary graph. We first list the connected components of G in linear time [14]. All the connected components of G are clique substitutions of P_3. Next, for each connected component, we have from Lemma 17 that we can reduce the aligned 1-2-coloring extension problem to a puzzle solving problem in linear time. Finally, we run our linear-time puzzle solving program on each of those puzzles (Theorem 2). The aligned 1-2-coloring extension problem is solvable if and only if all those puzzles are solvable. The total running time is $O(C \times K)$. □

C. Proof of Theorem 4

Theorem 4 **(Hardness)** states:

> *Register allocation with pre-coloring and spilling of families of variables for an elementary program is NP-complete.*

We reduce this problem to the maximal K-colorable subgraph of a chordal graph, which was proved to be NP-complete by Yannakakis and Gavril [47]. The key step is to show that any chordal graph is the interference graph of a program in SSA form. We first define a convenient representation of chordal graphs. Suppose we have a tree T and a family V of subtrees of T. We say that (T, V) is a *program-like decomposition* if for for all $\sigma \in V$ we have that (1) the root of σ has one successor. (2) each leaf of σ has zero or one successor, (3) each vertex of T is the root of at most one element of V, (4) a vertex of T is the leaf of at most one element of V, in which case it is not the root of any subtree, and (5) each element of V contains at least one edge. For each subtree $\sigma \in V$, we identify root_σ as the vertex of σ that is closest to the root of T.

In order to prove that any chordal graph has a program like decomposition, we rely on the concept of *nice tree decomposition* [33]. Given a nice tree T, for each vertex $x \in T$ we denote by K_x the union of all the subtrees that touch x. T satisfies the following properties: (1) Every node x has at most two children. (2) If $x \in T$ has two children, $y, z \in T$, then $K_x = K_y = K_z$. In this case, x is called a *joint* vertex. (3) If $x \in T$ has only one child, $y \in T$, then $K_x = K_y \cup \{u\}$, or $K_x = K_y \setminus \{u\}$. (4) If $x \in T$ has no children, then K_x is reached by at most one subtree, and x is called a *leaf* node. Figure 31 (b) shows a nice tree decomposition produced for the graph in Figure 31 (a). The program like decomposition is given in Figure 31 (c).

LEMMA 20. *A graph is chordal if and only if it has a program like tree decomposition.*

Proof. $\Leftarrow$: immediate.

$\Rightarrow$: A graph is chordal if and only if it has a *nice* tree decomposition [33]. Given a chordal graph, and its nice tree decomposition, we build a *program like* decomposition as follows:
(1) the only nodes that have more than one successor are the joint nodes. If a joint node v is the root of a subtree, replicate v. Let v' be the replicated node. Add the predecessor of v as the predecessor of v', and let the unique predecessor of v be v'. Now, v' is the root of any subtree that contains v.
(2) this is in accordance to the definition of nice tree, for joint nodes are never leaves of subtrees.
(3) If there is $v \in T$ such that v is the root of $\sigma_x, \sigma_y \in V$, then replicate v. Let v' be the replicated node in such a way that $K_{v'} = K_v \setminus \{x\}$. Add the predecessor of v as the predecessor of v', and let the unique predecessor of v be v'. Now, v' is the root of any subtree that reaches v, other than σ_y.
(4) If there is $v \in T$ such that v is the leaf of $\sigma_x, \sigma_y \in V$, then replicate v. Let v' be the replicated node in such a way that $K_{v'} = K_v \setminus \{x\}$. Add the sucessor of v as the successor of v', and let the unique successor of v be v'. Now, v' is the leaf of any subtree that reaches v, except σ_y.
(5) If there is a subtree that only spans one node, replicate that node as was done in (1).

□

We next define simple notions of *statement* and *program* that are suitable for this paper. We use v to range over program variables. A statement is defined by the grammar:

$$\begin{array}{rcll} \text{(Statement) } s & ::= & v = & \text{(definition of } v\text{)} \\ & | & = v & \text{(use of } v\text{)} \\ & | & \texttt{skip} & \end{array}$$

A program is a tree-structured flow chart of a particular simple form: a program is a pair (T, ℓ) where T is a finite tree, ℓ maps each vertex of T with zero or one successor to a statement, and each variable v is defined exactly once and the definition of v dominates all uses of v. Notice that a program is in strict SSA form.

The *interference graph* of a program (T, ℓ) is an intersection graph of a family of subtrees V of T. The family of subtrees consists of one subtree, called the *live range*, per variable v in the program; the live range is the subtree of the finite tree induced by the set of paths from each use of v to the definition of v. Notice that a live range consists of both vertices and edges (and not, as is more standard, edges only). That causes no problem here because we don't allow a live range to end in the same node as another live range begins.

From a chordal graph G presented as a finite tree T and a program-like family of subtrees V, we construct a program $P_G = (T, \ell)$, where for each subtree $\sigma \in V$, we define $\ell(\text{root}_\sigma) tobe``v_\sigma ="$, and for each subtree $\sigma \in V$, and a leaf $n$ of $\sigma$, we define $\ell(n) tobe``= v_\sigma''$. Figure 31(d) shows the program that corresponds to the tree in Figure 31 (c).

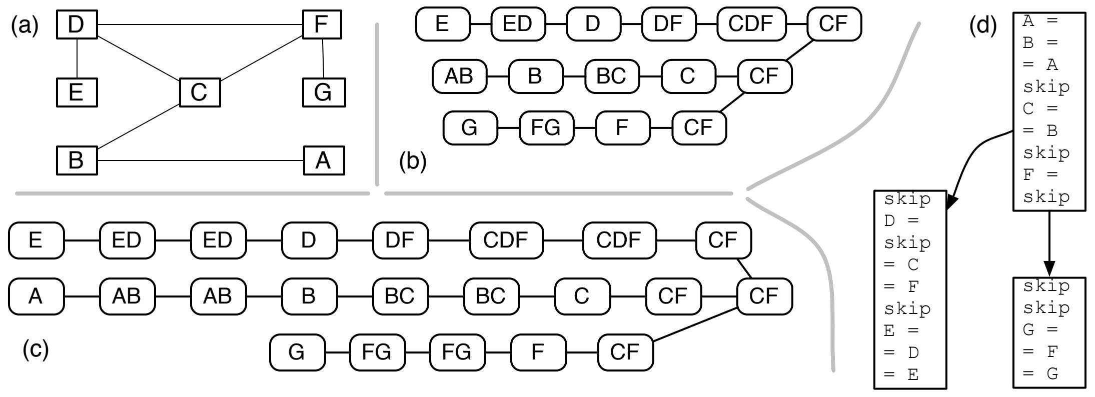
LEMMA 21. *G is the interference graph of P_G.*

Proof. For all $\sigma \in V$, the live range of v_σ in P is σ. □

In Section 4 we introduced families of variables in an elementary program. This concept is formally defined as:

DEFINITION 22. *Let P_s to be a strict program, and let P_e to be the corresponding elementary program. Given a variable $v \in P_s$, the set Q_v of all the variables in P_e produced from the renaming of v is called the family of variables v.*

We emphasize that the union of the live ranges of all the variables in a family Q_v is topologically equivalent to the live range of v. We state this fact as Lemma 23.



```
A =
B =
= A
skip
C =
= B
skip
F =
skip
```

```
skip
D =
skip
= C
= F
skip
E =
= D
= E
```

```
skip
skip
G =
= F
= G
```

Figure 31. A chordal graph represented as a program.

LEMMA 23. *Let P_s be a strict program, and let P_e be the elementary program derived from P_s. Let v and u be two variables of P_s, and let Q_v and Q_u be the corresponding families of variables in P_e. The variables v and u interfere if, and only if, there exists $v' \in Q_v$ and $u' \in Q_u$ such that v' and u' interfere.*

Proof. Follows from definition 22. ☐

THEOREM 24. *The maximal aligned 1-2-coloring extension problem for elementary graphs is NP-complete.*

Proof. The problem of finding the maximum induced subgraph of a chordal graph that is K colorable is NP-complete [47]. We combine this result with Lemmas 21 and 23 for the proof of this theorem. ☐

The proof of Theorem 4 is a corollary of Theorem 24:

Proof. Follows from Theorem 24. ☐

D. Pseudocode

The algorithm given below is an expansion of the program presented in Figure 10.

```
• $S$ = empty;
• $L$ = undefined for all registers;
• For each basic block $b$, in a pre-order traversal of the dominator
  tree of the program:
    ▪ For each instruction $i \in b$:
        1. if $i$ is the first instruction of $b$:
             – Head[$b$] = $L$;
        2. Let $p$ be a puzzle build from live-in, live-out and
           variables in $i$.
        3. while $p$ is not solvable:
             – choose and remove a piece $v$ from $p$; assign a
               memory address to $v$;
        4. $S'$ := a solution of $p$, guided by S.
        5. Update $L$ with the variables that $S$ places on the board.
        6. if there is instruction $i' \in b$ that precedes $i$:
             – implement the parallel copy between $i'$ and $i$ using
               $S$ and $S'$.
        7. $S$ = $S'$;
        8. if $i$ is the last instruction of $b$:
             – Tail[$b$] = $L$;
```

Important characteristics of our register assignment phase are:

- the size of the intermediate representation is kept small, i.e, at any moment the register allocator keeps at most one puzzle board in memory;
- the solution of a puzzle is guided by the solution of the last puzzle solved;
- parallel copies between two consecutive instructions i_1 and i_2 in the same basic block can be implemented after the puzzle for i_2 is solved. To implement a parallel copy means to insert copies/swaps to transfer a solution found to i_1 to i_2;
- we record the locations of variables at the beginning and at the end of each basic block in tables called Head and Tail. These recordings guide the elimination of φ-functions and π-functions.

The list L is a mapping of registers to variables. For instance, $L[v] = r$ denotes that register r is holding the value of variable v.

Once all the basic blocks have been visited, our register allocator procedes to implement φ-functions and π-functions. We use basically the technique described by Hack *et al.* [24]; however, the presence of aliasing complicates the algorithm. We are currently writing a technical report describing the subtleties of SSA-elimination after register allocation.