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 [12, 17, 36], integer linear programming [2, 19], partitioned Boolean quadratic optimization [21, 35], and multi-commodity network flow [25]. These abstractions represent different tradeoffs between compilation speed and quality of the produced code. For example, linear scan [33, 38] is a simple algorithm based on the coloring of interval graphs that produces code of reasonable quality with fast compilation time; iterated register coalescing [17] 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 [36] 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 [13], 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.* [20]).

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 [14], the iterated register coalescing algorithm of George and Appel [17] with the extensions proposed by Smith, Ramsey, and Holloway [36], and the partitioned Boolean quadratic optimization algorithm [21]. 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. Unsurprisingly, the slower Boolean optimization algorithm produces the fastest code.

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 extended version of our paper [32] has appendices with the proofs of the four theorems stated in this paper; we have omitted the proofs in this version due to space constraints.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

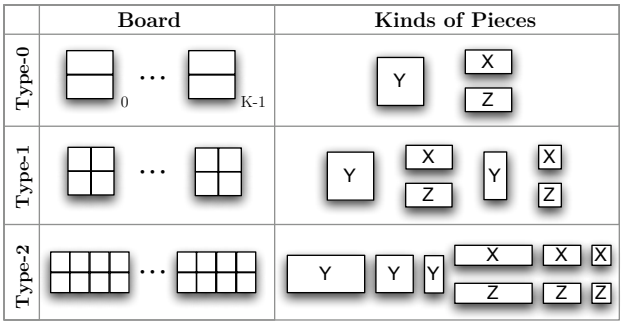


Figure 1. Three types of puzzles.

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 a 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.

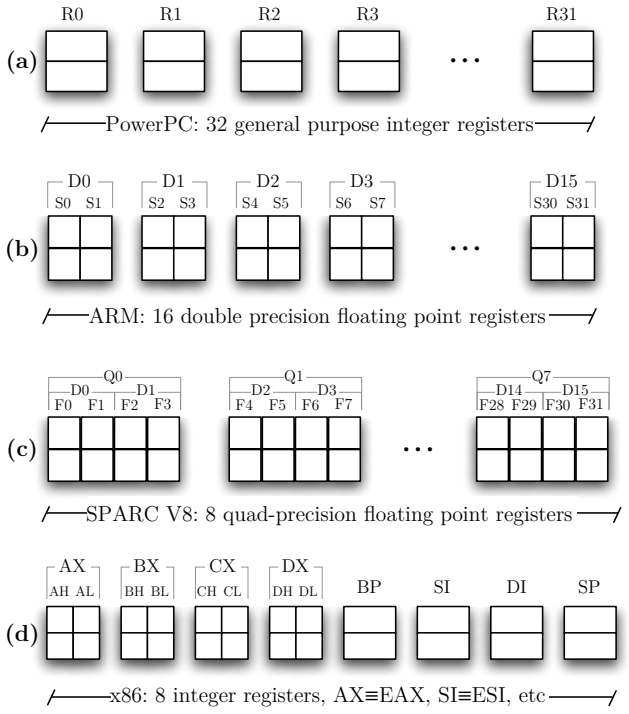


Figure 2. Examples of register banks mapped into puzzle boards.

Type-2 puzzles. SPARC V8 [22, 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 [39, 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.

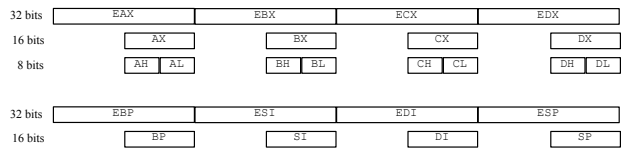


Figure 3. General purpose registers of the x86 architecture

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 [13] 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 [23].) 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 precolored registers. We use $x = y$ to denote an instruction that uses y and defines x; it is not a simple copy.

Cytron *et al.* [13] 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 a 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.

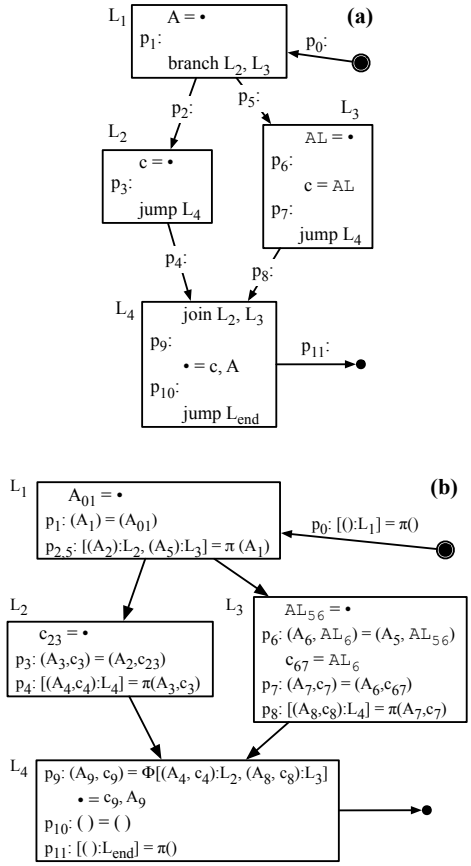


Figure 4. (a) Original program. (b) Elementary program.

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).

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.*

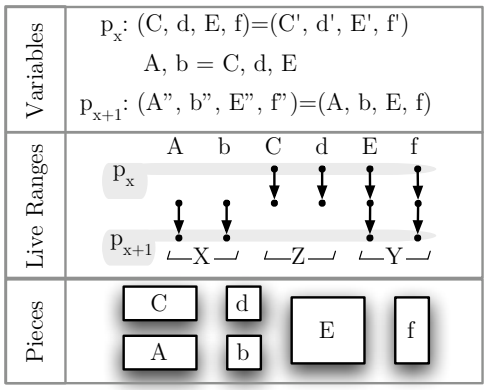


Figure 5. Mapping program variables into puzzle pieces.

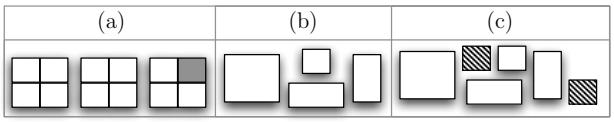


Figure 6. Padding: (a) puzzle board, (b) pieces before padding, (c) pieces after padding. The new pieces are marked with stripes.

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. In the full version [32] we show 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 $r : s$. We now informally explain the meaning of rules, statements, and programs.

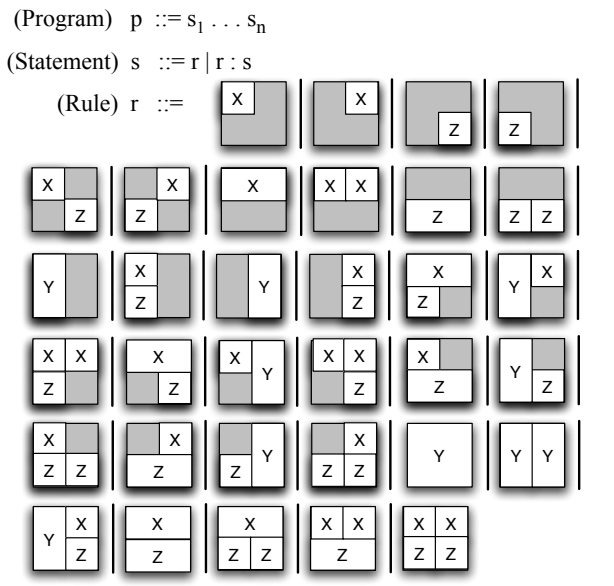


Figure 7. A visual language for programming puzzle solvers.

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

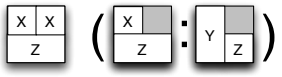
because the pattern of the rule does not match this area.

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.

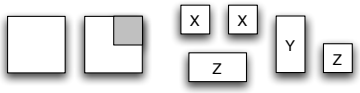
Programs. The execution of a program $s_1 \ldots s_n$ on a puzzle $\mathcal{P}$ proceeds as follows:

- 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.

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:

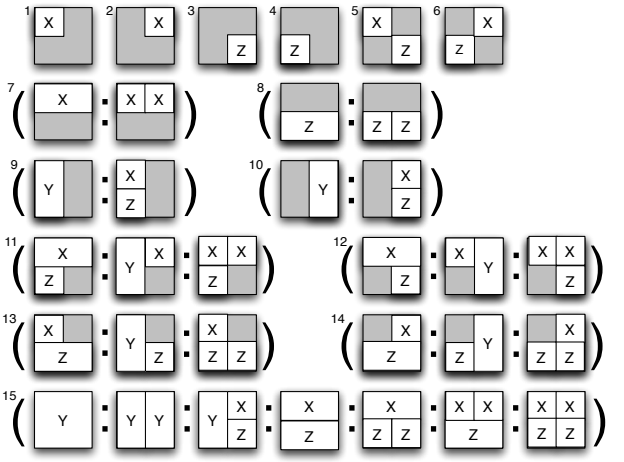
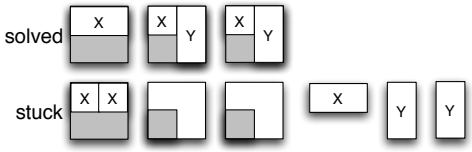
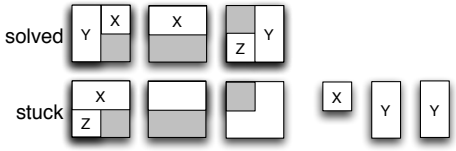


Figure 8. Our puzzle solving program



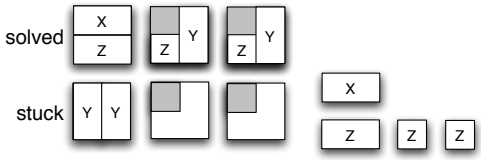
Because statement 7 prefers a size-2 X-piece over two size-1 X-pieces, the example is impossible. Notice that our program also prefers the size-2 pieces over the size-1 pieces in statements 8–15 for reasons similar to our analysis of statement 7.

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:



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:

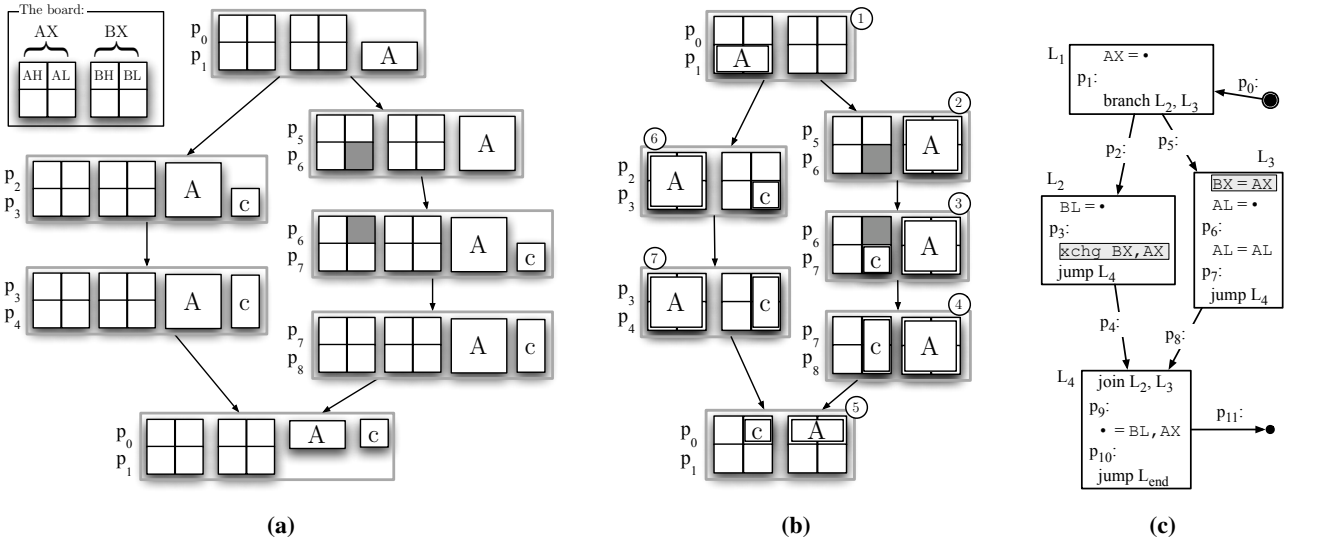
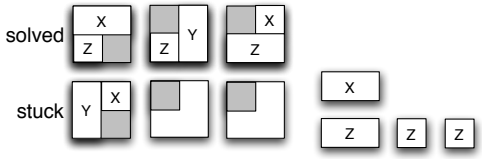


Figure 9. (a) The puzzles produced for the program given in Figure 4(b). (b) An example solution. (c) The final program.



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.*

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.

- 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

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

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 [32]. 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 choos-

ing 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 [33] in linear-scan register allocation. The furthest-first strategy spills a 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.* [38].

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

We now describe three optimizations that we have found useful in our implementation of register allocation by puzzle solving for x86.

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 [8]. The second transformation converts the target program into a variation of SSA-form called *Conventional SSA-form* (CSSA) [37]. 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 [11]. These two transformations are enough to handle the 'swap' and 'lost-copy' problems pointed out by Briggs *et al.* [8].

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, 31]. 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.

6. Experimental Results

Experimental platform. We have implemented our register allocator in the LLVM compiler framework [26], 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 11. All the figures use short names for the benchmarks; the full names are given in Figure 11. We order these benchmarks by the number of non-empty puzzles that they produce, which is given in Figure 13.

Puzzle characteristics. Figure 12 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 13, 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 13 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 14 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 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.

	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 11. 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).

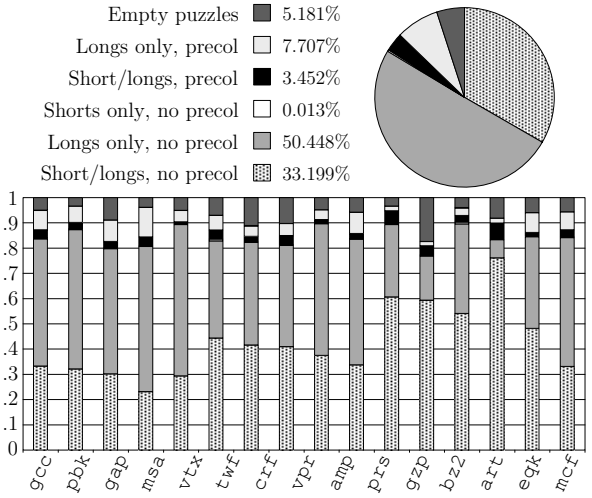


Figure 12. The distribution of the 1,486,301 puzzles generated from SPEC CPU2000.

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.* [40] and Evlogimenos [14]. 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 [17] with extensions by Smith, Ramsey, and Holloway [36] for handling register aliasing. We use EIRC (Extended Iterated Register Coalescing) to denote this register allocator.

The third register allocator is based on partitioned Boolean quadratic programming (PBQP) [35]. 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 [21]. 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.

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 13. 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.

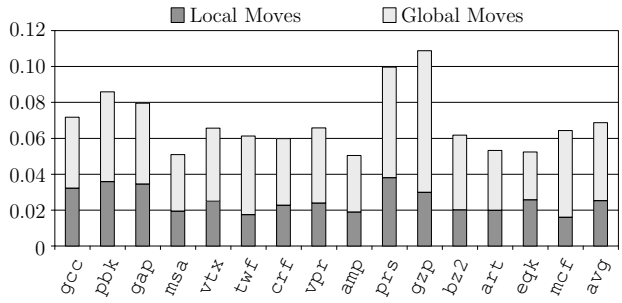


Figure 14. Number of copy and swap instructions inserted per puzzle.

Stack size comparison. The top half of Figure 15 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 15 compares the number of load/store instructions in the assembly code. The puzzle solver inserts marginally fewer memory-access instructions than PBQP, 1.2% fewer memory-access instructions than EIRC, and 9.6% 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 of the live range of a spilled variable.

Run-time comparison. Figure 16 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

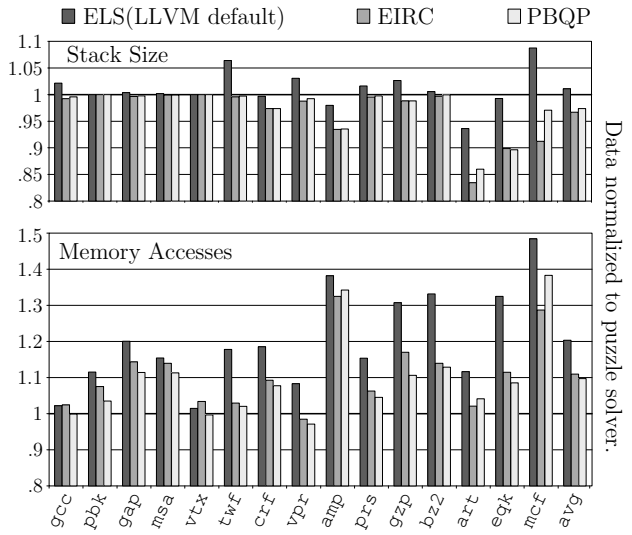


Figure 15. In both charts, the bars are relative to the puzzle solver; shorter bars are better for the other algorithms. **Stack size:** Comparison of the maximum amount of bytes reserved on the stack. **Number of memory accesses:** Comparison of the total static number of load and store instructions inserted by each register allocator.

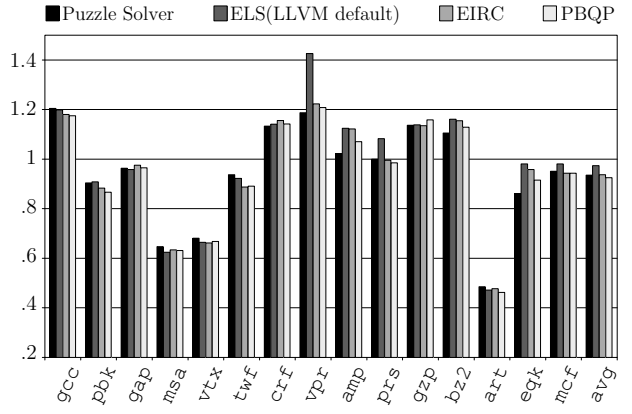


Figure 16. Comparison of the running time of the code produced with our algorithm and other allocators. The bars are relative to gcc -O3; shorter bars are better.

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 time demanding PBQP.

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 17 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.

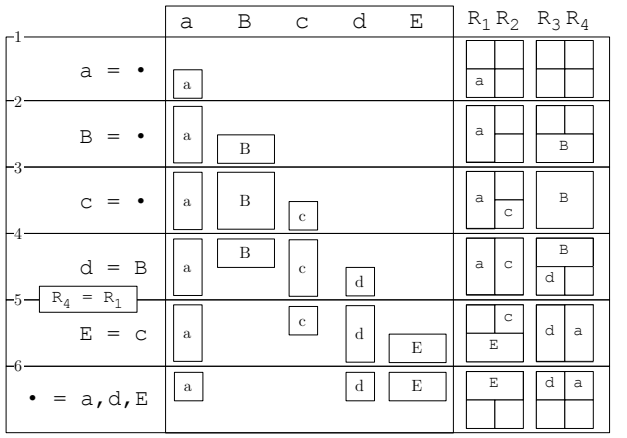


Figure 17. (left) Example program. (center) Puzzle pieces. (right) Register assignment.

Compile-time comparison. Figure 18 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 compilation speeds from statements in previous papers. The experiments shown in [21] 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.* [36] 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 [33] suggests that graph coloring is about twice as slow as linear scan, while Traub *et al.* [38] gives an slowdown of up to 3.5x for large programs, and Sarkar and Barik [34] 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 21 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 19 shows the interference graph of the elementary program in Figure 4(b). Any graph can be the interference graph of a general program [12]. SSA-form programs have chordal interference graphs [6, 9, 20, 30], and the interference graphs of SSI-form programs are interval graphs [10]. We call the interference graph of an elementary program an *elementary graph* [32]. Each connected com-

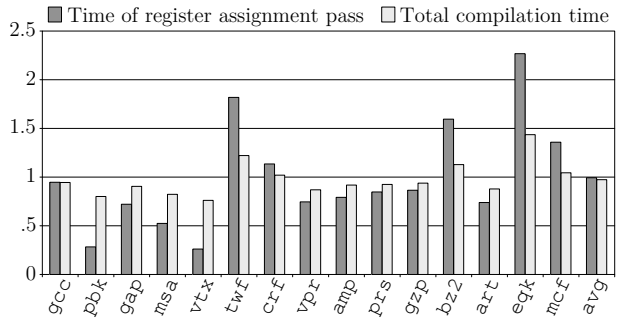


Figure 18. Comparison between compilation time of the puzzle solver and extended linear scan (LLVM's default algorithm). The bars are relative to the puzzle solver; shorter bars are better for extended linear scan.

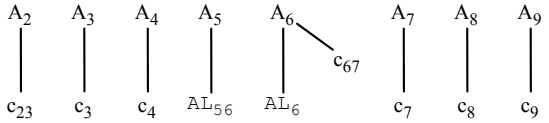


Figure 19. Interference graph of the program in Figure 4(b).

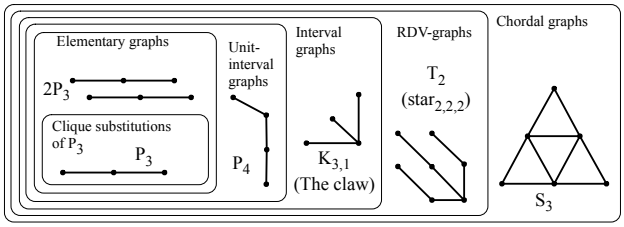


Figure 20. Elementary graphs and other intersection graphs. RDV-graphs are intersection graphs of directed lines on a tree [29].

ponent 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 20 illustrates these inclusions. Elementary graphs are also *Trivially Perfect Graphs* [18], as we show in [32]. 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 [12] because coloring general graphs is NP-complete. However, this problem has a polynomial time solution for SSA-form programs [6, 9, 20] because chordal 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,

	Class of graphs			
Program	general	SSA-form	SSI-form	elementary
Problem	general	chordal	interval	elementary
ALIGNED 1-2-COLORING EXTENSION	NP-cpt [24]	NP-cpt [4]	NP-cpt [4]	linear [TP]
ALIGNED 1-2-COLORING	NP-cpt [24]	NP-cpt [27]	NP-cpt [27]	linear [TP]
COLORING EXTENSION	NP-cpt [24]	NP-cpt [4]	NP-cpt [4]	linear [TP]
COLORING	NP-cpt [24]	linear [16]	linear [16]	linear [16]

Figure 21. Algorithms and hardness results for graph coloring. NP-cpt = NP-complete; TP = this paper.

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 [27].

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 [28].

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 [32] 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.* [15]. 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 [41].

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:Iinterval 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] 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.

[9] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *IWLS*, pages 447–454. 2005.

[10] 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.

[11] 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.

[12] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

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

[15] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA*, pages 564–573. ACM Press, 1998.

[16] 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.

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

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

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

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

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

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

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

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

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

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

[27] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation for straight-line programs is NP-complete. In *ICALP*, pages 680–691. Springer, 2007.

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

[29] 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.

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

[31] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is NP-complete. In *FOSSACS*, pages 79–93. Springer, 2006.

[32] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving, 2008. `http://compilers.cs.ucla.edu/fernando/projects/puzzles/`.

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

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

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

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

[37] 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.

[38] 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.

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

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

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