Optimal Spilling for CISC Machines with Few Registers

or, reducing register allocation to the unsolved problem of optimal coalescing

PRELIMINARY DRAFT

Andrew W. Appel
Princeton University

Lal George
Bell Labs, Lucent Technologies

August 24, 2000

Note: Since writing this early draft, we have found a good heuristic for coalescing that effectively removes the subtitle of the August 24th draft, i.e. it "solves" the problem of optimal coalescing. Actually, our coalescing algorithm is not optimal, but it is good enough to make optimal spilling useful and practical. The more recent (November 14th, 2000) draft of the paper (without subtitle) is at
http://ncstrl.cs.Princeton.EDU/expand.php3?id=TR-630-00
but the August 24th version may still be useful to some people, because it describes some aspects of the linear-programming "solution" to coalescing that are not in the more recent draft.

Abstract

Register allocation based on graph coloring performs poorly for machines with few registers, if each temporary is held either in machine registers or memory over its entire lifetime. With the exception of short-lived temporaries, most temporaries must spill – including long lived temporaries that are used within inner loops. Live-range splitting before or during register allocation helps to alleviate the problem but prior techniques are sometimes complex, make no guarantees about subsequent colorability and thus require further iterations of splitting, pay no attention to addressing modes, and make no claim to optimality. We formulate the register allocation problem for CISC architectures with few registers in two parts: an integer linear program that determines the optimal location to break up the implementation of a live range between registers and memory, and a register assignment phase that we guarantee to complete without further spill code insertion. Our linear programming model considers the various addressing modes available to an instruction and finds an optimal solution quickly. The second phase will complete without spilling, but at the expense of register-register copies remaining in the program. The task of coloring while leaving behind the minimum number of splits (*optimal coalescing*) is left as an open problem; we discuss two unsatisfactory solutions (one fast and suboptimal, the other optimal but far too slow).

1 Introduction.

Register allocation by graph coloring has been a big success for machines with 30 or more registers. The instruction selector generates code using an unlimited supply of temporaries; liveness analysis constructs an interference graph with an edge between any two temporaries that are live at the same time (and thus cannot be allocated to the same register); a graph coloring algorithm finds a K-coloring of the interference graph (where K is the number of registers on the machine). If the graph is not K-colorable, then some nodes are spilled: the temporaries are implemented in memory instead of registers, with a cost for loading them and storing them when necessary. Graph coloring is NP-complete, but simple algorithms can often do well.

An important improvement to this algorithm was the idea that the live range of a temporary should be split into smaller pieces, with move instructions connecting the pieces. This relaxes the interference constraints a bit, making the graph more likely to be K-colorable. The graph-coloring register allocator should coalesce two temporaries that are related by a move instruction if this can be done without increasing the number of spills.

Unfortunately, this approach has not worked well for machines like the Pentium, which have $K = 6$ allocable registers (there are 8 registers but usually two are dedicated to specific purposes). What happens is that there will typically be many nodes with degree much greater than K, and there is an enormous amount of spilling. Of course, with few registers there will inevitably be spilling, as the live variables cannot all be kept in registers; but if a variable is spilled because it has a long live range, then it stays spilled even (for example) in some loop where it is frequently used.

There have been much research on shrinking and splitting of live ranges[LGAT97, Bri92, CH90, LH86], but some of these approaches are quite complicated and it was not clear how best to integrate them with graph-coloring or other register allocation algorithms.

At one point we attempted a simple solution to the problem: split all the live ranges into quite small pieces (one basic block per piece), then use iterated register coalescing [GA96] to join as many pieces as possible without spilling. It was obvious to us that this approach would work, but in fact it failed. The problem is that many nodes have very high degree, so no conservative coalescing was possible. Iterated register coalescing guarantees to keep a colorable graph colorable, but it cannot find the optimal set of nodes to spill in an uncolorable graph.

In the last few years some researchers have taken a completely different approach to register allocation: formulate the problem as an integer linear program (ILP) and solve it exactly with a general-purpose ILP solver. ILP is NP-complete, but approaches that combine the simplex algorithm with branch-and-bound can be successful on some problems. Unfortunately, the work to date in optimal register allocation via ILP has not quite been practical: Goodwin's optimal register allocator can take hundreds of seconds to solve for a large procedure [Goo96, GW96]. Goodwin has formulated "near-optimal register allocation (NORA)" as an ILP; our solution can be viewed as a different approach to near-optimal register allocation.

A two-phase approach. Our new approach decomposes the register allocation problem into two parts: spilling, then register assignment. Instead of asking, "at program point p, should variable v be in register r?" we first ask, "at program point p, should variable v be in a register or in memory?" Clearly, this is a simpler question, and in fact we can formulate an integer linear program (ILP) that solves it optimally and efficiently (tens of milliseconds). This phase of register allocation finds the optimal set of spills.

Not only does our algorithm compute where to insert loads and stores to implement spills, but it also optimally selects addressing modes for CISC instructions that can get operands directly from memory. For example, the add instruction on the Pentium takes two operands s and d, and computes $d \leftarrow d + s$. The operands can be in registers or in memory, but they cannot both be in memory. On a modern implementation of the instruction set, the instruction $m[x] \leftarrow m[x] + s$ is no faster than the sequence of instructions $r \leftarrow m[x]$; $r \leftarrow r + x$; $m[x] \leftarrow r$. However, the latter sequence requires an explicit temporary r, and if there are many other live values at this point, some other value will have to be spilled; the former sequence wouldn't require the spilling of some other value. Therefore, it is important to make use of the CISC instructions.

The second phase is to allocate the unspilled variables to registers. This is still difficult to do efficiently, as we will discuss in section 5.

In judging our decomposition into two phases, there are three important questions to ask:

1. When we decompose the problem into two subproblems (spilling, coloring) and solve each subproblem optimally, does that lead to an optimal solution to the original problem? We will present empirical evidence that the solutions are excellent, but there is no theoretical reason that they will be optimal.

2. Can the spilling subproblem be solved optimally and efficiently? We will show that it can, using integer linear programming.

3. Can the coloring subproblem be solved optimally and efficiently? We can do it optimally but slowly using ILP; we can do it quickly but suboptimally using iterated register coalescing; but at present we cannot do it optimally and efficiently. However, the coloring subproblem is an interesting simplification of the general graph-coloring register-allocation problem, and one that we think merits further study as a problem in algorithms.

2 Optimal spilling via ILP

We model the register-spilling problem as a 0-1 linear program, that is, an optimization problem with constraints that are linear inequalities, a linear cost function, and the additional constraint that every variable must take the value 0 or 1. We use AMPL [FGK93] to describe, generate, and solve the linear program. AMPL is a programming language for describing the constraints and objective in a linear program as a mathematical model. The AMPL compiler derives an instance of the optimization problem by instantiating the model with data specific to the task being solved, and feeds the

resulting system (in a suitable form) to a standard off-the-shelf simplex solver. The data is in the form of symbolic sets and scalar parameters.

The input to our spiller is a control-flow graph containing Intel IA-32 instructions. At the lowest level, our model contains a set of symbolic variables **V** corresponding to temporaries in the program, and a set **P** of points within the flowgraph. There is a point between any two sequential instructions. A branch instruction terminates in a single point that is then connected to all points at the targets of the branch. In the AMPL model, these sets are declared simply as:

```
model
set V;
set P;
```

There are several different classes of instructions in the IA-32 instruction set, such as two address binary instructions ($d \leftarrow d \oplus s$), and unary instructions ($d \leftarrow \mathrm{f}(s)$), for example. If there is an add instruction between program point p_1 and a successor point p_2, with source variable v_1 and destination variable v_2, we model this by writing, $(p_1, p_2, v_1, v_2) \in$ `Binary`, and similarly for `Unary`. That is, set `Binary` is a subset of $\mathbf{P} \times \mathbf{P} \times \mathbf{V} \times \mathbf{V}$ and is declared in the AMPL model using:[1]

```
set Binary ⊂ (P×P×V×V) ;
set Unary ⊂ (P×P×V×V) ;
```

For any variable v_1 that is live at a point p_1, we write $(p_1, v_1) \in$ `Exists`. The `Exists` set is similar to the live set but not identical: if an instruction between points p_1 and p_2 produces a result v that is immediately dead, then v is nowhere live but $(p_2, v) \in$ `Exists`.

If a variable v_1 is live and carried unchanged from point p_1 to p_2, then we say that $(p_1, p_2, v_1) \in$ `Copy`. If, from point p_1 to point p_2 variable v_1 is copied to variable v_2 (e.g., by a move instruction), we write $(p_1, p_2, v_1, v_2) \in$ `Copy2`.

```
set Exists ⊂ (P×V) ;
set Copy ⊂ (P×P×V) ;
set Copy2 ⊂ (P×P×V×V) ;
```

The compiler selects instructions that use temporaries ($v, w, \ldots$) that are to be assigned registers, but sometimes refers to specific registers (`%eax`, `%esp`, ...), either because a machine instruction requires an operand in a specific register or because of parameter-passing conventions.

Now consider the instruction

```
movl    %eax, %v
```

that moves the contents of register `%eax` to the variable `v`. We model this as an instruction that takes no argument (because no temporary is a source operand) and produces a result into `v`. `Binary` instructions (such as `movl`) can take their source or destination operands from registers or memory, but they cannot both be from memory. In this

[1] AMPL actually uses the word `cross` instead of the symbol $\times$, and `within` instead of $\subset$. In general, we will use mathematical notation instead of strictly AMPL notation in the body of the paper, and give the exact AMPL code in the appendix.

case, since the source (`%eax`) is known to be a register, the destination can be a register or memory. The class of instructions that take no argument and produce a register or memory result we call `Nullary`.

In contrast, in the instruction

```
movl     4(%esp), %v
```

that moves the contents of memory at `4(%esp)` to `v`, the operand `v` must be a register. The instruction class that take no argument and produce a register-only result we call `NullaryReg`.

$$\texttt{set Nullary} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$
$$\texttt{set NullaryReg} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$

Some instructions accomplish $v \leftarrow \mathrm{f}(v)$, where v can be in a register or memory (e.g. `addl($256, %v)`, that adds an immediate to the variable `v`); others require that v must be in a register and nothing else (e.g. `addl(4(%esp), %v)`). We call these `Mutate` and `MutateReg` respectively:

$$\texttt{set Mutate} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$
$$\texttt{set MutateReg} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$

For cases where no results are produced, the instruction may take two operands of which at most one can be in memory (e.g., the *compare* instruction); or take one operand which can be either a register or memory (e.g. `addl(%v, %eax)`); or take one operand that must be in a register. We call these three instruction-classes `UseUp2`, `UseUp`, and `UseUpReg` respectively:

$$\texttt{set UseUp2} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V}\times\mathbf{V})\ ;$$
$$\texttt{set UseUp} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$
$$\texttt{set UseUpReg} \subset (\mathbf{P}\times\mathbf{P}\times\mathbf{V})\ ;$$

Consider a branch instruction between points p_1 and p_2 that branches to p_4 if $v_1 = 0$, but otherwise falls through to p_3. It is necessary to know about points such as p_2 that are associated with a branch, as we cannot insert spill or reload instructions at p_2. We therefore have

$$\texttt{set Branch} \subset \mathbf{P}$$

with $p_2 \in$ `Branch`. Suppose v_3 is live throughout, and v_1 is live only in the p_4 successor. Then it is necessary to propagate this liveness along the edges of the branch, and we represent this by generating

$$(p_1, p_2, v_1) \in \texttt{UseUp}$$

$$\{(p_1,p_2,v_3),(p_2,p_3,v_3),(p_2,p_4,v_3),\\ (p_1,p_2,v_1),(p_2,p_4,v_1),\} \quad \subset \texttt{Copy}$$

The model declares several scalar and vector parameters (that are indexed symbolically using sets such as **P**). Each point in the program has an estimated frequency of execution that is used to weight the cost of spill or reload instructions in our optimal spilling framework. The frequencies can be obtained by profiling or by static estimation [WL94]. In our model we have the vector parameter `weight`:

```
param weight {P};
```

to say that there is one `weight` parameter for each point.

In the case where the compiler has explicitly used a machine register (e.g., `movl %eax,%v`), that register (e.g., `%eax`) is not available for coloring temporaries live at the same point. We communicate this to the model via a parameter `K` specific to each point:

```
param K {P};
```

Finally we have some scalar cost parameters:

`param` $C_{\text{load}},\ C_{\text{store}},\ C_{\text{move}},\ C_{\text{instr}}$

C_{load}, C_{store} and C_{move} are the cost of executing a load, store, and move instruction. C_{instr} is the cost of fetching and decoding one instruction byte. Presumably, $C_{\text{load}} > C_{\text{store}} > C_{\text{move}} > C_{\text{instr}}$. (In fact, C_{instr} really measures the cost of a slight extra pressure on the instruction cache.)

Example. Figure 1 shows the Intel IA-32 instructions that may be generated for the factorial function, and Figure 2 shows the corresponding flowgraph annotated with points surrounding each instruction. For this program the following AMPL sets are generated:

```
set P := {p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14}
set V := {t1 t2}
set Branch := {p7 p11}
set NullaryReg := {(p3,p4,t1)}
set UseUp2 := {(p5,p6,t1,t2)}
set UseUp := {(p8,p9,t1) (p12,p13,t2)}
set Mutate := {(p9,p10,t1)}
set MutateReg := {(p8,p9,t2)}
set Binary := {(p8,p9,t1,t2)}
set Copy :=
   {(p4,p5,t1) (p5,p6,t1) (p6,p7,t1) (p7,p8,t1) (p8,p9,t1) (p10,p11,t1) (p11,p8,t1)
    (p5,p6,t2) (p6,p7,t2) (p7,p8,t2) (p9,p10,t2) (p10,p11,t2) (p11,p8,t2)}
set Exists :=
   {(p4,t1) (p5,t1) (p6,t1) (p7,t1) (p8,t1) (p9,t1) (p10,t1) (p11,t1)
    (p5,t2) (p6,t2) (p7,t2) (p8,t2) (p9,t2) (p10,t2) (p11,t2) (p12,t2) (p13,t2)}
```

The `imull` instruction is not classified as a `Binary` instruction as the destination must be a register operand, and cannot be memory. Therefore, `imull` is classified as `MutateReg` for the destination operand and `UseUp` for the source operand, as the source can be in either class.

Missing in the data are the concrete parameters such as the execution frequency of each point, the costs, and the value of `K` at each point. If we assume that `%esp` and `%ebp` are dedicated, then the value of `K` at all points in the flowgraph is 6, except at point `p_13` where `%eax` is defined, and the value of `K` is 5.

```
fac:   pushl  %ebp           ;; save old frame pointer
       movl   %esp, %ebp     ;; new frame pointer
       movl   8(%ebp), t1    ;; n
       movl   #1, t2         ;; fac := 1
       testl  t1, t1         ;; cc := n ∧ n
       je     L1             ;; if n=0 goto L1
L2:    imull  t1, t2         ;; fac := n * fac
       decl   t1             ;; n := n - 1
       jnz    L2             ;; if n <> 0 goto L2
L1:    movl   t2, %eax       ;; return register
       leave                 ;; done
       ret
```

Figure 1: Intel IA-32 instructions for the factorial function

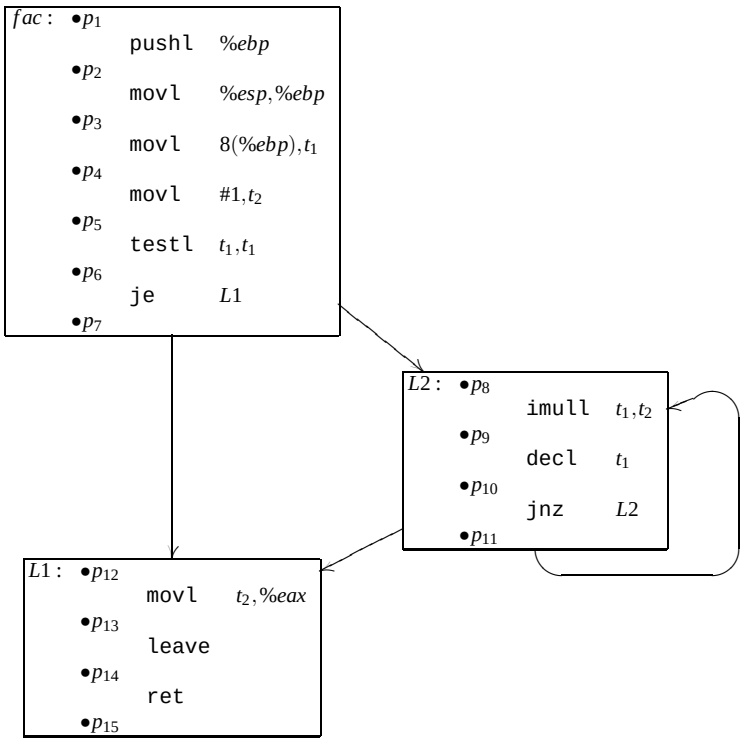

Figure 2: Flowgraph annotated with points

3 Variables and constraints

Spilling is the insertion of loads and stores between the instructions of the program. Each instruction of our program spans a pair of points, and "between the instructions" means "at a point." Thus, we will insert loads/stores at points, not between them.

Consider a variable v live at a program point p. The variable v could arrive at p in a register and depart in a register – we call this $r_{p,v}$. Or it could arrive in memory and depart in memory – $m_{p,v}$. It could arrive in a register and depart in memory – $s_{p,v}$; or v could arrive in memory and depart in a register – $l_{p,v}$. A solution to the spilling problem is just the description of where the loads and stores are to be inserted.

We model this as follows:

```
var r {Exists} binary;
var m {Exists} binary;
var l {Exists} binary;
var s {Exists} binary;
```

This says that for each (p,v) in `Exists` – that is, for each variable v live at a program point p – there are linear-program variables $r_{p,v}$, $m_{p,v}$, $l_{p,v}$, and $s_{p,v}$; the `binary` keyword says that the variable must take on the value 0 or 1. We wish to find the values of these variables subject to a set of linear constraints.

Exists: The first constraint is that exactly one of these variables is set for any p and v:

$$\forall (p,v) \in \texttt{Exists}.\ \ l_{p,v} + r_{p,v} + s_{p,v} + m_{p,v} = 1$$

Branch: At a branch-point it's not possible to load or store, because we can't insert an instruction after a conditional-branch instruction but before its targets.

$$\forall (p,v) \in \texttt{Exists} \text{ s.t. } p \in \texttt{Branch}.\ \ l_{p,v} + s_{p,v} = 0$$

Coloring: At any point p, all the stores can be performed before all the loads. However, the variables to be stored originate in registers, therefore the sum of variables that remain in registers and those that are to be spilled must be no more than the number of registers available for coloring.

$$\forall p \in \mathbf{P}.\ \ K[p] \ \geq \sum_{(p,v) \in \texttt{Exists}} r_{p,v} + s_{p,v}$$

Similarly, after all the loads have been done at a point, the number of variables in registers should be no more than K.

$$\forall p \in \mathbf{P}.\ \ K[p] \ \geq \sum_{(p,v) \in \texttt{Exists}} r_{p,v} + l_{p,v}$$

Copy propagation: If a variable v is copied from p_1 to p_2, then either it departs from p_1 in a register and arrives at p_2 in a register, or it departs from p_1 in memory and arrives at p_2 in memory. If it departs from p_1 in a register it must have already been in a register (i.e. $r_{p_1,v} = 1$), or was loaded into a register at p_1 ($l_{p_1,v} = 1$). If it arrives at p_2 in a register, it can either continue in a register at p_2 ($r_{p_2,v} = 1$) or it can be stored at p_2 ($s_{p_2,v} = 1$):

$$\forall (p_1, p_2, v) \in \texttt{Copy}. \quad l_{p_1,v} + r_{p_1,v} = s_{p_2,v} + r_{p_2,v}$$

The constraint $s_{p_1,v} + m_{p_1,v} = l_{p_2,v} + m_{p_2,v}$ is redundant and must not be specified (redundant constraints will – with the inevitable rounding errors – overconstrain the problem so that the LP solver fails to find a solution).

If a variable v_1 at p_1 is copied to a variable v_2 at p_2, then if it departs v_1 in a register it must arrive v_2 in a register. The constraint is similar to the `Copy` case except that two variables are involved.

$$\forall (p_1, p_2, v_1, v_2) \in \texttt{Copy2}.$$
$$l_{p_1,v_1} + r_{p_1,v_1} = s_{p_2,v_2} + r_{p_2,v_2}$$

3.1 Specifying the CISC instructions

On the IA-32 (x86, Pentium), if there is a `Binary` instruction (e.g., two-operand add) between p_1 and p_2, operating on source variable v_1 and destination variable v_2, then at least one of v_1 and v_2 must depart p_1 in registers:

$$\forall (p_1, p_2, v_1, v_2) \in \texttt{Binary}$$
$$l_{p_1,v_1} + r_{p_1,v_1} + l_{p_1,v_2} + r_{p_1,v_2} \geq 1$$

Furthermore, the destination operand v_2 must be in registers departing p_1 if and only if it is in registers arriving p_2:

$$\forall (p_1, p_2, v_1, v_2) \in \texttt{Binary}$$
$$l_{p_1,v_2} + r_{p_1,v_2} = s_{p_2,v_2} + r_{p_2,v_2}$$

There are similar constraints for the other classes of instructions, as shown in the appendix. They say that the result of a `NullaryReg` must arrive p_2 in a register; at least one operand of a `UseUp2` must be in a register; the operand of a `UseUpReg` must be in a register; the operand of a `Mutate` must depart p_1 in the same storage class as it arrives p_2; the operand of a `MutateReg` must depart p_1 in a register and arrive p_2 in a register; and that at least one operand of a `Unary` must be in a register.

These constraints are all Pentium-specific, but by illustrating how easily they are specified we hope to convince the reader that many kinds of CISC instructions could be specified within this framework.

3.2 Objective function

The objective function of our linear program calculates the estimated runtime cost of the spill-related loads, stores, and CISC operands. The first component of the cost comes from loads and stores:

$$
\begin{array}{l}
\text{minimize COST}: \\
(\sum_{(p,v)\in \texttt{Exists}} \\
\quad \texttt{weight}_p((C_{\text{load}} + 3C_{\text{instr}})l_{p,v} + \\
\qquad\qquad (C_{\text{store}} + 3C_{\text{instr}})s_{p,v})) \\
+ \quad \ldots
\end{array}
$$

The cost of executing a load is C_{load}. The cost of a 3-byte load instruction (in i-cache occupancy) is $3C_{\text{instr}}$. For each point p and variable v such that there is a spill-load of v at p we incur this cost; and similarly for stores.

If the destination operand of a `Binary` instruction is in memory, we incur a cost C_{load} and C_{store}, and only one byte of C_{instr} cost to specify the operand. If the source operand is in memory, then we incur only a load cost and one instruction-byte cost:

$$
\begin{array}{l}
+(\sum_{(p_1,p_2,v_1,v_2)\in \texttt{Binary}} \\
\quad \texttt{weight}_{p_1}((C_{\text{load}} + C_{\texttt{instr}})(1 - (r_{p_1,s} + l_{p_1,v_1})) \\
\qquad + (C_{\text{load}} + C_{\text{store}} + C_{\texttt{instr}})(1 - (r_{p_2,v_2} + s_{p_2,v_2})))) \\
+ \quad \ldots
\end{array}
$$

There are similar clauses to account for the cost of memory operands of the other classes of instructions: `Unary`, `Mutate`, and so on.

This completes the description of our linear-program model of spill costs.

3.3 Special cases of instructions

Consider an add instruction whose destination is known to be in memory: $m[x] \leftarrow m[x] + v$. This could occur because x is the address of an array element, for example. Then v must be in a register, and x must be in a register. We can model this as

$$
\begin{array}{l}
(p_1, p_2, x) \in \texttt{UseUp} \\
(p_1, p_2, v) \in \texttt{UseUp}
\end{array}
$$

Similarly, the instruction $v \leftarrow v + m[x]$ is modeled as

$$
\begin{array}{l}
(p_1, p_2, v) \in \texttt{Mutate} \\
(p_1, p_2, x) \in \texttt{UseUp}
\end{array}
$$

Or consider the case where the source operand is a constant, $v \leftarrow v + c$:

$$(p_1, p_2, v) \in \texttt{Mutate}$$

There are many variations on this theme, but the point is that each special case of an instruction (where one of the operands is forced to be in memory, or in registers, or constant) reduces to a case that can also be described in the model. The compiler does this reduction before generating the data set sent to AMPL.

4 Solving the model.

Our compiler [AM91][GGR94] feeds the data associated with a flowgraph together with the model to AMPL. AMPL generates a linear program with variables, constraints, and an objective function. From the example in Figure 2 the variables:

$$r_{p_4,t_1},\ \ l_{p_4,t_1},\ \ s_{p_4,t_1},\ \ m_{p_4,t_1}$$

would be generated for t_1 corresponding to the point p_4, since $(p_4,t_1) \in$ `Exists`. A constraint corresponding to the **Exists** formula (Section 3) would establish the equation:

$$r_{p_4,t_1} \ + \ l_{p_4,t_1} \ + \ s_{p_4,t_1} \ + \ m_{p_4,t_1} \ = \ 1$$

In a typical large cluster of basic blocks spanning several source-program functions, there will be a few thousand points p and several hundred temporaries v, yielding tens of thousands of linear-program variables.

AMPL first runs a "presolve" phase in which as many variables as possible are eliminated; for example, any use of $m_{p,v}$ could be replaced by $1-(r_{p,v}+l_{p,v}+s_{p,v})$. After the presolve, AMPL formats the linear program in a way acceptable to the back end, which is any one of several commercial and noncommercial LP solvers. Some of these solvers can do integer linear programs using a combination of the simplex method with branch-and-bound; others can do only continuous LP's using simplex alone. We have used CPLEX [CPL00] and IBM's OSL [Hun93]; CPLEX is an order of magnitude faster but sometimes dumps core.

After the ILP solver is finished, AMPL formats the results – a table of r,l,s,m for each (p,v). Our compiler computes all the spilling from this information inserting load and store instructions at points where $l_{p,v}$ and $s_{p,v}$ is set, and introduces memory operands at instructions for which $m_{p,v}$ is set. A prior phase assigns a logical spill location for every temporary, ensuring that nonoverlapping live ranges share the same memory location.

5 Optimal register coalescing

In a flowgraph where no more than K variables are ever simultaneously live, it may still be the case that there is no K-coloring of the variables – that K registers do not suffice. If x interferes with y at point p_1, y interferes with z at point p_2, and z interferes with x at point p_3, then no more than two of these variables are simultaneously live, but no two of them can share a register.

Our solution is to copy every variable to a freshly named temporary at every program point. At point p_1 we will copy $x_2 \leftarrow x_1$ and $y_2 \leftarrow y_1$, at p_2 we copy $y_3 \leftarrow y_2 || \ z_3 \leftarrow z_2$, and so on. We assume the copies are done in parallel, so that y_2 interferes only with x_2 and not with x_1 or z_3. Then no temporary interferes with more than $K-1$ others, and the graph is colorable.

Whenever there is an edge from program point p_1 to p_2 such that the optimal-spill model has a `Copy` or `Copy2` relation, we also introduce a copy in the optimal-coalescing graph. That is, all the variables copied across an edge are formed into a

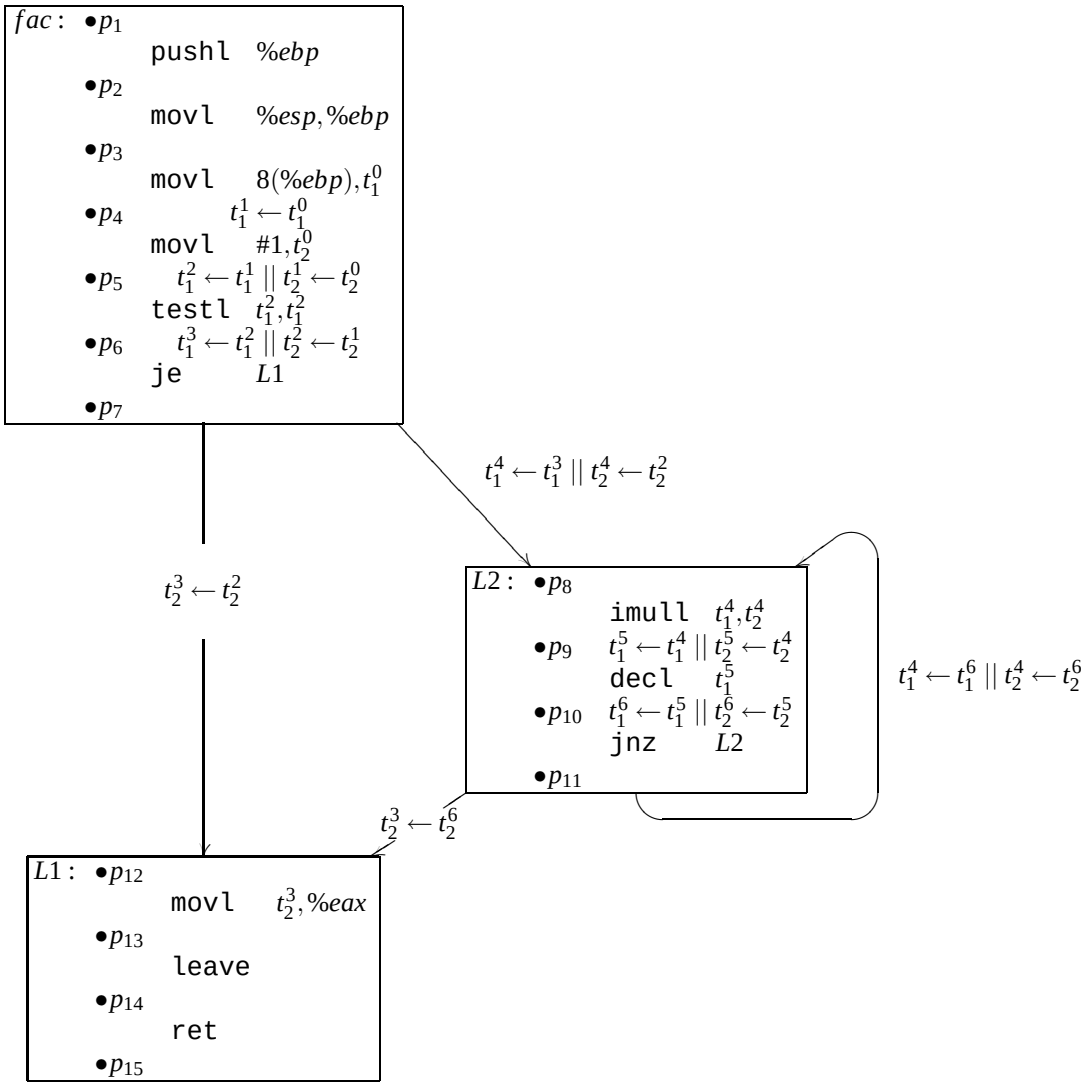

Figure 3: Flowgraph with internal splits

parallel copy that is meant to occur simultaneously with any other instruction executed at the edge. For edges that don't contain any "real" instruction, a new basic block must sometimes be introduced; this is called *edge splitting* and is common in register-allocation problems [App98, figs. 19.2–3].

After the graph is colored, each $K-way$ parallel copy must be implemented by a sequence of K register-register move instructions. If the parallel copy corresponds to a permutation with one or more cycles, then extra work (and extra storage) may be required to move a value out of the way and then move it back. Fortunately, the `xchg` (exchange two registers) on the IA-32 avoids the need for extra storage.

Because there are no more than K live variables at any time, and because a variable-span live at one time is never live at any other time (only *related* to other live ranges), the graph is trivially K-colorable. Any conflicts that arise at an instruction can be removed by an appropriate set of parallel copies before the instruction. That is, from the result of the spill phase, we can construct an interference graph in which every node[2] has degree less than K. Such a graph can be easily colored by Kempe's algorithm [Kem79] (rediscovered 102 years later by Chaitin [CAC+81]).

Having K "artificial" move instructions before every "natural" instruction would be expensive. Given a move instruction $u \leftarrow v$, if u and v can be colored the same – assigned to the same register – then the move can be deleted. The *register coalescing* problem is to find a coloring so that as many moves as possible have source and destination colored the same. When we formulate the coloring problem, we say that u and v are *move-related*.

Briggs [BCT94] developed an algorithm called *conservative coalescing* that K-colors a graph while attempting to color move-related nodes the same. Our *iterated register coalescing* algorithm [GA96] is an improvement to Briggs's algorithm; it is a greedy algorithm that will coalesce two nodes in the interference graph whenever it can be proved, using a simple heuristic, that the action will not render the graph non-K-colorable.

Iterated register coalescing is efficient (linear time) and in many applications (especially on underconstrained problems such as RISC machines with 32 registers) it produces excellent results. However, the our optimal spilling phase generates highly constrained problems, and the greedy algorithm yields poor solutions.

We therefore implemented an integer-linear-programming solution to the coloring problem, which we describe in the next section. It produces optimal (and in fact excellent) results, in a matter of only a few hours for tiny programs. It is useful primarily in establishing that coalescing problems generated by our spilling phase do in fact have good solutions. This means that we have not lost valuable information by splitting the problem into two phases. If good approximation algorithms can be found for the coalescing problem, then the whole approach will be effective and feasible.

The coloring/coalescing problem is significantly simpler than the problem handled by most graph-coloring register allocators, because the spills have already been identified and the graph is guaranteed K-colorable. Therefore it's worth stating exactly what

[2]The situation is more complicated for machines with instructions that both overwrite some of the input operands and generate new result operands. (Neither the IA-32 (Pentium), MIPS, Sparc, or Alpha have such instructions.) The interference graph after optimal spilling may have some nodes of degree $\geq K$, but these nodes won't have high-degree neighbors, so the graph will still be trivially colorable by Kempe's algorithm.

the algorithmic problem is.

Optimal register coalescing. Given an undirected graph of maximum degree $K-1$ (these are the *interference* edges), and an additional set of weighted edges (these are the *move* edges), find a K-coloring of the graph such that

1. No two nodes connected by an interference edge have the same color;
2. There is the lowest possible cost, where cost is the sum of the weights of those move edges whose endpoints are colored differently.

This problem is clearly NP-complete; it reduces the general graph-coloring problem (though we won't show the reduction here). However, it is entirely conceivable that good approximation algorithms can be found.

6 Optimal register coalescing via ILP

The linear program for graph coloring with register coalescing makes use of two variables:

$$x_{v,r} = \begin{cases} 1 & \text{if variable } v \text{ is in register } r \\ 0 & \text{otherwise} \end{cases}$$

where $r \in \mathbf{R}$ a set of physical registers on the machine, and

$$s_{c,r} = \begin{cases} 1 & \text{if copy } c \text{ is coalesced/subsumed[Cha82] using register } r \\ 0 & \text{otherwise} \end{cases}$$

The set `Interfere` contains the element (v_1, v_2) if there is an edge from v_1 to v_2 in the interference graph.

```
set R;
set Interfere ⊂ (V×V);
set Copy2  ⊂ (P×P×V×V) ;
var x {V, R} binary;
var s {Copy2, R} binary;
```

Specifiying the constraints for interference graph coloring and coalescing using the above could not be simpler.

Assign: For starters, only one register can be assigned to a variable:

$$\forall v \in \mathbf{V}.\ \sum_{r \in \mathbf{R}} x_{v,r} = 1$$

Interfere: If two variables interfere then they cannot be assigned to the same register:

$$\forall (v_1, v_2) \in \texttt{Interfere}. \forall r \in \mathbf{R}.\ (x_{v_1,r} + x_{v_2,r}) \leq 1;$$

Constrained: A copy is constrained if the source and destination interfere:

$$\forall c_{=(p_1,p_2,v_1,v_2)} \in \texttt{Copy2}.\ \forall r \in \mathbf{R}\ .$$
$$s.t.\ (v_1,v_2) \vee (v_2,v_1) \in \texttt{Interfere}.\ s_{c,r} = 0;$$

Coalesced: If a copy c is coalesced then both the source and destination of the copy must be in the same register. If v_1 and v_2 are the source and destination associated with the copy c, then for any register r then the following situations are permitted:

$x_{v_1,r}$	$x_{v_2,r}$	$s_{c,r}$
0	0	0
0	1	0
1	0	0
1	1	1

This relation can be expressed by $s_{c,r} = x_{v_1,r} \cdot x_{v_2,r}$, but that's a nonlinear relationship between the variables and is not permitted by the ILP solver. Instead, we take advantage of the fact that the variables are constrained to be 0 or 1, and use the following two linear inequalities:

$$x_{v_1,r} + x_{v_2,r} \geq 2s_{c,r}$$

$$x_{v_1,r} + x_{v_2,r} - 1 \leq 2s_{c,r}$$

That is, we express the coalescing constraints by the equations

$$\forall c_{=(p_1,p_2,v_1,v_2)} \in \texttt{Copy2}.\ \forall r \in \mathbf{R}.$$
$$(x_{v_1,r} + x_{v_2,r}) \geq 2 \cdot s_{c,r};$$

$$\forall c_{=(p_1,p_2,v_1,v_2)} \in \texttt{Copy2}.\ \forall r \in \mathbf{R}.$$
$$(x_{v_1,r} + x_{v_2,r} - 1) \leq 2 \cdot s_{c,r};$$

Cost function: The linear program is set to minimize the number of uncoalesced copies weighted by their execution frequency,

$$\sum_{c_{=(p_1,p_2,v_1,v_2)}} \texttt{weight}_{p_1} \cdot (1 - \sum_r s_{c,r})$$

Thus, the optimal coalescing problem can be expressed as an integer linear program. However, it does not seem as natural an expression as that of the optimal spilling problem: there are many more variables and we believe that fewer of the constraints can be solved by Gaussian elimination after each branch-and-bound step.

7 Benchmarks

We evaluate the method as follows:

- How costly is the optimal spilling algorithm?

- How many spills remain, compared to other algorithms?
- How costly is the optimal coalescing algorithm? How costly is the (greedy) iterated coalescing algorithm?
- How many moves remain, compared to other algorithms?
- How much suboptimality is caused by splitting the problem into two phases, spilling and coalescing?

7.1 Optimal Spilling

Figure 4 provides a graphical and tabular representation of the spill statistics. In the tabular form, the **Spill** and **Reload** columns shows the number of spill and reload instructions inserted into the program. In other words, these columns are a count of the number of memory loads and stores from spill instructions. Some spills and reloads can be combined with addressing modes, and the number of instructions affected is shown in the last column. No distinction is made between instructions that use memory as an operand and destination, and those that use memory for an operand only. In any case, we would expect the LP columns to be better in this regard as it is designed to reduce this cost. Each column shows the base SML/NJ compiler (version 110.23), and the same compiler modified to use our new algorithm for optimal spilling.

The base compiler uses static single-assignment (SSA) form, which divides each program variable into several temporaries based on the relation of definitions of the variable to the dominator tree of the program. Then a Chaitin-style spiller implements each temporary either entirely in registers or entirely in memory. Briggs [Bri92] conjectured that SSA was the best way to split the variables prior to coloring with coalescing. Our current paper can be viewed as a test of his conjecture; we have described an entirely different method for splitting the variables.

A characteristic of SSA form is that there will typically be one spill and multiple reloads for any temporary that is spilled. The number of spill and reloads from the base compiler is 30% higher than the LP version. In particular the number of spills in the LP version is higher than the base compiler. This can easily be explained as the LP model is splitting a live range into multiple parts, some subset of which are implemented in registers and the others in memory. In other words, there is only one transition from register to memory in the base compiler, but multiple transitions in the LP model.

A different story applies to the **Reload** column. The LP column reloads less than half as many variables as the base compiler, as the LP model effectively keeps active temporaries in registers.

The **Memory instructions** column again demonstrates that the optimal-spill has made much better use of external registers.

7.2 Optimal-spill performance

Figures 5, 6, and 7 shows the size of the AMPL model and the speed of generating an optimal solution. Each dot in these figures represents a cluster, and each benchmark is made up of multiple clusters. A cluster is a call graph in which every function in

	Spills		Reloads		Memory Instructions	
	Base	LP	Base	LP	Base	LP
barnes-hut	131	200	319	186	550	269
boyer	301	189	156	103	211	108
count-graphs	52	122	184	110	446	195
knuth-bendix	141	194	423	208	769	322
lexgen	292	347	525	277	961	332
life	41	70	90	57	180	62
logic	73	103	270	109	509	347
mandelbrot	10	11	38	16	67	32
mlyacc	752	1001	2006	1017	3605	1623
ray	91	134	204	103	428	240
simple	375	671	729	241	1295	480
tsp	45	94	92	53	224	76
Total	2304	3136	5036	2480	9245	4086

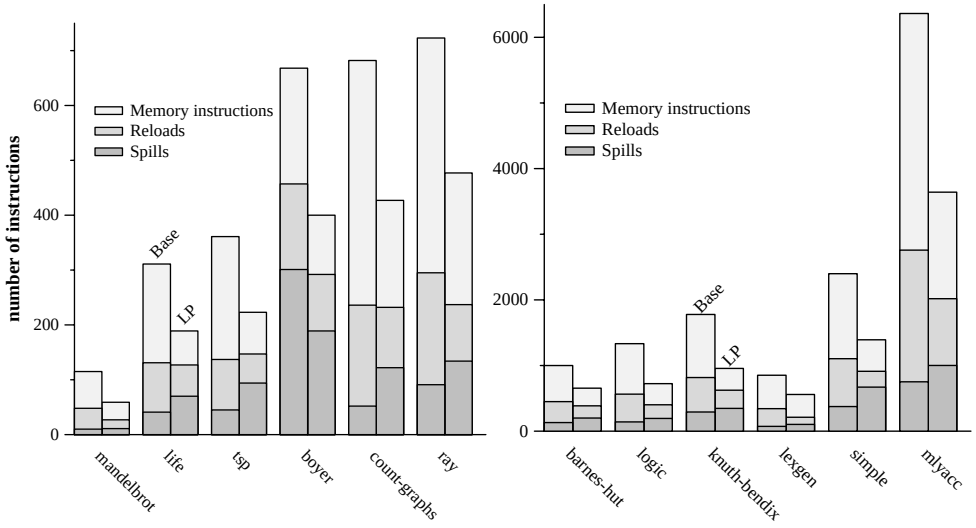

Figure 4: Comparison of spill statistics for SML/NJ v110.23 (Base) using previous algorithm (SSA splitting and iterated register coalescing) and same compiler based on optimal spilling via integer linear programming (LP)

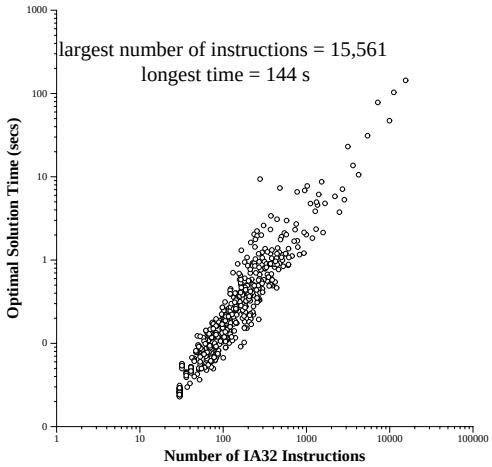

Figure 5: Solve time versus program points

Figure 6: Constraints generated versus program points

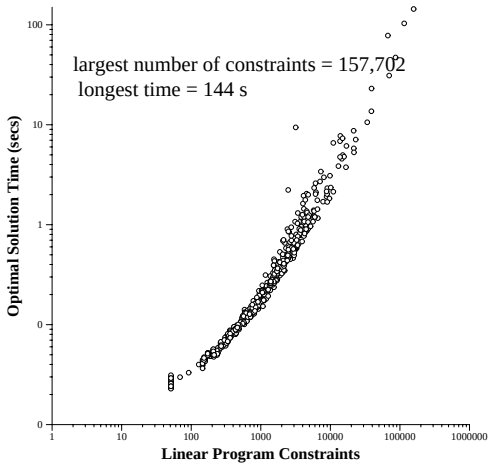

Figure 7: Solve time versus number of constraints

Benchmark	**Base**	**LP**	**Speedup**
Mandelbrot	28.09s	23.99s	17%
Life	22.01s	16.54s	33%
Barnes-hut	4.21s	3.85s	9%
Geometric mean of speedup			*19%*

Figure 8: Execution speed

the graph has at least one call-edge with another function in the graph. Since this is a continuation passing style (CPS) compiler, there are usually a large number of clusters for each benchmark.

The most important result is Figure 5 which is the time to solve the model as a function of the number of instructions. Two minutes was the longest time taken on a 150MHz, SGI, MIPS processor, with most clusters being solved within 10 seconds. The complexity is close to linear ($O(n^{1.3})$ – taking the least square fit), and is significantly better than the $O(n^{2.5})$ reported by Goodwin and Wilken[GW96] for general purpose processors. In all fairness, Goodwin and Wilken are solving the entire register allocation problem for an architecture with many more registers. Kong and Wilken[KW98] get much better performance but do not report any empirical complexity result, but the same caveat applies as they also solve the complete register allocation problem. The number of constraints also grows almost linearly with the program size ($O(n^{1.3})$) which is significantly better than the models solved by Wilken et al.[GW96, KW98].

7.3 Register Allocation

The AMPL model described in Section 6 was not viable in practice. A tiny function consisting of about 100 instructions (generated from an unrolled version of factorial) took in excess of two hours to solve, however, the resulting program had practically no uncoalesced moves at all.

In order to get practical validation we used the following approach for each cluster:

1. Perform iterated register coalescing as described by us[GA96], but record the number of freezes required to completely color the color.
2. Repeat iterated register coalescing as above, but perform only a fraction of the total number of freezes required.
3. If the resulting interference graph is of reasonable size and a small number of freezes were used, then feed the resulting interference graph to the ILP solver as described in Section 6.
4. Otherwise, abort and use the base compiler (SML/NJ v110.23) for this cluster.

Step 2 is performed to simplify the interference graph as much as possible so that the generated model could be solved in a reasonable amount of time. In some cases the clusters were so big that a large number of freezes were required before a reasonable sized model was obtained. In many cases such clusters would be converted to go through the regular compiler phases and not through the AMPL model.

It must be emphasized that this approach is highly unsatisfactory. The freezes done in the iterated register allocator may be at the cost of constraining more expensive splits that eventually remain in the program — note that the conservative coalescing heuristic may have been too weak to coalesce the expensive move to begin with. Despite these caveats we were able to get three benchmarks (the smallest three) to go through, and Figure 8 shows the results.

On this admittedly inadequate set of benchmarks, optimal spilling followed by optimal coalescing generates code that is about 19% faster than the code generated from SSA-based splitting followed by iterated register coalescing.

8 Algorithmic challenge

The algorithmic challenge is to find a better algorithm for the optimal coalescing problem, as described at the end of Section 5. We invite others to take up the challenge.

We have provided a web site

`http:www.cs.princeton.edu/~appel/coalesce`

containing a database of problem instances, suboptimal solutions as found by iterated register coalescing[GA96], and a simple `check.c` program that validates and evaluates solutions.

A technical note about how the problem instances were generated. The interference graphs naturally generated by our optimal spilling algorithm may have some nodes of $\geq K$. In general, such nodes will not have high-degree neighbors, so the graphs are naturally K-colorable by Kempe's algorithm; therefore, the high-degree nodes are not likely to be a problem for any coalescing algorithm. But just to make the input data more regular for those who would take up our challenge, we have transformed the graphs to have maximum degree $K-1$ by the following procedure:

- Assume v_0^1, v_1^1, $\cdots$, v_n^1 are live and passed unchanged between point p_1 and p_2.
- If a new temporary is not defined by the instruction between p_1 and p_2, then assuming the instruction has the form $v_0^1 := v_0^1 \oplus v_1^1$, the interference graph is generated assuming the following instructions between p_1 and p_2:

  $$\bullet\ p_1$$

  $$v_0^2 \leftarrow v_0^1 \ \|\ v_1^2 \leftarrow v_1^1 \ \|\ \cdots\ v_n^2 \leftarrow v_n^1$$
  $$v_0^2 := v_0^2 \oplus v_1^2$$

  $$\bullet\ p_2$$

  All the fall-through temporaries (no more than $K-1$) are copied into fresh temporaries, and the rewritten instruction is executed in sequence.

- If a new temporary or temporaries are defined, then assuming the instruction has the form $v_m^1 := v_0^1 \oplus v_1^1$ the interference graph is generated assuming the following instruction between p_1 and p_2:

  $$\bullet\ p_1$$

  $$v_0^2 \leftarrow v_0^1 \ \|\ v_1^2 \leftarrow v_1^1 \ \|\ \cdots\ v_n^2 \leftarrow v_n^1 \ \|\ v_m^2 := v_0^1 \oplus v_1^1$$

  $$\bullet\ p_2$$

  That is to say the fall-through variables and instruction are executed in parallel. It is necessary to do it this way because if one of the fall-through variables v_0^1 used in the instruction should go dead at p_2, then the new temporary v_m^2 will not interfere with it.

9 Conclusions

We have formulated the register allocation problem for CISC architectures with few registers to one involving optimal placement of spill code, followed by optimal register coalescing. We have shown empirically that dividing the problem into these two phases does not significantly worsen the overall quality of the solution. We have demonstrated

an efficient algorithm using integer linear programming for optimal spill-code placement. The optimal coalescing has a significantly simpler structure than the general register-allocation problem, as the spilling has already been taken care of. We have shown an inefficient algorithm for optimal coalescing, and an efficient algorithm for suboptimal coalescing.

Programs compiled with optimal spilling followed by optimal coalescing run about 19% faster than when compiled with SSA-based splitting followed by iterated register coalescing (though this number is based on an inadequate set of small programs). This refutes a conjecture by Briggs [Bri92] that the splits induced by SSA would be appropriate for register allocation and spilling.

An efficient algorithm for optimal coalescing is left as an open problem.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, Oct 1990.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Proceeding of the ACM SIGPLAN '82 Symposium on Compiler Construction.

[CPL00] CPLEX mixed integer solver. www.cplex.com, 2000.

[FGK93] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, South San Francisco, CA, 1993. www.ampl.com.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 208–218, New York, Jan 1996. ACM Press.

[GGR94] Lal George, Florent Guillame, and John Reppy. *A portable and optimizing back end for the SML/NJ compiler*, volume 786 of *LNCS*, pages 83–97. Springer-Verlag, 1994.

[Goo96] David William Goodwin. *Optimal and Near-Optimal Global Register Allocation*. PhD thesis, University of California at Davis, 1996.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software—Practice and Experience*, 26(8):929–965, 1996.

[Hun93] Ming S. Hung. *Optimization with IBM-OSL*. Scientific Press, South San Francisco, CA, 1993.

[Kem79] A. B. Kempe. On the geographical problem of the four colors. *American Journal of Mathematics*, 2:193–200, 1879.

[KW98] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *31st International Microarchitecture Conference*. ACM, December 1998.

[LGAT97] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. In *Languages and Compilers for Parallel Computing*, pages 246–265. Springer Verlag, LNCS 1239, August 1997.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986.

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *27th IEEE/ACM International Symposium on Microarchitecture (MICRO-27)*, November 1994.

A Appendix: AMPL model for spilling

```
model;
set Vars;
set Pts;

set Exists within (Pts cross Vars);

set Binary within (Pts cross Pts cross Vars cross Vars);
set Unary  within (Pts cross Pts cross Vars cross Vars);
set Copy   within (Pts cross Pts cross Vars);
set Copy2   within (Pts cross Pts cross Vars cross Vars);
set Nullary within (Pts cross Pts cross Vars);
set NullaryReg within (Pts cross Pts cross Vars);
set Mutate within (Pts cross Pts cross Vars);
set MutateReg within (Pts cross Pts cross Vars);
set UseUp2 within (Pts cross Pts cross Vars cross Vars);
set UseUp   within (Pts cross Pts cross Vars);
set UseUpReg within (Pts cross Pts cross Vars);

param weight Pts;
param K, loadCost, storeCost, moveCost, instrCost;

var inReg Exists binary;
var inMem Exists binary;
var load  Exists binary;
var store Exists binary;

subject to EXISTS  (p,v) in Exists:
  load[p,v] + inReg[p,v] + store[p,v] + inMem[p,v] = 1;

subject to BRANCH  (p,v) in Exists : p in Branch:
    load[p,v] + store[p,v] = 0;

subject to REGISTERS_K1  p in Pts:
   sum (p,v) in Exists (inReg[p,v]+store[p,v]) <= K;

subject to REGISTERS_K2  p in Pts:
   sum (p,v) in Exists (inReg[p,v]+load[p,v]) <= K;

subject to COPY_PROPAGATE   (p1,p2,v) in Copy:
   load[p1,v] + inReg[p1,v] = store[p2,v] + inReg[p2,v];

subject to COPY2_PROPAGATE  (p1,p2,v1,v2) in Copy2:
 load[p1,v1] + inReg[p1,v1] = store[p2,v2] + inReg[p2,v2];

subject to BINARY_IN_REG  (p1,p2,s,d) in Binary:
  load[p1,s] + inReg[p1,s] + load[p1,d] + inReg[p1,d] >= 1;

subject to BINARY_PROPDST (p1,p2,s,d) in Binary:
   load[p1,d] + inReg[p1,d] = store[p2,d] + inReg[p2,d];

subject to NULLARY_REG   (p1,p2,v) in NullaryReg:
   store[p2,v] + inReg[p2,v] = 1;
```

```
subject to USEUP2  (p1, p2, s1, s2) in UseUp2:
 load[p1,s1] + inReg[p1,s1] + load[p1,s2] + inReg[p1,s2] >= 1;

subject to USEUP_IN_REG1   (p1,p2,v) in UseUpReg:
   load[p1,v] + inReg[p1,v] = 1;

subject to MUTATE_PROPDST   (p1,p2,v) in Mutate:
    load[p1,v] + inReg[p1,v] = store[p2,v] + inReg[p2,v];

subject to MUTATE_REG1    (p1,p2,d) in MutateReg:
    load[p1,d] + inReg[p1,d] = 1;

subject to MUTATE_REG2    (p1,p2,d) in MutateReg:
     store[p2,d] + inReg[p2,d] = 1;

subject to UNARY_BINARY_IN_REG   (p1,p2,s,d) in Unary:
  load[p1,s] + inReg[p1,s] + store[p2,d] + inReg[p2,d] >= 1;
```

```
minimize COST:
   (sum v in Vars, p in Pts: (p,v) in Exists
        weight[p] *
          (  loadt[p,v] * (loadCost + 3 * instrCost) +
             load[p,v]  * (loadCost + 3 * instrCost) +
             store[p,v] * (storeCost + 3 * instrCost)))
 + (sum (p1,p2,src,dst) in Binary
        weight[p1] *
         ( (1 - (inReg[p1,src] + load[p1,src] + loadt[p1,src])) *
                        (loadCost + instrCost) +
           (1 - (inReg[p2,dst] + store[p2,dst])) *
                        (loadCost + storeCost + instrCost)))
 + (sum (p1,p2,src,dst) in Copy2
        weight[p1] *
         ((1 - (inReg[p1,src] + load[p1,src] + loadt[p1,src])) *
                        (loadCost + instrCost) +
          (1 - (inReg[p2,dst] + store[p2,dst])) *
                         (storeCost + instrCost)))
 + (sum (p1,p2,src,dst) in Unary
        weight[p1] *
         ( (1 - (inReg[p1,src] + load[p1,src] + loadt[p1,src])) *
                     0.8 * (loadCost + instrCost) +
           (1 - (inReg[p2,dst] + store[p2,dst])) *
                        (storeCost + instrCost)))
 + (sum (p1,p2,dst) in Mutate
        weight[p1] *
         ( (1 - (inReg[p2,dst] + store[p2,dst])) *
                        (loadCost + storeCost + instrCost)))
 + (sum (p1,p2,dst) in Nullary
        weight[p1] *
         ( (1 - (inReg[p2,dst] + store[p2,dst])) *
                        (storeCost + instrCost)))
 + (sum (p1,p2,src) in UseUp
        weight[p1] *
         ( (1 - (inReg[p1,src] + load[p1,src] + loadt[p1,src])) *
                        (loadCost + instrCost)))
 + (sum (p1,p2,src1,src2) in UseUp2
        weight[p1] *
         ( (1 - (inReg[p1,src1] + load[p1,src1] + loadt[p1,src1]))*
                        (loadCost + instrCost)+
           (1 - (inReg[p1,src2] + load[p1,src2] + loadt[p1,src2])) *
                        (loadCost + instrCost)));
```