Optimal Spilling for CISC Machines with Few Registers

Andrew W. Appel
Princeton University
appel@cs.princeton.edu

Lal George
Lucent Technologies
Bell Laboratories
george@research.bell-labs.com

ABSTRACT

Many graph-coloring register-allocation algorithms don't work well for machines with few registers. Heuristics for live-range splitting are complex or suboptimal; heuristics for register assignment rarely factor the presence of fancy addressing modes; these problems are more severe the fewer registers there are to work with. We show how to optimally split live ranges and optimally use addressing modes, where the optimality condition measures dynamically weighted loads and stores but not register-register moves. Our algorithm uses integer linear programming but is much more efficient than previous ILP-based approaches to register allocation. We then show a variant of Park and Moon's optimistic coalescing algorithm that does a very good (though not provably optimal) job of removing the register-register moves. The result is Pentium code that is 9.5% faster than code generated by SSA-based splitting with iterated register coalescing.

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. On our test suite of 600 basic-block clusters comprising 163,355 instructions, iterated register coalescing produces 84 spill instructions for a 32-register machine, but 22,123 spill instructions for an 8-register machine. This is about 14% of all instructions, which is worth the trouble to improve.

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 [11, 12]. 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 splits and 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 in a way that leaves as few as possible register-register moves in the program. This is difficult to do optimally, but we will show an efficient algorithm can get very good results.

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

1. When we decompose the problem into two subproblems (spilling and 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. For the entire class of allocators that do not use rematerialization, and keeps no more than one copy of each variable at a time, our algorithm provably generates the least number of (weighted) loads, stores, and memory-operand instructions. Rematerialization can be easily incorporated into our model, but we have not yet done so; variables that live in several locations at once require further research -- our initial attempts produce integer linear programs that are too costly to solve.

3. Can the coloring subproblem be solved optimally and efficiently? We can do it optimally but far too slowly using integer programming; we can do it quickly and adequately (though suboptimally) using *optimistic coalescing*.

2. OPTIMAL SPILLING VIA ILP

We model the register-spilling problem as a 0-1 linear program: 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 [8] to describe and generate the linear program, and CPLEX [7] to solve it. The AMPL compiler derives an instance of the optimization problem by instantiating a mathematical model with problem-specific data, and feeds the resulting linear program (in a suitable form) to a standard off-the-shelf simplex solver such as CPLEX.

The AMPL model consists of variable, set, and parameter declarations, and templates to generate the constraints for the linear program. The sets, in their simplest form, are a symbolic enumeration and declared in the model using a declaration similar to:

```
set T;
set R;
```

Sets may also be built from cartesian products of other sets. Variables are usually indexed over sets, so a declaration such as:

```
var x {T,R};
```

defines a set of variables $x_{i,j}$ where i ranges over T and j over R. Parameter declarations inject concrete values into the model, so a declaration such as:

```
param cost {T};
```

defines a parameter cost that is indexed over elements in the set T. The equations are generated from templates and are derived from
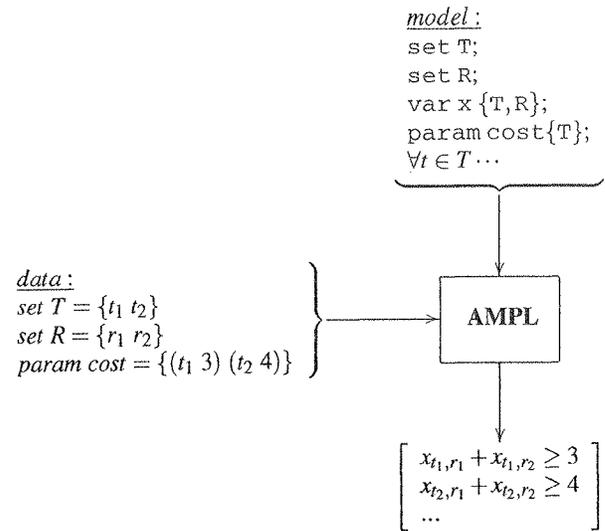

Figure 1: AMPL modeling system

logical connections among the sets. For example:

$$\forall t \in T \, . \, \sum_{r \in R} x_{t,r} \geq cost[t]$$

If $T = \{t_1 \ t_2\}$ and $R = \{r_1 \ r_2\}$ then, the template above will generate two equations, one for each member of T:

$$x_{t_1,r_1} + x_{t_1,r_2} \geq cost[t_1]$$
$$x_{t_2,r_1} + x_{t_2,r_2} \geq cost[t_2]$$

This AMPL example is illustrated in Figure 1 which shows the model, data, and system of linear equations that is generated.

Set Declarations: The description of our ILP formulation of optimal spilling begins with the various set declarations required to characterize the input flowgraph 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:

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

The remaining data declarations deal with liveness properties and a characterization of the type of IA-32 instructions between two points. 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 ($d \leftarrow f(s)$), for example. If there is an add instruction $v_2 \leftarrow v_2 + v_1$ 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 C (P×P×V×V) ;
set Unary C (P×P×V×V) ;
```

---

[1]AMPL actually uses the word cross instead of the symbol ×, and within instead of C. The actual AMPL code is shown in the appendix.

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 will sometimes refer to specific hardware 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 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 address (%esp+4) 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.

```
set Nullary ⊂ (P × P × V) ;
set NullaryReg ⊂ (P × P × V) ;
```

Some instructions accomplish $v \leftarrow f(v)$, 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:

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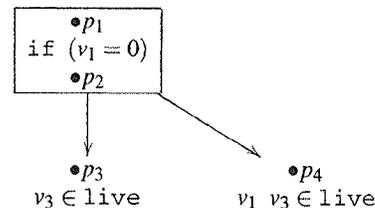
For cases where no results are produced, the instruction may take two operands of which at most one can be in memory (e.g., the *compare* instruction); or take one operand which can be either a register or memory (e.g. addl(%v, %eax)); or take one operand that must be in a register. We call these three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

```
set UseUp2 ⊂ (P × P × V × V) ;
set UseUp ⊂ (P × P × V) ;
set UseUpReg ⊂ (P × P × V) ;
```

If there is a branch instruction between p_1 and p_2, then it is necessary to know about points such as p_2, associated with a branch, as we cannot insert spill or reload instructions at p_2. We therefore declare a set of branching points:

```
set Branch ⊂ P
```

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. Suppose v_3 is live throughout, and v_1 is live only along the successor containing p_4.

It is necessary to propagate this liveness information along the edges of the branch, and we represent this by generating:

$$(p_1, p_2, v_1) \quad \in \text{UseUp};$$

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

Note that v_1 is used and propagated between the points p_1 and p_2, and the other variables are propagated along the appropriate branch edges.

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:

$$(p_1, p_2, x) \in \text{UseUpReg}$$
$$(p_1, p_2, v) \in \text{UseUpReg}$$

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

$$(p_1, p_2, v) \in \text{MutateReg}$$
$$(p_1, p_2, x) \in \text{UseUpReg}$$

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

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

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.

Parameter Declarations: 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. We obtain the frequencies by static estimation from branch predictions, propagated using Kirchoff's laws as described by Wu and Larus [18]; better frequencies could be obtained by dynamic profiling. In our model we have:

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

to associate the frequency of execution with each point.

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

```
fac:    pushl   %ebp            ;; save 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 2: Intel IA-32 instructions for the factorial function

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

where K[p] is the number of available registers at point p.

Finally we have some scalar cost parameters:

```
param  Cload, Cstore, Cmove, Cinstr
```

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, $C_{load} > C_{store} > C_{move} > C_{instr}$. (In fact, C_{instr} really measures the cost of a slight extra pressure on the instruction cache.)

Example. Figure 2 shows the Intel IA-32 instructions that may be generated for the factorial function, and Figure 3 shows the corresponding flowgraph annotated with points surrounding each instruction. The AMPL sets generated are:

```
set P := {p₁ p₂ p₃ ... p₁₄ p₁₅}
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, while the source operand can be in either class. Therefore, imull is classified as `MutateReg` for the destination operand and `UseUp` for the source operand.

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.

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
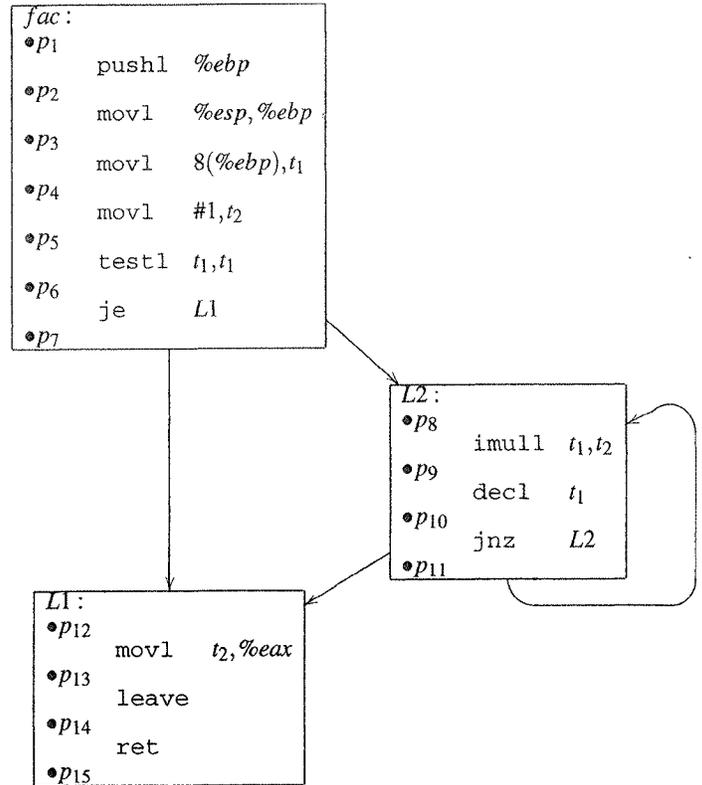

Figure 3: Flowgraph annotated with points

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 – $r_{p,v}$,

- arrive in memory and depart in memory – $m_{p,v}$,

- arrive in a register and depart in memory – $s_{p,v}$ (for *stored*),

- or arrive in memory and depart in a register – $l_{p,v}$ (for *loaded*).

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:

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

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 are already in registers and those that are to be spilled must be no more than the number of registers available for coloring at p.

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

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 ($s_{p_2,v} = 1$):

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

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:

$$\text{minimize COST}:$$
$$(\Sigma_{(p,v)\in\texttt{Exists}}$$
$$\text{weight}_p((C_{\text{load}} + 3C_{\text{instr}})l_{p,v} +$$
$$(C_{\text{store}} + 3C_{\text{instr}})s_{p,v}))$$
$$+ \ \cdots$$

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

$$+(\Sigma_{(p_1,p_2,v_1,v_2)\in\texttt{Binary}}$$
$$\text{weight}_{p_1}((C_{\text{load}} + C_{\text{instr}})(m_{p_1,s} + s_{p_1,v_1})$$
$$+(C_{\text{load}} + C_{\text{store}} + C_{\text{instr}})(m_{p_2,v_2} + l_{p_2,v_2})))$$
$$+ \ \cdots$$

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

3.3 Temporary loads

When we execute a load instruction to bring a value from memory to registers, the value becomes accessible from both places, and similarly when we store from registers to memory. The model we have described does not account for this fact; it acts as if a value lives only in one place at a time. We constructed a more ambitious model that accurately accounts for values that continue to live in both memory and registers after a load or store, but we had little success with it: the equations seem to be sufficiently underconstrained that the integer LP solvers do enormous amounts of branch-and-bound search. Therefore we use the model that assumes that each value lives in one place (memory or registers) at a time. Our spilling is optimal only with respect to this model.

However, we were able to incorporate one useful special case into our model. A variable can be loaded from (a spill location in) memory to a register for use in the very next instruction, with the assumption that the register is then dead and the memory value lives on. We have not described this mathematically in the body of the paper, but our implemented AMPL model includes this feature.

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

4. SOLVING THE MODEL.

Our compiler [2][10] 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 3 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 $(p_4, t_1) \in \texttt{Exists}$. 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 or noncommercial LP solvers. Some of these solvers can solve 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 [7] and IBM's OSL [13]; 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. REGISTER COALESCING

The resulting flowgraph has no more than K variables simultaneously live at any point, but it may still be the case that there is no K-coloring of the variables – that K registers do not suffice. If x_1 interferes with y_1 at point p_1, y_1 interferes with z_1 at point p_2, and z_1 interferes with x_1 at point p_3, then even though there are only two temporaries live at any time, there is no 2-coloring of the interference graph.

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 $y_3 \leftarrow y_2$ and $z_3 \leftarrow z_2$, 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 each parallel copy moves at most K variables, and each temporary interferes with no 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 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 [1, figs. 19.2–3]. The resulting flowgraph for the example in Figure 2 is shown in Figure 4.

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) instruction on the IA-32 avoids the need
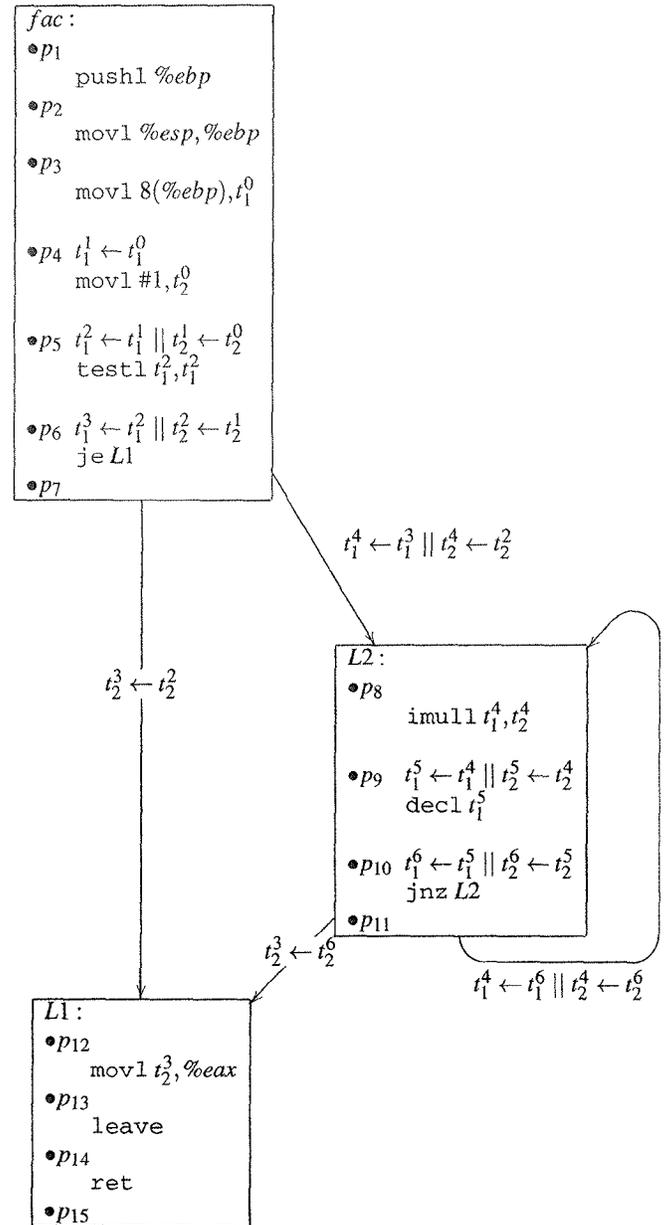

Figure 4: Flowgraph with internal splits

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 [14] (rediscov-

---

[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.

ered 102 years later by Chaitin [5]).

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*.

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 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).

6. ALGORITHMS FOR COALESCING
We have tried three approaches to the coalescing problem: iterated register coalescing [9], integer linear programming, and optimistic coalescing [17]. The first two don't work: iterated coalescing is fast but too conservative for the highly constrained problems that result from our optimal spiller, and our integer programs produce optimal solutions but not in a reasonable amount of time.

Optimistic coalescing. Our third approach is based on Park and Moon's optimistic coalescing [17] and works as follows: We perform aggressive coalescing (á là Chaitin), which may overconstrain the graph so that it becomes uncolorable. We do this coalescing in priority order, so that the expensive moves get coalesced first. Of course, we do not coalesce nodes that interfere – hence the need for priorities.

We then do a Briggs-style [4] optimistic coloring: that is, we remove nodes of degree $< K$ and push them on a stack. When the graph contains only nodes of degree $\geq K$, we select a *spill candidate* using Chaitin's heuristics and remove it from the graph, pushing it on the stack. Briggs called this optimistic because there is always the chance that in the stack-popping (coloring) phase, several neighbors of the spill candidate will be colored the same, so that a color is available.

If no color is available, Briggs would spill the node. Park and Moon point out that we can instead undo the coalescing that caused this node to have high degree. We go even further: in our context, because we start with a graph where *all* nodes have low degree, we know that it will *always* be possible to undo the coalescing of a spill candidate and color the nodes individually.

However, we don't always need to undo this coalescing all the way. We first split the spill candidate into its constituent primitive nodes. Then we reconsider each move instruction, and coalesce it if the resulting node is colorable in the current context.

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 optimistic coalescing algorithm? We will not even perform measurements to answer this question; the algorithm is clearly linear-time (for any given K), and should be about as fast as Briggs's algorithm, which is known to be very efficient when implemented carefully.

- How many moves remain, compared to other algorithms?

It would also be interesting to know how much suboptimality is caused by splitting the problem into two phases, spilling and coalescing. Answering this question would require an optimal algorithm for coalescing; although we have implemented one using integer programming, it blows up on any but the tiniest examples.

We measure the performance of our algorithms on the Standard ML of New Jersey benchmark suite. These are not microbenchmarks; the mean size of the 14 programs is 10,117 instructions of generated Pentium code (exclusive of spill-related instructions).

7.1 Optimal Spilling
Figure 5 shows the spill statistics. The first column ("Base") in each pair shows the number of spills produced by iterated register coalescing on the Pentium (6 general-purpose registers and 2 conventionally reserved); this is the measured performance of SML/NJ version 110.23. The second column ("Opt") shows the performance of our new algorithm – optimal spilling with optimistic coalescing – on the Pentium.

Within each column, the **Spills** and **Reloads** sections show the number of spill and reload instructions inserted into the program. In other words, these subcolumns 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 top subcolumn. No distinction is made between instructions that use memory as a source operand and a destination, and those that use memory for a source only. The height of each column is $100 \cdot \text{spills}/(\text{spills} + \text{nonspills})$. Figure 6 aggregates the values across the columns.

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 [3] 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
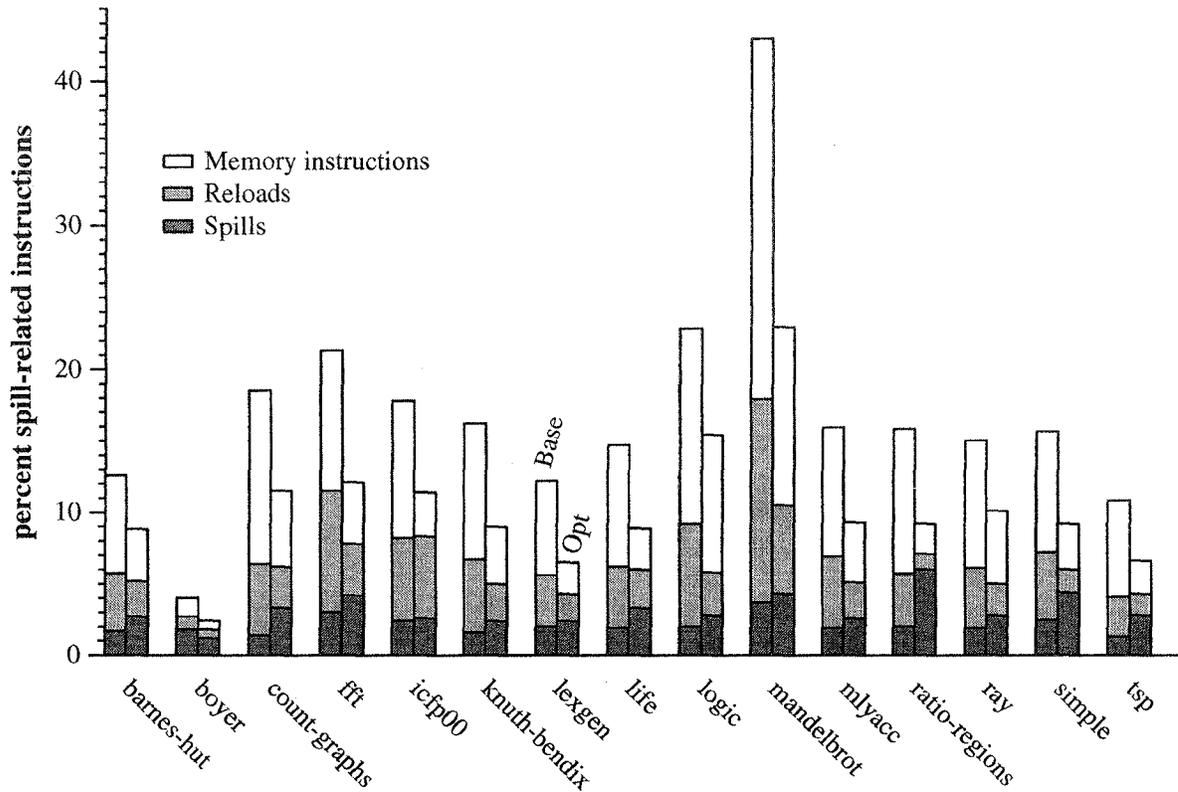
249

Figure 5: Comparison of static 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 (Opt)

	Spills		Reloads		Memory Instructions	
	Base	Opt	Base	Opt	Base	Opt
Total	3040	4310	6771	3804	12312	5009

Figure 6: Aggregate values of spill statistics from Figure 5

of spill and reloads from the base compiler is 21% higher than the Opt version, however the number of spills in the Opt version is higher than the base compiler. This can easily be explained as the ILP 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 ILP model.

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

The **Memory instructions** column demonstrates that the optimal spilling has done a significantly better job at keeping temporaries in registers.

7.2 Optimal-spill performance

Figure 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 the graph has at

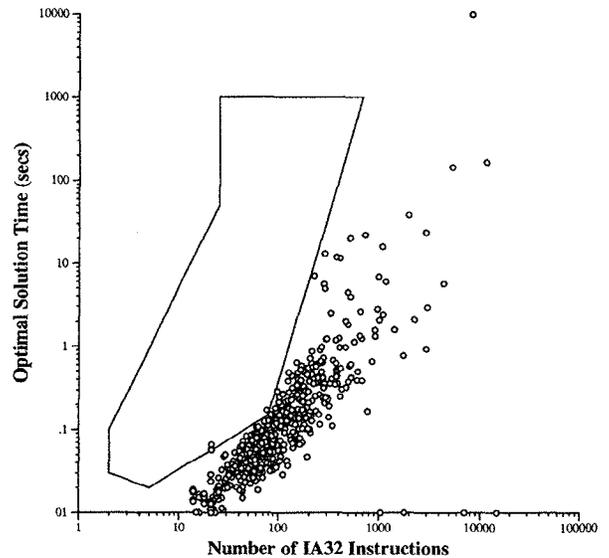**Figure 7: Solve time versus program points.**
The circles show the performance of our algorithm; the polygon shows the approximate performance of Goodwin's algorithm (as reported by him [11]) on a different data set.

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. Superimposed on

250

	Splits		Non-splits	Instructions
	Base	Opt	Opt	Per Split
barnes-hut	391	326	7430	23
boyer	489	254	16495	65
count-graphs	223	215	3705	17
fft	145	212	3669	17
icfp00	1413	1008	19332	19
knuth-bendix	776	648	7912	12
lexgen	1767	1352	14543	11
life	230	203	2118	10
logic	201	163	3653	22
mandelbrot	26	16	262	16
mlyacc	3184	2559	39267	15
ray	403	311	4735	15
simple	1288	930	15133	16
tsp	292	291	3395	12
Geometric Mean				17

Figure 8: Number of splits and instructions

Benchmark	Base	Opt	Speedup
barnes-hut	2.92	2.92	0.0%
boyer	12.57	12.49	0.0
mlyacc	9.14	9.11	0.0
tsp	6.92	6.77	2.2
lexgen	9.08	8.84	2.7
count-graphs	24.07	22.15	8.7
icfp00	109.29	99.72	9.6
fft	8.58	7.80	10.0
logic	5.10	4.61	10.6
knuth-bendix	8.08	7.22	11.9
mandelbrot	27.92	23.21	20.3
life	19.03	15.24	24.9
simple	31.53	25.12	25.5

Figure 9: Execution speed

scatter plot is a crude bounding box obtained from Figures 3.4 in Goodwin's thesis [11] .[3]

The most important result from Figure 7 is that every block-cluster of fewer than 5000 instructions can be solved within 30 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 [12] for general-purpose processors. Goodwin and Wilken's performance is so much worse because they compute optimal coloring via ILP, whereas we compute only optimal register pressure (assuming that the cost of coalescing will be insignificant). Kong and Wilken [15] get much better performance (though they do not report any empirical complexity result), and they also solve the whole register allocation problem. Our number of constraints grows almost linearly with the program size ($O(n^{1.3})$) which is significantly better than the models solved by Wilken et al. [12, 15].

7.3 Register Allocation

Figure 8 shows the number of splits (uncoalesced moves) remaining in the SSA-based compiler with iterated register coalescing (Base), and the number remaining using ILP and optimistic coalescing (Opt). The third column is the number of non-split instructions in the Opt compiler; the corresponding column for the Base compiler should be similar. ILP with optimistic coalescing produces programs in which 1 in 17 instructions are moves, and the static number of splits in all but one benchmark is better than our SSA-based splitting with iterated register coalescing. We don't know how many of these are required by the two-address nature of the instruction set or by other constraints – that is, we don't know how many moves an optimal coalescer would leave. However, we have measured the overall performance of several standard ML benchmarks using our old algorithm (SSA-based splitting with iterated register coalescing [9]) and our new one (optimal spilling with optimistic coalescing). The results (in Figure 9) show a speedup of 9.5% improvement in execution speed (taking the geometric mean of ratios). Some of the benchmarks have a significant improvement in static spills (Figure 5) but no speedup; perhaps this is because we weight the spill costs by static estimation, and perhaps dynamic

---

[3]We believe the machine on which Goodwin got his results (HP9000/770) is about as fast as our machine (SGI Origin 2100 @ 250 MHz).

Iterated Register Coalescing		Optimal Spilling, Optimistic Coalescing	
Liveness and		Liveness	1.3*
Interference graph	8.19*	I/O to AMPL	82.7*
Iter. reg. coalesce	27.11*	AMPL	419.2†
		CPLEX	594.1†
		I/O from AMPL	30.7*
Insert load/stores	9.27*	Insert load/stores	1.8*
		Liveness	1.2*
		Interference graph	30.6*
		Optimistic coalesce	360.4*
TOTAL (seconds)	44.57	TOTAL (seconds)	1522.0

Figure 10: Time spent in the register allocators
*433 MHz Pentium II   †250 MHz SGI Origin 2100

profiling would significantly improve the performance of the optimal spiller.

Figure 10 compares the compile-time cost of the old (iterated register coalescing) algorithm to the new one, totalled over register-allocating all but one of the benchmark programs (153,836 machine instructions). We have omitted the *ratio-regions* benchmark – which produces the uppermost point of Figure 7 – but with it the total times would be 57 and 11306 seconds, respectively. The new allocator is only a prototype, however, and is not highly engineered for efficiency.

8. RELATED WORK

Goodwin and Wilken [12] address several optimization problems such as live range splitting, register assignment, spill placement, rematerialization, callee/caller-save register management, and copy elimination, within the single framework of 0-1 integer linear programming. They do not handle CISC instruction selection, though our new result implies that instruction selection could be incorporated into their framework.

Our optimal spilling algorithm can handle problem sizes at least an order of magnitude larger than theirs, as figure 7 shows. We believe this is an important benefit of separating spilling from coloring.

Goodwin and Wilken's algorithm has a different (and incomparable) optimality guarantee than ours. They guarantee an optimal set of spills and register-register moves, given a predetermined set of

potential split points. We guarantee an optimal set of spills (but not optimal moves) over all possible split points. In principle, one could run their algorithm on an input that specifies a split point at every possible place, but we believe the resulting problem size would swamp their algorithm in practice.

Each optimization performed by Goodwin and Wilken can be done in one of the two phases that we have described:

Optimal spilling	Optimistic coalescing
• Spilling	• Register assignment
• Live range splitting	• Copy elimination
• Callee/caller-save management	
• Rematerialization	

We did not implement rematerialization, but it should fit naturally into our spilling model.

Kong and Wilken [15] extend the work of Goodwin and Wilken to handle irregular architectures and in particular the IA-32 instruction set, but their treatment of addressing modes appears to be much weaker than ours. Many of the extensions deal with special aspects of register assignment on the Intel architecture, such as the penalty in code size for using addressing modes involving registers %esp and %ebp, and the use of short (8 and 16 bit) registers; we do not deal with these issues. They also consider the insertion of splits before commutative operations, i.e., a commutative operation such as $S3 \leftarrow S1 + S2$ could be translated by either moving S1 or S2 into S3 and performing the appropriate two address instruction – the choice is made by the linear program. They do not consider the possibility of inserting splits at any program point.

Lueh, Gross, and Adl-Tabatabai [16]. *Fusion*-based register allocation breaks up register allocation into a per-region basis, where the simplest region is a basic block. Spilling is performed inside the region so that the resulting interference graph is simplifiable. It may be necessary to spill *transparent* live ranges for the graph to be simplifiable, but the actual spilling of transparent live ranges is delayed. A transparent live range is one that is live on entry and exit to a region and is not used within the region. It is similar to members of our *Copy* set. As neighboring regions are fused together, each region can be individually colored by inserting splits for all the transparent live ranges at the boundaries of the region and coloring each region individually. Of course this naive strategy is undesirable, and great effort is made to stretch the lifetimes of transparent live ranges in memory or registers across the multiple regions being fused together. This is precisely what our linear programming phase does, but with a lot less bookkeeping, and our version is simpler to specify.

Chow and Hennessy [6] use *priority-based coloring* before instruction selection. Higher-priority temporaries are more important to keep in registers. They assign colors to the interference graph in order of priority; when a temporary is uncolorable, they use a greedy heuristic to split it into smaller live ranges. Some of these live ranges will be colorable (with copies between one and the next, if they have different colors), and some will spill. This algorithm is not particularly simple to implement, makes no guarantee of optimality, and they describe results only for the relatively unconstrained problem of a 32-register RISC machine.

9. CONCLUSIONS

We have formulated the register allocation problem for CISC architectures with few registers into one involving optimal placement of spill code, followed by optimal register coalescing. We have given some empirical evidence that dividing the problem into these two phases does not significantly worsen the overall quality of the solution, but a full demonstration of this fact would require optimal solutions to the overall problem that no one has been able to calculate. We have demonstrated an efficient algorithm using integer linear programming for optimal spill-code placement.

The optimal coalescing problem has a significantly simpler structure than the general register-allocation problem, as the spilling has already been taken care of, and every node in the graph has small degree. Because of this, our adaptation of Park and Moon's optimistic coalescing algorithm is simpler and stronger than the original.

Although optimistic coalescing performs well, it is not optimal. We have formulated the optimal coalescing problem (at the end of section 5) in such a simple way – significantly simpler than traditional register-allocation problems that require spilling – that other researchers can continue to investigate optimal coalescing.

Programs compiled with optimal spilling followed by optimistic coalescing run about 9.7% 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 [3] that the splits induced by SSA would be appropriate for register allocation and spilling.

10. REFERENCES

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

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag.

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

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

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[6] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501–536, October 1990.

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

[8] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, South San Francisco, CA, 1993. www.ampl.com.

[9] L. George and A. 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.

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

[11] D. W. Goodwin. *Optimal and Near-Optimal Global Register Allocation*. PhD thesis, University of California at Davis, 1996.

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

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

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

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

[16] 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.

[17] J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques*, pages 196–204, 1998.

[18] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th IEEE/ACM International Symposium on Microarchitecture (MICRO-27)*, Nov. 1994.

Appendix: AMPL model for spilling

```
model;
set Vars; set Pts;
set Copy within (Pts cross Pts cross Vars);
set Copy2 within (Pts cross Pts cross Vars cross Vars);
set Mutate within (Pts cross Pts cross Vars);
set MutateReg within (Pts cross Pts cross Vars);
set Unary within (Pts cross Pts cross Vars cross Vars);
set Binary within (Pts cross Pts cross Vars 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);
set Nullary within (Pts cross Pts cross Vars);
set NullaryReg within (Pts cross Pts cross Vars);
set Branch within Pts;
set Connect within (Pts cross Pts);
set Exists within (Pts cross Vars);

param weight {Pts}; param K {Pts};
param loadCost; param storeCost; param moveCost; param instrCost;

var inReg {Exists} binary;    var inMem {Exists} binary;
var load {Exists} binary;     var loadt {Exists} binary;
var store {Exists} binary;

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

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

subject to REGISTERS_K1 {p in Pts}:
    sum { (p,v) in Exists }
```

```
        (inReg[p,v] + store[p,v]) ≤ K[p];
subject to REGISTERS_K2 {p in Pts}:
    sum { (p,v) in Exists }
        (inReg[p,v] + load[p,v] + loadt[p,v]) ≤ K[p];

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

subject to COPY2_PROPAGATE {(p1,p2,src,dst) in Copy2}:
    load[p1,src] + inReg[p1,src] = store[p2,dst] + inReg[p2,dst];

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

subject to USEUP2{(p1, p2, src1, src2) in UseUp2}:
    loadt[p1,src1] + load[p1,src1] + inReg[p1,src1] +
    loadt[p1,src2] + load[p1,src2] + inReg[p1,src2] ≥ 1;

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

subject to BINARY_PROPDST {(p1,p2,src,dst) in Binary}:
    load[p1,dst] + inReg[p1,dst] = store[p2,dst] + inReg[p2,dst];

subject to BINARY_IN_REG {(p1,p2,src,dst) in Binary}:
    loadt[p1,src] + load[p1,src] + inReg[p1,src] +
    loadt[p1,dst] + load[p1,dst] + inReg[p1,dst] ≥ 1;

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

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

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

subject to UNARY_BINARY_IN_REG {(p1,p2,src,dst) in Unary}:
    loadt[p1,src] + load[p1,src] + inReg[p1,src] +
    store[p2,dst] + inReg[p2,dst] ≥ 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] *
        ( (inMem[p1,src] + store[p1,src]) * (loadCost + instrCost) +
        (1 - (inReg[p2,dst] + store[p2,dst])) * 0.8 *
                    (loadCost + storeCost + instrCost)))
+ (sum {(p1,p2,src,dst) in Copy2}
        weight[p1] *
        ((inMem[p1,src] + store[p1,src]) * (loadCost + instrCost) +
        (1 - (inReg[p2,dst] + store[p2,dst])) * (storeCost + instrCost)))
+ (sum {(p1,p2,src,dst) in Unary}
        weight[p1] *
        ( (inMem[p1,src] + store[p1,src]) * 0.8 * (loadCost + instrCost) +
        (1 - (inReg[p2,dst]+store[p2,dst])) * 0.8 * (storeCost+instrCost)))
+ (sum {(p1,p2,dst) in Mutate}
        weight[p1] * ( (1 - (inReg[p2,dst] + store[p2,dst])) * 0.8 *
                    (loadCost + storeCost + instrCost)))
+ (sum {(p1,p2,dst) in Nullary}
        weight[p1] * ( (1 - (inReg[p2,dst] + store[p2,dst])) * 0.8 *
                    (storeCost + instrCost)))
+ (sum {(p1,p2,src) in UseUp}
        weight[p1] *
        ( (inMem[p1,src] + store[p1,src]) * 0.8 * (loadCost + instrCost)))
+ (sum {(p1,p2,src1,src2) in UseUp2}
        weight[p1] *
        ( (inMem[p1,src1] + store[p1,src1]) * 0.8 * (loadCost + instrCost)+
        (inMem[p1,src2] + store[p1,src2]) * 0.8 * (loadCost + instrCost)));
```