Optimal Spilling for CISC Machines with Few Registers

Andrew W. Appel  
Princeton University  
apel@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 related by a move instruction if this can be done without increasing the number of spills.

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.

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 unsplilled 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
variables are usually indexed over sets, so a declaration such as:

```AMPL
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 logical connections among the sets. For example:

```AMPL
\forall t \in T . \sum_{r \in R} x_{t,r} \geq \text{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 \( T \):

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

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

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:

```AMPL
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, \((p_1, p_2, v_1, v_2) \in \text{Binary}, \) and similarly for \( \text{Unary} \). That is, set \( \text{Binary} \) is a subset of \( P \times P \times V \times V \) and is declared in the AMPL model using:

```AMPL
set Binary C \{(P \times P \times V \times V)\};
set Unary C \{(P \times P \times V \times V)\};
```

For any variable \( v_1 \) that is live at a point \( p_1 \), we write \((p_1, v_1) \in \text{Exists}. \) The \( \text{Exists} \) 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 \((p_2, v) \in \text{Exists}. \) If a variable \( v_1 \) is live and carried unchanged from point \( p_1 \) to \( p_2 \), then we say that \((p_1, p_2, v_1) \in \text{Copy}. \) 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 \((p_1, p_2, v_1, v_2) \in \text{Copy2}. \)

```AMPL
set Exists C \{(P \times V)\};
set Copy C \{(P \times P \times V)\};
set Copy2 C \{(P \times P \times V \times V)\};
```

1 AMPL actually uses the word `cross` instead of the symbol `×`, and `within` instead of `⊂`. In general, we will use mathematical notation instead of strictly AMPL notation in this paper, and give the full AMPL code (93 lines long) in our forthcoming technical report.
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 C (P × P × V);
set NullaryReg C (P × P × V);
```

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

```
set Mutate C (P × P × V);
set MutateReg C (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 C (P × P × V × V);
set UseUp C (P × P × V);
set UseUpReg C (P × P × V);
```

Consider a branch instruction between points p1 and p2 that branches to p4 if v1 = 0, but otherwise falls through to p3. It is necessary to know about points such as p2 that are associated with a branch as we cannot insert spill or reload instructions at p2. We therefore have a set of branching points:

```
set Branch ⊂ P
```

with p2 ∈ Branch. Suppose v3 is live throughout, and v1 is live only in the p4 successor.

```
   • p1
   if (v1 = 0)
   • p2

   • p3
   v3 ∈ live
   • p4

   v1, v3 ∈ live
```

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

\[
(p1, p2, v1) ∈ UseUp;
\]

\[
\{(p1, p2, v1), (p2, p3, v1), (p2, p4, v3),
(p1, p2, v1), (p2, p4, v1),\} ∈ COPY
\]

Note that v1 is used and propagated between the points p1 and p2, 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] ← 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:

\[
(p1, p2, v) ∈ UseUpReg
\]

\[
(p1, p2, v) ∈ UseUpReg
\]

Similarly, the instruction v ← v + m[x] is modeled as:

\[
(p1, p2, v) ∈ MutateReg
\]

\[
(p1, p2, v) ∈ UseUpReg
\]

Or consider the case where the source operand is a constant, v ← v + c:

\[
(p1, p2, v) ∈ Mutate
\]

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. The frequencies can be obtained by profiling or by static estimation [18] (which we use in our implementation). 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:

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

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

Finally we have some scalar cost parameters:

```
param C_{load}, C_{store}, C_{move}, C_{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, \(C_{load} > C_{store} > C_{move} > C_{instr}\). (In
fact, $C_{\text{inst}}$ really measures the cost of a slight extra pressure on the 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:

\begin{align*}
\text{set } P & := \{ p_1, p_2, p_3, \ldots, p_{14}, p_{15} \} \\
\text{set } V & := \{ t_1, t_2 \} \\
\text{set } \text{NullaryReg} & := \{ (p_1, p_4, t_1) \} \\
\text{set } \text{UseUp2} & := \{ (p_5, p_6, t_2) \} \\
\text{set } \text{UseUp} & := \{ (p_8, p_9, t_1) (p_{12}, p_{13}, t_2) \} \\
\text{set } \text{Mutate} & := \{ (p_8, p_{10}, t_1) \} \\
\text{set } \text{MutateReg} & := \{ (p_8, p_9, t_2) \} \\
\text{set } \text{Binary} & := \{ (p_6, p_9, t_2) \} \\
\text{set } \text{Copy} & := \{ (p_4, p_5, t_1) (p_5, p_6, t_1) (p_6, p_7, t_1) (p_7, p_8, t_1) (p_8, p_9, t_1) (p_{10}, p_{11}, t_1) (p_{11}, p_{12}, t_1) (p_9, p_{13}, t_2) (p_{10}, p_{11}, t_2) (p_{11}, p_{12}, t_2) \} \\
\text{set } \text{Exists} & := \{ (p_4, t_1) (p_5, t_1) (p_6, t_1) (p_7, t_1) (p_8, t_1) (p_9, t_1) (p_{10}, t_1) (p_{11}, t_1) (p_{12}, t_2) (p_{13}, t_2) \}
\end{align*}

The \text{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, \text{imull} is classified as \text{MutateReg} for the destination operand and \text{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 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:

\begin{itemize}
  \item arrive at $p$ in a register and depart in a register – $r_{p,v}$,
  \item arrive in memory and depart in memory – $m_{p,v}$,
  \item arrive in a register and depart in memory – $s_{p,v}$ (for stored),
  \item or arrive in memory and depart in a register – $l_{p,v}$ (for loaded).
\end{itemize}

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:

\begin{align*}
\text{var } r (\text{Exists}) & \text{ binary;} \\
\text{var } m (\text{Exists}) & \text{ binary;} \\
\text{var } l (\text{Exists}) & \text{ binary;} \\
\text{var } s (\text{Exists}) & \text{ binary;
\end{align*}

This says that for each $(p,v)$ in $\text{Exists}$ – 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 $v$:

$$\forall (p,v) \in \text{Exists}. \quad l_{p,v} = 0 \quad r_{p,v} = 0 \quad m_{p,v} = 0 \quad s_{p,v} = 0$$

**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.

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

Figure 2: Intel IA-32 instructions for the factorial function

Figure 3: Flowgraph annotated with points
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 \( p \).

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

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

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

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 \( p_2 \) (\( s_{p_2,v} = 1 \)).

\[
\forall (p_1,p_2,v) \in \text{Copy}. \ l_{p_1,v} + r_{p_1,v} = s_{p_2,v} + r_{p_2,v}
\]

The constraint \( s_{p_2,v} + m_{p_2,v} = l_{p_2,v} + m_{p_2,v} \) 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.

\[
\forall (p_1,p_2,v_1,v_2) \in \text{Copy2}, \ l_{p_1,v_1} + r_{p_1,v_1} = s_{p_2,v_2} + r_{p_2,v_2}
\]

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:

\[
\forall (p_1,p_2,v_1,v_2) \in \text{Binary} \quad l_{p_1,v_1} + r_{p_1,v_1} + l_{p_1,v_2} + r_{p_1,v_2} \geq 1
\]

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

\[
\forall (p_1,p_2,v_1,v_2) \in \text{Binary} \quad l_{p_1,v_2} + r_{p_1,v_2} = s_{p_2,v_2} + r_{p_2,v_2}
\]

There are similar constraints for the other classes of instructions, as shown in the full technical report. 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:

\[
\text{minimize COST}:
\]

\[
\text{weight}_p ((C_{\text{load}} + 3C_{\text{instr}})l_{p,v} + (C_{\text{store}} + 3C_{\text{instr}})s_{p,v}) + \ldots
\]

The cost of executing a load is \( C_{\text{load}} \). The cost of a 3-byte load instruction (in L-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 \( C_{\text{load}} \) and \( C_{\text{store}} \), and one extra byte of \( C_{\text{instr}} \) cost to specify the operand. If the source operand is in memory, then we incur a load cost and one instruction-byte cost:

\[
+(\sum_{(p_1,p_2,v_1,v_2) \in \text{Binary}} \text{weight}_{p_1} ((C_{\text{load}} + C_{\text{instr}})(m_{p_1,v_1} + s_{p_1,v_1}) + (C_{\text{load}} + C_{\text{store}} + C_{\text{instr}})(m_{p_2,v_2} + l_{p_2,v_2})) + \ldots
\]

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:

\[ r_{p_1} \cdot t_{p_2} \cdot s_{p_{n_1}} \cdot m_{p_{n_1}} \]

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

\[ r_{p_4} + t_{p_4} + s_{p_4} + m_{p_4} = 1 \]

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 \( z_2 \leftarrow x_1 \) 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 \( z_2 \) 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 \text{Copy} or \text{Copy2} relation, we also introduce a copy in the optimal-coalescing graph. That is, all the variables copied across an edge form 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 it is common in register-allocation problems [1, Lgs. 19.2–3]. The resulting flowgraph for the example in Figure 2 is shown in Figure 4.

![Figure 4: Flowgraph with internal splits](image)

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 \text{exch} (exchange two registers) instruction 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

\(^2\)The situation is more complicated for machines with instructions that both overwrite some of the input operands and generate new
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 (à la 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.

#### 7.1 Optimal Spilling

Figure 5 shows the spill statistics. The Spills and Reloads columns show 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 a source operand and a destination, and those that use memory for a source only. Each column shows the base SML/NJ compiler (version 110.23), and the same compiler modified to use our new algorithm for optimal spilling (Opt).

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 of spill and reloads from the base compiler is 30% 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 again demonstrates that the optimal spilling has made much better use of effective addressing modes.

### 7.2 Optimal-spill performance

Figure 6 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 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 scatter plot is a crude bounding box obtained from Figures 3.4 in Goodwin’s thesis [11].³

The most important result from Figure 6 is that two minutes was the longest time taken on a 250MHz 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 [12] 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 [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 7 shows the number of splits 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 is not shown. 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 8) show a speedup of 9.5% improvement in execution speed (taking the geometric mean of ratios), which we feel can be improved even further by the use of accurate profiles instead of static estimates.

³The technical report explains why we believe the machine on which Goodwin got his results is about as fast as our machine.
8. RELATED WORK

**Goodwin and Wilken** [12] address several optimization problems such as live range splitting, register assignment, spill placement, rematerialization, callee/caller-save 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 6 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:

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

**Figure 7:** Number of splits and instructions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions Opt</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes-hut</td>
<td>2.92</td>
<td>0.0%</td>
</tr>
<tr>
<td>boyer</td>
<td>12.57</td>
<td>0.0</td>
</tr>
<tr>
<td>mlyacc</td>
<td>9.14</td>
<td>0.0</td>
</tr>
<tr>
<td>tsp</td>
<td>6.92</td>
<td>2.2</td>
</tr>
<tr>
<td>lexgen</td>
<td>9.08</td>
<td>2.7</td>
</tr>
<tr>
<td>count-graphs</td>
<td>24.07</td>
<td>8.7</td>
</tr>
<tr>
<td>icfp00</td>
<td>109.29</td>
<td>9.6</td>
</tr>
<tr>
<td>fft</td>
<td>8.58</td>
<td>10.0</td>
</tr>
<tr>
<td>logic</td>
<td>5.10</td>
<td>10.6</td>
</tr>
<tr>
<td>knuth-bendix</td>
<td>8.08</td>
<td>11.9</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>27.92</td>
<td>20.3</td>
</tr>
<tr>
<td>life</td>
<td>19.03</td>
<td>24.9</td>
</tr>
<tr>
<td>simple</td>
<td>31.53</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Figure 8:** Execution speed

Our optimal spilling can handle problem sizes at least an order of magnitude larger than theirs, as Figure 6 shows. We believe this is an important benefit of separating spilling from coloring.

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 $S_3 \leftarrow S_1 + S_2$ could be translated by either moving $S_1$ or $S_2$ into $S_3$ 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 spills 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 no 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


