A New MLRISC Register Allocator

(version 110.25)

Allen Leung
New York University
leunga@cs.nyu.edu

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

Abstract

This report describes the new graph coloring register allocator that we have developed for the MLRISC customizable code generator. The new allocator has a more modular design, uses a sparse method for constructing the interference graph, and has incorporated some new features and algorithmic improvements, including support for frequency sensitive allocation, and spill reduction strategies. Preliminary benchmarks (by compiling the SML/NJ compiler) indicate that the new allocator is about 20% faster on RISC machines, which translates into 4% to 6% overall speedup in the compiler; but on the Intel x86, no speedup is obtained. The code quality, when measured as the amount of spill code and the amount of generated code, uniformly has improved on all the architectures that we have tested. On RISC architectures, we have obtained over 10% reduction in spill code.

1 Introduction

This report describes the design and implementation of the new register allocator for the MLRISC customizable code generator [Geo97]. This new allocator, like the current allocator distributed with MLRISC, is a Chaitin-style graph coloring allocator that uses the iterated coalescing algorithm [GA96]. The new allocator, however, has a different client interface, uses different data structures, and has incorporated the following new features and improvements.

Priority Modern compiler optimizations are often driven by execution frequency information gathered either through profiling or static estimation techniques. The goal to is make optimizations sensitive to the payoff, so that frequently executed parts of the program are highly optimized, possibly by trading off performance gains in parts of the program that are not executed as often. This idea is actually quite old, and has been incorporated into many register allocation frameworks. Chow and Hennessy's allocator [CH90], which uses a frequency sensitive heuristic to guide the order of coloring and splitting, is a classic example.

The new allocator can now take advantage of frequency information supplied by the client to prioritize decisions such as coalescing and spilling. This addition should improve code quality, especially for programs with tight loops.

Sparse interference graph construction The new allocator uses a sparse method for constructing the interference graph. This algorithm can also operate incrementally, which makes it possible to incrementally reconstruct the interference graph when transforming the program[1], such as during spilling or splitting.

Improved spilling heuristics The new allocator has smarter spill insertion strategies. In the old allocator, spills and reloads that are obviously useless, such as a spill followed immediately by a reload of the same value, are sometimes generated. The new allocator can now avoid these types of unproductive code.

More modular design Lastly, the new allocator has a more modular parametric design which makes it possible for the client to replace specific components. For example, it is now possible for the client to replace the default spilling heuristics with another, simply by composing the modules that make up register allocator in a different way.

Changes in the client interface also makes it easier to implement features such as rematerialization [BCT94], and live-range splitting. Support for register pair coalescing and coloring, and support for predicated architectures, can also be more elegantly accomodated in the new design.

1.1 Overview

The rest of this report is organized as follows. Section 2 describes the details of the algorithm and our extensions. Section 4 describes related work. Section 5 presents the preliminary benchmark results. Appendix A shows how the allocator can be specialized, and how to implement the client callbacks that is needed by the allocator.

2 The Algorithm

2.1 Clusters

All optimizations in MLRISC are performed on *clusters*, a unit of compilation that may include multiple related procedures. MLRISC represents a program at the level of the target machine instructions. From the view of the allocator, a cluster has two components: the control flow graph of the program, and a *regmap*, a mapping from register name to register name. Before allocation, the regmap is (usually) the identity, and after allocation, the regmap maps

[1]This feature has yet been implemented.

each virtual register in the cluster into the appropriate physical register. The allocator does not directly transform the instructions in a cluster, except during spilling, when spill code is inserted.

2.2 Parallel Copies

The iterated coalescing algorithm can effectively eliminate copies in a program, by mapping the source and target of a move with the same register. All MLRISC supported instruction sets contains pseudo parallel copies instructions for integer and floating point registers.

```
COPY{dst, src, tmp, ...}
FCOPY{dst, src, tmp, ...}
```

The semantics of these instructions is to copy all the registers in `src` in parallel to all the registers in `dst`. When there are more than one sources and destinations, we also associate a *copy temporary* `tmp` with the copy instruction. During code emission a parallel copy has to be linearized to a sequence of actual instructions. This copy temporary is used for shuffling values if the linearization contains cycles.

George and Appel [GA96], and Reig Galilea and George [GG99] described how paralell copies can be used to aid code generation for SML and C-- [JON98, JNON97].

2.3 Phases in the allocator

Figure 1 shows the phase ordering of the iterated coalescing framework [GA96] used in the allocator. Phases surrounded by dashes are parameterizable and may be replaced by other implementations when appropriate, as they are dependent on the program representation, machine architecture and/or instruction semantics. For example, hyperblocks [MLC+92], which we also support in MLRISC, require a different method for computing live-ranges that is predicate-aware [JS96, ED95], thus a different module for constructing the interference graph is needed[2].

The phases follow the basic framework described in [GA96], so we only highlight the differences in each phase:

Build/Rebuild In addition to constructing the interference graph, this phase is also responsible for collecting the following information for each live range v:

1. The definition sites $\mathit{defs}(v)$ and use sites $\mathit{uses}(v)$,
2. The spill cost, which is defined as

$$\mathit{spillcost}(v) = \sum_{u \in \mathit{uses}(v)} \mathit{loadcost} \times \mathit{freq}(u) + \sum_{d \in \mathit{defs}(v)} \mathit{storecost} \times \mathit{freq}(d)$$

[2] Hyperblock register allocation has not implemented yet.

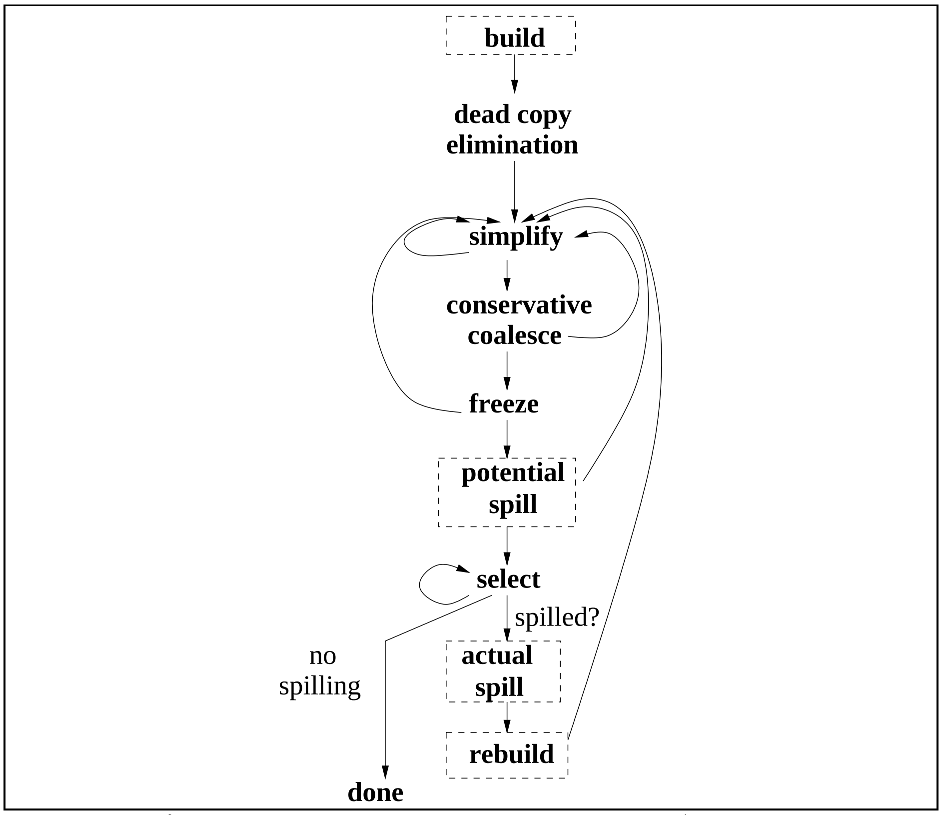

Figure 1: Phases in the Iterated Coalescing Algorithm

where *freq*(i) is the execution frequency of instruction i, and *loadcost* and *storecost* are the execution cost of each load and store[3].

1. Compute *defs*(v), *uses*(v) and *spillcost*(v) for all variables v by scanning the program.

2. **for** $u \in uses(v)$ **do**

```
    Visited := Ø
    LiveIn(v, u)

procedure LiveOut(v, u)
    for v' ∈ defs(u) do
        if v' ≠ v then
            Add interference edge (v, v')
    if v ∉ defs(u) then
        LiveIn(v, u)

procedure LiveIn(v, u)
    if u ∉ Visited then
        Visited := Visited ∪ {u}
        let b be the current block where u resides
        if u is the first instruction in b then
            for b' ∈ pred(b) do
                let w be the last instruction in b'
                LiveOut(v, w)
        else
            let w be the previous instruction of u
            LiveOut(v, w)
```

Figure 2: Sparse interference graph construction

Our allocator uses the sparse interference graph construction algorithm shown in Figure 2. This algorithm is adapted from the one presented in [App98], and it works in two phases. In the first phase, we compute the definition and use sites for all variables. This phase may be skipped if such information is already available from previous phases of the compiler. In the second phase, we perform a liveness analysis for all variables v individually. For all $u \in uses(v)$, we walk backward in the control flow graph, visiting each instruction no more than once (this can be accomplished by marking each instruction visited.). Whenever a definition $v' \leftarrow \ldots$ where $v' \neq v$ is encountered, we add the edge (v, v') to the interference graph. We stop the traversal at definition sites *defs*(v).

[3] A simple sequential execution model is assumed.

This algorithm can be also be adapted to work in linear-time for SSA [CFR+89] or machine SSA forms [LG99].[4] When applied to non-SSA programs, the algorithm is not linear in the worst case; nevertheless, it is effectively linear in practice.

Traditionally, interference graph construction first uses an iterative dataflow liveness analysis to compute liveout information for each basic block, then builds the interference graph by scanning each basic block individually, while performing a local liveness analysis. In contrasts, the sparse approach does not require a separate local liveness analysis, and can easily operate incrementally, since no local liveness information needs to be updated.

The new allocator makes the distinction between the first time an interference is constructed (**build**) and subsequent rounds of interference graph construction after spilling (**rebuild**). In the current implementation, we just clear the interference graph and rebuild it from scratch after the first round. In future releases, incremental reconstruction will be added, exploiting the flexibility of the sparse inference graph construction algorithm.

Dead copy elimination This optional phase eliminates dead copies from the program. A copy $dst \leftarrow src$ is dead if dst has no use. We may safely discard the move, and remove dst and all its edges from the interference graph.

This optimization benefits mainly the x86 platform. On RISC machines, dead copies can usually be coalesced away, leaving no dead code in the final program. However, on the x86 these copies are often retained, because the few numbers of colors available inhibits coalescing, and because copies are sometimes transformed into loads or stores during the RA32 to RA8 conversion phase [Geo99]; these loads and stores cannot be handled by coalescing.

Although the dead copy elimination heuristic is strictly less powerful than dead code elimination, it is also much cheaper to run since it does not require dataflow analysis, but uses information readily available from interference graph construction.

Conservative Coalescing The new allocator coalesces moves in non-increasing order of its cost. Given a move $dst \leftarrow src$, we define its cost as its execution frequency in the program. Note that the order of coalescing, unlike that of simplification, is significant; coalescing a moves may prohibit another from coalescing, or vice versa. Thus prioritized coalescing has the effect of coalescing moves that are "expensive" first before trying cheaper moves.

Freeze Like coalescing, this phase is also prioritized in the new allocator; the node chosen is the one with the lowest freeze cost. We define the freeze cost of v to be the sum of the cost of all the uncoalesced moves associated with v.

[4] The algorithm in [App98] only marks the end of basic blocks as they are visited, and in various scenarios fails to exhibit linear complexity, since the same instructions may be visited multiple times.

Thus this heuristics has the effect of discarding "cheaper" moves first before more "expensive" moves.

Potential Spill The spill node selection heuristic is parameterized in the new allocator. The default potential spill module uses a modified Chaitin function $chaitin(r)$ that is weighted by execution frequencies. This is defined as:

$$chaitin(r) = \begin{cases} \infty & \text{if } defs(r) = \{p\} \text{ and } uses(r) = \{q\} \\ & \text{and } p \text{ immediately precedes } q \\ -1 - deg(r) & \text{if } uses(r) = \emptyset \\ spillcost(r)/deg(r) & \text{otherwise} \end{cases}$$

where $deg(r)$ is the degree of the node r.

The purpose of the first clause is to avoid spilling a short live-range. The second clause gives higher priority for live-ranges that have no use than live-ranges that have uses. These live-ranges are either dead values, in which case they can be killed, or else temporaries associated with parallel copies[5]. In the latter case, we expect that these temporary are frequently unnecessary[6] so using them as potential spill candidates is usually a good decision. The term $-deg(r)$ is incorporated because it is generally more profitable to spill live-ranges with higher degree of interference.

The third clause is the basic Chaitin heuristic, but weighted by the execution frequency of each definition and use sites.

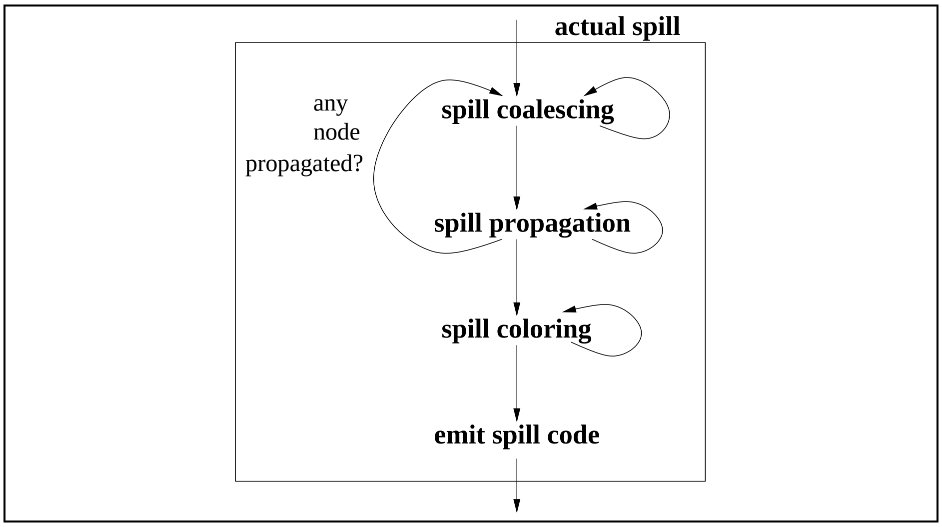

Figure 3: Actual Spill Phase

[5]In our implementation, copy temporaries have no use sites associated with them.
[6]The temporaries are only necessary to break cycles in parallel copies.

Actual Spill The default spilling strategy that we use is Chaitin's: for each spilled live-range v, we insert spill code before definitions sites $\mathit{defs}(v)$ and reload code before each use sites $\mathit{uses}(v)$.

In addition to this basic strategy, our allocator has incorporated a few spill improvements strategies that can reduce the amount of spill/reload code and the size of the stack frame for holding spilled variables. These are: (1) **spill coalescing**, which coalesces the spill locations of live-ranges in order to eliminate redundant memory to memory moves; (2) **spill coloring**, which attempts to merge the memory locations of spilled variables in order, and (3) **spill propagation**, which, in combination with spill coalescing, can further reduce spill and reload cost by evicting colorable but "pass through" live-ranges to memory. These phases are structured as in Figure 3.

Spill Coalescing Chaitin-style spilling naively assigns spill locations in memory in a first come first serve basis. If two spilled live-ranges a and b are assigned different memory locations, it may be necessary to insert memory to memory move instructions if a and b are connected by copies. This problem can become severe when the register pressure is high, or in architectures with few registers, such as the x86.

Suppose we have a move instruction $a \leftarrow b$ where both a and b cannot be colored. With conservative coalescing, the move will not be coalesced, and since both a and b cannot be colored and must both be spilled. Chaitin-style spilling will transform the move $a \leftarrow b$ into a sequence of memory move instructions. Figure 4 illustrates this effect.

$$
\begin{array}{ccc}
b & \leftarrow & \ldots \\
 & \vdots & \\
a & \leftarrow & b \\
 & \vdots & \\
\ldots & \leftarrow & a
\end{array}
\quad \overset{\Longrightarrow}{\text{spill } a, b} \quad
\begin{array}{ccl}
t_1 & \leftarrow & \ldots \\
\mathtt{st} & t_1, & \mathtt{mem}(b) \\
 & \vdots & \\
t_2 & \leftarrow & \mathtt{ld}\ \mathtt{mem}(b) \\
\mathtt{st} & t_2, & \mathtt{mem}(a) \\
 & \vdots & \\
t_3 & \leftarrow & \mathtt{ld}\ \mathtt{mem}(a) \\
\ldots & \leftarrow & t_3
\end{array}
\qquad
\begin{array}{ccl}
t_1 & \leftarrow & \ldots \\
\mathtt{st} & t_1, & \mathtt{mem}(ab) \\
 & \vdots & \\
 & \vdots & \\
t_3 & \leftarrow & \mathtt{ld}\ \mathtt{mem}(ab) \\
\ldots & \leftarrow & t_3
\end{array}
$$

(a) (b) (c) With spill coalescing

Figure 4: Spill coalescing example

Spill coalescing can be used to overcome this deficiency. Suppose we have a move $a \leftarrow b$ and both a and b must be spilled, then a and b may be assigned the same spill location if a does not interfere with b. Now, by targeting a and b both to the same spill location, we can avoid spilling and reloading the copy $a \leftarrow b$. In effect, we have coalesced the spill locations of a and b (see Figure 4(c)).

Now suppose we have two copies $a \leftarrow b$ and $a \leftarrow c$, where a, b and c are non-colorable, and suppose further that b interferes with c. Clearly, only one pair of spill locations out of a, b and c may be merged in memory. Taking in account of this effect, our spill coalescing algorithm (see Figure 5) prioritizes moves by their costs, so that more expensive moves are considered first during the coalescing step.

procedure SpillCoalescing(nodes to coalesce C, spilled nodes S)
    let $M := [\{a, b\} \mid \text{all moves between } S \text{ with source or destination in } C]$
    while there are duplicates $m_1, m_2 \in M$ **do**
        Eliminate the duplicates and count the move cost as the sum of m_1's and m_2's.
    let I be the current interference graph.
    for $\{a, b\} \in M$ in non-increasing move cost **do**
        If (a, b) does not interfere in I **then**
            Merge the nodes a and b in I
            Coalesce the spill locations of a and b

Figure 5: Spill Coalescing Algorithm

Spill Propagation Contrary to intuition, sometimes it is beneficial to spill instead of keeping values in registers. Consider the code fragment in Figure 6(a), where we have three live-ranges a, b and c, connected by copies. Suppose a and c are not colorable but b is. Using the conservative coalescing heuristic, the copies $b \leftarrow a$ and $c \leftarrow b$ will not be coalesced. Instead nodes a and c will be spilled. The code that remains after spilling looks like the sequence in Figure 6(b).

$$
\begin{array}{ccc}
a & \leftarrow & \ldots \\
 & \vdots & \\
b & \leftarrow & a \\
 & \vdots & \\
c & \leftarrow & b \\
 & \vdots & \\
 & \leftarrow & c
\end{array}
\quad \xrightarrow[\text{spill } a, c]{} \quad
\begin{array}{ccc}
t_1 & \leftarrow & \ldots \\
\texttt{st} & t_1, & \texttt{mem}(a) \\
 & \vdots & \\
b & \leftarrow & \texttt{ld mem}(a) \\
 & \vdots & \\
\texttt{st} & b, & \texttt{mem}(c) \\
 & \vdots & \\
t_2 & \leftarrow & \texttt{ld mem}(c) \\
 & \leftarrow & t_2
\end{array}
\qquad
\begin{array}{ccc}
t_1 & \leftarrow & \ldots \\
\texttt{st} & t_1, & \texttt{mem}(abc) \\
 & \vdots & \\
 & \vdots & \\
 & \vdots & \\
t_2 & \leftarrow & \texttt{ld mem}(abc) \\
 & \leftarrow & t_2
\end{array}
$$

(a) (b) (c) with spill propagation

Figure 6: Spill propagation example

However, note that b is only used to connect the live-ranges a and c. Suppose the live-range b is also spilled with a and c, then the instructions that load $\texttt{mem}(a)$

into b and spill b into `mem`(c) can be made superfluous with spill coalescing, and all a, b and c can be merged into the same spilled location. Thus if we *propagate* the spill from a and c into b, we can obtain an improved sequence of code. Figure 6(c) shows the result of this transformation.

We formalize this strategy as follows. Given a colorable live-range v, define the lists $L(v)$ and $S(v)$ as:

$$L(v) = [u \leftarrow v \mid u \text{ is non-colorable and } u \leftarrow v \text{ is a non-interfering copy}]$$

$$S(v) = [v \leftarrow w \mid w \text{ is non-colorable and } v \leftarrow w \text{ is a non-interfering copy}]$$

The live-ranges in $L(v)$ and $S(v)$ are all non-colorable and each may be merged with v in memory if v were spilled. We define the function $savings(v)$ as follows.

$$savings(v) = \sum_{u \leftarrow v \in L(v) \cup S(v)} (loadcost + storecost) \times freq(u \leftarrow v)$$

Function $savings(v)$ measures the savings in terms of memory to memory moves, in the case where v is spilled instead of kept in a register. Intuitively, if $savings(v)$ is greater than $spillcost(v)$, v should be spilled instead of kept in a register, even if it is colorable. This is because $spillcost(v)$ is a worst case measure that assumes the Chaitin-style spilling strategy, while $savings(v)$ measures the savings in spill coalescing.

For instance, in the example in Figure 6, we have $savings(b) > spillcost(b)$, since:

$$\begin{aligned} spillcost(b) &= loadcost \times freq(b \leftarrow a) + storecost \times freq(c \leftarrow b) \\ savings(b) &= (loadcost + storecost) \times freq(b \leftarrow a) + \\ &\quad\; (loadcost + storecost) \times freq(c \leftarrow b) \end{aligned}$$

Thus b should be spilled.

We use the worklist based algorithm presented in Figure 7. It takes as input a set of spilled nodes, and returns a (potentially larger) set of spilled nodes.

Spill propagation and spill coalescing are integrated using the algorithm shown in Figure 8. This algorithm iterates between spill coalescing and spill propagation. This process repeats until no new propagated spill nodes are discovered.

Spill Coloring Spill coloring attempts to map multiple spilled live-ranges into the same spill location. When spilling occurs often in a program, this strategy will reduce the size of the stack frame and improve spatial locality[7].

The spill coloring algorithm that we use is similar to the select phase, except that there is no arbitrary limit to the number of spill locations. We start with only a single spill location, and whenever a node becomes uncolorable, we allocate a new location to the available set. Given a set of spilled nodes S, our algorithm considers only the interference edges induced by S. We currently use a simple round robin scheme for choosing spill locations.

[7] However, it may in some cases introduce false dependencies, thereby reducing the ILP of the program.

```
procedure SpillPropagation(nodes to propagate P, spilled nodes S)
    L := ∅
    NewP := ∅
    (* initialize worklist L *)
    for v ∈ P do
        InsertCandidates(v)
    (* main loop *)
    while L ≠ ∅ do
        Remove w from L
        if Savings(w, S) ≥ spillcost(w) then
            (* spill w since it is cheaper *)
            S := S ∪ {w}
            NewP := NewP ∪ {w}
            InsertCandidates(w)
    return (NewP, S)

procedure InsertCandidates(A spilled node w)
    for all nodes u where u is related to w via a move do
        (* insert only colorable nodes into worklist *)
        if u is not spilled and u ∉ Processed then
            L := L ∪ {u}
            Processed := Processed ∪ {u}

procedure Savings(a colorable node w, S)
    savings := 0
    (* sum up all the savings due to spill coalescing *)
    for all nodes u that is related to w via a move m do
        if u does not interfere with w and u ∈ S then
            let f be the execution frequency of m
            savings := savings + f × (loadcost + storecost)
    return savings
```

Figure 7: Spill Propagation Algorithm

```
procedure SpillPropagationCoalescing(spilled nodes S)
    Processed := ∅
    P := S
    repeat
        SpillCoalescing(P, S)
        (P, S) := SpillPropagation(P, S)
    until P = ∅
    return S
```

Figure 8: Spill Propagation and Coalescing

3 Improvements to Iterated Coalescing

The iterated coalescing algorithm [GA96] is very effective in targeting the registers in copies without causing extra spills. However, in certain pathological cases the original algorithm can spend an excess amount of time iterating between the simplify, coalesce and freeze phases.

Recall that in the original algorithm, an active move (i.e. a move not on the worklist and is not frozen) $x \longleftrightarrow y$ is put onto the worklist whenever one of x or y's neighbors goes from high degree to low degree. Suppose we have an active move $x \longleftrightarrow y$ where x has N neighbors of high degree, where $N \gg K$. Every time $x \longleftrightarrow y$ loses one of its high degree neighbors, a coalescing check may have to be performed, and in the worst case, $O(N)$ checks may have to performed on $x \longleftrightarrow y$.

In version 110.25 of MLRISC we use a slightly different implementation of the iterated coalescing that eliminates this wasteful behavior. The intuition is that an active move should not be put onto the worklist if we know that it has an excess of high degree neighbors. To perform this pruning, each active move m keeps track of a number $hi(m)$, which is a lower bound on the number of high degree neighbors that prohibit m from being coalesced. An active move m is only put onto the worklist when $hi(m) \leq 0$.

The number $hi(x \leftrightarrow y)$ is initialized in two different ways, depending which coalescing test applies to x and y:

Briggs' conservative coalescing test If both x and y are pseudo registers, then define $hi(x \leftrightarrow y)$ as the number of high degree neighbors of the coalesced node xy, minus $K - 1$.

George's coalescing test If x is a precolored node and y is a pseudo register, then define $hi(x \leftrightarrow y)$ as the number of high degree neighbors of y which do not interfere with x.

3.1 The Algorithm

The routines presented in the Appendix of [GA96] are modified in the following manner. Changes to the original are marked with numbers.

procedure Simplify()
    let $n \in$ simplifyWorklist
    simplifyWorklist := simplifyWorklist - { n }
    push(n, selectStack)
1 **if** degree[n] $\geq K$ **then**
1   EnableMoves(Adjacent(n))
    forall $m \in$ Adjacent(n) **do**
        DecrementDegree(m)

procedure DecrementDegree(m)
    let d = degree[m]

```
    degree[m] := d − 1
    if d = K then
1       EnableMoves(Adjacent(m))
        spillWorklist := spillWorklist - {m}
        if MoveRelated(m) then
            freezeWorklist := freezeWorklist ∪{m}
        else
            simplifyWorklist := simplifyWorklist ∪{m}
```

Note 1: function Simplify removes nodes from the simplify worklist. Removing a node from a graph involves decrementing the degree of all its current neighbors. If Simplify is removing a node n of high degree from the graph[8], we call EnableMoves to process all moves related to n's neighbors. Function DecrementDegree is modified so it only calls EnableMoves for all nodes that have lost a high degree neighbor.

```
procedure EnableMoves(nodes)
    (* nodes must have lost a high degree neighbor *)
    forall n ∈ nodes do
        forall m ∈ NodeMoves(n) do
            if m ∈activeMoves then
2               hi(m) := hi(m) − 1
2               if hi(m) ≤ 0 then
                    activeMoves := activeMoves - {m}
                    worklistMoves := worklistMoves ∪{m}
```

Note 2: the modified version of EnableMoves decrements the count $hi(m)$ for all moves m related to the input list of nodes. An active move m is only moved onto the worklist when $hi(m)$ reaches zero.

```
procedure George(u ↔ v)
    (* u must be a precolored-node and v must be non-precolored *)
3   k := 0
3   forall w ∈ Adjacent(v) do
3       if degree[w] ≥ K ∧ (u, w) ∉ adjSet then
3           k := k + 1
3   if k > 0 then
3       hi(u ↔ v) := k
3       return false
3   else
3       return true

procedure Briggs(u ↔ v)
```

[8]Note that this can happen with Briggs' optimistic spilling heuristics, i.e. the potential spill phase can select a high degree node to be simplified.

```
    (* u and v must both be non-precolored *)
    k := 0
    forall n ∈ Adjacent(u) ∪ Adjacent(v) do
        if degree[n] ≥ K then k := k + 1
4   if k < K then
4       return true
4   else
4       hi(u ↔ v) := k − (K − 1)
4       return false
```

Notes 3 and 4: the coalescing tests initializes $hi(m)$ with the appropriate count whenever they fail.

```
procedure Combine(u, v)
    if v ∈ freezeWorklist then
        freezeWorklist := freezeWorklist - {v}
    else
        spillWorklist := spillWorklist - {v}
    coalescedNodes := coalescedNodes ∪{v}
    alias[v] := u
5   if u is precolored then
5       forall m ∈ nodeMoves[v]
5           let x ↔ y = m
5           if x and y are both non-precolored then
5               hi(m) := 0
5               nodeMoves[u] := nodeMoves[u] ∪{m}
5   else
5       nodeMoves[u] := nodeMoves[u] ∪ nodeMoves[v]
6   losingHiDegNode := degree[u] ≥ K ∧ degree[v] ≥ K
    forall t ∈ Adjacent(v) do
        AddEdge(t, u)
        DecrementDegree(t)
    if degree[u] ≥ K ∧ u ∈ freezeWorkList then
        freezeWorklist := freezeWorklist - {u}
        spillWorklist := spillWorklist ∪{u}
6   if losingHiDegNode then EnableMoves(Adjacent(u))
```

Function Combine is used to merge two nodes u and v. We guarantee that v cannot be a precolored node. **Note 5:** if u is a precolored node, then when v is merged with u we have to iterate over all the moves $v \leftrightarrow w$ where w is non-precolored and reset the count $hi(v \leftrightarrow w)$. This is because $hi(v \leftrightarrow w)$ is initialized by the routine Briggs, but now George's heuristics applies, and it is no longer valid. By reseting the count we guarantee that it will be recomputed later if required.

Note 6: The variable `losingHiDegNode` is set to true if by coalescing u and v, the neighbors of u and v lose one neighbor of high degree[9]. If this is true we will call EnableMoves on all moves that are adjacent to the merged node uv.

4 Related Work

Spill coloring is similar to the scheme described in Cooper and Harvey [CH98]. In their scheme, after the first round of allocation, anything that spills is allocated into the compiler controlled memory (CCM), which for our purposes can be identified with spill locations. The original interference graph is consulted to determine the assignment of spill nodes into CCM. In our scheme, we do not keep around the original interference graph and thus the spill coloring decisions between different rounds of spilling are kept separate. While this may reduce the opportunity for compaction, from our observations we believe that this has minimal impact, since multiple rounds of spilling are infrequent and the number of spilled live-ranges become fewer and fewer in subsequent spilling rounds.

Cooper and Harvey also mentioned coalescing in their algorithm, which is based on the algorithm of Briggs et al [BCT94]. However, from their presentation it seems that Briggs' conservative coalescing strategy is used for coalescing spill locations. It is likely because, like the register file, the amount of compiler controlled memory is limited. In our scheme spill coalescing uses Chaitin's aggressive strategy (but prioritized by move cost) since we assume that the size of the stack frame is unbounded.

5 Benchmarks

5.1 Old versus New Allocator

Some preliminary benchmark results are obtained by running the new allocator on SML/NJ version 110.21. We use compilation of the new compiler source code as the test.

Unfortunately, not all the features of the new allocator are exercised here. For example, frequency information are currently unavailable from SML/NJ's FLINT front-end. By default, all basic blocks are assigned an execution frequency of 1. Although we can estimate this information from the program structure in the backend, this adds about 30% overhead on compile time, which we reason is counter-productive for SML/NJ.

Spill coalescing, spill propagation and spill coloring are also not incorporated into these benchmarks. We delay these tests for the next release of the allocator. We expect that the incorporation these phases into our allocator will reduce spill cost even further, with only a minimal increase in compilation time.

[9] We compute this before the subsequent loop because that may alter the degree of u.

	Sparc	PA-RISC	X86
Old ra	344/1414	161/820	371/823
New ra	274/1360	133/768	368/814
	25%/4%	21%/6.7%	<1%/1%

Table 1: Register allocation time versus total compile time (in seconds)

	Sparc	PA-RISC	X86
Old ra	2866+3695	2575+3318	n.a.
New ra	2452+3274	2278+3018	113328+142351
	16.8%/12.8%	13%+10%	

Table 2: Integer spills and reloads

Compilation speed The basic results indicate that the new allocator is faster than the old one in terms of compilation time, even though the new allocator is more complex. On RISC machines, the new allocator is over 20% faster than the old one (see Table 1). This translates into around 4-6% percent of speedup in SML/NJ overall, when measuring the time to bootstrap the compiler. On the Intel x86 architecture, however, no speed up is obtained. This can be attributed to the fact that the x86 requires lots of spilling and incremental interference graph reconstruction has not been implemented.

Spilling Spilling on all architectures have been improved. On RISC architectures, integer spills and reloads reduce by about 10% with the new allocator (see Table 2). There are no floating point spills or reloads in the SML/NJ compiler. Definite spill/reload numbers are unavailable from the old x86 compiler.

The reduction in spilling also results in fewer iterations of the allocator. The phases from **simplify** to **select** constitute one round of allocation; when spilling occurs then more rounds are needed. Table 3 shows the number of extra rounds of allocation versus the number of times the allocator is invoked. In all cases fewer rounds of allocation is needed.

Lastly, Table 4 compares the image sizes of the SML/NJ compiler. The row labeled as "110.21" shows the size of version 110.21 of the compiler. The row labeled as "Old ra" shows the size of the new compiler bootstrapped with the old allocator. Finally row "New ra" shows the resulting image size when the new allocator is used to bootstrap itself. On the x86, about 59K of code has been removed. We believe that this is mainly due to the dead copy elimination phase in the new allocator[10].

[10] Dead copy elimination is only enabled on the x86. On RISC machines, dead copies can usually be successfully coalesced away.

	Sparc	PA-RISC	X86
Old ra	278/46459	240/46480	14896/91811
New ra	191/46459	164/46480	14175/91811

Table 3: Extra allocation rounds

	Sparc	PA-RISC	X86
110.21	7836816	7538796	7848564
Old ra	7793164	7537432	7836888
New ra	7793164	7533336	7777496

Table 4: Size of images

5.2 Effects of Spill Propagation

To test the effectiveness of spill propagation, we have conducted another set of benchmarks [11]. Since SML/NJ spills very little on RISC machines, the benefits of spill propagation are minimal. Instead, we will concentrate on compilation for the x86.

The SML/NJ x86 backend uses a two-phase approach for register allocation [Geo99], in order to accommodate the lack of registers on the x86. The first phase, **ra32**, is used to compute spill locations used in the second phase. The program is first allocated using 32 pseudo memory registers, which are mapped into a small area on the stack frame. In the second phase, **ra8**, the program is allocated using 6 physical registers on the x86 [12]; spills are mapped into the pseudo memory registers computed in the first phase.

In our experiment we compare three different schemes: *(i)* **ra32+ra8**, which is used by the release 110.23 of the SML/NJ compiler; *(ii)* **ra8+coal**, which directly allocates for 6 physical registers but also incorporates the spill coalescing and spill coloring heuristics in the actual spill phase; and *(iii)* **ra8+prop**, which is identical to **ra8+coal** except that spill propagation is also added. Since the spill decisions we get in all three ra8 sub-phases are similar, we believe our results can be used to measure the effectiveness of spill coalescing and propagation heuristics.

In the first set of benchmarks, we compile and run the set of test programs from the SML/NJ distribution. Table 5 shows the result of **ra8+coal** versus **ra8+prop**. Spill propagation seems to be very effective in reducing the amount of the spilling in all cases. Spills and reloads are reduced by an average of 5.7% and 6.5%. On the whole, compilation time improves by 3.28% on average. We believe the compilation speedup is due to the fact that less spilling is required in **ra8+[rop**, and this in turns translates into less rounds through the allocation algorithm.

[11]These benchmarks are conducted on release 110.23 of SML/NJ.
[12]The stack pointer and the allocation pointer are reserved.

Name	Compilation Time (secs)			Spill+Reload		
	ra8+coal	**ra8+prop**	Improv.	**ra8+coal**	**ra8+prop**	Improv.
barnesHut	3.993	3.975	0.46%	639+812	627+781	1.9%+4.0%
boyer	12.538	12.697	-1.25%	487+270	482+267	1.0%+1.1%
count-graphs	1.327	1.482	-10.46%	357+391	338+369	5.6%+6.0%
fft	0.997	0.957	4.18%	176+245	167+230	5.4%+6.5%
knuthBendix	4.087	4.083	0.08%	447+580	431+558	3.7%+3.9%
lexgen	8.970	9.083	-1.25%	822+859	797+806	3.1%+6.6%
life	0.962	0.995	-3.35%	173+175	173+172	0.0%+1.7%
logic	2.555	2.562	-0.26%	586+681	565+643	3.7%+5.9%
mandelbrot	0.112	0.098	13.56%	45+45	33+35	36.4%+28.6%
mlyacc	26.868	27.582	-2.59%	3453+4552	3302+4344	4.6%+4.8%
nucleic	5.213	5.122	1.79%	210+150	204+140	2.9%+7.1%
ratio-regions	20.483	13.485	51.90%	802+504	753+438	6.5%+15.1%
ray	1.703	1.753	-2.85%	363+344	354+337	2.5%+2.1%
simple	7.470	7.882	-5.22%	1480+1255	1438+1195	2.9%+5.0%
tsp	1.192	1.128	5.61%	241+201	223+199	8.1%+1.0%
vliw	26.672	26.123	2.10%	2340+3073	2291+2929	2.1%+4.9%
Average			3.28%			5.7%+6.5%

Table 5: Compilation time of **ra8+coal** versus **ra8+prop**

Table 6 compares the number of spill coalescing and spill propagation that is performed in the **ra8+coal** and **ra8+prop** schemes. On the average, spill propagation increases the number of spill coalescing by over 110%.

Table 7 compares the execution times of **ra+coal** and **ra+prop**. On the average the execution times improves by 5.5% in **ra+prop**. In some cases, such as lexgen, ratio-regions and ray, the execution times degrade slightly (less than 2%) with spill propagation. However, large improvements are obtained in benchmarks such as life and mandelbrot.

We also compare the schemes **ra32+ra8** and **ra8+prop**. Table 8 compares the compilation time and the amount of spill code in the two schemes. In almost all cases, **ra8+prop** is faster. This is expected since **ra32+ra8** has to perform allocation twice for each flowgraph. On the average, 11.5% percent of improvements are obtained. This improvement is not close to 50% since the ra32 phase is much cheaper than the ra8 since spilling is rare in the first phase. The amount of spill code is reduced by over 23%. However, this number is somewhat misleading since this only measures the number of times the spill and reload callbacks are invoked, and not the actual number of spill and reload instructions in the program. The **ra32+ra8** scheme performs targeting of spill locations within the callbacks and can in some cases remove spill and reload instructions.

Table 9 compares the runtime performance of **ra32+ra8** and **ra8+prop**. On the average, **ra8+prop** is faster than **ra32+ra8** by 5%. The benchmarks life and mandelbrot improve by over 20%. We believe that this improvement can be attributed solely to spill propagation; the same large amount of improvements

Name	Spill Coalescing Counts			Spill Propagation Counts	
	ra8+coal	**ra8+prop**	Improv.	**ra8+coal**	**ra8+prop**
barnesHut	276	690	150.00%	0	318
boyer	138	192	39.13%	0	54
count-graphs	312	672	115.38%	0	270
fft	330	504	52.73%	0	150
knuthBendix	480	900	87.50%	0	312
lexgen	570	1134	98.95%	0	534
life	66	162	145.45%	0	72
logic	576	1056	83.33%	0	390
mandelbrot	84	186	121.43%	0	78
mlyacc	1896	3816	101.27%	0	1674
nucleic	294	666	126.53%	0	348
ratio-regions	420	858	104.29%	0	366
ray	342	606	77.19%	0	234
simple	612	1806	195.10%	0	1044
tsp	90	180	100.00%	0	84
vliw	882	2328	163.95%	0	1284
Average			110.14%		

Table 6: Spill Coalescing in **ra8+coal** versus **ra8+prop**

Name	Execution Time (secs)		
	ra8+coal	**ra8+prop**	Improv.
barnesHut	2.838	2.833	0.18%
boyer	0.257	0.255	0.65%
count-graphs	22.117	22.065	0.23%
fft	0.865	0.792	9.26%
knuthBendix	0.803	0.770	4.33%
lexgen	0.815	0.818	-0.41%
life	0.112	0.082	36.73%
logic	4.595	4.370	5.15%
mandelbrot	0.605	0.518	16.72%
mlyacc	0.452	0.440	2.65%
nucleic	0.138	0.137	1.22%
ratio-regions	100.743	102.322	-1.54%
ray	2.725	2.737	-0.43%
simple	2.853	2.667	7.00%
tsp	6.628	6.377	3.95%
vliw	1.565	1.528	2.40%
Average			5.5%

Table 7: Execution time of **ra8+coal** versus **ra8+prop**

Name	Compilation Time (secs)			Spill+Reload		
	ra8+ra32	**ra8+prop**	Improv.	**ra8+ra32**	**ra8+prop**	Improv.
barnesHut	4.535	3.975	14.09%	732+897	627+781	16.7%+14.9%
boyer	17.125	12.697	34.88%	476+276	482+267	-1.2%+3.4%
count-graphs	1.695	1.482	14.40%	435+462	338+369	28.7%+25.2%
fft	0.972	0.957	1.57%	255+320	167+230	52.7%+39.1%
knuthBendix	4.707	4.083	15.27%	622+744	431+558	44.3%+33.3%
lexgen	9.162	9.083	0.86%	948+1004	797+806	18.9%+24.6%
life	1.090	0.995	9.55%	178+196	173+172	2.9%+14.0%
logic	2.888	2.562	12.75%	691+805	565+643	22.3%+25.2%
mandelbrot	0.128	0.098	30.51%	61+62	33+35	84.8%+77.1%
mlyacc	30.375	27.582	10.13%	4059+5156	3302+4344	22.9%+18.7%
nucleic	5.010	5.122	-2.18%	255+200	204+140	25.0%+42.9%
ratio-regions	13.717	13.485	1.72%	861+499	753+438	14.3%+13.9%
ray	1.882	1.753	7.32%	412+442	354+337	16.4%+31.2%
simple	8.325	7.882	5.62%	1544+1374	1438+1195	7.4%+15.0%
tsp	1.343	1.128	19.05%	251+237	223+199	12.6%+19.1%
vliw	28.367	26.123	8.59%	2520+3204	2291+2929	10.0%+9.4%
Average			11.51%			23.7%+25.4%

Table 8: Compile time of **ra32+ra8** versus **ra8+prop**

are obtained when comparing **ra8+coal** and **ra8+prop**.

However, some benchmarks are slower in **ra8+prop**. We believe this is because currently **ra8+prop** does not perform dead copy elimination, nor does it perform coalescing of pseudo memory registers in the program (it only coalesces spill locations.) Improving these aspects of **ra8+prop** can enhance its performance further.

Since the benchmark programs are rather small, we have conducted another round of tests in which we compile release 110.23 of the SML/NJ compiler using all three schemes [13]. The results are summarized in Table 10 and 11.

Comparing the results of **ra8+coal** and **ra8+prop**, spill propagation doubles the number of live-ranges that can be coalesced, and reduces the number of spills and reloads by 6.8% and 8.1%. Overall, the number of extra allocation rounds decreased by 3.4% when spill propagation is enabled.

Comparing the results of **ra32+ra8** and **ra8+prop**, the compilation time reduces by 15.7% when **ra8+prop** is used. However, the image size of increases slightly 1.18%. We again attribute this to the lack of dead copy elimination and the lack of pseudo memory register coalescing in **ra8+prop**.

In summary, spill propagation seems to be a very effective spill reduction strategy.

[13] The compilation time differs from results in the last section because different releases are used.

Name	Execution Time (secs)		
	ra8+ra32	**ra8+prop**	Improv.
barnesHut	2.977	2.833	5.06%
boyer	0.240	0.255	-5.88%
count-graphs	21.512	22.065	-2.51%
fft	0.882	0.792	11.37%
knuthBendix	0.743	0.770	-3.46%
lexgen	0.767	0.818	-6.31%
life	0.112	0.082	36.73%
logic	4.525	4.370	3.55%
mandelbrot	0.625	0.518	20.58%
mlyacc	0.423	0.440	-3.79%
nucleic	0.135	0.137	-1.22%
ratio-regions	99.547	102.322	-2.71%
ray	3.210	2.737	17.30%
simple	2.872	2.667	7.69%
tsp	6.620	6.377	3.82%
vliw	1.537	1.528	0.55%
Average			5.05%

Table 9: Execution time of **ra32+ra8** versus **ra8+prop**

	Extra Rounds Versus Total	Spills+Reloads	Coalescing Count	Propagation Count
ra8+coal	13623/46911	97642+126484	10892	0
ra8+prop	13170/46921	91395+116961	22967	10062
Improvement	3.4%	6.8%+8.1%	12075	
ra32+ra8	14137/93181	114683+143799	0	0
ra8+prop	13170/46921	91395+116961	22967	10062

Table 10: Compiling the Compiler

	Image Size	Compilation Time
ra8+coal	n/a	758.92
ra8+prop	905372	746.53
Improvement		1.7%
ra32+ra8	812188	863.69
ra8+prop	905372	746.53
Improvement	1.18%	15.7%

Table 11: Compilation time and image size

6 Acknowledgements

We thank Fermin Reig Galilea for reading and commenting on various drafts of this paper.

References

[App98] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, UK, 1998.

[BCT94] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM TOPLAS*, 16(3):428–455, 1994.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single assignment form. In *ACM SIGPLAN POPL*, pages 25–35, 1989.

[CH90] F. C. Chow and J. L. Hennessy. The priority–based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, 1990.

[CH98] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. *In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1998.

[ED95] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th International Workshop on Microprogramming and Microarchitecture*, pages 180–191, 1995.

[GA96] L. George and A. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3), May 1996.

[Geo97] L. George. Mlrisc: Customizable and reusable code generators (`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/index.html`). Technical report, Bell Laboratories, Murray Hill, 1997.

[Geo99] L. George. SMLNJ: Intel x86 back end Take [I] (`http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-nodes/k32.ps`). 1999.

[GG99] F. J. Reig Galilea and L. George. An implementation of C– using MLRISC `http://www.dcs.gla.ac.uk/ reig/c--/cmmc.ps`. 1999.

[JNON97] S. L. Peyton Jones, T. Nordin, D. Oliva, and P. Nogueira. The C– manual. available from `http://www.research.microsoft.com/users/simonpj/c--.html`. 1997.

[JON98] S. L. Peyton Jones, D. Oliva, and T. Nordin. C–: A portable assembly language. *In Proceedings of the 1997 Workshop on Implementing Functional Languages. Springer Verlag LNCS*, 1998.

[JS96] R. Johnson and M. Schlansker. Analysis of predicated code. In *Micro-29, International Workshop on Microprogramming and Microarchitecture*, 1996.

[LG99] A. Leung and L. George. Static single assignment form for machine code. *ACM SIGPLAN PLDI*, 1999.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Micro-25*, pages 45–54, 1992.

[Sit92] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

A Using the New Allocator

The new allocator has the signature `RA` defined in Figure 9.

The first parameter is a list of callbacks and flags which determine how to perform the register allocation:

`cellkind` is the cell kind of the registers to be allocated. For example, if we want to allocate general purpose integer registers, we use `GP`; for floating point, we use `FP`.

`spillProh` is a list of ranges of registers that are prohibited from spilling.

> User Hints: Originally, the purpose of `spillProh` is to tell the allocator that certain registers only appear in very short live-ranges and thus should not be spilled. In the new allocator, however, these type of live-ranges are automatically detected. So specifying `spillProh` is usually unnecessary, and can be left as the empty list. □

`K` is the number of colors available.

`dedicated` is an array of booleans indicating whether a register is dedicated. The array should be of length $r + 1$, where r is the maximum cell id of the physical register in question. All references to dedicated registers are removed from consideration during register allocation. Typically the stack pointer, the frame pointer, assembly temporary register, and the zero register (if there are any) should be specified as dedicated.

`getreg` is a function for computing a new color. This function can be obtained by specializing the functor `GetReg`.

```
signature RA = sig
   structure I : INSTRUCTIONS
   structure C : CELLS
   structure F : RA_FLOWGRAPH
   structure G : RA_GRAPH = RAGraph
      sharing F.I = I
      sharing I.C = C

   type getreg = ...
   type mode = G.mode

   val NO_OPTIMIZATION   : mode
   val DEAD_COPY_ELIM    : mode
   val BIASED_SELECTION  : mode
   val SPILL_COLORING    : mode
   val SPILL_COALESCING  : mode
   val SPILL_PROPAGATION : mode
   val COPY_PROPAGATION  : mode

   type raClient =
   { cellkind     : C.cellkind,
     spillProh    : (C.cell * C.cell) list,
     K            : int,
     dedicated    : bool Array.array,
     getreg       : getreg,
     copyInstr    : F.Spill.copyInstr,
     spill        : F.Spill.spill,
     spillSrc     : F.Spill.spillSrc,
     spillCopyTmp : F.Spill.spillCopyTmp,
     reload       : F.Spill.reload,
     reloadDst    : F.Spill.reloadDst,
     renameSrc    : F.Spill.renameSrc,
     mode         : mode
   }

   val ra : raClient list -> F.flowgraph -> F.flowgraph
end
```

Figure 9: Signature of the allocator

`copyInstr` is a function for constructing a parallel copy from an existing parallel copy.

```
val copyInstr : (C.cell list * C.cell list) * I.instruction
                  -> I.instruction
```

`spill, spillSrc, spillCopyTmp` are the client callbacks for inserting spill code. The function `spill` has the following signature.

```
val spill :
   {instr       : I.instruction,
    reg         : C.cell,
    spillLoc    : int,
    kill        : bool,
    regmap      : C.cell -> C.cell,
    annotations : Annotations.annotations ref
   } ->
   {code   : I.instruction list,
    proh   : C.cell list,
    newReg : C.cell option
   }
```

Here, `instr` is the instruction where the spill has occurred, `reg` is the register that is being spilled, `spillLoc` is the *logical spill location* of the spilled live-range, `kill` is a flag indication whether the definition of `reg` can be killed, `regmap` is the current allocation map, and `annotations` is the list of client-defined annotations in the basic block where `instr` resides.

The client is responsible for mapping the logical spill location to the physical spill slot in the client's stack frame, and insert the appropriate spill code using this spill slot. This callback should return `code`, a list of spill code; `proh`, a list of registers prohibited from further spilling; `instr`, the spilled instruction, and `newReg`, an optional new pseudo register containing the spilled value[14].

The function `spillSrc` has the following signature.

```
val spillSrc :
   {src         : C.cell,
    reg         : C.cell,
    spillLoc    : int,
    annotations : Annotations.annotations ref
   } -> I.instruction list
```

[14]In the new allocator, it is possible for different registers to be mapped into the same logical spill location (and thus the same physical spill slot). The `spillLoc` parameter, and not `reg` must be used for selecting the correct spill slot.

It is responsible for inserting an instruction writing the value in register `src` to `spillLoc`.

The function `spillCopyTmp` has the following signature.

```
type spillCopyTmp =
   {copy     : I.instruction,
    spillLoc : int,
    annotations : Annotations.annotations ref
   } -> I.instruction
```

This function is dedicated to spilling the temporary associated with `copy` to `spillLoc`.

`reload, reloadDst, renameSrc` are the client callbacks for inserting reload code. The parameters in `reload` have similar interpretation as the `spill` function.

```
val reload :
   {instr       : I.instruction,
    reg         : C.cell,
    spillLoc    : int,
    regmap      : C.cell -> C.cell,
    annotations : Annotations.annotations ref
   } ->
   {code   : I.instruction list,
    proh   : C.cell list,
    newReg : C.cell option
   }
```

The function `reloadDst` is the dual of `spillSrc`; it is responsible for reading the value from `spillLoc` to the register `dst`.

```
val reloadDst :
   {dst      : C.cell,
    reg      : C.cell,
    spillLoc : int,
    annotations : Annotations.annotations ref
   } -> I.instruction list
```

The function `renameSrc` is used for renaming all uses of register `fromSrc` in `instr` to `toSrc`.

```
val renameSrc :
   {instr    : I.instruction,
    fromSrc  : C.cell,
    toSrc    : C.cell,
```

```
 regmap   : C.cell -> C.cell
} ->
{code     : I.instruction list,
 proh     : C.cell list,
 newReg   : C.cell option
}
```

`mode` is a flag indicating which addition optimization phases described in Section 2 should be included in the algorithm. These are:

- DEAD_COPY_ELIM – enable the dead copy elimination phase
- BIASED_SELECTION – uses Briggs' biased selection during the select phase.
- SPILL_COLORING – enable the spill coloring phase.
- SPILL_COALESCING – enable the spill coalescing phase.
- SPILL_PROPAGATION – enable the spill coalescing and propagation phases.
- COPY_PROPAGATION – when this is set, the allocator stops when all moves that can be coalesced without spilling are coalesced.
- NO_OPTIMIZATION – this is the default mode.

A.1 Specializing the Allocator

The functor `RegisterAllocator` implements the signature `RA`. This (curried) functor is parameterized with respect to the spilling heuristics module, which has the signature `RA_SPILL_HEURISTICS`; and the abstract flowgraph module, which has the signature `RA_FLOWGRAPH`. The spilling heuristics module implements the **potential spill** phase of the algorithm, while the flowgraph module is responsible for implementing phases **build**, **actual spill** and **rebuild**.

```
functor RegisterAllocator
  (SpillHeuristics : RA_SPILL_HEURISTICS)
  (Flowgraph : RA_FLOWGRAPH) : RA
```

The default spilling heuristics module is `ChaitinSpillHeur`, which implements the modified Chaitin's heuristic mentioned in Section 2. An alternative module is `ChowHennessySpillHeur`, which implements the priority function described in allocation framework of Chow and Hennessy [CH90]. The default abstract flowgraph module specialized to the *cluster* representation (see [Geo97]), is `ClusterRA`.

For example, here is how the functor `RegisterAllocator` can be specialized for the Alpha architecture [Sit92]:

```
structure Ra =
  RegisterAllocator
    (ChaitinSpillHeur)
    (ClusterRA
```

```
    (structure Flowgraph = AlphaFlowGraph
     structure Asm = AlphaAsmEmitter
     structure InsnProps = AlphaInsnProps
    )
  )
```

Structures `AlphaFlowGraph`, `AlphaAsmEmitter` and `AlphaInsnProps` are the flowgraph, assembly emitter, and machine instruction properties modules specialized to the Alpha instruction set.

Below we specialize two round robin allocators for integer and floating point registers. These are called `GR` and `FR`. We also initialize the arrays of dedicated integer (`dedicatedR`) and floating points registers (`dedicatedF`).

```
local
   val {low,high} = Cells.cellRange Cells.GP
in
   structure GR = GetReg(val first=low val nRegs=high-low+1
                         val available= list of available registers)
   val dedicatedR = Array.array(high+1,false)
   val _ = app (fn r => Array.update(dedicatedR,r,true) [sp,fp,at,...]
end
local
   val {low,high} = Cells.cellRange Cells.FP
in
   structure FR = GetReg(val first=low val nRegs=high-low+1
                         val available= list of available registers)
   val dedicatedF = Array.array(high+1,false)
   val _ = app (fn r => Array.update(dedicatedF,r,true) [...]
end
```

The following instructions are used to construct parallel copy instructions given a pair of destinations and sources. `copyR` is used for integer copies, while `copyF` is used for floating point copies.

```
fun copyR((rds as [_], rss as [_]), _) =
    I.COPY{dst=rds, src=rss, impl=ref NONE, tmp=NONE}
  | copyR((rds, rss), I.COPY{tmp, ...}) =
    I.COPY{dst=rds, src=rss, impl=ref NONE, tmp=tmp}
fun copyF((fds as [_], fss as [_]), _) =
    I.FCOPY{dst=fds, src=fss, impl=ref NONE, tmp=NONE}
  | copyF((fds, fss), I.FCOPY{tmp, ...}) =
    I.FCOPY{dst=fds, src=fss, impl=ref NONE, tmp=tmp}
```

We can instantiate the module for renaming registers on the Alpha as follows.

```
structure Rewrite = AlphaRewrite(AlphaInstr)
```

Next we define the spill and reload callbacks. In the actual spill phase, the allocator calls the spill callback to insert a spill and the reload callback to insert a reload. The integer spill callback can be defined as follows:

```
fun spillR{annotations, kill=true, regmap, reg, spillLoc, instr} =
    if instr  defines only reg  and has no side effect
    then {code=[], proh=[], newReg=NONE}
    else spillR{annotations=annotations, kill=false, regmap=regmap,
                reg=reg, spillLoc=spillLoc, instr=instr}
  | spillR{annotations, kill, regmap, reg, spillLoc, instr} =
    let val loc =  find the spill location for spillLoc
        val newR = Cells.newReg()
        val instr' = Rewrite.rewriteDef(regmap, instr, reg, newR)
    in {code=[I.STORE{stOp=I.STQ, b=sp, d=I.IMMop loc, r=newR, ...},
              instr'], proh=[newR], newReg=SOME newR
        }
    end
```

The function spillR first checks if the instruction being spilled is dead. If it is, and if it only defines the spilled reg and is an instruction without side effect, then we can remove the instruction from the flowgraph. Otherwise, we find a spill location for the logical spill location spillLoc. A typical implementation can use a hash table to keep track of the mapping between logical and physical spill locations. The definition of reg is rewritten with a new temporary newR, and insert the store instruction after the rewritten instruction. We also return newR in he list proh so that the allocator will avoid spilling newR is subsequent rounds.

To spill the temporary associated with a copy, the function spillCopyTmp is used. This can be defined as follows.

```
fun spillCopyTmpR{copy=I.COPY{dst, src, impl, ...},
                  spillLoc, annotations} =
    I.COPY{dst=dst,src=src,impl=ref NONE,tmp=
            I.Displace{base=sp, disp= spill location for spillLoc}
          }
```

The function spillSrcR is defined as follows.

```
fun spillSrcR{src, reg, spillLoc, annotations} =
    let val loc =  find the spill location for spillLoc
    in  [I.STORE{stOp=I.STQ, b=sp, d=I.IMMop loc, r=src, ...}]
    end
```

The reload callbacks, reloadR, reloadDstR and renameSrcR are defined as follows.

```
fun reloadR{annotations, regmap, reg, spillLoc, instr} =
let val loc =  find the spill location for spillLoc
    val newR = Cells.newReg()
    val instr' = Rewrite.rewriteUse(regmap, instr, reg, newR)
in  {code=[I.LOAD{ldOp=I.LDQ, b=sp, d=I.IMMop loc, r=newR, ...},
           instr'],
```

```
      proh=[newR]
      newReg=SOME newR
    }
end

fun reloadDstR{annotations, reg, dst, spilllLoc} =
let val loc =  find the spill location for spillLoc
in  [I.LOAD{ldOp=I.LDQ, b=sp, d=I.IMMop loc, r=dst, ...}]
end

fun renameSrcR{regmap, fromSrc, toSrc, instr} =
let val instr' = Rewrite.rewriteUse(regmap, instr, fromSrc, toSrc)
in  {code=[instr'], proh=[], newReg=SOME toSrc}
end
```

The spill and reload callbacks for floating point registers can be analogously defined. We can integrate everything together as the function `ra`.

```
fun ra cluster =
   ( initialize spill location map;
    Ra.ra
      [{ cellkind      = AlphaCells.GP,
         spillProh     = [],
         K             =  number of available integer registers,
         dedicated     = dedicatedR,
         getreg        = GR.getreg,
         copyInstr     = copyR,
         spill         = spillR,
         spillSrc      = spillSrcR,
         spillCopyTmp  = spillCopyTmpR,
         reload        = reloadR,
         reloadDst     = reloadDstR,
         renameSrc     = renameSrcR,
         mode          = Ra.NO_OPTIMIZATION
       },
       { cellkind      = AlphaCells.FP,
         spillProh     = [],
         K             =  number of available fp registers,
         dedicated     = dedicatedF,
         getreg        = FR.getreg,
         copyInstr     = copyF,
         spill         = spillF,
         spillSrc      = spillSrcF,
         spillCopyTmp  = spillCopyTmpF,
         reload        = reloadF,
         reloadDst     = reloadDstF,
         renameSrc     = renameSrcF,
```

```
        mode          = Ra.NO_OPTIMIZATION
      }] cluster
  )
```

A.2 Redefining the Spill Cost Function

```
signature RA_SPILL_HEURISTICS = sig
   structure G : RA_GRAPH = RAGraph

   exception NoCandidate

   val init : unit -> unit

   val chooseSpillNode :
       { graph          : G.interferenceGraph,
         spillWkl       : G.node list,
         hasBeenSpilled : int -> bool
       } ->
       { spillWkl : G.node list,
         node     : G.node option,
         cost     : real
       }
end
```

Figure 10: Computing spill heuristics

Spill cost computation modules are given the signature presented in Figure 10. The function `chooseSpillNode` are given a set of spill candidates, and a function whether a node has already been spilled. It returns a new spill `node`, the new spill worklist `spillWkl` with the node removed, and the cost of the spill node.

A.3 Customizing the Flowgraph

It is possible to customize the new allocator to use different program representations. The signature `RA_FLOWGRAPH` shown in Figure 11 defines an abstract view a flowgraph required by the new register allocator.

In order to allow different representations to share the same register allocator core, each representation should implement a set of *services*. These are:

`regmap` – retrieves the regmap from the flowgraph representation.

`dumpFlowgraph` – dumps out the flowgraph into textual form. This is used for debugging purposes.

`build` – this method implements the **build** phase of the algorithm.

```
signature RA_FLOWGRAPH = sig

   structure I     : INSTRUCTIONS
   structure C     : CELLS
   structure G     : RA_GRAPH = RAGraph
   structure Spill : RA_SPILL
   structure W     : FREQ
     sharing Spill.I = I
     sharing I.C = C

   type flowgraph

   val regmap : flowgraph -> C.regmap

   val dumpFlowgraph :
        string * flowgraph * TextIO.outstream -> unit

   val services : flowgraph ->
       { build   : G.interferenceGraph * C.cellkind->
                      G.move list,
         spill   : {copyInstr : Spill.copyInstr,
                    spill     : Spill.spill,
                    reload    : Spill.reload,
                    graph     : G.interferenceGraph,
                    nodes     : G.node list,
                    cellkind  : C.cellkind
                   } -> G.move list,
         freq    : G.programPoint -> W.freq
       }
end
```

Figure 11: Abstract flowgraph interface

`spill` – this method implements the **actual spill** and **rebuild** phase of the algorithm.

`freq` – this method extracts the execution frequency given a program point.

A.4 Flags and Counters

The following flags and counters are available from the `MLRiscControl` structure.

`dump-cfg-before-ra` prints the control flowgraph before register allocation.

`dump-cfg-after-ra` prints the control flowgraph after register allocation

`dump-cfg-after-spilling` prints the control flowgraph after spilling.

`ra-debug-spilling` prints out the nodes that are spilled and their associated cost

`ra-count` counts the number of times the register allocator is invoked.

`ra-rebuild` counts the number of times the **rebuild** phase is executed.

`ra-dead-code` if this is set, we also count the number of dead variables in the program.

`ra-count-dead-code` counts the number of dead variables.

`ra-spill-coalescing` counts the number of spill locations merged by spill coalescing.

`ra-spill-propagation` counts the number of live-ranges spilled by spill propagation.

In the SML/NJ compiler, the following counters are also available:

`ra-int-spills` counts the number of integer spills

`ra-int-reloads` counts the number of integer reloads

`ra-float-spills` counts the number of floating point spills

`ra-float-reloads` counts the number of floating point reloads