Register Allocation via Hierarchical Graph Coloring

David Callahan Brian Koblenz

Tera Computer Company
400 N 34th Street, Suite 300
Seattle, WA 98103 *

Abstract

We present a graph coloring register allocator designed to minimize the number of dynamic memory references. We cover the program with sets of blocks called tiles and group these tiles into a tree reflecting the program's hierarchical control structure.

Registers are allocated for each tile using standard graph coloring techniques and the local allocation and conflict information is passed around the tree in a two phase algorithm. This results in an allocation of registers that is sensitive to local usage patterns while retaining a global perspective. Spill code is placed in less frequently executed portions of the program and the choice of variables to spill is based on usage patterns between the spills and the reloads rather than usage patterns over the entire program.

1 Introduction

We examine the problem of efficiently allocating registers to hold program variables and compiler temporaries. In this problem, a program is represented as a control flow graph consisting of basic blocks connected with edges representing possible transfers of control. Each basic block consists of a sequence of instructions accessing variables. The target machine has a finite set R of physical registers and an unbounded set M of memory locations. Each reference to a program variable must be associated with either a physical register or a memory location during compilation. The goal of the register allocator is to minimize the number of dynamic memory references in the program by placing heavily used variables in registers.

Most of the work in recent years has cast the register allocation problem in terms of coloring a graph where the nodes represent variables and the edges represent conflicts. Two variables in the graph are connected if they cannot simultaneously share a register at some point in the program. The goal of the allocator is to assign a register ("color") to every node such that each node has a different color than any of its neighbors.

When it is impossible to color every node differently from its neighbors then some form of spilling is required.[1] The placement of code to implement spill decisions has not received as much attention as the question of which variables to spill. Even though newer processor designs have more registers, we believe that the appropriate placement of spill code will become more important in the near future. The reason for this is that these processors have longer pipelines, longer latencies between the time an operation is issued and the time the result is available, and more operations executing concurrently. In order to find these concurrent operations and hide the pipeline latency, aggressive loop unrolling and operation scheduling are required, both of which increase register pressure at various points in the program[5].

In Chaitin's allocator[6] the decision to spill a variable is based on a weighted reference count and the number of conflict edges in the interference graph. This heuristic suffers because the program flow structure is not represented in the interference graph and local reference patterns are not visible. In addition,

*This research was supported by the United States Defense Advanced Research Projects Agency under Contract MDA972-89-C-0002. The views and conclusions contained in this document are those of Tera Computer Company and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U. S. Government.

[1] A variable is *spilled* over a section of the program when references to that variable are associated with memory rather than registers. The term *reload* is used to indicate a transition point in the program where a spilled variable becomes associated with a register and the term *spill* refers to the opposite transition.

once a variable is spilled to memory all references must fetch its value from memory and all definitions must store the value back. Simple methods within a basic block[2][6] can be used to avoid multiple loads (or stores) of a variable if the register has not been used since the previous load, but no global mechanism allows the variable to be allocated a register over some larger portion of the program. In the example shown in Figure 1, Chaitin's allocator will spill either $g1$ or $g2$ for the entire program resulting in the poor execution of one of the loops.

Our work describes a graph coloring allocator that is sensitive to program flow structure. Spilling occurs in less frequently executed portions of the program. The choice of variables to spill is based on usage patterns between the spills and the reloads rather than usage patterns over the entire program. The method allows a variable to be assigned to one register over a portion of the program, memory in a second portion, and a different register in yet a third portion. Profiling information can be trivially incorporated to improve the selection of spilled variables and the location of the spill code because all analysis is based on the probability of being in a particular basic block or flowing along a particular control flow edge.

The main idea is to represent the program's loop and conditional structure by a tree of *tiles*. Tiles are visited in a bottom up fashion and a local interference graph is created and colored (using pseudo registers) for each tile. A tile's local spill decisions are made based on local usage and a compact summary of the local interference graph is passed to the parent tile to be incorporated into its interference graph. After the bottom-up pass has allocated variables to pseudo registers for the entire tree, a top down walk binds pseudo registers to physical registers and introduces spill code where desirable and required, but not necessarily where the decision to spill was made.

In addition to better spill code placement, our approach also allows smaller conflict graphs to be constructed. We are not claiming to be asymptotically smaller, but with this technique it is not necessary to construct the full conflict graph at any one time. This is similar to the benefit of clique separators described by Gupta, Soffa, and Steele[11] yet we are able to retain a global view of the program permitting better spill analysis.

The next section describes the tile tree. Section 3 gives details for the allocation and section 4 describes how spill decisions are made. We conclude with comparisons to other work and some observations about the generality of our approach.

2 Tiles and Tile Trees

Our goal is to represent the hierarchical structure of a program as a tree because the tree representation is easy to reason about and separates areas of high and low execution frequency.

We start with some definitions:

A **program** is represented by a control flow graph $G = (B, E, start, stop)$ where B is the set of basic blocks, E is the set of control flow edges between elements of B, $start \in B$ is the unique block with no predecessors and $stop \in B$ is the unique block with no successors.

Let T be a collection of sets of basic blocks which covers the set B. We say that T is a **tile tree** and each element of T a **tile** if the following conditions hold:

1. Each pair of sets in T are either disjoint or one is a proper subset of the other. If $t_2 \subset t_1$ and there is no $t \in T$ such that $t_2 \subset t \subset t_1$, then we say that t_2 is a **child** or a **subtile** of t_1 and t_1 is the **parent** of t_2, denoted by $t_1 = parent(t_2)$. We also define the set $blocks(t)$ to be the set of basic blocks which are members of t but which are not members of any child of t.

2. For each edge $e = \langle n, m \rangle \in E$ and tile t such that $m \in t$, we have $n \in t$ or $n \in blocks(parent(t))$. If $n \in blocks(parent(t))$ we say that e is an **entry edge**.

3. For each edge $e = \langle m, n \rangle \in E$ and tile t such that $m \in t$, we have $n \in t$ or $n \in blocks(parent(t))$. If $n \in blocks(parent(t))$ we say that e is an **exit edge**.

4. There is some tile t_0 such that $blocks(t_0) = \{start, stop\}$ This tile is called the **root tile**.

The first restriction is central to our hierarchical approach to making spill decisions. The second and third are somewhat technical and we observe that empty basic blocks can be inserted along an edge to allow the original endpoints to be further "apart" in the tile tree. Intuitively each empty block becomes a point where spill code can be inserted if needed. The final condition ensures a simple boundary case.

Many collections T satisfy the above definition including the trivial tree consisting of two tiles, one of which is the root tile. Our goal is to make the tile tree represent the structure of the program so we combine a couple of heuristics to build the tree from the control flow graph. The algorithm to construct the tile tree is given in appendix A.

Figure 1 shows the tile tree resulting from the program in figure 1 superimposed on the new control flow

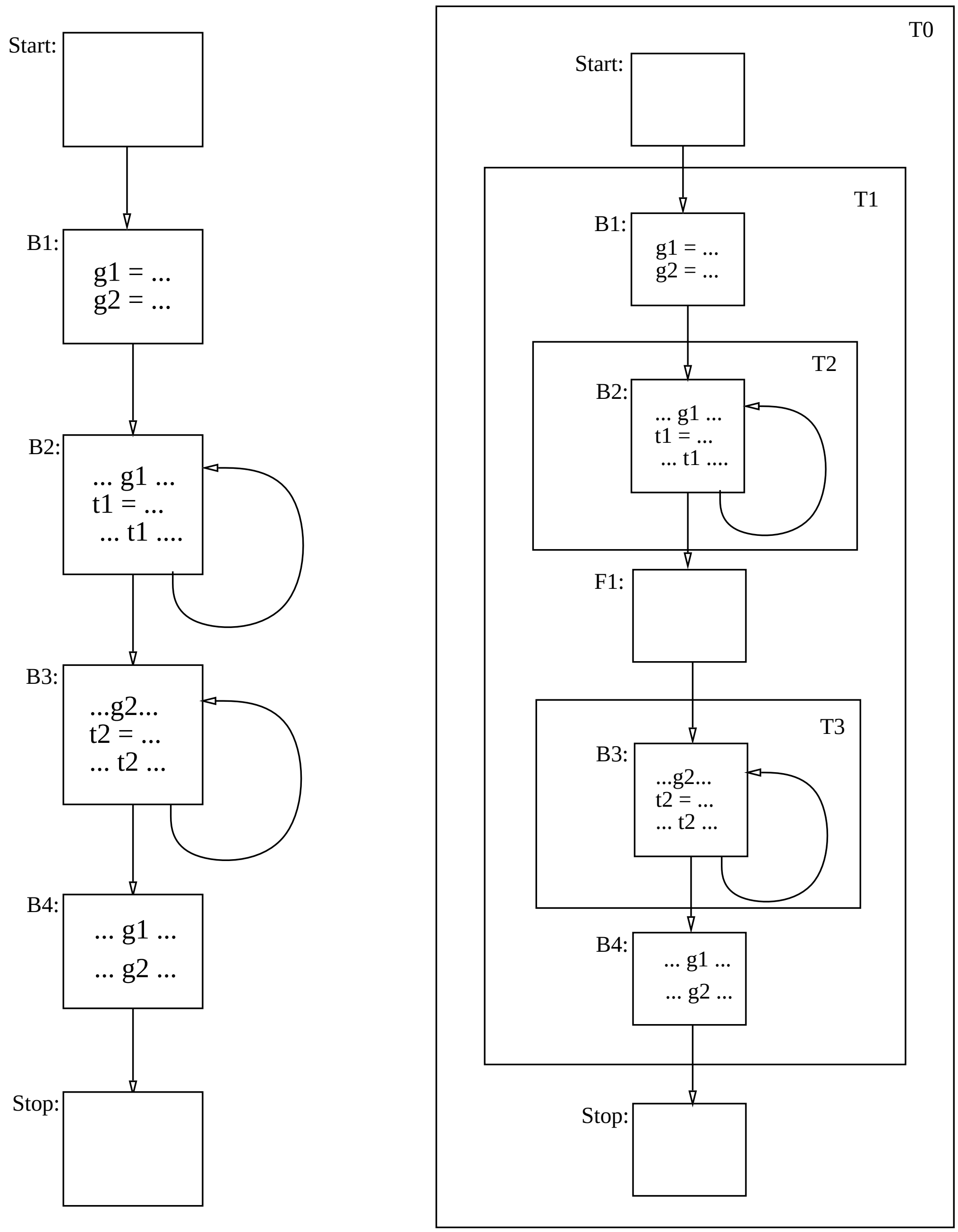

Figure 1: Example allocation problem involving two loops, four variables and a two-register machine. Optimal allocation requires g_2 to be spilled before block B_2 and reloaded before B_3. g_1 should be spilled after B_2. The right hand figure shows the hierarchical tile tree and the basic block inserted during tree construction.

graph. The block labeled `F1` is generated by the fix-up code in figure 3.

Though representing only the loop hierarchy may suffice in many cases, we include both loops and conditionals in our hierarchy. By including the conditionally executed portions of the program the size of the interference graphs are further reduced and the placement of spill code is improved.

For example, consider a variable v that is used inside a deeply nested conditional that is rarely executed. There may not be enough register pressure to cause the variable to spill until higher in the tile tree, but the point higher in the tree is executed more frequently than the conditional where the variable is used. This corresponds to the case where it is desirable to spill a variable inside a conditional lower in the tile tree than is necessary.

Alternatively, consider a pair of nested loops and a variable v that cannot be allocated a register for the inner loop (represented by tile t). It is possible to spill inside of the outer loop (corresponding to spilling along t's entry and exit edges), but if there are no references to v in the outer loop it is better to spill the variable outside of the outer loop, in a tile still higher in the tree.

3 Allocation

Each basic block is populated with uses and definitions of **variables**. We assume that each program variable has been fully renamed[9] and that variables correspond to an unbounded set of pseudo-registers.[2]

We say a variable v is **local** to a tile t if all references to v are made by blocks within t and v is not live along any entry or exit edge of t. Variables referenced in a tile but not local to the tile are called **global** with respect to that tile.

The allocation process has two phases. In the first phase, physical and pseudo registers are allocated in a bottom-up fashion for each tile using graph coloring. Local spill decisions are made and a summary of the tile's allocation is passed to its parent. Once the entire tile tree has had pseudo registers allocated, the second phase makes a top-down walk of the tile tree mapping pseudo registers to physical registers and updating spill decisions. We distinguish **physical** and **pseudo** registers in that a pseudo register will be bound to some physical register during the second phase. When certain values must be in particular physical registers, *e.g.* to satisfy linkage conventions, those variables are assigned to the appropriate physical registers during the first phase.

[2]Renaming is not required but allows a variable with distinct live ranges within a tile to receive distinct registers for each live range.

phase1(tile t)
    foreach subtile s of t **do**
        phase1(s)
    endfor
    compute conflicts based on references in *blocks(t)*
    add preferences based on references in *blocks(t)*
    foreach subtile s of t **do**
        incorporate s's tile summary variable conflicts
        foreach variable g global in s that is also in a register in s **do**
            incorporate g's conflicts
        endfor
        add preferences from s
    endfor
    color tile interference graph
    update spill information
    save conflict and preference information
        for *parent(t)*

phase2(tile t)
    reconstruct interference from global variables and
        tile summary variables
    include global variables in registers in *parent(t)*
    set preferences based on allocation in *parent(t)*
    color interference graph using physical registers
    save allocation for each subtile s of t
    foreach subtile s of t **do**
        phase2(s)
    endfor

Figure 2: Overview of the two passes for register allocation. The initial call passes the root tile.

Figure 2 gives a brief algorithmic overview of the allocation process. The steps are described in more detail in the succeeding sections.

Allocation and Conflict Summary After the first phase has processed a tile, each variable referenced in the tile will be either assigned a physical or pseudo register, or be spilled to memory. The first phase summarizes the allocation of local variables by creating one new variable for each distinct register to which a local variable is allocated. These new variables are referred to as **tile summary variables** and we define the mapping $ts_t(v)$ which maps each local variable v in t to the tile summary variable associated with the register allocated to v. Tile summary variables represent the coalescing of local variables that were allocated to the same register and allow an efficient representation of the conflicts between local and global variables.

The number of tile summary variables for any given tile is bounded by the number of physical registers $||R||$ and the conflicts with other tile summary variables can be represented with a small bit matrix.

For each global variable g that was assigned a register in tile t, we retain two sets of conflicting variables. The first set, $c_t(g)$, describes the local variables in t that conflict with g. If L_t is the set of variables local to tile t and $X(v_1, v_2)$ is the relation that v_1 and v_2 conflict then we define:

$$c_t(g) = \{ts_t(v) | v \in L_t \text{ and } X(v, g)\}$$

The second set of conflicts describes the global variables in t that both conflict with g and were assigned to registers in t. We observe that at most $||R||$ global variables can be in registers at any entry or exit point to the tile and so the total amount of information in the summary is $O((x_t \cdot ||R||)^2)$ where x_t is the number of blocks which are the destination of entry edges to tile t or sources of exit edges from tile t. For structured programs, this number is 2.

Tile Interference Graph For each tile processed in a bottom-up manner, an interference graph is built. The interference graph for tile t consists of two kinds of variables: variables referenced in t, and tile summary variables for each child of t. The first kind of variable corresponds to the typical uses and definitions found in t, while the second kind of variable corresponds to the registers used by variables local to a subtile of t.

A subtle implication is that variables that are live but not referenced in t do not require a variable in t's interference graph. The motivation here is that if any variable must be spilled, these variables would be the first candidates. Omitting unreferenced live variables also provides a reduction in the size of the conflict graphs. An example of this is in figure 1 where tile $T2$ does not need to represent g_2 in its interference graph.

Edges in the graph represent conflicts between variables and prevent two variables from being assigned the same register. These conflict edges come from a variety of sources.

1. A conflict edge exists between two variables if they conflict in a block in $blocks(t)$. This initial conflict graph is built ignoring the subtile information using an algorithm similar to Chaitin[6].
2. A variable conflicts with tile summary variables and other global variables as indicated in the conflict summaries for each subtile.
3. Any variable that is live in a subtile but was not part of the subtile's allocation summary, conflicts with all of the subtile's tile summary variables. This includes variables that are not referenced anywhere in the subtree rooted at the subtile plus variables that have been spilled in the subtile but are live in the current tile. A second savings in graph size occurs because variables live across a tile normally conflict with all local variables in the tile. These conflicts are more compactly represented as conflicts with the tile summary variables which represent a set of variables local to the subtile.
4. Each node corresponding to a tile summary variable conflicts with other tile summary variables from the same subtile as indicated in the conflict summaries for each subtile. Tile summary variables do not conflict with tile summary variables from sibling tiles because tile summary variables in sibling tiles correspond to variables with non-overlapping live ranges.

Coloring Once the interference graph has been constructed it is colored using a standard heuristic: all variables with less than $||R||$ conflicts are placed on a colorable stack along with their edges. When a variable is placed on the stack all associated edges are removed from the graph enabling other variables to now be placed on the stack.

When the remaining graph consists solely of variables with at least $||R||$ conflicts, spill analysis is used to determine the next variable to push on the colorable stack. Variables that are least valuable for keeping in a register — as determined in section 4 — are pushed on the colorable stack next. Eventually every variable is on the colorable stack as suggested by Briggs *et al.*[3].

At this point the assignment of physical and pseudo registers begins. A set of colors is maintained consisting of physical and pseudo register colors assigned to variables. The set is initialized to contain the physical registers that certain local variables require.

The coloring process pops variables from the colorable stack and gives the variable a color from the color set excluding any color already allocated to a conflicting variable. If less than $||R||$ colors have been used and the variable requires a new color then a pseudo register is selected and added to the set of used colors.

An exception is made for global variables live at the tile boundaries and those variables that are preferenced to them (see below). In order to avoid overly constraining these variables, we attempt to select a color that is separate from any other color already used subject to the constraint of using only $||R||$ colors. Because we keep a summary of the tile's conflicts, it is possible to bind these distinct local colors to the same physical register during the top down phase. Alternatively, if we had coalesced local and global variables during the bottom up phase, it would be impossible to separate them even if it was desirable. An obvious implication of the preceeding discussion is that when coloring a local variable we try to avoid colors that are allocated to uncoalesced global variables.

Since the algorithm may place a variable with $||R||$ or more conflicts on the colorable stack, some variables on the stack may not receive a color. However, due to the stack ordering of variables, it is guaranteed that the more important variables, based on the spill analysis, will receive registers before the less important ones.

After all variables have either been given color assignments or spilled, we compute the tile conflict summary described earlier for variables still in a register.

Preferencing It is sometimes desirable to allocate different variables to the same register. Sources of these preferences include:

1. If a variable is used as a procedure call argument or result and the linkage convention requires the argument or result be in a particular physical register, then the variable becomes preferenced to that register.
2. If there is a simple assignment from one variable to another, than those variables become preferenced.

Explicit preferencing is used as an alternative to *coalescing*[6] where nodes satisfying the above conditions, especially the last, would simply be subsumed into a single node before coloring. We feel our method is superior because the separate live ranges of the individual variables allow more precise spill decisions to be made.

The above cases are different in that the first case has specific register requirements, while the second case simply desires to have two variables in the same, arbitrary register. We handle these cases separately.

To implement the first case of preferencing we associate an optional *local preference* register with each variable. When coloring a variable which has a local preference, if the desired register is available, then that register is assigned; otherwise the preference is ignored. When coloring a variable without a local preference, a color is found that does not conflict with either the already colored conflicting variables or with the local preferences of still uncolored conflicting variables. If no such color exists, we revert to standard coloring techniques choosing a color which is distinct from already colored conflicting variables. This mechanism prevents a variable from arbitrarily choosing a register that is the local preference of a conflicting variable. The only time a local preference will not be awarded is if a higher priority variable (based on the order of the coloring stack) desires the preferenced register, or every available register is some other variable's local preference.

The second type of preferencing described above does not set a local preference value because there is no particular register the two variables must share. Instead, each variable is added to the other's *preference list*. When a variable v is colored, each uncolored variable on v's preference list is given v's color as its local preference. Thus, after one variable in a group of preferenced variables is colored the mechanism described in the previous paragraph will then work to keep the register available for the other member's of the group.

In order to support inter-tile preferencing and register targeting a couple of special cases are implemented:

1. If a global variable is allocated to a physical register in a subtile, then that physical register becomes a local preference in the parent tile. This propagates preferences up the tile tree and allows earlier definitions of a variable to target a desirable location.
2. If two global variables are preferenced in a subtile and are allocated to the same pseudo register (the preferencing was successful in the subtile), then the pair is added to a list of pairs of variables that should be preferenced in the parent tile. When the parent tile's interference graph is constructed the variables on this propagated preferences list are preferenced to one another.
3. We also add preferences between global variables and tile summary variables if there is a local variable that was preferenced with the global variable and both variables are assigned the same color. This handles the case where there is a preference between the global and a

local associated with the tile summary variable without the need to retain additional preference information between phases.

Mapping pseudo registers to physical registers When the root of the tile tree is colored, the final assignment of pseudo registers to physical registers occurs in a top-down fashion. At each tile, the parent will have already placed certain global variables into registers and assigned some tile summary variables to registers. Other global variables will be spilled to memory and some tile summary variables will also be spilled to memory.

To make the final register assignment, the tile conflict graph is recreated from the summary information and is colored based on current preferences. Since, during the bottom-up walk, we ignored global variables that were not referenced in the subtree rooted at the current tile, we must now include those variables that received registers in the parent tile and are live across the subtile. We make these variables conflict with every other variable in the conflict graph and preference them to the physical register they received in the parent.

Variables with physical preferences are assigned to those physical registers. Global variables assigned to a register in the parent are preferenced to those registers if there is no physical local preference.

Inserting Spill Code Once the final coloring for a tile is known, spill code may need to be added on entry and exit edges. There are four general situations:

Spill When the parent assigns a register to a global variable v that is spilled in the child, then on each entry edge where v is live a store to memory is inserted and on each exit edge where v is live a load is inserted.

Transfer When the parent assigns a register to a global variable which is assigned a different register in the child, then on each entry or exit edge where the variable is live a register move is inserted.

Reload When the parent spills to memory a variable assigned to a register in the child, and it is cost effective to reload that variable, then on each entry edge where the variable is live a load from memory is inserted and on each exit edge where the variable is live a store to memory is inserted.

No Change When both the parent and the child allocate the variable to memory, no processing is necessary because there is a single memory location associated with each spilled variable.

The phrase "inserted on an edge" means that a new basic block is created which is executed only when this edge is traversed; fix-up code is placed in this block. The fix-up code on entry and exit edges must be ordered: stores and moves from a register must preceed loads and moves to a register. It is possible for a cycle of register to register moves to exist — permuting the contents of a set of registers — in which case an idle register is used to break the cycle. In the worst case a register must be spilled just to provide an idle register.

The above description of when to spill and reload is actually a bit too pessimistic. Consider the case of a definition of a variable v prior to a loop, some uses of v inside the loop, and finally another use of v after the loop. Assume that v gets a register for the tile associated with the loop, but does not get a register outside of the loop. As described, there would be a reload of v before the loop is entered, and a spill of v after the loop exit. The spill is unnecessary because v was never modified in the loop so the correct value is already in memory. A simple solution is to mark each tile where a variable is defined and propagate the information up the tile tree. Thus, a variable is reloaded only if there is a register definition of the variable deeper in the tile tree and there has been no spill of that variable since the definition.

4 Spilling

There are two primary issues to consider when spilling:

- which variables should be spilled, and
- where should the spill code be located.

We address both of these issues here.

When the allocator reaches a point where all variables not on the colorable stack have at least $||R||$ conflicts, then one of the remaining variables may have to be spilled. Following Briggs *et al.*[3] we prioritize the remaining variables and next place the least valuable variable on the stack; we delay the actual decision to spill until a variable fails to find a valid color.

Chaitin[6] spills the variable with the lowest spill cost to conflict count ratio where the spill cost is the penalty of accessing this variable from memory. This is the variable that is least likely to benefit from a register and most likely to enable other variables to become colorable. Bernstein *et al.* [2] use a complementary combination of spill heuristics and pick the best set of variables to spill over the entire program. Our algorithm could easily use either method but is implemented using Chaitin's heuristic with our cost metric.

As was previously mentioned, the best place for spill code may not be around the tile where the decision to spill the variable is made. Therefore, the information to make the spill location decision is computed on the way up the tile tree and the actual spill code insertion is made on the walk back down the tree.

To properly determine spill locations for variables, each variable that may want to be spilled higher or lower in the tile tree is tracked in the subtiles. This does not include every variable since the only variables that may want to be spilled higher in the tree are those variables that have already been spilled, or those that are referenced in this subtree and are visible to the parent tile. Local variables will never be spilled higher in the subtree because they are not live.

The following formulas are used to determine which variables are most deserving of registers and where spill code should be inserted. Assuming unit cost to load or store a variable:

$$\begin{aligned}
\mathit{Local_weight}_t(v) &= \sum_b \mathit{Prob}(b) \cdot \mathit{Ref}_b(v) \\
\mathit{Transfer}_t(v) &= \sum_e \mathit{Prob}(e) \cdot \mathit{Live}_s(v) \\
\mathit{Weight}_t(v) &= \sum_s (\mathit{Reg}_s(v) - \mathit{Mem}_s(v)) + \\
&\quad \mathit{Local_weight}_t(v) \\
\mathit{Reg}_t(v) &= \mathit{Reg?}_t(v) \cdot \\
&\quad \mathit{min}(\mathit{Transfer}_t(v), \mathit{Weight}_t(v)) \\
\mathit{Mem}_t(v) &= \mathit{Mem?}_t(v) \cdot \mathit{Transfer}_t(v)
\end{aligned}$$

where e ranges over entry and exit edges, b ranges over the blocks in $\mathit{blocks}(t)$, and s ranges over the subtiles of tile t. $\mathit{Live}_e(v)$ is 1 if variable v is live along edge e and 0 otherwise. $\mathit{Prob}(b)$ is the probability of b being executed and $\mathit{Prob}(e)$ is the probability of flowing along edge e. $\mathit{Ref}_b(v)$ is the number of references to variable v in block b. $\mathit{Reg?}_t(v)$ is 1 if variable v was allocated a register in tile t and 0 otherwise. Similarly, $\mathit{Mem?}_t(v)$ is 1 if variable v was not allocated a register in tile t and 0 otherwise.

$\mathit{Local\_weight}_t(v)$ corresponds to the value of keeping v in a register contributed by blocks in tile t.

$\mathit{Transfer}_t(v)$ is the cost of spilling (and/or reloading) variable v on entry to and exit from tile t.

$\mathit{Weight}_t(v)$ is used to drive the heuristic of which variable should be spilled. It is based on the number of uses of the variable in the tile, the penalty of allocating v to memory in this tile if it desires a register in some subtiles, and the penalty of allocating v to a register in this tile if it must be spilled in some subtiles. The weight can be negative if there is disincentive to allocate v to a register. This occurs if the cost of spilling v back to memory in some subtiles outweighs the benefit of having v in memory for tile t and the subtiles that want v in a register.

Tile summary variables have zero $\mathit{Local\_weight}$; their weight is based on their value in the subtile and the cost of transferring a variable on all of the tile entry and exit edges. This transfer cost approximates the penalty of spilling and reloading conflicting variables that are live and in registers at the child tile's boundaries.

$\mathit{Reg}_t(v)$ is the penalty of having the parent tile allocate v to memory if this tile allocates it to a register. This is the lesser cost choice of doing a transfer between memory and registers at t's boundary or simply changing the allocation of v to be in memory in the subtile. If $\mathit{Reg}_t(v) < 0$ then there is incentive for the parent to place v in memory because some descendants of t will need to spill v regardless of t's decision and there are not enough local references to overcome the cost of the spill.

$\mathit{Mem}_t(v)$ is the penalty of having the parent tile allocate v to a register if this tile allocates it in memory. This is simply the cost of moving v between registers and memory around the current tile because v must remain in memory in the subtile once a decision to spill it has been made.

The preceeding equations are used in the bottom-up walk of the tile tree to determine which variables are most important to allocate to registers and also to determine if a variable is not worth allocating a register even if the parent tile can place the variable in a register. They are also used in the top-down assignment of physical registers to determine when the allocation of a variable to a register should be changed to a memory allocation because the variable is in memory in the parent tile.

Making Spill Decisions When walking up the tile tree, it is possible to determine that some variables are not worth allocating to registers. For example, a variable that is defined in the current tile but already spilled in all subtiles may not be worth keeping in a register for this tile because the subtile spills can be eliminated if the variable is in memory. In this case, $\mathit{weight}_t(v) < 0$ demonstrating the disincentive of allocating v to a register. If this disincentive overshadows the benefit even when the parent tile allocates v to a register, then we know v should be in memory in tile t regardless of whether the parent of t places v in a register. Thus, each variable v satisfying the inequality $\mathit{transfer}_t(v) + \mathit{weight}_t(v) < 0$ is marked as not receiving a register for tile t.

All other spilled variables (during the bottom up walk) are spilled based on register pressure and the decision of which variables to spill is based on the local benefit of keeping the variable in a register.

Placement of Spill Code When walking back down the tile tree, spill code must be inserted at appropriate locations. There are two relevant cases for a variable v.

If v is in memory in tile t and in a register in the parent of t then we insert code to move v to and from memory on t's boundary edges. This is because spill decisions are never undone. If the parent found it profitable to place v in a register then there were enough uses to overcome the disincentive represented by the cost of transferring v to memory around tile t.

If v is in a register in t and in memory in the parent of t and if $weight_t(v) > transfer_t(v)$ we generate memory to register transfers otherwise we change the allocation of v in t to reflect that it should be in memory.

5 Related Work

Our work is primarily an extension of the work done by Chaitin[6]. We retain the accuracy of Chaitin's interference graph but, unlike Chaitin, we are able to benefit from local usage patterns and can place spill code intelligently.

Chow and Hennessy[8] and Larus and Hilfinger[13] handle conflicts more coarsely than we do. They represent each variable as a set of contiguous basic blocks — a *live range* — where the variable must occupy a register. When the interference graph cannot be colored with the available registers a live range is split into a pair of live ranges which are treated as separate variables and allocated to registers independently. Spill code is inserted at the basic blocks that form the boundary of two live ranges to coalesce variables that were split from a common ancestor. The splitting of live ranges is greedy and based solely on the edges in the interference graph; the new live range continues to grow as long as it is colorable. This can result in live range boundaries inside of loops even though spilling outside the loop would produce better code. Another disadvantage of the Chow and Hennessy scheme is that they separate local and global variables and color them from distinct register sets. This results in extra copies that coalescing and preferencing can avoid and also reduces the number of available registers for global coloring.

Briggs, Cooper and Torczon[4] find the loop structure of the program and perform live range splitting (with limited coalescing) at loop boundaries. They spill intelligently, but their interference graphs are large and they cannot spill inside of conditional statements.

Meltzer and Knobe[12] independently attempted to incorporate the program structure into a register allocator and have a similar notion of local and global variables. They construct a "control tree" based on the work of Sharir[14]. This is similar to our tile tree for structured programs but we believe our algorithm is more easily adapted to varying tile granularities. Also, Sharir may view branches out of nested loops as if statements where we continue to see the natural loop structure.

Meltzer and Knobe do not go into detail about making spill decisions, but they claim the "optimal" location for spill code is at the highest point in the control tree. We believe their claim to be in error based on the discussion in section 2 and have tried to give a complete discussion of spill analysis to support our position.

6 Summary and Observations

To summarize, our method for register allocation covers the program with sets of blocks called tiles. The tiles are grouped into a tree reflecting the program's structure and resulting in an allocation of registers that is sensitive to local usage patterns while retaining a global perspective. Registers are allocated in each tile using standard graph coloring techniques and spill code is inserted at infrequently executed tile boundaries to rectify different allocations between an ancestor tile and its descendants.

We conclude with a brief discussion of special topics which are handled well by this method.

When a variable corresponding to a machine instruction operand is spilled, some provision must still be made to deliver the variable's value to the processor. Most modern processor designs allow such operands to only come from registers and so a register must be made available to hold the value immediately before its use. A simple solution[13] to this problem is to reserve registers specifically for this purpose. Another solution[6] is to introduce temporary variables for each such use and then repeat the entire register allocation process considering these temporary variables. Repeating the allocation is expensive[3][11] and the potential for register spills is increased with aggressive optimization techniques. Our method avoids the need to iterate by introducing these temporaries as local variables with infinite spill cost. Since these variables are visible in only one tile and no tile has a large number of them, they do not contribute significantly to the cost of allocation.

Another anomaly in register allocation is handling procedure calls. A linkage convention may specify that particular registers should be saved by the caller, saved by the callee, used as parameters, or used to return a value. The various conventions for handling registers at call sites and procedure entry and exit can be handled uniformly with our technique. Parameter passing and return values can be easily handled with preferencing. Spilling caller-save variables can be modeled by introducing, at the point of the call, a local variable with infinite spill cost and a preference to a physical register. Handling callee-save registers is analogous to spilling variables corresponding to the callee-saved registers that are live across the entire procedure but not referenced anywhere in the procedure. When processing the root tile, each callee-save register is assumed to contain a live variable with weight commensurate with the save and restore cost and a preference to the callee-save register. This variable competes with other variables during the spill analysis.

For example, consider a case where a routine first has a quick return check and then does lots of computation. The bulk of the computation will be in a subtile with lower execution frequency than the root tile. In this case, the cost analysis will indicate subtile summary variables should be spilled in the root tile, rather than the variables occupying callee-save registers. The net effect is the same as the "shrink wrapping" discussed by Chow[7]: a callee-save register is not saved until an execution path which actually requires the register is selected.

Inline expansion — replacing a function call with a copy of the body of the function — is an optimization used to reduce the overhead of function calls and to allow more effective optimization in the vicinity of the function call. However, inline expansion can have a detrimental effect on traditional register allocators since a natural spill point (the call site) has been removed. Since our method retains natural spill points such as loop boundaries and nested control we should not suffer any side effects. Further, since the local variables of the inlined-function will all be local to the function's tile, the cost of coloring after inline expansion should be proportional to the combined cost of coloring each function separately.

Some machines have more levels of programmer addressable memory hierarchy than just registers and main memory. Our techniques can be easily extended to handle this hierarchy by moving variables between one hierarchical level and another at the tile boundaries. Allocation entails placing the variable at the highest level where it can be allocated and relying on the spill analysis to eliminate unprofitable moves between levels.

Finally, our original goal for this work was to find an efficient register allocation scheme that could execute in parallel on our machine and thus speed the entire allocation process. We achieved this goal since sibling subtrees can be processed concurrently in both the bottom-up and top-down passes. The amount of parallelism depends on the shape of the tile tree. Our primary language is Fortran and our expectations — confirmed by early experiments — are that there is adequate breadth in the tree to expect benefit from parallel evaluation.

References

[1] F. E. Allen. Control Flow Analysis. In *Proceedings of the SIGPLAN '70 Symposium on Compiler Construction*, pages 1–19, July 1970.

[2] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, pages 258–263, June 1989.

[3] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, pages 275–284, June 1989.

[4] P. Briggs, K. Cooper, and L. Torczon. Aggressive Live range Splitting. Technical report, Rice University, 1991.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN 90 Conference on Program Language Design and Implementation*, pages 53–65, June 1990.

[6] G. Chaitin. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[7] F. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, pages 85–94, June 1988.

[8] F. Chow and J. Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of*

the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6, pages 222–232, June 1984.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pages 25–35, January 1989.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] R. Gupta, M. L. Soffa, and T. Steele. Register Allocation via Clique Separators. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, pages 264–274, June 1989.

[12] K. Knobe and A. Meltzer. Control Tree based Register Allocation. Technical report, COMPASS, 1990.

[13] J. Larus and P. Hilfinger. Register Allocation in the SPUR Lisp Compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices Vol. 21, No. 7*, pages 255–263, June 1986.

[14] M. Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5:151–153, 1980.

[15] R. E. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

A Tile Tree Construction

Many tile trees can be constructed from a program. We construct a tile tree by starting with a tile graph corresponding to the control flow graph and group nodes in the tile graph together until it forms a legal tile tree.

The first step is to identify the loop structure based on intervals in the flow graph[1, 15]. An interval is a set of basic blocks which form a loop in the program. Like tiles, intervals nest. We define a **loop top** as the single basic block that has incoming back edges and dominates every basic block in its loop. Although irreducible loops do not have a loop top, all blocks in an irreducible loop that are reached by a forward control flow edge from a basic block outside the loop can be combined in the tile tree and treated as a single summary loop top. This summary node will dominate every basic block in the loop. Similarly for loops with multiple exits, we add a summary exit node so that each node in the interval will be post-dominated by a node in the interval. Each interval will be a tile.

The interval structure could be used directly as a tile tree but we can further capture the control structure within each interval. For each interval I, we form a graph $G_I = (N_I, E_I)$. Each interval strictly contained in I is represented by a single node in N_I and each block in I not in a subinterval is represented by a node in N_I. Edges E_I are induced by control flow edges between blocks in I as if blocks in subintervals were coalesced together. Self-loops and interval exit edges are ignored. We next find equivalence classes of nodes in each G_I which are totally ordered by both the dominator and post-dominator relations:[3] for $S_i = \{n_1, \ldots, n_k\}$, we have n_j dominates and is post-dominated[4] by n_{j+1}. From each S_i we construct S_i^* by including any node dominated

define $t(n)$ to be the smallest tile which
    contains block n.
foreach edge $e = (n, m)$ **do**
    if $n \notin t(m)$ and $m \notin t(n)$ **then**
        let a be the smallest tile containing
            both n and m
        create a block n_a in a and in
            all tiles containing a
        replace e with (n, n_a) and (n_a, m).
    endif
endfor
while $\exists e = (n, m)$ where $m \notin parent(t(n))$ **do**
    create n' in $parent(t(n))$ and all ancestor tiles
    replace e with (n, n') and (n', m)
endwhile
while $\exists e = (m, n)$ where $m \notin parent(t(n))$ **do**
    create m' in $parent(t(n))$ and all ancestor tiles
    replace e with (n, m') and (m', m)
endwhile

Figure 3: Tile tree fix-up: eliminate edges which violate conditions 2 or 3 by inserting empty blocks.

[3] These sets are almost the same as sets of nodes with the same control dependences[10]. Since not all edges in the control flow graph are considered, the dominator and post-dominator relations are for this graph are not subsets of the corresponding relations of the control flow graph.

[4] We assume a both dominates and post-dominates itself.

by a node in S_i and post-dominated by a node in S_i. Each of the sets S_i^* will be a tile.

Since conflict graphs may have $O(n^2)$ edges for n variables, it is desirable to control the size of $blocks(t)$ plus the number of subtiles of t. This suggests that we break large tiles up into pieces to bound this number. A natural way to break tiles is to partition large S_i into disjoint pieces where all nodes in one piece dominate those in another. A tile can, however, be broken into arbitrary connected components if no natural partition exists.

Once an initial covering of tiles has been selected, empty basic blocks are added where edges violate the second or third conditions on a tile tree. Figure 3 shows the algorithm to add these basic blocks. The first loop finds edges which cross between sibling tiles and introduces an empty block in the smallest containing tile. The next two loops identify edges which cross from a tile to a containing tile which is not the parent. Each such edge is "shortened" by adding an empty block in the parent tile and replacing the invalid edge with an edge to the parent block and and edge from the parent block to the other endpoint. The former of these new edges satisfies restriction two and the latter is "shorter" than the original edge so this process must terminate. Execution time is $O(\|E\| \cdot h(T))$ where $h(T)$ is the height of the tile tree: the length of the longest chain of tiles totally ordered by subset inclusion. It is expected that actual times will not approach this bound in practice.

Execution time of finding intervals is $O(\|E\| + \|N\|)$ and the execution time of finding tiles within intervals is dominated by the time to compute the dominator relation, $O(\|E\| \log \|N\|)$.