Global Register Allocation Based on Graph Fusion

Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai

March 1996
CMU-CS-96-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A register allocator must effectively deal with three issues: live range splitting, live range spilling, and register assignment. This paper presents a new coloring-based global register allocation algorithm that addresses all three issues in an integrated way: the algorithm starts with an interference graph for each region of the program, where a region can be a basic block, a loop nest, a superblock, a trace, or another combination of basic blocks. Region formation is orthogonal to register allocation in this framework. Then the interference graphs for adjacent regions are fused to build up the complete interference graph. The algorithm delays decisions on splitting, spilling, and register assignment, and therefore, the register allocation may be better than what is obtained by a Chatin-style allocator. This algorithm uses execution probabilities, derived from either profiles or static estimates, to guide fusing interference graphs, allowing an easy integration of this register allocator into a region-based compiler.

This research was sponsored in part by the Advanced Research Projects Agency/ITO monitored by SPAWAR under contract N00039-93-C-0152.

Keywords: Register allocation, graph coloring, graph fusion, compiler optimization, code generation

1 Introduction

Graph coloring is a well established approach to register allocation and a crucial component of many optimizing compilers. However, recent developments in compiler design and processor technology warrant a new look at global register allocation. Modern compilers include aggressive transformations (e.g, function inlining or loop unrolling) to exploit instruction level parallelism well beyond a single basic block. These transformations, together with global instruction scheduling, create two challenges: (i) the register pressure increases, since there are more values that may be allocated to a register; and (ii), the register allocator should be sensitive to the other compiler optimizations, i.e., if a compiler heavily optimizes one region of the program, then the register allocator should consider this information when deciding when and where to insert spill code.

We developed a register allocation framework that addresses these two concerns, and Figure 1 illustrates the benefits that can be obtained using this new approach for the `alvinn` program from the SPEC suite. This figure shows the total number of data movement operations executed that are due to register allocation (i.e., operations to save or restore a register, and register copies). All functions called by the main loop of this program are inlined. On the left, we see the number of data movement operations for an enhanced Chaitin-style register allocator [5]. In the middle, we see the results for the approach described in this paper which removes about 50 % of the data movement operations compared to a Chaitin-style register allocator. A simple enhancement, based on the observation that using caller-saved or callee-saved registers implies different costs, improves the results further and reduces the data movement overhead by 80%, as shown on the right.

The framework presented in this paper is *region-based* [10]: the register allocator operates on groups of basic blocks formed using either profile information or static analysis. The register allocator does not dictate how these regions are formed and thus provides a general and flexible approach to register allocation; form and priorities of regions are parameters to our algorithm. Regions can be individual basic blocks, traces [15], superblocks [11], or any other grouping used in the compiler (e.g., the loop structure). This framework fits nicely into code generators that take a similar approach to instruction scheduling [15, 6]: the register allocator now uses the same units of compilation and the same execution probability estimates as the instruction scheduler. This framework allows us to model a number of different approaches to register allocation [4, 9, 16], including the classical Chaitin-style register allocation [5] if a region is a function.

The key idea of our approach is to incrementally build up the interference graph. Consider the task of allocating registers for a compilation unit, e.g., a function. Instead of building an interference graph for the function and then cutting it down to make it colorable, we start with the interference graphs of smaller units and then build up towards the interference graph of the complete function. This approach allows the register allocator to delay decisions like which live range to spill or where to split, until more parts of a program have been analyzed. The order in which parts of a program are analyzed determines where overhead operations are placed. That's how the register allocator takes the priorities of regions into account when making spilling and splitting decisions: Live ranges are split only when necessary and then at boundaries to lower priority regions, thus minimizing the cost of additional data movement code. Spilling and splitting are well integrated: only those segments of a live range that have low spill cost but span regions of high priority and high register pressure are spilled. The base of our register allocator is a powerful *fusion* operator: the interference graphs of regions are fused to build up the complete interference graph.

A major benefit of this region-based approach is that the register allocator can custom-tailor the use of caller-saved and callee-saved registers. Many discussions of register allocation do not pay attention to the common practice of designating some registers as callee-saved and others as caller-saved. This register allocator can now split a live range L in a manner that isolates (and spills) those parts of L that cross frequent function calls.

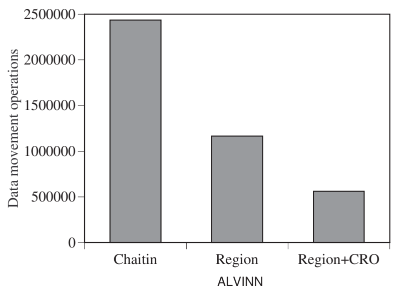

Figure 1: Impact of different register allocation strategies.

1.1 Terminology

A common approach is to model the register allocation problem as a graph coloring problem by assigning colors (physical registers) to nodes (live ranges of virtual registers) in some heuristic order. This process blocks when either all vertices have degree greater than N (the number of registers) [5] or when no legal color exists for a node [7, 3]. When coloring blocks, the compiler must somehow lower the maximum degree of the vertices in the interference graph to allow coloring to proceed. Two major techniques have been developed to lower the degree of the interference graph:

Splitting: Live range splitting segments a long live range $lr(x)$ into smaller live ranges $lr_i(x)$. *Shuffle* code is then needed to move the data value x when control passes from a segment lr_1 to another segment lr_2. Splitting does not reduce register pressure, but rather reduces the degree of the interference graph. The expectation is that each $lr_i(x)$ has a lower degree than $lr(x)$ and that the new graph can then be colored.

Spilling: A live range L is assigned a location in memory, and all references to L are done by memory accesses (loads/stores), which are referred to as *spill code*. A spilled live range is removed from the interference graph since it is no longer a register assignment candidate, thus lowering the register pressure.

The *spill cost* or *split cost* is then the execution cost of the spill or shuffle code. These two basic techniques still leave the compiler with a wide range of options, and the compiler's decisions influence the quality of the code. Among the decision that a compiler must make are:

- How to order the live ranges considering them for coloring.
- Whether to spill or split when coloring fails, which live ranges to spill or split, and in the case of splitting, where to place the shuffle code.
- Whether to assign a callee-save or caller-save register to a live range.

Our register allocator isolates these three aspects and thereby allows a compiler to pick the heuristic or strategy that is most in line with the rest of the compiler design.

2 Background and prior work

A frequently employed technique is to first allocate a *virtual register* from an infinite pool of registers to each register allocation candidate. Candidates can be user variables, constants, or compiler generated temporaries, depending on the strategy chosen [7]. Then, liveness analysis and reaching analysis determine the live range for each virtual register. Live ranges that are constructed in this manner, however, may be comprised of disjoint segments, resulting in an unnecessarily high number of conflicts for a live range. Renumbering [5] and web analysis [12] are two techniques to construct concise live ranges, which result in interference graphs of potentially lower degree. Virtual registers are mapped to physical registers by the register allocator, and spill or shuffle code is added to the program, as necessary.

2.1 Chaitin-style register allocation

Chaitin's algorithm is based on the observation that if a vertex V has degree $< N$, then V can be trivially colored since no matter what colors are assigned to V's neighbors, a legal color will remain for V. Such a vertex with degree less than N is called *unconstrained*. Given an interference graph, Chaitin's algorithm proceeds by successively removing unconstrained vertices from the graph. Each time a vertex V is removed, the edges that are incident upon V are also removed, and the degrees of V's neighbors are decremented. This process is known as *simplification*. Once all vertices have been removed, colors can be assigned to vertices in the *reverse order* in which they were removed. Simplification blocks when all remaining vertices have degree $\geq N$ and at this time a live range is picked to be spilled based on a heuristic cost function. This cost function is based on the spill cost of a live range as well as the benefit of removing the live range from the interference graph (the degree of the live range). Since registers must also be assigned to spill code, the process of building the interference graph and performing simplification is repeated until no more spilling is necessary.

Several refinements to this basic algorithm have been implemented. The coloring based register allocation algorithm used in the RS/6000 compiler [1] improves on this basic algorithm in two ways. First, the interference graph is colored three times, each time using a variation of the cost function, and the coloring with the least resulting total spill cost is selected. Second, when a live range L is selected for spilling, instead of inserting a load before each use and a store after each definition of L, an attempt is made to insert at most a single load and store inside each basic block. In effect, L is split into segments that span at most a basic block.

Simplification is a heuristic approach to coloring, and as such may miss legal coloring opportunities. Optimistic coloring [3] improves simplification by attempting to assign colors to live ranges that would have been spilled by the basic algorithm. Optimistic coloring delays spill decisions until the register assignment phase. As in the basic algorithm, a *spill candidate* L is chosen when simplification blocks, but rather than deciding to spill L, this live range is removed from the graph and added to the set of live ranges that will be assigned colors. Spill decisions are made during the register assignment phase: when no legal color exists for the next live range to be colored, this live range is spilled.

The Chatin-style approach is simple and fast, and produces good results for programs with colorable interference graphs. However, when an interference graph cannot be colored using simplification, this approach makes spilling decisions that are all-or-nothing: *all* definitions and uses of a spilled live range go through memory even though some parts of the live range could have been allocated a register. It is more beneficial to spill only the troublesome segments of a live range (i.e., segments that contain few or no references and span regions of high register pressure), while keeping in registers those segments that have references in regions of high execution frequency. Besides, despite all the attempts to improve spill code, in practice there are situations where splitting produces better results [2].

It is difficult to adopt live range splitting into Chaitin's approach. Since the interference graph does not encode information about the program control flow structure and reference patterns of live ranges, this graph cannot be used to guide partitioning of live ranges. Moreover, it is difficult to make splitting decisions when simplification blocks, for several reasons. First, it is difficult to decide how to partition a live range in a manner that not only allows registers to be assigned in the most critical regions of code and minimizes split cost but also allows simplification to proceed. Second, the interference graph must be *rebuilt* each time live ranges are split. This is in contrast to spilling where simplification can *resume* after a spilled live range is removed from the interference graph. Third, it is difficult to decide between spilling and splitting, i.e., it is difficult to distinguish between cases of high register pressure where spilling is absolutely necessary, and cases where the interference graph has high degree and splitting can reduce the interference graph's degree.

2.2 Splitting live ranges before coloring

Several recent approaches have tried to fit live range splitting into Chaitin's coloring framework by splitting live ranges prior to coloring. The motivation is to reduce the degree of the interference graph and to allow the spilling of only those live range segments that span program regions of high register pressure. Aggressive live range splitting [2] uses the Static Single Assignment (SSA) representation of a program to determine split points. A live range is split at a ϕ node when the incoming values to the ϕ node result from distinct assignments. The approach also splits all live ranges that span a loop by splitting these live ranges immediately before and after the loop. Kolte and Harrold [13] partition a live range at a finer granularity by considering the ranges of instructions between loads and stores of a virtual register.

These approaches to splitting live ranges before coloring have several drawbacks. First, decisions regarding which live ranges to split and where to split them are made prematurely. Thus, live ranges are split unnecessarily, resulting in a performance degradation due to unnecessary shuffle code. Various heuristics have been developed to eliminate shuffle code by increasing the chance that the same color is given to partner live ranges, e.g., biased-coloring or conservative coalescing [2]. Rather than determining the critical regions where splitting and spilling are beneficial, these approaches split live ranges arbitrarily and greedily, with the hopes that later heuristical steps will clean up unnecessary splits. Second, the points in the program where live ranges are split by these approaches are not necessarily points of low execution probability. Although these approaches may use execution probabilities to determine the points to split live ranges, there is no guarantee that the resulting live ranges will fit into registers without spilling. That is, the register allocator runs the risk of either splitting too much (leading to unnecessary shuffle code), or not enough (with the consequence that high-frequency live ranges are spilled).

2.3 Priority-based coloring

The priority-based coloring [7] approach is an alternative framework that allows splitting decisions to be delayed until coloring blocks. In contrast to Chaitin's approach, where the register allocator assigns physical registers to virtual registers, this approach begins with each live range assigned a home location in memory. In effect, the algorithm begins with all live ranges spilled to memory. Coloring greedily assigns colors to live ranges in a heuristic order determined by a priority function. The priority function captures the savings in memory accesses from assigning a register to a live range rather than keeping the live range in memory. This priority function can be based on either profile information or static estimates, e.g., live ranges that have references within deeply nested-loops can be given high priority [7]. Before colors are assigned, unconstrained live ranges are removed from the interference graph, since unconstrained live ranges can always be assigned a legal color, after colors have been assigned to other live ranges. Unlike simplification, the degree of nodes neighboring the removed unconstrained nodes are not decremented.

Color assignment blocks when no legal color exists for the next live range L to be colored, i.e., when all N colors have been taken up by L's neighbors. At this point, L is split. To facilitate splitting, the live range for a candidate V is defined as a collection of *live units*, where each live unit is a basic block within which V is live. Splitting forms a new live range L' by starting from a seed live unit and incrementally adding live units to the new live range until adding one more live unit renders L' uncolorable. In [7], live units are added in a breadth-first traversal of the control flow graph, preferably starting from a live unit where the first reference to

V is a definition. A live range is spilled (i.e., remains in its home location) when no live units that comprise the live range can be given a register.

An important consideration in live range splitting is selecting the points where shuffle code is inserted. To reduce split cost, shuffle code should be placed at points of low execution probability. The priority based approach does not take execution frequency into account when inserting shuffle code, and there is no guarantee that split points do not end up along frequently executed edges. Shuffle code induced by a split may end up, e.g., on a loop back arc. Code motion techniques are used after register assignment, to optimize placement of shuffle code [7].

2.4 Program structure based approaches

Several more recent approaches to register allocation attempt to make graph coloring sensitive to program structure by dividing a program into regions and prioritizing the regions according to execution probabilities. The register allocator than colors regions in order of their priorities, and shuffle is code is inserted at the boundaries of these regions. Representative program structure based approaches include the Tera [4], Multiflow [9] and RAG [16] compilers.

The Tera compiler (as described in [4]) constructs a tile tree for a program; this tree corresponds to the control-flow hierarchy of the program. Register allocation colors the tiles in two phases. The first phase traverses the tile tree from the bottom up and allocates pseudo registers to the live ranges in each tile using graph coloring. Pseudo registers capture the constraint that two virtual registers are to be allocated to the same physical register. Once two variables in a tile are assigned to the same pseudo register, the parents of the tile must adhere to this decision. The second phase walks through the tile tree top-down and binds pseudo registers to physical registers. Biased-coloring is used to avoid unnecessary shuffle code at tile boundaries. As coloring is performed hierarchically, shuffle code tends to be outside of the innermost loops. The requirement to observe the lower-level coloring imposes more and more constraints as coloring moves toward the root of the tile tree.

Figure 2 illustrates unnecessary constraints imposed on the register allocation by premature coloring decisions. The code in Figure 2(a) consists of two tiles, one for the loop (the shaded region) and one for blocks B_1 and B_6. The loop region is colored first. The interference graph *for this region* is depicted in Figure 2(b). Inside this region, the live range $lr(y)$ is live in B_2 and B_5. There are two live ranges for x, $lr(x)$ and $lr'(x)$, in the loop. The live range $lr(x)$ is live in B_2 and B_5, and $lr'(x)$ is live in B_4. Coloring the graph with two registers, $lr'(x)$ is placed into the same pseudo register as $lr(x)$ or into the same pseudo register as $lr(y)$. Based on the information inside the loop region, both choices are reasonable. However, if $lr'(x)$ and $lr(y)$ use the same pseudo register, a shuffle move is inevitable either on $\langle B_4, B_6\rangle$ or on $\langle B_5, B_6\rangle$, since $lr(x)$ and $lr'(x)$ merge in B_6. If $lr'(x)$ and $lr(x)$ use the same pseudo register, no shuffle code is required, but that this mapping is beneficial can only be determined when dealing with B_6. This paper presents an approach to address this deficiency by delaying binding decisions. In contrast to [4], our algorithm takes a lazy approach to color assignment so as to avoid premature coloring decisions.

Instead of a tile tree, the RAG compiler colors the region nodes in a function's Program Dependence Graph (PDG), proceeding in a hierarchical manner from the leaves to the root [16]. Chaitin's algorithm is used at each region node. The coloring decision made for a region is preserved by the parent nodes of the region. This may prevent graph coloring from finding the best solution and affects live range splitting as well. A live range is split when that live range is spilled in the parent region; such a live range is split at the boundaries of the parent region and all subregions in which the live range is live. All live ranges that are determined to be in the same register in the subregions are considered as one node in the parent region and therefore interfere with the union of the live ranges with which the individual live ranges interfere.

The Multiflow compiler employs trace scheduling as a framework for both register allocation and schedul-

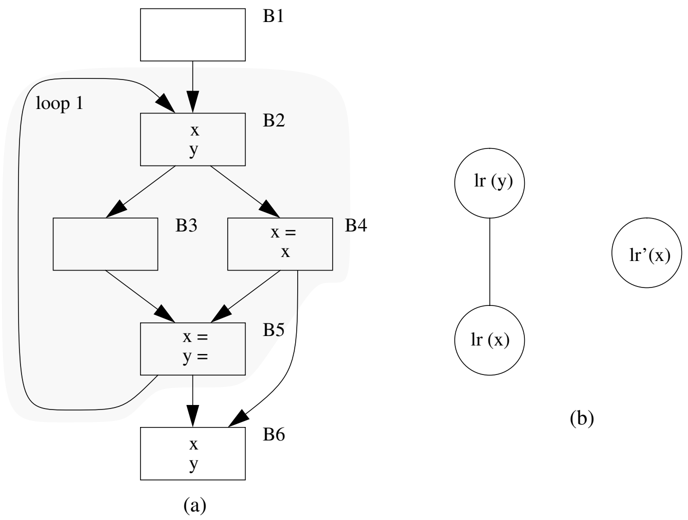

Figure 2: Premature coloring decision with $N = 2$.

ing [9]. The trace scheduler picks a trace and then passes it to the code scheduler; the code scheduler then performs register allocation and scheduling together. The code scheduler records register usage preferences for the scheduled trace; this information is maintained for each exit from or entry into a trace. This information is subsequently used when translating traces that connect to these exit or entry points and shuffle code between two traces is not needed if the value can be kept in the same register. Traces that are compiled first have more freedom in using registers, and shuffle code ends up boundaries to traces that are compiled later. As long as the trace picker presents the traces in an order that reflects the execution frequency, this scheme favors the most frequently executed parts of a program. One drawback of the approach is that coloring is not used for register allocation.

2.5 Other approaches

There exist other approaches to register allocation and live range splitting. For example, probabilistic register allocation [17] is a hybrid of the priority-based and program structure based approaches. The approach consists of three steps, local register allocation, global register allocation, and register assignment. Variables initially reside in memory instead of virtual registers (with load/store for every use/definition). An estimate of the probability that a variable reference is allocated to a register is used as priority in the local and global register allocation steps when determining which references get registers. Real register assignment is deferred till the register assignment step. The global register allocation step partitions a program into regions based on the loop hierarchy and proceeds from the innermost loops to the outermost loops. When a variable is assigned to a register, shuffle code is placed in the pre-header and post-exit of a loop to load the value of the variable into the register and restore it back to memory (if the value is updated). It is not obvious in the paper how outer loops deal with those shuffle code.

Kurlander and Fischer [14] perform live range splitting *after* register allocation to free up registers that can be used to improve code scheduling. Empty delay slots in the final schedule are filled with shuffle code to split and spill live ranges. Spilling frees up registers and these additional registers are used to remove false dependencies induced by the reuse of registers.

3 Overview of register allocation based on graph fusion

The key idea is to *build up* the interference graph, starting off with live ranges that extend at most a single region. Each region initially has its own interference graph. A region can be as small as a single basic block, or as large as a whole function (in which case the algorithm is identical to Chaitin-style register allocation). Regions are connected via control-flow edges, which are prioritized. These edges are then considered in priority order, and the interference graphs of two regions connected by an edge are merged by *fusing* the interference graphs. This fusion operator coalesces live ranges that span the control flow edge and maintains the invariant that the resulting merged interference graph is colorable, if necessary by splitting (suppressing the coalescing) of a live range. Thus the fusion operator makes spilling and splitting decisions when it becomes clear that it is impossible or unprofitable to keep all live ranges in registers. After all control flow edges have been considered, a single colorable interference graph remains.

There are four phases in this framework: *region formation*, *graph simplification*, *graph merging*, and *register assignment*. In the *region formation phase*, regions are formed using any number of possible techniques. For example, a region can be a single basic block, a trace [15], a superblock [11], a region as defined in [10], the blocks at a particular static loop nesting level [4], or the blocks within a PDG region node [16]. Control flow edges that lie outside of regions are then ordered according to some priority function consistent with the region formation approach, e.g., edges entering innermost loop regions are ordered before those entering outermost loop regions. One particularly attractive priority function is the use of execution probabilities. These can be derived either from profile information [8, 18], from static estimates such as loop nesting depth [7], or from static branch estimates [15]. The choice of edge ordering is orthogonal to register allocation but of course impacts the quality of the code, as our register allocation framework is edge order sensitive: shuffle code is less likely to end up on edges that are ordered first, and spilling decisions are delayed until later edges are processed.

During region formation, an interference graph G_R is built for each region R. Given a virtual register x, the segment of x's live range $lr(x)$ that extends block B_i is denoted by $lr_i(x)$, and there is one or more segment for every region R where x is live and reaching.

The objective of the *graph simplification phase* is to determine how many live ranges must be spilled within each region. If an interference graph G_R can be simplified, then no spill code is necessary within region R. But if G_R cannot be simplified, then from R's perspective the cheapest live ranges to spill within R are those that are *transparent*, i.e., live ranges $lr(y)$ that span R with no definition or use of y in R. From a global perspective, the choice of *which* live ranges are the best ones to spill cannot be determined at this point in the algorithm. Thus the decision on which live range to spill is delayed until more global knowledge of reference patterns is available; this phase determines only *how many* transparent live ranges need to be spilled within each region. The next phase, graph merging, determines *which* live ranges are the best ones to spill. This technique is referred to as *delayed spilling* and is discussed in more detail in Section 4.3. There, we also describe what to do if the compiler must spill more live ranges than there are transparent ones.

The *graph merging phase* takes the sequence of control flow edges determined by the region formation phase and fuses interference graphs along each edge. Graph merging is based on a powerful *fusion* operator that maintains the invariant that the resulting interference graph is simplifiable (i.e., can be simplified). Live range splitting decisions are made by the fusion operator: if fusing two graphs G_1 and G_2 along an edge E results in an interference graph that cannot be simplified, then one or more live ranges that span E are split. Only the live ranges that span E need to be considered, because in the worst case, splitting all such live ranges partitions the graph $G_1 \cup G_2$ back to the two original graphs, G_1 and G_2, both of which are simplifiable. At the end of the graph merging phase, we are left with one simplifiable interference graph; we know how many live ranges to spill for each region and where to place the shuffle code, but no physical registers are committed to any live range. The invariant is the lazy approach which allows us to avoid making any coloring decisions prematurely

$lr_1(x)$	$lr_2(x)$ Spilled	Transparent	Non-transparent
Spilled	Spilled	Spill (Section 4.3)	Split (Section 4.3)
Transparent		Coalesced, transparent	Coalesced, non-transparent
Non-transparent			Coalesced, non-transparent

Figure 3: Different possibilities when coalescing $lr_1(x)$ and $lr_2(x)$.

during the graph merging phase. Therefore there is no coloring assignment decision to any live range while graphs of the loop region are fused in Figure 2. All we know is that the interference graph of the loop region is colorable. The actual coloring decision is deferred till the register assignment phase.

As more graphs are fused, the delayed spilling mechanism gradually spills live ranges. The net effect of combining splitting with delayed spilling is that the register allocator may spill only those segments of a virtual register's live range that do not contain references but span regions of high register pressure.

In the *register assignment phase*, physical registers are assigned to live ranges and shuffle code is generated. The simplifiability invariant guarantees that once all interference graphs have been fused, the resulting interference graph is colorable and spilling or splitting decisions have already been made. Shuffle code is inserted at an edge $E = \langle B_1, B_2 \rangle$ for those live ranges that extend across E but have different storage locations in B_1 and B_2. Shuffle code is either a load, store or register to register move; shuffling between memory locations is not necessary as all live ranges of a virtual register x are spilled to the same location in memory. This straight forward insertion of shuffle code may result in partial redundancies among the shuffle code. While a general partial redundancy elimination algorithm could be used to optimize the shuffle code, a simple technique is effective in practice, details of which are discussed in Section 5. There exist further opportunities for improving the code in this phase: biased coloring [2] may eliminate shuffle code, optimistic coloring [3] may assign registers to live ranges that have been spilled by simplification, and if we notice that a live range gets a caller-saved register in a region with high call frequency, then the register assigner may decide to spill this live range nevertheless.

4 Fusing two interference graphs

Consider two basic blocks B_1 and B_2 that are in two regions R_1 and R_2, and connected by an edge $E = \langle B_1, B_2 \rangle$ in the control flow graph. Let x be a virtual register that is live along edge E. Each of these live range segments lr_i can be in one of these three states: it has been spilled, it is represented in G_{R_i} and transparent, or it is represented in G_{R_i} and non-transparent (i.e., there is a reference). The fusion operation attempts to coalesce the live range segments $lr_1(x)$ and $lr_2(x)$ range in the interference graph $G_{R_1 \cup R_2}$ so that no shuffle code is needed for x along E. If both segments are spilled, no shuffle code is needed. If one is spilled and the other is transparent, we spill the combined live range in $R_1 \cup R_2$ and thereby reduce the register pressure, since the live range is already spilled in one region and there are no references to it in the other. If one is spilled and the other non-transparent, we have a split point. Otherwise the segments are coalesced, although if the new interference graph is not simplifiable, coalescing may be suppressed (i.e., a split). Figure 3 enumerates these cases.[1]

Each live range has three attributes that are propagated during graph fusion: *caller-save cost*, *spill-cost* and *has-def*. The *caller-save cost* attribute is the estimated cost of assigning a caller-saved register to the live range. This cost is the number of saves and restores that must be executed around function calls if this live range is assigned a caller-saved register. The *spill-cost* attribute is the estimated spill cost of a live range. The *caller-save cost* and *spill-cost* properties are used during register assignment to decide whether a live range is assigned a

[1]Empty entries in this table are symmetric cases.

callee-save register, a caller-save register, or whether it is more profitable to spill the live range. The *has-def* property is true for a live range $lr(x)$, if there is a definition of x within $lr(x)$. This information indicates whether x's value needs to be stored to memory at exits from $lr(x)$; if a live range does to modify the value of a virtual register, then no stores are required at exits from the live range.

For each basic block B, two data flow sets are maintained: $ReachIn(B)$ and $LiveOut(B)$. Given a virtual register x, $x \in ReachIn(B)$ if a definition of x reaches the entry of B, and $x \in LiveOut(B)$ if x is live at the exit of B. The $ReachIn$ and $LiveOut$ sets indicate which live ranges span edge E.

The fusion operator ensures that the simplifiability invariant holds by performing simplification on $G_{R_1 \cup R_2}$. If simplification of $G_{R_1 \cup R_2}$ blocks, then a coalesced live range $lr(x)$ is chosen for splitting from among the set of remaining constrained nodes. $lr(x)$ will be split along E back into $lr_1(x)$ and $lr_2(x)$. After splitting a live range, the degree of the interference graph may be lowered, allowing the simplification to proceed from the point where it blocked. If the simplification blocks again, additional live ranges are split. Since the original graphs G_{R_1} and G_{R_2} are simplifiable, then at worst all live ranges that span E are split.

As graph merging proceeds further, more interference graphs are fused, resulting in denser interference graphs. Thus, the earlier an edge is considered by the fusion process, the less likely it is that live ranges are split along that edge. This provides a nice property: if we don't want shuffle code on a particular edge, we can fuse the interference graphs along that edge first. Consequently, the decision of *where* to split is prioritized according to the edge ordering, and shuffle code ends up on less frequently executed edges.

4.1 Example

We illustrate the above steps with an example. Consider assigning registers to the program fragment shown in Figure 4(a). Assume we have only two physical registers (N = 2) and that regions are basic blocks. There are three virtual registers (x, y and z) in this program with initial live ranges indicated by the vertical bars. The interference graph of each basic block prior to fusion is depicted in Figure 4 (b). Suppose edges 1 and 3 form the most frequently executed path and edges are fused in the order 1, 3, 2, 4. After we fuse along edges 1, 3 and 2, we obtain the interference graph of Figure 4 (c) (interference graph nodes list live ranges that have been fused). If we now fuse the interference graphs along edge 4, $lr_3(y)$ and $lr_4(y)$ are combined, since the only live range spanning edge 4 is $lr(y)$. However, fusing along edge 4 makes the graph unsimplifiable (clique of size 3), so the algorithm undoes combining $lr_3(y)$ and $lr_4(y)$. $lr(y)$ is effectively split at the less frequently executed point in this control flow graph. The result is that all values can be kept in registers with an additional register move instruction at the end of block B_3.

4.2 Splitting to reduce call cost

Dividing the registers into two sets, callee-save and caller-save registers, provides the register allocation more freedom when minimizing the call overhead [7]. There is a cost associated with each kind of register assigned to a live range. When a live range lr ends up in a caller-save register, we must pay the cost of saving and restoring lr's value at all function calls that are crossed by lr. Similarly, if lr ends up in a callee-save register, then this register may have to be saved/restored at entry/exit to the function.

To allow the register allocator to choose the right kind of register for a live range, we model the cost with two functions, $f_{benefit\_caller}$ and $f_{benefit\_callee}$, respectively. These functions are defined for each live range lr. For each lr, $f_{benefit\_caller}(lr)$ (resp. $f_{benefit\_callee}(lr)$) is defined as the weighted reference counts of the spill code minus the weighted caller-save (resp. callee-save) cost. That is, these two functions indicate the estimated number of loads and stores that are eliminated if a caller-save (or callee-save) register is assigned to lr. During the register assignment phase, the selection of the kind of register to use is based on these two functions. If $f_{benefit\_callee}(lr) > f_{benefit\_caller}(lr)$, finding an available *callee-save* register for lr is attempted prior to

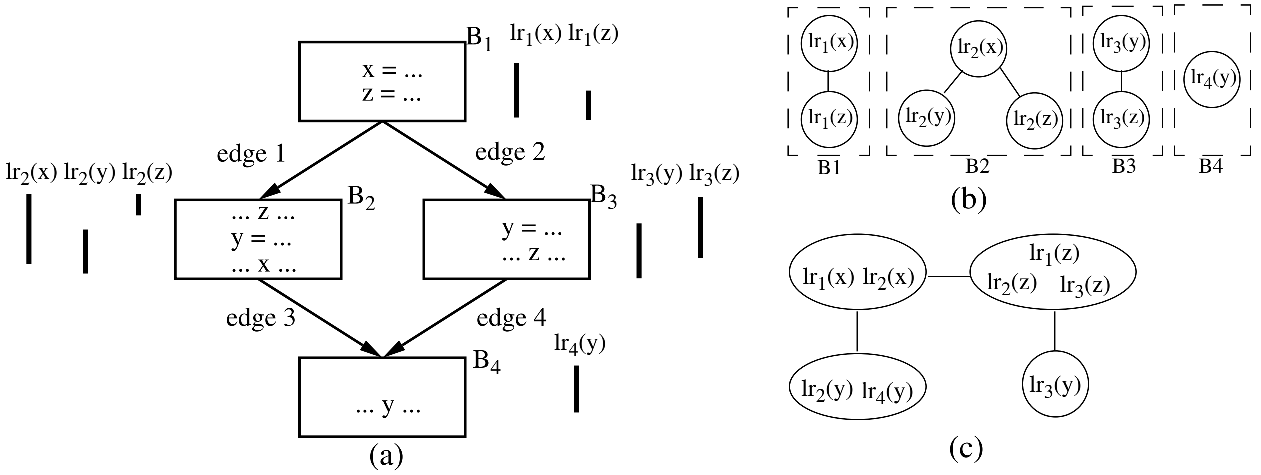

Figure 4: Simple code fragment for a world with $N = 2$.

finding an available caller-save register. If $f_{benefit\_callee}(lr) \leq f_{benefit\_caller}(lr)$, it is preferable to put lr into a *caller-save* over using a callee-save register. However, sometimes *not* using a register (i.e., spilling a live range) is better than using the wrong kind of register. For example, the spill-cost may be less than the caller-save cost. In such a case, the whole live range is spilled to reduce the overall load/store counts even though there is an available register, since this register is of the wrong kind.

Fusing two interference graphs tends to grow live ranges aggressively to reduce the shuffle cost as long as the colorability invariant is maintained. One improvement to the algorithm sketched above is to limit the growth of live ranges that cross function calls, by constraining coalescing. In this manner, the (smaller) live range has a chance to get the best kind of register or be spilled (at lower spill-cost), thus reducing the overall load/store counts. Even though the graph is colorable, it may be better if the live range is split at infrequently executed edges; then we pay the *lower* shuffle cost (along these edges) instead of the *higher* caller-save cost at all call sites. When applied to the `alvinn` program, this improvement results in the 80% reduction in data movement operations shown in Figure 1.

The heuristic we use to constrain coalescing is based on two functions: $f_{threshold}$ and f_{split}. $f_{threshold}$ is a threshold function that determines good candidates for splitting, and f_{split} is a function that determines whether splitting would result in savings. Consider two live ranges $lr_1(x)$ in region R_1 and $lr_2(x)$ in R_2. R_1 and R_2 are joined by edge E. $f_{threshold}(lr_1, lr_2, E)$ is a function of the edge E's execution frequency, the number of registers available for assignment, and the weighted reference counts of $lr_1(x)$ and $lr_2(x)$. If $f_{threshold}(lr_1, lr_2, E)$ decides that edge E is a good place to suppress the coalescing of lr_1 and lr_2, $f_{split}(lr_1, lr_2, E)$ judges whether this decision could drive down the number of data movement operations. This function tests if one live range has a caller-save cost less than its spill-cost and the other live range has a caller-save cost greater than its spill cost plus the shuffle-cost along E. If that is the case, then the two live ranges are not coalesced since adding $lr_1 \cup lr_2$ to the interference graph is likely to hurt performance if $lr_1 \cup lr_2$ ends up in a caller-save register.

Figure 5 depicts an example of how caller-save cost is reduced by splitting. In this example, the live range x is live through out the whole program. Assume $lr(x)$ is not split, i.e. x occupies one register in Figure 5 (a). If this register is a caller-save register , $lr(x)$ pays a high caller cost (saving and restoring x around function `foo`). It is much cheaper to split the live range of x into $lr_1(x)$ and $lr_3(x)$ along the edge $\langle B_2, B_3\rangle$, as illustrated by Figure 5 (b). Now $lr_1(x)$ can either reside in memory, or a callee-save register (if we determine that the callee-save cost is low).

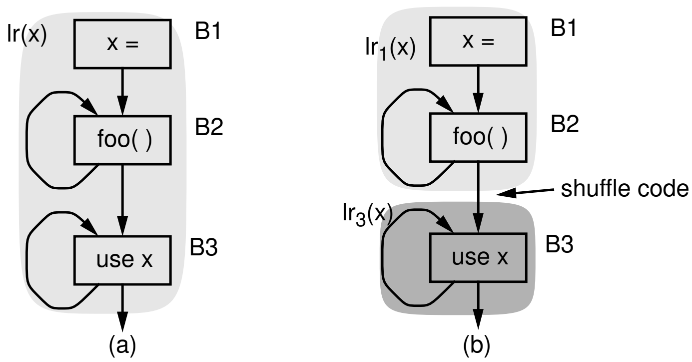

Figure 5: Eliminating caller-save cost by splitting.

4.3 Delayed spilling

When a region R needs M physical registers to be colored, and $M > N$, then $M - N$ live ranges must be spilled. Considering only local spill costs inside R, the best spill choice is a transparent live range L_t, since L_t has a high degree in the interference graph, and the cost of spilling L_t is zero inside R. If we assume that there are T transparent live ranges in region R, then there are three cases that must be considered:

$M - N < T$**:** In this case, the number of transparent live ranges is more than the number of live ranges that must be spilled. However, choosing the transparent live ranges for spilling should be delayed until the compiler obtains more global information about the reference patterns of the transparent live ranges. Searching the region's immediate neighbors does not solve the problem because transparent live ranges may be transparent across many basic blocks. *Delayed spilling* deals nicely with this case, as explained below. A large number of transparent live ranges is common while processing the first, high-priority edges.

$M - N = T$**:** In this case, the spill needs are satisfied by spilling all transparent live ranges.

$M - N > T$**:** In this case, all transparent live ranges, as well as $M - N - T$ live ranges with a reference inside R, are spilled.

We note that spill decisions may not be globally optimal. Although spilling a transparent live range is very attractive, we can construe flow graphs for which this decision is not optimal. And if $M - N > T$, we must spill some non-transparent live ranges, which are selected by a heuristic based on spill cost, area, and the degrees in the interference graph [1].

We now describe in more detail the delayed spilling technique used to handle the case where $M - N < T$. Since all transparent live ranges conflict with each other, these live ranges form a clique in a region's interference graph. The transparent live ranges are therefore collected into a single *clique summary node* C in the interference graph, as depicted in Figure 6. This node C contains an edge to all other nodes in the graph, since the transparent live ranges interfere with all other live ranges in a region. The clique summary node is annotated with the number of transparent live ranges that it represents, i.e., $T(C)$. We record that $\psi(C) = M - N$ of the T transparent live ranges must be spilled, without specifying which ones. The actual size of the clique is thus $T(C) - \psi(C)$. The clique is dealt with as a single unit; eventually $\psi(C)$ live ranges will be spilled. By keeping a summary node in the graph, we can keep more live ranges in the interference graph than there are registers, and we delay the decision on *which* range(s) to spill until more information is available.

Given an edge $E = \langle B_1, B_2 \rangle$, the live ranges that span across E are merged. When a live range lr_1 is merged with lr_2, and one of the two is transparent, there are three cases to consider:

1. If lr_1 is a spilled live range, lr_2 is in a clique C (transparent): lr_2 is removed from C and spilled. Both $T(C)$ and $\psi(C)$ are decremented by one. This decision grows a spilled live range, thereby allowing the compiler to avoid shuffle code (a contiguous range of spilled or non-spilled live range segments does not incur shuffle code; only a transition from spill to non-spilled or vice versa requires shuffle code).

2. If lr_1 is a non-transparent live range, lr_2 is in a clique C: lr_2 is removed from C, but not spilled. Only $T(C)$ is decremented. This decision enlarges a non-spilled live range; since this live range is in a register in an adjacent region, the compiler favors it over the other live ranges in the clique.

3. If lr_1 is in clique C_1 and lr_2 is in clique C_2 : lr_1 and lr_2 are coalesced and added to a new clique, $C = C_1 \cap C_2$. After processing all live ranges across E, there are three cliques C, C'_1 and C'_2, where $C'_1 = C_1 - C$ and $C'_2 = C_2 - C$. $\psi(C)$ establishes how many ranges of C must be spilled and is computed as a function of $\psi(C_1)$ and $\psi(C_2)$.

If $T(C) = \psi(C)$ for a clique C at any time during this phase, then this means that all live ranges of C must be spilled. At that point, C is removed from the interference graph.

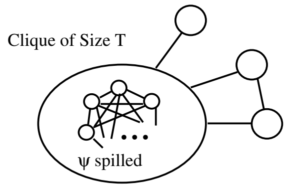

Figure 6: Clique summary node.

5 Placement of shuffle code

After the register assignment (coloring) phase is done, shuffle code is inserted as necessary along edges. At an edge $E = \langle B_1, B_2 \rangle$, shuffle code is inserted for a virtual register x that has been split at E, i.e., if $lr_1(x)$ and $lr_2(x)$ have not been assigned the same storage location. Shuffle code can be of three types: register-to-register ($lr_1(x)$ and $lr_2(x)$ are assigned to different registers, so a move operation is required), register-to-memory ($lr_1(x)$ is assigned to a register and *has-def* is set, and $lr_2(x)$ is spilled), or memory-to-register ($lr_1(x)$ is spilled and $lr_2(x)$ is assigned to a register). There is no need for memory-to-memory shuffle code, because all live ranges belonging to the same virtual register x are spilled to a single location in memory (i.e., $lr_1(x)$ and $lr_2(x)$ are spilled to the same memory location).

The insertion of shuffle code is simple and straight forward, but this step may result in partial redundancies in the code. For instance, the has-def property of a live range $lr(x)$ determines whether shuffle stores of x are needed on the exit edges of $lr(x)$. When the definitions of x are in less frequently executed blocks, and the exit edges (splitting points) are more frequently executed than the definitions, the shuffle stores are executed more often than necessary. Based on the estimated costs of *def-cost* and the estimated costs of the shuffle move (register-to-register), store (register-to-memory) and load (memory-to-register) operations, a simple technique is used to optimize the shuffle stores. For each live range, the costs of the shuffle move, store and load can easily be obtained since each edge is annotated with the estimated execution frequency (either static or profile). The def-cost of a live range is the estimated (weighted) number of definition within the live range. If the sum of the def-cost plus the cost of shuffle moves, which move values into the live range, is less than the cost of shuffle store, then all shuffle stores are eliminated by inserting a new shuffle store right after each definition and the

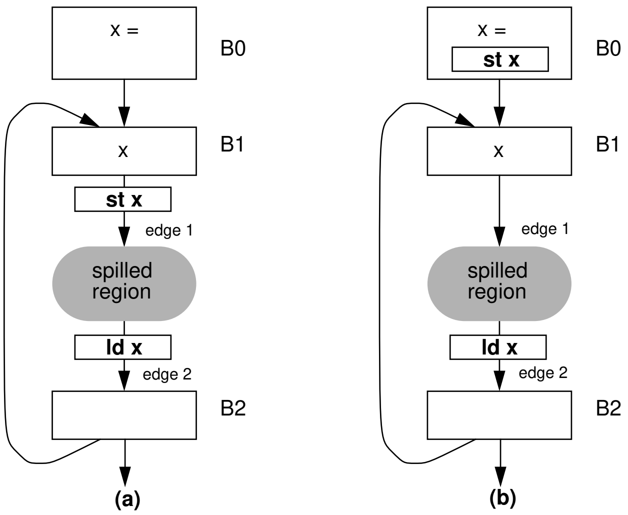

Figure 7: Placement of shuffle store.

shuffle moves. In other words, for each definition (which includes the shuffle moves), there is a store writing the new value back to the memory location of the live range so as to keep the value in memory up-to-date.

The technique is quite effective in practice. The reason is that the code motion step of the global optimization phase moves loop-invariant common subexpressions out of loops, which the definitions of those common subexpressions are less frequently executed, and they may be spilled inside part of the loops due to the high register pressure. Figure 7 depicts the optimized placement of the shuffle code. The live range x is defined in B_0 and live through out the whole program. x is spilled within the loop, as indicated by the shaded region. The shuffle code is highlighted in bold face. The straight forward shuffle code insertion requires two shuffle code, depicted in Figure 7(a), one shuffle store on Edge 1 (from B_1 to the spilled region) and one shuffle load on Edge 2 (from the spilled region to B_2). The shuffle store is executed on every loop iteration in despite of the fact that x is never modified in B_1 and B_2. Since x is defined in B_0, which is less frequently executed than Edge 1 is traversed, the placement of the shuffle store can be optimized by eliminating the shuffle store on Edge 1 and inserting a new shuffle store immediately after the definition of x (depicted in Figure 7(b)).

6 Evaluation

The framework is implemented in the *cmcc* compiler, an optimizing retargetable compiler developed at CMU. Our data are based on the code generator for the MIPS; dynamic numbers have been obtained on a DECStation 5000. The runtime cost of our algorithm is moderate. A version of `cmcc` that has been compiled with debug support on (`-g`, i.e., without optimizations), runs 2 – 4 times slower than the native C compiler[2].

We measured the impact of this register allocation strategy for various SPEC programs (`li`, `alvinn`, `espresso`, `compress`, `eqntott`, `ear`, `sc`, `matrix300`, `doduc`, `spice`, `nasa7` and `fpppp`). We first contrast our baseline algorithm (i.e., without the improvement of Section 4.2) with results using the well-known (enhanced) Chaitin-style approach. We use all registers on the MIPS, adhering to the standard calling convention.

[2]This figure is to be taken as preliminary. `cmcc` has not been tuned at all. Since the compiler used to compile `cmcc` does not support debugging of optimized code, we are forced to use the `-g` flag, since `cmcc` is still undergoing active development. Finally, `cmcc` is implemented in C++, and we cannot assess if this choice of language implies a performance penalty relative to the native compiler.

We start with the smallest region size, one basic block, and grow these until we have the interference graph for the function. All functions within the main loop of `alvinn` are inlined.

For both approaches, we run two experiments. First, we use only static information to guide the register allocation. That is, we use loop depth to estimate the execution frequency of a block. In our region-based approach, when considering the basic blocks inside a loop, we use breath-first order to select edges for fusing. In a second experiment, we use profile information from a prior execution.

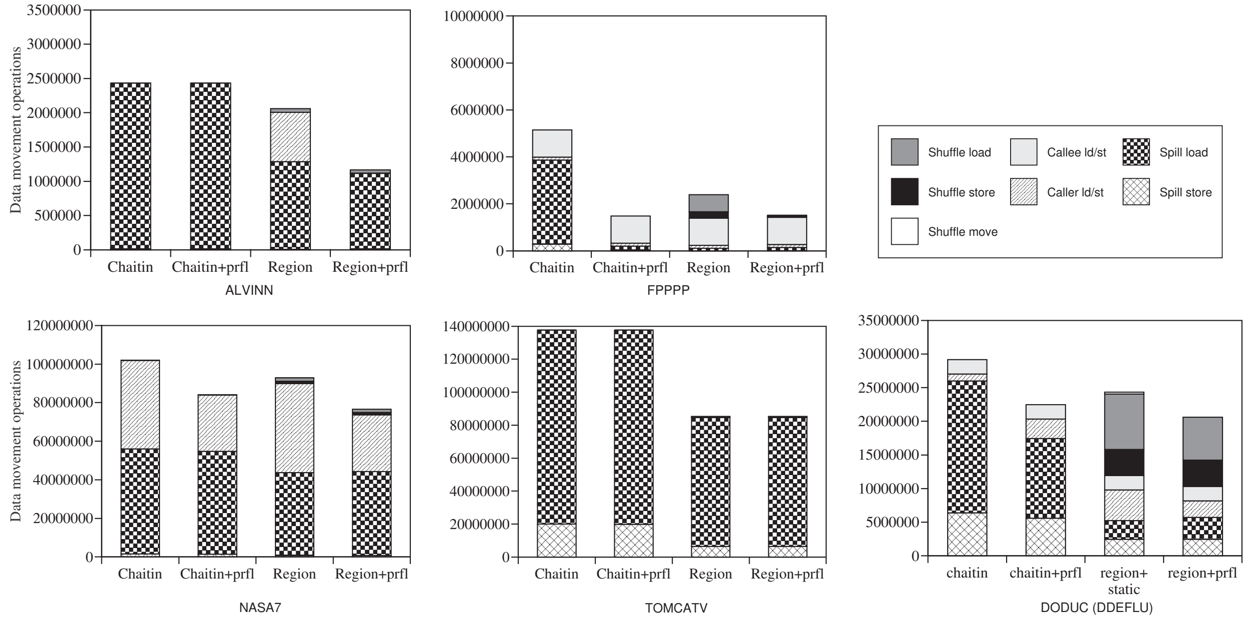

Figure 8: All available registers (26 int, 16 double).

6.1 All available registers

As expected, for programs with many small functions, Chaitin's algorithm works fairly well, and our approach produces identical results. For most of these programs, use of profile information improves the result, independent of the register allocation strategy. We therefore turn our attention to programs with significant register pressure.

`alvinn`, `nasa7`, `tomcatv`, `doduc` and `fpppp` are the programs with high register pressure, and here we see the limitations of the all-or-nothing approach to spilling that is the foundation of Chaitin-style register allocation. Figure 8 shows register allocation based on graph fusion is able to split live ranges properly and thereby cuts for `alvinn` nearly 52 % of the overhead required by Chaitin-style allocation. Chaitin-style allocation finds the "best" complete live ranges to put into registers, therefore, the result is not improved by using profile information. Our approach breaks live ranges, as can be seen by the lower number of spill loads. And profile information provides a better cost function for the register assignment phase (when deciding if a live range should go into a caller-save register or be spilled), resulting in a further improvement. Looking at the results for `fpppp` using static information, we see the same story. The overhead of `fpppp` is reduced by 54 %; the large contribution of shuffle code indicates that live ranges have been split. If we use profile information, then both approaches find the "right" live ranges and give the same result. For `tomcatv`, our approach removes 40 % of the overhead operations. Because the estimated static information provides a good estimation, the profile information does not help in this case. There are 3 big functions in `doduc`: `subb`, `supp` and `ddeflu`. These three account for the majority of data movement operations. `subb` and `supp` consist of only one big block. For those types of functions, our register allocator produces the same amount of data movement operations as a Chaitin-style allocator, since the sole basic block is the complete compilation unit. One simple way to deal with such blocks is to partition the big block into smaller blocks so that our approach can be applied to make better spilling and splitting decisions. We show in Figure 8 the result for `ddeflu`, where large portions of the spill loads and stores are replaced by shuffle code (17 % reduction for edge ordering based on static information and 30 % reduction for profile-based edge ordering).

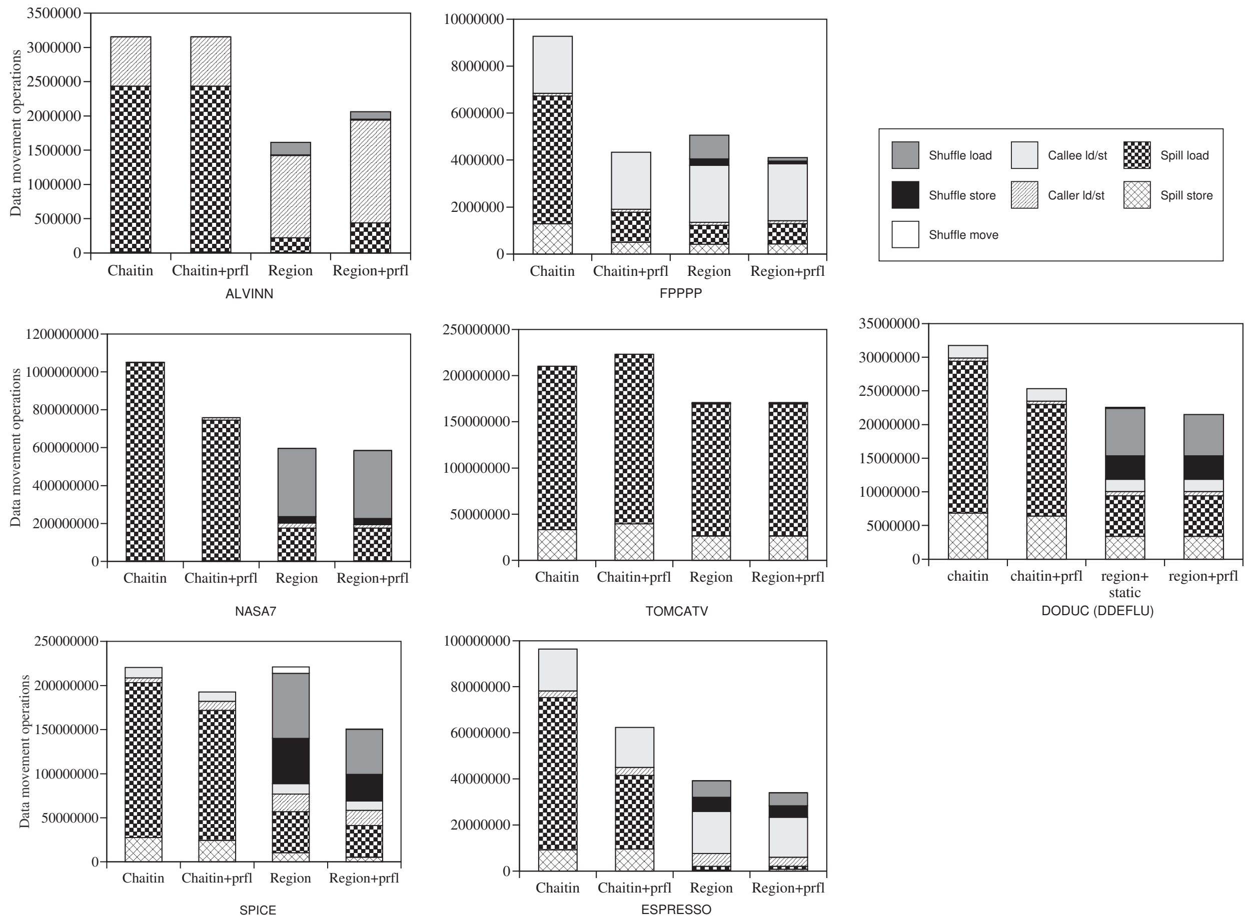

Figure 9: Smaller number of registers (14 int, 14 double).

6.2 Smaller number of registers

There are two causes for high register pressure: function inlining and a small number of registers. We now explore the second dimension and run our experiments with a reduced number of registers: 4 integer and 2 double argument registers, 2 integer and 2 double return registers (these are always managed by our register allocator), plus 2 integer and 4 double caller-save, and 6 integer and 6 double callee-save registers. These results are depicted in Figure 9. We notice that the overhead for Chaitin-style allocation includes now a fair number of operations to move values into and out of caller-saved registers. Interestingly, the result obtained by our algorithm using static information is better with a smaller number of registers. This indicates that maybe our our cost-functions can be improved further, since with fewer registers, live ranges do not grow as much. And it is easier for the register assignment phase to give a small live range the kind of register it requires (caller-saved or callee-saved). The reason why register allocation based on static information is superior to allocation based on profile information is related to this issue. If a live range is spilled during graph merging, the spill cost may be lower than if spilling happens during register assignment to avoid a high call overhead. The latter happens if profile information is used. `espresso` is another program where our approach is able to produce a better register allocation. Notice that a significant number of the overhead operations are shuffle load or stores. That is, our algorithm splits a number of live ranges so that one part ends up in a register and the other in memory. We see here the impact of dealing with spilling and splitting: Our register allocator eliminates spill load/store cost by introducing shuffle code on the less frequently executed edges. The overhead of `nasa7` is reduced by 46 % which is equal to 8 % of the total number of loads and stores.

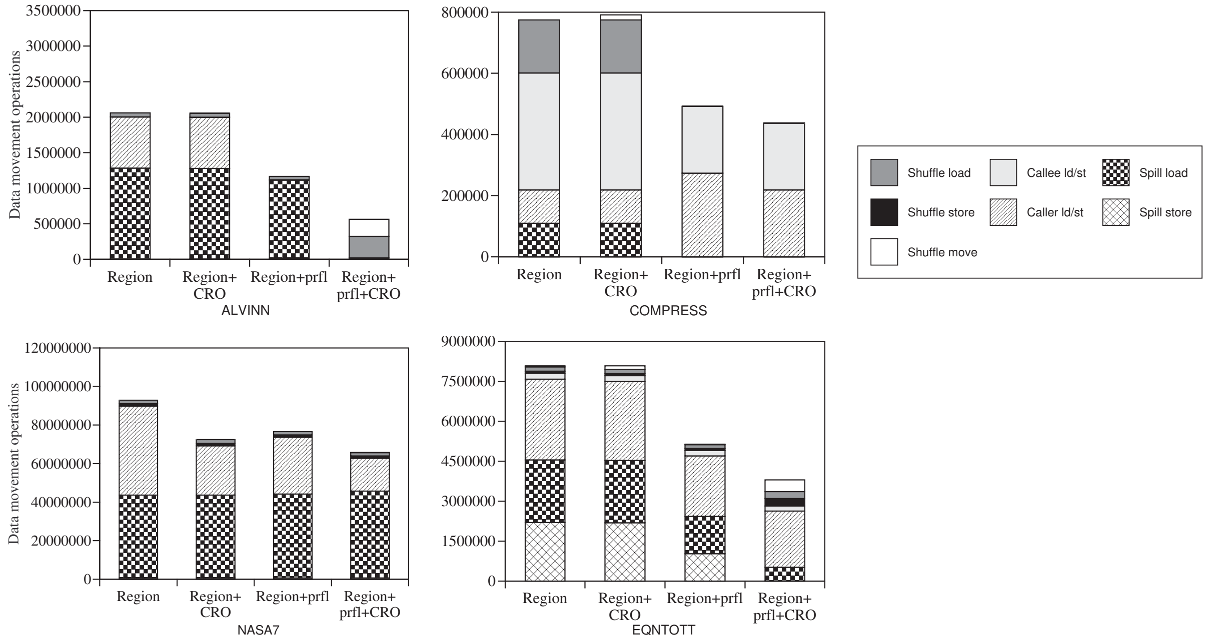

Figure 10: Impact of call cost optimization (CRO).

6.3 Call cost optimization

Given the importance of getting the right kind of register in a last experiment, we take positive action to limit the growth of live ranges, as described in Section 4.2. Figure 10 presents the results for `alvinn`, `compress`, and `eqntott` for this call cost optimization. Without profile information, the differences are negligible. (This is not surprising given the definition of the threshold and cost functions.)

Recall that a live range that is assigned a caller-save register but has high caller-save cost and low spill cost is spilled to memory during the register assignment phase, even though there are enough registers to hold the live range. Once a live range is picked to be spilled at this stage, no splitting is attempted. In other words, all references go through memory. The effect of such a decision is to increase the amount of spill code to reduce loads and stores of the caller-saved registers. The spill code that we see in Figure 10 (i.e., for `alvinn` and `eqntott`) is due to such live ranges that we spilled during register assignment. The call cost optimization of Section 4.2 suppresses coalescing of two live range segments lr_1 and lr_2 if one of them (say lr_2) has high caller-save cost (because the coalesced live range has high caller-save cost as well). Without coalescing, lr_1(with low caller-save cost) gets the desired caller-save register; lr_2 ends up in a memory or in a callee-saved register. If lr_2 is in memory, extra shuffle loads and stores are required. If lr_2 is in a callee-save register instead, no shuffle loads and stores are needed, but shuffle *moves* must be inserted. However, on most modern machines, moves are cheaper than loads/stores.

The results shown in Figure 10 illustrate the benefits nicely. This optimization succeed in isolating high caller-save cost regions at less frequently executed edges. Consequently, the register assignment phase spills fewer live ranges to memory. In the case of `alvinn`, about half of all overhead operations are shuffle moves. For `eqntott`, there are more shuffle moves than shuffle loads or stores.

Overall, in the case of `alvinn`, our approach of dealing with spilling and splitting, including call cost optimization, reduces total data movement overhead by 80 % compared to a Chaitin-style allocator.

7 Conclusion

In this paper we presented a new approach to register allocation. Our algorithm produces good results in those situations where a conventional Chaitin-style allocator breaks down. We have measured a reduction of the runtime cost of register allocation by up to 80 %. Of course, there are programs without significant register pressure. However, as aggressive compiler transformations become more common, even those simple programs are turned into challenges for a global register allocator. This algorithm is therefore especially attractive for compilers that look in regions beyond basic blocks for optimization opportunities or perform global scheduling. For those compilers, our register allocator presents an effective way to deal with programs that exhibit high register pressure.

Acknowledgements

We appreciate comments and feedback by D. Bernstein, IBM Israel, F. Chow, Silicon Graphics/MIPS, S. Freudenberger, HP Laboratories, and J. Ruttenberg, Silicon Graphics/MIPS.

References

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

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

[3] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 275–284. ACM, July 1989.

[4] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. SIGPLAN'91 on Programming Language Design and Implementation*, pages 192–203, Toronto, June 1991. ACM.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation by coloring. Research Report 8395, IBM Watson Research Center, 1981.

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proc. 18th Intl. Symp. on Computer Architecture*, pages 266–275. ACM/IEEE, May 1991.

[7] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12:501–535, Oct. 1990.

[8] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch direction from previous runs of a program. In *Proc. Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 85–97. ACM, October 1992.

[9] S. Freudenberger and J. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, pages 146–170. Springer Verlag, 1992.

[10] R. Hank, W. Hwu, and B. Rau. Region-based compilation: An introduction and motivation. In *Proc. 28th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, page (to appear), Ann Arbor, Nov 1995. ACM/IEEE.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1,2):229–248, March 1993.

[12] M. S. Johnson and T. C. Miller. Effectiveness of a machine-level global optimizer. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compler Construction*, pages 99–108. ACM, July 1986.

[13] P. Kolte and M. J. Harrold. Load/store range analysis for global register allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 268–277. ACM, June 1993.

[14] S. M. Kurlander and C. N. Fischer. Zero-cost range splitting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 257–265. ACM, June 1994.

[15] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1,2):51–142, March 1993.

[16] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 266–277. ACM, June 1994.

[17] T. A. Proebsting and C. N. Fischer. Probablistic register allocation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310. ACM, June 1992.

[18] D. Wall. Predicting program behavior using real or estimated profiles. In *Proc. ACM SIGPLAN '91 Symposium on Compiler Construction*, pages 59–70. ACM, June 1991.

Appendix: A more detailed example

In this appendix, we provide an example to show the interaction between edge ordering and shuffle code placement. The code fragment of Figure 11 (a), is taken from the function update_weights of `alvinn`.

Consider an edge ordering based on loop hierarchy:

$\langle B_3, B_3\rangle, \langle B_4, B_2\rangle, \langle B_3, B_4\rangle, \langle B_2, B_3\rangle, \langle B_5, B_5\rangle, \langle B_1, B_2\rangle, \langle B_4, B_5\rangle$.

B_3 requires all 3 registers for `tmp74, tmp76` and `tmp77`, which are all referenced within the block. In addition, there are two transparent live ranges induced by `tmp73` and `tmp75`. These two live ranges are spilled when the interference graph G_3 is simplified before fusion takes place. Fusing G_3 along edge $\langle B_3, B_3\rangle$ doesn't make any other splitting or spilling decision since the simplifiability invariant still holds for the resulting graph. B_2 also needs all 3 registers, therefore, the only transparent live range (`tmp74`) is spilled.

While fusing G_2 and G_4 along edge $\langle B_4, B_2\rangle$, the spilled live range of `tmp74` grows (to cover B_4 and B_2). After fusing along the remaining edges, the final resulting graph requires shuffle code to move values among different locations. This shuffle code is highlighted in bold face in Figure 11 (b). For `tmp75`, because there is no definition within the loop, the shuffle store doesn't have to be on edge $\langle B_2, B_3\rangle$ and is placed right after its definition, which is outside the loops.

If we change the edge ordering so that $\langle B_4, B_2\rangle$ and $\langle B_3, B_4\rangle$ are switched, then the register allocator ends up with a different spill decision, as seen in Figure 11 (c). The shuffle load of `tmp74` on $\langle B_4, B_5\rangle$ in Figure 11 (b) is not needed because B_4 is no longer in `tmp74`'s spill region. Furthermore, the shuffle load of `tmp75` is placed on $\langle B_4, B_2\rangle$ instead of $\langle B_3, B_4\rangle$, because the spill region for `tmp75`, indicated by a dotted line, contains B_4. (So instead of being in the loop, it is now on the loop back edge.)

```
tmp73 = ....
tmp74 = ....
tmp75 = ....
for ( ; tmp73 < limit1 ; ) {
     .. = .. tmp73 ..
     .. = .. tmp75 ..
  tmp76 = .. tmp75
  for ( ; tmp76 < limit2 ; ) {
    tmp77 = *(tmp76);
    *(tmp76) = tmp74 * tmp77;
    tmp76 = tmp76 + 4;
  }
  tmp73 = tmp73 + 400;
}
for ( ; tmp74 < limit3 ; ) {
  .. = .. tmp74 ..
}
```

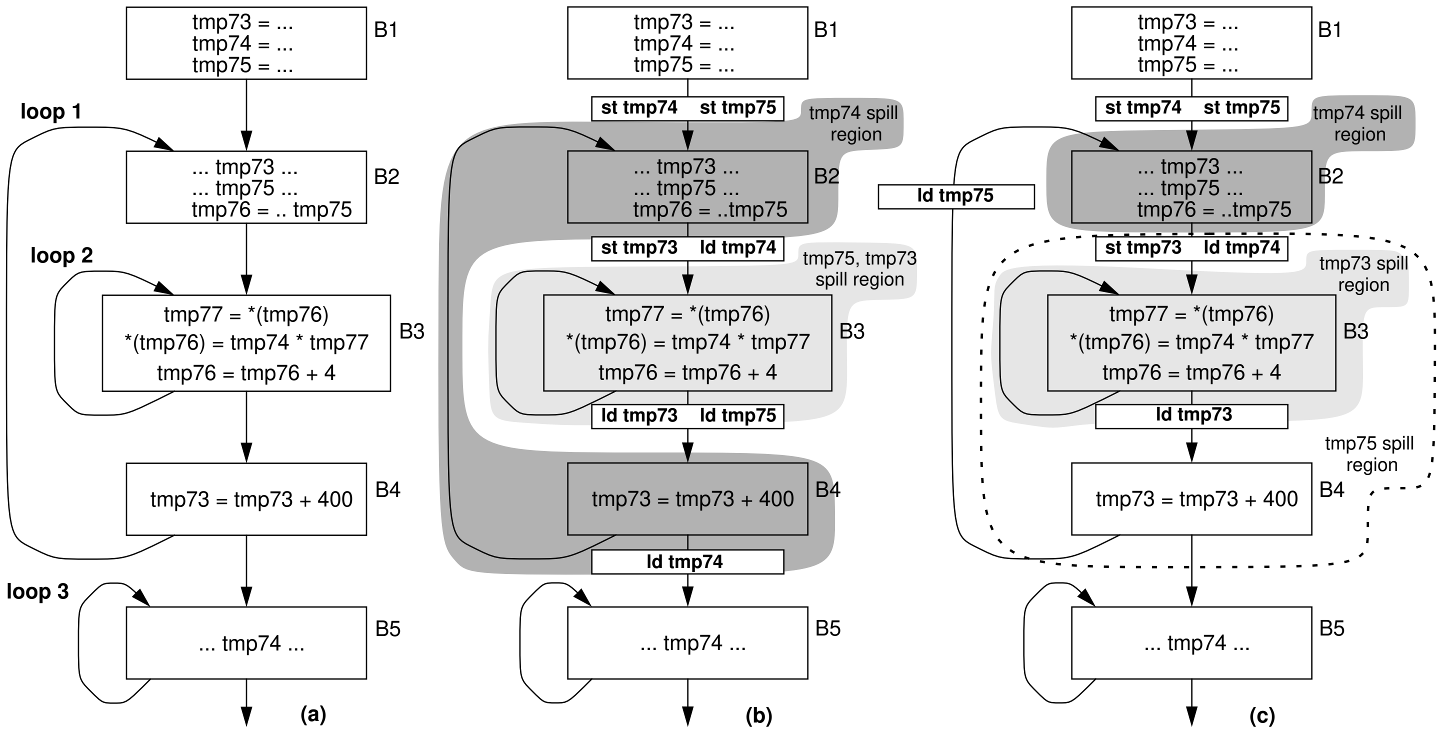

Figure 11: Code fragment from `alvinn` with N = 3.