Fusion-Based Register Allocation

Guei-Yuan (Ken) Lueh

May 2, 1997

CMU-CS-97-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.
Department of Electrical and Computer Engineering

Thesis Committee:
Allan Fisher, Chair
John Shen
Thomas Gross
Jaspal Subhlok

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, SPAWAR, or the U.S. Government.

Keywords: Code generation, register allocation, fusion-based register allocation, region-based compilation, spilling, splitting, delayed spilling, rematerialization

Abstract

Nowadays, compilers are looking for more optimizing opportunities by performing aggressive code transformations which introduce high register pressure. High register pressure potentially increases register overhead operations. Register allocation must deal with high register pressure well so that the performance gain of the code transformations is not thrown away by the increased overhead operations.

Register allocation must deal with three issues: spilling, live-range splitting, and register assignment. These issues are closely related to each other. Focusing on one issue and ignoring the others may deteriorate the quality of register allocation. This dissertation proposes a novel register-allocation approach that is fusion-based. Fusion-style register allocation starts off with constructing regions and applies graph fusion along control-flow edges to combine the interference graphs of regions into the interference graph for the whole function. Graph fusion integrates spilling, splitting, and register assignment in a seamless fashion. Fusion-based register allocation is sensitive to the ordering of control-flow edges that connect regions. Splitting is unlikely to happen at high priority edges (frequently executed) and likely to happen at low priority edges (infrequently executed).

This dissertation presents a register allocation framework that allows us to model various register allocation approaches, e.g., Chaitin-style, optimistic, priority-based, and fusion-style coloring. Fusion-style coloring also provides a nice framework to model various live-range splitting approaches such as the Tera, SSA, PDG, and Multiflow approaches.

This dissertation evaluates the effectiveness of fusion-style coloring under the register pressure caused by scope expanding transformations such as unrolling and inlining, whereas the traditional Chaitin-style coloring breaks down. The experimental results show that the number of register overhead operations using fusion-style coloring increases slowly. In other words, fusion-style coloring provides a solution to region-based compilation so that register allocation is no longer an obstacle to region-based compilation.

This dissertation presents and evaluats several enhancements to register allocation. These enhancements combine the strengths of both Chaitin-style and priority-based coloring into one. The integration of these enhancements provides a model of dealing with call cost (register saves/restores). The experimental results show that the model actually outperforms some existing approaches that have been adopted by researchers.

Acknowledgments

I would like to thank my technical advisor Prof. Thomas Gross for his guidance, wisdom, support, and infinite patience. I would also like to thank Allan Fisher, John Shen, and Jaspal Subhlok for spending their precious time to serve on my thesis committee.

I am very grateful to Ali-Reza Adl-Tabatabai, my officemate, colleague, and long time friend for his wisdom, sharp questions, and technical discussions. It was a great pleasure and wonderful experience to build the `cmcc` compiler with Ali.

Many thanks go to my friends for their generous help during my wife's pregnancy: Peter and Helen Lieu, Leaf and Shilling Huang, Thomas and Hui-Ru Tsai, Felix and Sharon Ng. Their help allowed me to continue focusing on my research work.

I would like to show my gratitude to the members of iWarp/Nectar group who made the research environment enjoyable and colorful.

I would like to thank my family and my wife's family for their encouragement. Finally, my deepest thanks go to my wife, Chia-Chian TuanMu, for her unselfish sacrifice, encouragement, and support. I am especially indebted to her for taking care of my beloving son, Sheen, and daughter, Anne, when I worked on my thesis. Without her genuine love, this thesis would not have been possible. I dedicate this thesis to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advanced VLSI technologies, performance of processors is approximately doubled every 18 months, and the density of memory is doubled every 24 months. However, the gap between the access time of memory and the processor speed enlarges. Based on the *principle of locality*, a memory hierarchy is introduced to keep values that processors operate as close to processors as possible [35]. The memory hierarchy usually consists of registers, caches (first-level, second-level, and third-level), and memory. Registers are the storage locations that are closest to the CPU and can be read and written at the clock speed of the CPU. Accessing data items that are not in registers incurs penalties because the data must be brought in from the next level of the memory hierarchy. The access time of data items between two levels may differ in order of magnitude. Because a superscalar machine can issue multiple instructions in each cycle[39], the penalties for the superscalar machine are more severe than for a non-superscalar machine. One task of the compiler is managing register storage locations to keep values that processors are most likely to access in registers so as to reduce the traffic of going up and down the memory hierarchy.

1.1 Compiler structure

A compiler is a piece of sophisticated and complicated software that translates one source language to a target machine's object code. To achieve modularity and reusability, a typical compiler is composed of three major phases: front end, global optimization, and back end (code generation). The front end usually parses the source code, performs syntax and semantic checks, and finally generates an internal representation (IR). Global optimization performs code improving transformations on IR. The back end translates IR into target machine code. The main purpose of IR is to cut loose the dependence among the three phases. For instance, adding a new language to the compiler requires only a new front end. Ideally, the global optimization and back end do not require any change. Likewise, translating a source language to a new machine code requires to rewrite only the back end.

There are some phases in the compiler that make the assumption that the target machine has an infinite number of registers. For instance, global optimization usually performs code improving transformations based on IR without any knowledge about the source language and the target machine. Global optimization, therefore, ignores the possibilities that the temporaries generated during code transformations may not reside in registers. Furthermore, parts of the back end may also make the same assumption. One implementation of the back end consists of code selection, code scheduling, register allocation, and code emission. The code selection maps the IR into an internal instruction representation that is very similar to the target machine code. The code selection phase uses *virtual registers* to hold values of expressions, variables, and constants. The code scheduling phase reorders instructions to exploit ILP. Register allocation then maps virtual registers to physical registers. The final phase, code emission, emits machine code. The assumption of an infinite number of registers simplifies the tasks of those phases because they don't need to apply machine dependent heuristics to constrain their tasks.

1.2 Register allocation

The task of register allocation is to map virtual registers to a limited number of physical registers. A common approach is to model the register allocation problem as a graph coloring problem. The graph comprises nodes and edges. Nodes represent live ranges of virtual registers; two conflicting (simultaneously live) live ranges are connected by an edge. The graph is called the *interference graph*. The interference graph is colored in such a manner that two conflicting live ranges are assigned different colors (physical registers).

Liveness analysis and reaching analysis determine the live range for each virtual register. Live ranges that are constructed in this manner, however, may comprise disjoint segments, resulting in an unnecessarily high number of conflicts for a live range. Renumbering [13] and web analysis [38] are two techniques to construct concise live ranges, which result in interference graphs of potentially lower degree.

Because finding the minimal number of colors for an arbitrary graph is NP-complete [27], most register allocation approaches assign colors to live ranges in some heuristic order. The coloring blocks when an approach fails to find a legal color for a live range. We say that the degree (or register pressure) of the interference graph exceeds N (the number of colors). When coloring blocks, the compiler must somehow lower the degree of the interference graph to allow coloring to proceed. Two major techniques have been developed to lower the degree of the interference graph.

Spilling: A live range lr is assigned a location in memory, and all references to lr 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.

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 may reduce the degree of the interference graph because the smaller a live range, the more likely it is that the live range interferes with fewer live ranges. The expectation is that each $lr_i(x)$ has a lower degree than $lr(x)$ and that the new graph can then be colored.

Simplification [13] is a common technique that people use to determine the coloring order in which live ranges are assigned colors. Simplification is based on the observation that if a node lr has degree $< N$, then lr can be trivially colored no matter what colors are assigned to the residual graph. Namely, there is always a legal color for lr even if each of lr's neighbors is assigned a different color because at most $N - 1$ colors are assigned to lr's neighbors. Such a node with degree less than N is called *unconstrained.* Simplification proceeds by successively removing unconstrained nodes from the graph. Each time a node lr is removed from the graph, the edges that are incident upon lr are also removed, and the degrees of lr's neighbors are decremented. Once all nodes have been removed from the graph, colors are assigned to nodes in the *reverse order* in which they were removed.

Simplification blocks when all remaining nodes have degrees $\geq N$, and at this time a live range is picked to be spilled based on a heuristic cost function. Once the spilled live range is removed from the graph, simplification then may proceed where it blocked because the degrees of the neighbors of the spilled live range are lowered.

Figure 1.1(a) illustrates a sequence of simplifying an interference graph with $N = 3$. The graph has 6 live ranges. Shaded live ranges are unconstrained. Initially, only lr_1 is unconstrained (degree = 2). The first step removes lr_1 from the graph. lr_2 becomes unconstrained after removing lr_1 and then is removed from the graph. At this point of time, there are two unconstrained live ranges, lr_3 and lr_5. Removing either one of them is legitimate (let's say lr_3). After removing lr_3, all remaining nodes become unconstrained and can be removed in any order. Assume that lr_4, lr_5, and lr_6 is the sequence of simplifying the remaining graph. Figure 1.1(b) shows the sequence of assigning colors to live ranges in the reverse order in which they were removed. Each hatch pattern indicates a color.

1.2.1 Overview

There are 9 chapters in this thesis. The thesis starts off with a review of prior approaches and addresses the challenges of register allocation (Chapter 2). We then look at the infrastructure used to implement and evaluate the thesis ideas (Chapter 3). This thesis introduces a new approach to register allocation that is based on graph fusion. Details and some important techniques of graph fusion are presented (Chapter 4). Nowadays, compilers performs aggressive code transformations to improve the code quality. It is very important to evaluate how register allocation (the fusion-based approach) and other code transformations interact (Chapter 5). The thesis also describes and evaluates some enhancements of register allocation (Chapter 6). Chaitin-style register allocation with the enhancements serves as the base line of our comparison in the thesis. We evaluate the improved Chaitin-style register allocation with some prior approaches (Chapter 7). Ultimately, the thesis discusses how the fusion-based approach can be incorporated with other existing register-allocation techniques (Chapter 8) and then presents the conclusions (Chapter 9).

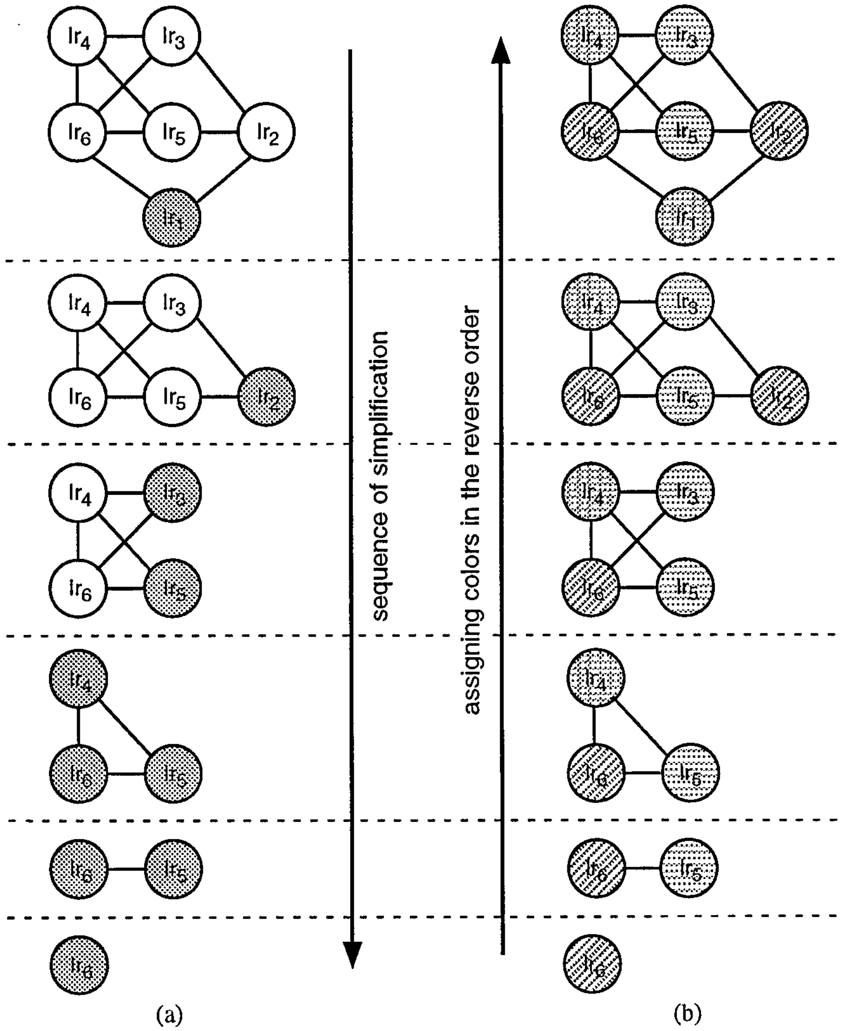

lr4
lr3
lr6
lr5
lr2
lr1
sequence of simplification
assigning colors in the reverse order
(a)
(b)

Figure 1.1: Simplification (N=3).

Chapter 2

Background

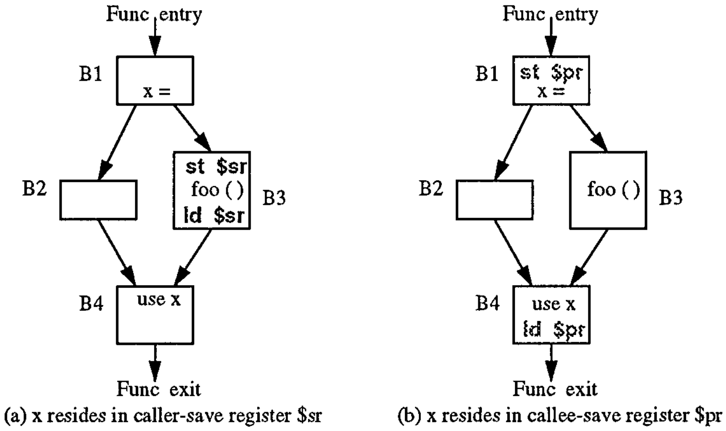
In this chapter, I present the cost model for register allocation (Section 2.1), review prior approaches to register allocation (Section 2.2), address shortcomings of the prior approaches (Section 2.3), and describe the challenges that a register allocator must deal with so as to obtain good quality register allocation.

2.1 Cost model for register allocation

Our basic machine model is a RISC processor that requires the operands of all operations to reside in registers. All our data are based on the code generator for the MIPS R3000 architecture. The register allocation cost includes all overhead operations that move operands in and out of a register; this cost includes, e.g., also overhead operations that move values from one register to another. That is, we compare the results of a register allocator against a perfect allocation with unbounded registers. The higher the cost (for a fixed number of registers), the worse the register allocator has performed.

The register allocation cost is the sum of three components: (i) spill cost (to move values to and from memory), (ii) call cost (to free up/restore registers upon procedure entry/exit), and (iii) shuffle cost (moving values from one live range to another).

Register allocation cost must include the call cost. If the register allocator focuses on the spill cost alone, the evaluation of the register allocator may be overly optimistic. The call cost, however, is influenced by the compiler's calling convention. Many compilers divide the registers into two sets, *callee-save* and *caller-save* registers, respectively. The distinction between these registers provides the register allocator with more choices when minimizing the call overhead [16]. There is a distinct cost associated with each kind of register assigned to a live range. Assume that there is a program that includes 3 functions, `car`, `bar`, and `foo`. Function `car` calls `bar`, and function `bar` calls `foo`. When a live range lr in `bar` ends up in a caller-save register and contains the function call `foo()`, we must pay the cost of saving and restoring lr's value around `foo()`. For instance, x in Figure 2.1 (a) resides in caller-save register $sr. The function call `foo()` divides lr_x, because the register allocator, without inter-procedural analysis, assumes that `foo` uses all caller-save registers. If lr ends up in a callee-save register, then this register must be saved (restored) at the entry to (exit from) `bar` (Figure 2.1 (b)), because the register allocator assumes that `car` demands all callee-save registers after the exit of `bar`. In Chapter 6, I present some heuristics that improve the quality of register allocation by reducing call cost.



(a) x resides in caller-save register $sr

(b) x resides in callee-save register $pr

Figure 2.1: Caller-save and callee-save costs.

A live-range splitting approach may partition a live range into smaller segments (smaller live ranges). Shuffle code is required at the splitting point to move value between two live ranges. Shuffle code can be of three types:

- *shuffle-move* (register-to-register): A `move` instruction shuffles the value between the two live ranges that are assigned to different registers.

- *shuffle-load* (memory-to-register): A `load` instruction moves the value from one spilled live range to the other live range.

- *shuffle-store* (register-to-memory): A `store` instruction moves the value from one live range to the other spilled live range.

There is no need for memory-to-memory shuffle code because the two live ranges of the same virtual register are *always* spilled to the same memory location.

2.2 Prior work

A number of improvements over Chaitin-style register allocation have been suggested. There are two perspectives to discuss prior approaches. We can focus on which kind of register overhead they attempt to reduce or address their shortcomings. We discuss the two perspectives separately. In this section, I investigate the major known register-allocation strategies, with a focus on which kind of register overhead they attempt to reduce. In Section 2.3, I address shortcomings of some approaches that are relevant to our register allocator.

2.2.1 Optimal

Goodwin and Wilken take the most aggressive approach [29] that formulates the register allocation problem as a 0-1 integer programming problem. The approach takes the spill, shuffle, and call costs into consideration and yields an optimal result under some assumptions (e.g., the register allocator does not reorder/reschedule instructions or perform inter-procedural register allocation).

2.2.2 Reducing spill cost

In essence, there exist two ways to reduce spill cost (i) finding better nodes to spill (i.e., better spilling heuristics) and (ii) splitting live ranges (in the expectation that the smaller live ranges interfere with fewer other live ranges).

Spilling heuristic

One of the main goals of register allocation is finding a way to assign registers to all live ranges of the interference graph. The register allocator must spill live range(s) to memory if the register pressure exceeds N. The register pressure is the minimum number of registers that the register allocator needs for assigning registers to *all* live ranges. Because the register allocation problem is NP hard [27], the minimum number of registers that the register allocator needs depends on the register allocation approach—this number may not be the same as the theoretical minimum number of colors for the interference graph. The heuristic that chooses which live ranges to spill has a direct effect on the quality of register allocation, so various heuristics have been investigated.

- Optimistic coloring [9] is based on simplification. Recall that simplification is a way to guarantee that there must be registers for the live ranges that are removed from the interference graph. Simplification is a heuristic approach to coloring which removes only live ranges whose degrees are less than N, and as such may miss legal coloring opportunities (too pessimistic). Optimistic coloring improves simplification by attempting to assign colors to live ranges that would have been spilled by the basic algorithm. Optimistic coloring delays spilling decisions until the color assignment phase. Spilling decisions are made during the color assignment phase: when no legal color exists for a live range to be colored (instead of when simplification blocks), this live range is spilled.

- Priority-based coloring [16] assigns registers to live ranges based on 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. The approach makes sure to assign registers to the most important live ranges and to spill the least important ones if necessary.

- Probabilistic register allocation [54], like priority-based coloring, also assigns registers to live ranges in a priority manner. Variables initially reside in memory instead of virtual registers (with load/store for every use/definition). The priority function is the probability that a variable x will reside in a register at a given use (load) multiplied by the benefit (savings) that this use accesses x's value from a register rather than memory. The inverse of the probability roughly indicates the distance between definitions of the value and its use. The longer the distance, the more likely it is that the register allocator assigns registers to other live ranges between the definitions and the use—therefore the lower the probability that the value resides in a register.

- The register allocation algorithm used in the RS/6000 compiler [5] improves on Chaitin's basic algorithm in two ways. First, the interference graph is colored three times, each time using a variation of the spilling heuristic (i.e., a different cost function), and the coloring resulting in the least total spill cost is selected. Second, when a live range lr is selected for spilling, instead of inserting a load before each use and a store after each definition of lr, the register allocator uses the *cleaning* technique that attempts to insert at most a single load and store inside each basic block. In effect, lr is split into segments that span at most one basic block.

- Rematerialization [10] spills live ranges that are cheaper to recompute than to store/load them back/from memory, e.g., loading constant values or computing addresses.

Live-range splitting

Several approaches have tried to split live ranges. The expense of splitting is that *shuffle code* must be inserted when a live range is broken into independent parts, since each part can be either in a different register or even in memory. 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. 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 then colors regions in order of their priorities, and shuffle code is inserted at the boundaries of these regions.

- Priority-based coloring [16] assigns colors to live ranges in a heuristic order determined by the priority function. Before colors are assigned, unconstrained (degree $< N$) live ranges are removed from the interference graph, since unconstrained live ranges can always be assigned legal colors after colors have been assigned to other live ranges. Color assignment blocks when no legal color exists for a live range lr to be colored, i.e., when all N colors have been taken up by lr's neighbors. At this point, lr is split.

Larus and Hilfinger implement the priority-based approach in the SPUR Lisp compiler [46].

- Briggs [7] uses the Static Single Assignment (SSA) representation of a program to determine splitting points. Splitting live ranges is taking place prior to coloring. 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.

- The Tera compiler (as described in [12]) 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. The second phase walks through the tile tree top-down and binds pseudo registers to physical registers. As coloring is performed hierarchically, shuffle code tends to be outside of the tiles at the bottom of the tree.

- The RAP compiler [53] colors the region nodes in a function's Program Dependence Graph (PDG), proceeding in a hierarchical manner from the leaves to the root. Chaitin's algorithm is used at each region node. This approach splits live ranges on region boundaries. A region is a collection of predicates and statements that are executed under the same control conditions.

- The Multiflow compiler employs trace scheduling as a framework for both register allocation and scheduling [25]. 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 on 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.

- The IMPACT compiler takes a similar approach as the Multiflow compiler [34]. The compiler selects a region and performs classical global optimizations, ILP optimization, code scheduling, and register allocation. Like the Multiflow approach, regions that are compiled first have more freedom in using registers; a similar binding technique is used to eliminate the shuffle code on region boundaries.

- Probabilistic register allocation [54] 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. 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).

- Coagulation code generation [50] integrates code generation and register allocation based on the ordering of prioritized control flow edges. The most important (frequently executed) regions have the maximum freedom in using registers, like in the Mulitflow compiler [25].

- Kurlander and Fischer [45] 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.

- Kolte and Harrold construct live ranges at a very fine granularity, essentially at every reference of live ranges [44].

2.2.3 Reducing shuffle code

Most live-range splitting approaches [12, 53, 25, 50, 54] reduce shuffle cost by prioritizing a function into regions and thereby performing register allocation in a hierarchical manner so as to place shuffle code in less important regions. Some approaches [7, 12, 53, 25] rely on some similar techniques such as biased coloring [7] to eliminate a shuffle move by assigning the same color to the two live ranges that the move connects.

2.2.4 Reducing call cost

A register may not be free at procedural boundaries, at call sites (caller-save) or at the entries/exits of procedures (callee-save). The cost of saving/restoring registers dominates the register overhead as the number of available registers increases. The register allocator can reduce the call cost intra-procedurally or inter-procedurally.

Intra-procedural allocation

The register allocator analyzes only the current function without any knowledge about calling or called functions. Selecting caller-save or callee-save registers for live ranges and optimizing placement of caller-save and callee-save code are two common approaches to reduce the call cost.

- The priority function used in the priority-based approach [16] considers both caller-save and callee-save cost. Register allocation assigns the kind of register to a live range which yields the maximum savings. If no register of the best type is available, the register allocator assigns the other kind of register to the live range. The approach spills live ranges whose savings for the assigned registers are negative.

- The generic callee-save convention saves/restores the values of callee-save registers at the exit/entry of the function. The code to save and restore is executed inevitably every time the function is invoked, so there may be redundant saves and restores if some callee-save registers are never used during the execution of the function. Chow uses *shrink-wrapping* [17] to move the callee-save savings and restorings close to the places where the callee-save registers are used. Hence save and restore operations are performed only when it is absolutely necessary.

- The "Chez Scheme" compiler optimizes the placement of caller-save code [55]. Measurements show that over two thirds of procedural activations actually make no calls [55]. The compiler, hence, favors using caller-save registers along the paths that contain no calls; it uses callee-save registers along the paths where function calls are inevitable. In other words, live ranges are split at the edge <P,S> where one (P/S) can reach the exit without making a call and all paths from the other one (S/P) to the exit contain calls. Shuffle code (a register move) is needed to move a value between two different classes of register at these splitting points.

Inter-procedural allocation

Inter-procedural register allocation usually assigns registers in a hierarchical manner over the call graph such that register allocation for a function f can take the register-usage information of functions g, h that are called by f into account.

- Wall defers register allocation until link time [61]. Variables that are not simultaneously live (inter-procedurally) are grouped together to use the same physical register. Variables of a procedure are never assigned to the same group as the variables of descendants on the call graph, so that the variables of f get different registers from g and h (f's descendants). Save/restore operations of registers across procedures, therefore, are unnecessary. The code generated by the code generation phase is annotated so that the linker knows what instructions must be deleted and can perform relocation once the register allocator assigns a register to a variable.

- Chow extends priority-based coloring to inter-procedural register allocation that constructs a call graph and compiles functions from leaves to the root [17]. The usage information of callee-save registers is propagated to the upper regions of the call graph. Function f tries to avoid using the same callee-save registers used by g and h such that save/restore operations of the registers become redundant and can therefore be eliminated.

- Steenkiste develops an inter-procedural register allocator in the context of a LISP compiler[58]. Because LISP programs tend to spend most of time in the bottom of the call graph, register allocation is performed from the leaves to the root over the call graph, like Chow's approach [17]. Different registers (not used by the descendants of the current function in the call graph) are assigned to live ranges of the current function.

- The HP compiler [56] performs inter-procedural register allocation using *global-variable promotion* and *spill-code motion.* Global-variable promotion transforms memory accesses to global variables into register references inside clusters of functions in the call graph. Spill-code motion elevates save/restore code for callee-save registers to infrequently executed functions.

2.3 Shortcomings of prior work

In the previous section, we study how prior approaches tackle each kind of register overhead. In this section, we look at prior approaches from the other perspective, i.e., how well they perform in terms of reducing overall register overhead. Because inter-procedural allocation is not implemented in our register allocation framework, I drop the inter-procedural approaches out of the discussion here. I classify some prior approaches that attempt to reduce the spill and shuffle costs into 5 categories: Chaitin-style coloring, optimal register allocation, splitting prior to coloring, splitting during coloring, and program structure based.

2.3.1 Chaitin-style coloring

The RS/6000 compiler [5] and optimistic coloring [9] improve Chaitin's basic algorithm. The former tunes the spilling heuristic, and the latter delays spilling decisions so as to assign registers to more live ranges than without delaying. The main problem with these two approaches is that the spilling decisions 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.

2.3.2 Optimal register allocation

Optimal register allocation [29] is an attractive approach which produces an optimal result. The major problem of the approach is the complexity of the algorithm compilation time and the resulting memory usage. Modern compilers perform aggressive scope enhancing transformations such as loop unrolling and function inlining so as to increase ILP and exploit more optimization opportunities. Optimal register allocation does not scale well in the presence of scope enhancing transformations. However, as the performance of processors keeps being doubled every 18 months, and faster integer programming solvers appear, the approach may become practical in the future.

2.3.3 Splitting prior to coloring

Briggs [7] determines splitting points using the SSA representation of a program. The approach also splits live ranges that span a loop at the loop boundary. Kolte and Harrold [44] partition a live range at a finer granularity by considering the ranges of instructions between loads and stores of a virtual register. These approaches make splitting decisions prior to coloring.

There are a couple of drawbacks in regard to making splitting decisions too early (prior to coloring). First, decisions regarding which live ranges to be split and where to split them are made prematurely. Thus, live ranges may be 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 [7]. Rather than determining the critical regions where splitting and spilling are beneficial, these approaches split live ranges arbitrarily and greedily, with the hope that later heuristic steps will clean up unnecessary splits. Second, the places where live ranges are split by these approaches are not necessarily the places of low execution probability. Although these approaches may use execution probabilities to splitting points, 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).

Even though Kolte and Harrold attempt to construct live ranges at the granularity of instructions between loads and stores, live ranges may be long under some circumstances. Consider a code for x that contains only one definition and use. Only one live range is constructed for x. The definition and use may be far apart, and the live range contains a lot of blocks with high register pressure. Splitting the live range is not possible even though splitting the parts of the live range that have high register pressure is desirable. Furthermore, it is unclear how they deal with the issue of eliminating shuffle code.

2.3.4 Splitting during coloring

The priority-based coloring [16] approach is an alternative framework that allows splitting decisions to be delayed until coloring blocks. Colors are assigned to live ranges in a priority fashion. When there is no legal color for a live range lr (coloring blocks), the approach splits lr. To facilitate splitting, the live range for a candidate x is defined as a collection of *live units*, where each live unit is a basic block within which x is live. Splitting forms a new live range $lr'(x)$ by starting from a seed live unit and incrementally adding live units to the new live range until adding one more live unit renders $lr'(x)$ uncolorable. 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 x 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 splitting points. To reduce shuffle cost, shuffle code should be placed at points of low execution probability. The priority based approach neither takes execution frequency nor program structure into account when splitting live ranges, and there is no guarantee that splitting 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 color assignment to optimize the placement of the shuffle code.

2.3.5 Program structure based

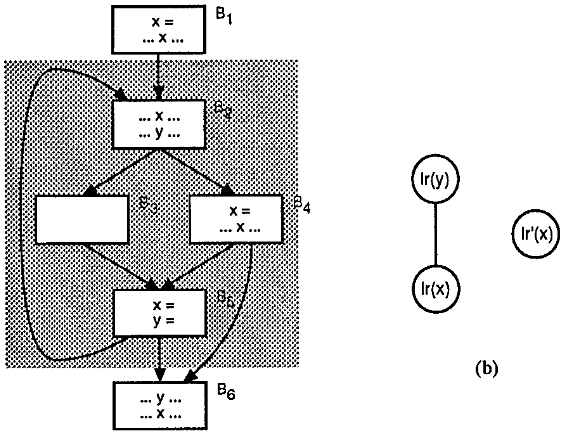
Program structure based approaches [12, 53, 25, 34, 50, 54] attempt to make graph coloring sensitive to program structure by dividing a program into regions and prioritizing the regions according to execution probabilities. The register allocator then colors regions in the order of their priorities, and shuffle code is inserted at the boundaries of these regions. As coloring is performed hierarchically, the higher priority regions have more freedom in using registers, and shuffle code tends to be outside the higher priority regions. As long as the regions reflect the execution frequency, they favor the most frequently executed parts of a program.

Those approaches all make early binding decisions when coloring is performed to a region—either assign real physical registers to live ranges or determine which live ranges should share the same registers. Early binding decisions impose unnecessary constraints to lower priority regions because the lower priority regions must bind to the already-made coloring decisions.

Example of early binding

Let us consider the Tera approach [12] as an example. Figure 2.2 illustrates unnecessary constraints imposed on register allocation by premature coloring decisions. The approach constructs a tile tree for a program. The code in Figure 2.2(a) consists of two tiles, one for the loop (the shaded region) and one for blocks B_1 and B_6. Recall that the Tera compiler performs register allocation for the tiles in two phases. The first phase colors each tile in a bottom up fashion (from the bottom of the tile tree up to the root tile). Pseudo registers are assigned to live ranges in the first phase. Once two variables in a tile are assigned to the same pseudo register, the parents of the tile must adhere to this decision.

The loop tile is colored first. The interference graph of the loop tile is depicted in Figure 2.2(b). Inside the loop tile, 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 either $lr(x)$ or $lr(y)$. 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 on either $\langle B_4, B_6\rangle$ or $\langle B_5, B_6\rangle$, since $lr(x)$ and $lr'(x)$ merge in



(a)

(b)

Figure 2.2: Premature coloring decision with $N = 2$.

B_6. The notation $\langle B_m, B_n \rangle$ represents a control-flow edge that goes from block B_m to B_n. 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.

2.4 Challenges

There are several challenges to register allocation. The compiler's decisions to those challenges influence the quality of register allocation.

- *coloring order*: In which order are live ranges assigned colors.

- *spilling versus splitting*: When the register allocator perceives that register pressure is high and ought to take some actions to lower the register pressure. Both spilling and splitting live ranges reduce the pressure. The register allocator has to choose which technique to use to lower the register pressure. Which live ranges are spilled or split?

- *splitting points*: In the case of splitting live ranges, how does the register allocator determine splitting points for the live ranges?

- *reconciliation*: How to reconcile register allocation of two regions is a fundamental challenge to the program structure based approaches. If the register allocator performs register allocation for a region and doesn't reconcile the region with other regions, excessive shuffle code may arise at region boundaries.
- *call cost*: What is the heuristic of assigning a callee-save or caller-save register to a live range?

2.5 Preview of fusion-based register allocation

In this chapter, I briefly review prior approaches on register allocation, and address their shortcomings. None of the heuristic (non-optimal) approaches taken so far considers the program structure during both splitting and coloring. However, splitting and coloring are closely related to each other. The decision to allow some live ranges to reside in the same register (at different times) affects subsequent splitting and coloring decisions. The ideal model is to postpone the splitting and register binding decisions as late as possible so that more information can be considered.

Fusion-based register allocation is a *bottom-up* approach that *builds 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. The 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.

Chapter 3

Experimental Framework

In this chapter, I describe cmcc, the Carnegie Mellon C Compiler, which provides the context for my implementation of register allocation. I also describe the register allocation framework that provides the infrastructure for various register allocation approaches.

3.1 The cmcc compiler

The cmcc compiler is an optimizing, retargetable compiler for a complete programming language (ANSI C). cmcc consists of three major phases, front end, global optimization, and code generation. The front-end phase uses the lcc ANSI C front end [24] (written in C). The code generation of lcc is modified to generate cmcc Intermediate Representation (IR) operations. The global optimization and code generation phases are coded in C++. The global optimization phase accepts cmcc's internal IR generated by the front-end phase and performs global optimizations. Table 3.1 lists the global optimizations performed by cmcc. The code generation phase performs code selection, local code scheduling (list scheduling), and register allocation. cmcc produces assembly language for a variety of target machine architectures: MIPS [40], SPARC [51], and iWarp [6]. The results (register overhead operations and execution time) in this dissertation are based on the MIPS architecture.

Table 3.2 contrasts the optimized code generated by cmcc with the optimized code produced by gcc and the native MIPS cc compiler for the MIPS architecture, and with the code produced by the native SUN cc compiler for the SPARC architecture. (This table presents the performance relative to these compilers; a number of less than one means that

Loop unrolling and peeling
Linear function test replacement
Induction variable expansion
Induction variable simplification
Constant propagation and folding
Induction variable elimination
Assignment propagation
Partial dead code elimination
Dead assignment elimination
Partial redundancy elimination
Strength reduction
Branch optimizations
Instruction scheduling
Global register allocation (using graph coloring)
Register coalescing

Table 3.1: Optimizations performed by cmcc

	MIPS		SUN
program	gcc -O2	cc -O	cc -O
alvinn	1.07	0.94	0.79
compress	0.84	0.95	1.04
ear	1.08	0.95	0.99
eqntott	1.14	1.09	0.59
espresso	1.06	1.05	1.02
li	0.98	1.05	1.16
sc	1.07	1.03	1.00

Table 3.2: Performance of optimized code generated by cmcc, relative to optimized code generated by gcc (version 2.3.2) and MIPS cc on a DECstation 5000/200, and SUN cc on a SPARC.

cmcc produces better code.) The sample programs are the C programs of the SPEC92 suite. Overall, even without serious performance tuning, cmcc produces code that is roughly of the same quality as the code produced by the other compilers.

3.2 The register-allocation framework

In the cmcc compiler, register allocation is performed after code scheduling. The cmcc register allocator takes a scheduled code sequence in which instructions use virtual registers and assigns physical registers based on graph coloring.

The cmcc compiler implements a register allocation framework [1] for experimentation of various register-allocation approaches. A framework partitions the task of the framework into abstract classes and defines their responsibilities and collaboration [26]. A framework fosters code reuse by capturing the common structure to its application domain. Various applications that share the same framework are easy to maintain because the code is concise and any change or extension to the framework benefits to all applications. Furthermore, a framework facilitates a *fair* comparison which is very important for researches. For example, if an improvement is added to some parts of register allocation (e.g., the construction of the interference graph, or color assignment) then *all* register-allocation approaches can benefit from the improvement. Using a framework avoids a bias in favor of a one particular approach.

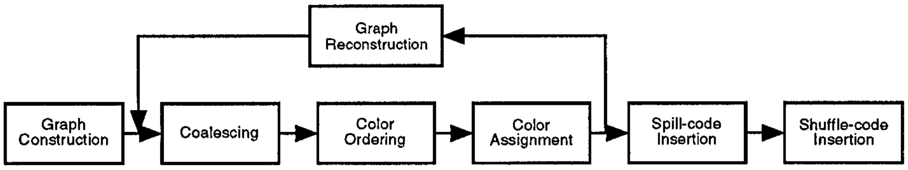


Figure 3.1: Structure of the register allocator.

I identify seven phases in the register allocator: *graph construction, live-range coalescing, color ordering, color assignment, graph reconstruction, spill-code insertion,* and *shuffle-code insertion* (as illustrated by Figure 3.1). This register-allocation structure allows us to model a wide range of register allocation approaches. A register-allocation approach customizes the register-allocation framework by providing specific implementations of the abstract classes (subclassing).

Graph construction

The graph-construction phase builds the interference graph for the input instruction sequence. This phase utilizes the data-flow analysis framework [1] to compute the *reaching* and *live* properties of virtual registers for building live ranges. *reaching* determines if a definition of a virtual register reaches the entry/exit of a basic block. *live* decides if the value of a virtual register is used/referenced after the entry/exit of a basic block. Various approaches may split live ranges to lower register pressure while constructing the interference graph. Details are presented in Chapter 4.

Coalescing

The coalescing phase eliminates copy instructions that move data values between two non-conflicting live ranges by coalescing the two live ranges into one. If a copy instruction moves a value from a live range to a physical register, or from a physical register to a live range (such as calling convention), the coalescing phase sets the *partner* information. A live range may have several partners. The partner relationship indicates that a live range would like to receive the same color as one of its partners. The later color-assignment phase can thereby take the partner information into consideration and assign colors to live ranges (biased coloring [7]).

Color ordering

There are two data structures that are used as the interface between the color-ordering and color-assignment phases: (1) the color stack ($\mathcal{C}$), and (2) the spill pool ($\mathcal{S}$). The color-ordering phase heuristically determines the order in which live ranges are to be assigned colors and pushes live ranges onto $\mathcal{C}$ based on the ordering — the higher the position within $\mathcal{C}$, the higher the priority. $\mathcal{S}$ keeps all spilled live ranges. While deciding the color ordering, this phase may make spilling decisions due to lack of colors and may add the spilled live range to $\mathcal{S}$. The spill-code insertion phase allocates spaces on the local heap for spilled live ranges in $\mathcal{S}$, and inserts actual spill code.

Color assignment

The color-assignment phase pops live ranges from $\mathcal{C}$ and finds legal colors (non-conflicting colors) for them, i.e. this phase assigns colors based on the ordering computed by the previous phase and uses the interference graph to make sure that conflicting live ranges are assigned different colors. Biased coloring, which considers the partner information, is used to find legal colors so as to eliminate copy instructions. That is, a non-conflicting partner's color of a live range may be assigned to the live range to eliminate the copy instruction that move the value between the partner and the live range. Details of biased coloring are presented in Section 4.5. If this phase is unsuccessful to find a legitimate color for a live range because all colors are already taken by its neighbors, the live range is then spilled and added to $\mathcal{S}$.

The spilling that we have discussed so far is called *out-of-color* spilling because live ranges are spilled with the goal of reducing register pressure. Register allocation may also make spilling decisions to reduce the overall overhead operations even though there are still enough colors for live ranges. This kind of spilling is called *call-cost* spilling (details are discussed in Chapter 6).

Graph reconstruction

For simplicity, the color-ordering and color-assignment phases make spilling decisions under the assumption that a spilled live range does not need any register resource. However, for *load/store* machines, memory accesses must be done by load/store instructions, which require to use a register as one operand (destination/source) [35]. In other words, at every occurrence of spilled live ranges, there is an instant requirement of a register. Hence, spill code needs to acquire registers. Reserving registers for spill code is one way of making sure that there are registers for spill code. A register allocator reserving registers for spill code, nevertheless, works with fewer colors than one without reserving registers. Our register allocator reserves no registers for spill code; that is, the register space is not divided into "local" and "global" registers. To ensure that there are registers for spill code, the register allocator rebuilds the interference graph (constructs live ranges for spill code) after spilling live ranges and restarts from the coalescing phase.

There are two ways to reconstruct the interference graph: (1) spill code (a load/store) is inserted into the schedule immediately before/after a use/definition, the interference graph is thrown away, and the whole coloring process restarts from the first phase; (2) the existing interference graph is modified by producing a new small live range for each occurrence of a spilled live range (no spill code is inserted; the newly produced live ranges span only one instruction), and the coloring process can then skip the graph construction phase and proceed from the coalescing phase with the modified graph. Those newly produced live ranges are called *spill-code live ranges*. The first approach reuses the code for constructing the interference graph. The second approach favors compilation speed, because the interference graph is not rebuilt from scratch. `cmcc` takes the second approach (i.e., it reuses the graph, to save compilation time).

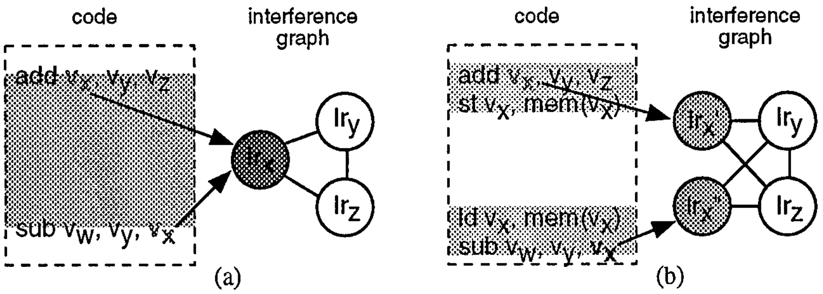


Figure 3.2: Spill-code live ranges.

Figure 3.2 illustrates an example of spill code. Enclosed in dotted lines are sequences of code. What we are interested in is the live range lr_x. The `add` instruction defines x, and the `sub` instruction is the last instruction that uses x. The shaded area in Figure 3.2(a) is the range in which x is live before lr_x is spilled. Any reference of x within the area accesses the value of lr_x. When lr_x is spilled, the first approach of reconstructing the interference graph first inserts the two spill code operations (one store and one load as shown in Figure 3.2(b)). The graph construction phase then rebuilds the interference graph for the new code sequence. As a result, the x defined in the `add` instruction and the x accessed by the `sub` instruction belong to two different live ranges, lr'_x and lr_x", because the `ld` instruction re-defines x (start of lr_x"). The second approach of reconstructing graph constructs the new interference graph by modifying the original interference graph of Figure 3.2(a). The `st` and `ld` instructions are not inserted into the code sequence until the later spill-code insertion phase.

The next pass of register allocation (restart from the coalescing phase) attempts to assign registers to all the live ranges including spill-code live ranges. Other live ranges may be spilled during this iteration, the register allocator repeats register allocation until no more live range is spilled. Because spill-code live ranges span only one instruction, spilling them does not help to lower register pressure but increases spill code instead. Therefore, they are marked *non-spillable* so that the register allocator won't spill them in subsequent iterations of register allocation.

Figure 3.3 illustrates an example such that the register allocator needs to perform multiple iterations of register allocation. A white circle in the figure indicates a definition, and a black circle indicates a use. Solid vertical lines indicate live ranges of virtual registers. The example has two colors and three live ranges, x, y, and z. The three live ranges overlap (as shown in Figure 3.3(a)) so they interfere with each other and must reside in distinct registers. Hence, the register allocator must spill one live range (assume that x is spilled). The live range x is broken into two spill-code live ranges (depicted in Figure 3.3(b)). The dotted line between the two spill-code live ranges indicates the value of x residing in memory instead of a register. Even though the live range x is spilled already, we see that the maximum number of register requirement is still three at the use of x (shaded range). Thus the register allocator must spill another live range (assume that z is spilled) during the second iteration of register allocation and re-start register allocation over again. After z is spilled, the maximum number of register requirement is two at any point of the program (shown in Figure 3.3(c)).

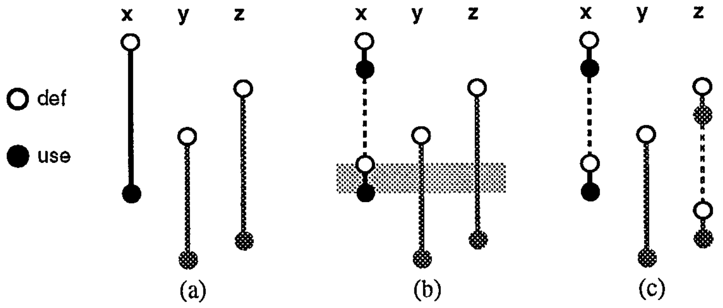


Figure 3.3: Spill code with $N = 2$.

Spill-code insertion

The spill-code insertion phase allocates spaces for spilled live ranges on S. We classify virtual registers into *global* and *local* virtual registers. Global virtual registers are those who appear in multiple blocks. Local virtual registers are those that appear within one single basic block. The spilled live ranges that correspond to the same virtual register are always spilled to the same location. In other words, spaces are allocated to virtual registers. No space is allocated to a virtual register if no live range of the virtual register is spilled. Global virtual registers have their own locations. Nonetheless, spaces for local virtual registers are squeezed/compacted using the same graph coloring technique. That is, the locations already allocated to the neighbors of a spilled live range are not allocated to the live range.

After spaces for spilled live ranges are allocated, the spill-code insertion phase traverses each instruction and inserts spill code for each reference of spilled live ranges. Each reference of a spilled live range has its own spill-code live range, which acquires a register for loading from or storing to memory. If there exists a source operand (use) of an instruction and the corresponding live range is a spill-code live range, then a load instruction is inserted before the instruction to load the value of the operand from memory. If there is a destination operand (definition) of an instruction which is a spill-code live range, then a store is appended after the instruction to spill the value of the live range to memory.

Shuffle-code insertion

Register allocation may include one or more live-range splitting approaches to split live ranges into smaller segments to reduce register pressure. In this register allocation framework, splitting can occur only on control-flow edges (not within a basic block). The shuffle-code insertion phase goes through control-flow edges and inserts shuffle code for those live ranges that should span across the edges but reside in different storage locations instead (details are discussed in Section 4.6).

3.2.1 Modeling existing register-allocation approaches

This section elucidates how various register allocation approaches fit into this register allocation framework. Table 3.3 shows how the register allocation framework can model various register-allocation approaches.

Chaitin-style coloring [13, 5] uses a *simplification* process to determine the coloring order. Simplification moves all *unconstrained* live ranges (degree $< N$) to the C stack; when

	Register allocation			
Phases	Chaitin-style	Priority-based	Optimistic	Fusion-style
Graph construction	function	function	function	region out-of-color spill split
Graph reconstruction	X	X	X	X
Coalescing	X	X	X	X
Color ordering	simplification out-of-color spill	priority	simplification	simplification
Color assignment	call-cost spill	out-of-color spill call-cost spill	out-of-color spill call-cost spill	call-cost spill
Spill-code insertion	X	X	X	X
Shuffle-code insertion				X

Table 3.3: Model different register-allocation approaches.

a node is removed, all edges incident upon the node are also removed, opening up further opportunities for simplification. If simplification blocks (all nodes have degree $\geq N$), a least-cost live range is spilled (added to S) so that the simplification process can proceed. The Chaitin-style approach makes out-of-color spilling decisions in the color-ordering phase.

Optimistic coloring [9] also uses simplification to determine the coloring order. When simplification blocks, however, rather than spilling, optimistic coloring pushes a least-cost live range onto C, removing the live range from the residual graph. In this manner, a live range that would have been spilled by the Chaitin-style approach is given another chance at receiving a register in the color-assignment phase. Optimistic coloring defers out-of-color spilling decisions until the color-assignment phase, which spills those live ranges that cannot be assigned legal colors (all colors are taken up by neighbors).

The priority-based coloring that I implement is Chow's priority-based coloring [16] *without* live range splitting. Priority-based coloring determines the coloring order according to a priority function. This approach first separates out unconstrained live ranges and pushes them onto C, since unconstrained live ranges can always be assigned legal colors. The priority function determines the order in which constrained live ranges are pushed onto C (from the smallest to the biggest). During the color-assignment phase, if a live range cannot be assigned a legal color (because of conflicts with the neighbors' colors), then priority-based coloring spills this live range.

This framework can also model other approaches to live-range splitting that take the program structure into account, such as the approaches taken by Tera [12], Multiflow [25], and RAP [53], as well as splitting based on the SSA representation [7], and the integrated approach to splitting and spilling provided by fusion-style coloring[48]. Fusion-style coloring makes splitting and spilling decisions in the graph construction phase. A more detailed discussion of how fusion-style coloring models other approaches is presented in Chapter 4.

This framework emphasizes what priority-based coloring [16] and the Chaitin-based approaches have in common: they all try to determine the color ordering in which live ranges are assigned colors (registers). Priority-based coloring uses cost analysis as the priority to determine the ordering and guarantees that the register assignment phase assigns registers to the most important (high savings of memory accesses) live ranges. Chaitin-style, optimistic, and fusion-style simplify the interference graph to decide the ordering. Optimistic coloring is similar to priority-based coloring in the sense that both approaches delay out-of-color spilling decisions until the color-assignment phase.

Fusion-style register allocation requires the shuffle-code insertion phase to insert code to move values at splitting points. Shuffle-code insertion is just a *nop* for the Chaitin-style, optimistic, and priority-based approaches because these approaches do not split live ranges.

Chapter 4

Fusion-Based Register Allocation

In chapter, I explain the structure of fusion-based register allocation, the fusion operation that fuses two interference graphs, and some techniques used during graph fusion. In Section 4.1, I present an overview of fusion-based register allocation. In Section 4.2, I present the fusion operator that fuses two graphs. In Section 4.3, I discuss how fusion-based register allocation models different splitting approaches. In Section 4.4, I describe delayed spilling that avoids making premature spilling decisions while two graphs are being fused. In Section 4.5, I describe the biased coloring technique that is used to eliminate unnecessary move operations. In Section 4.6, I present the ordering of shuffle code and the optimization of placing shuffle code. In Section 4.7, I measure the influence of fusion-based register allocation for various SPEC92 programs.

4.1 Overview

Fusion-based register allocation identifies regions, builds the interference graph for each region, and then merges all regions' interference graphs along control-flow edges. In this section, I concentrate on two essential phases of fusion-based register allocation, graph construction and color assignment, that make spilling, splitting, and register binding decisions. Graph fusion takes place during the graph construction phase which is further divided into three steps (shown in Figure 4.1): *region formation, graph simplification*, and *graph fusion.*

Here we define some notations that are used in this chapter.

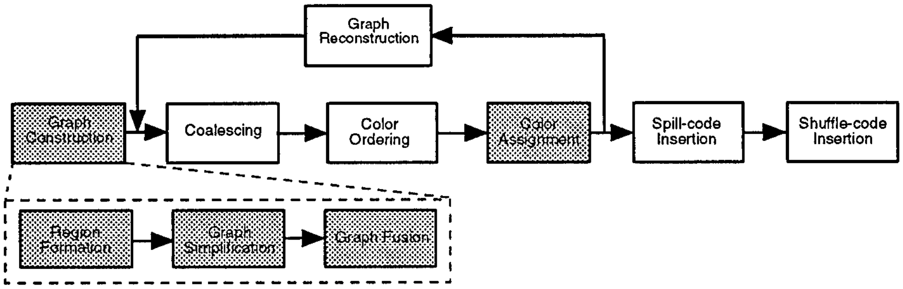


Figure 4.1: Structure of the fusion-based register allocator.

- $\langle B_m, B_n \rangle$ indicates a control-flow edge that goes from block B_m to B_n.
- G_R represents the interference graph of region R.
- $lr_i(x)$ denotes the segment of a virtual register x's live range $lr(x)$ that extends block B_i, and there is one or more segment for every region R where x is live and reaching.
- $[lr_x\ , lr_y]$ is the interference edge that connects live ranges lr_x and lr_y.
- lr_{ij} is the combined live range of lr_i and lr_j.
- $neighbor(lr)$ is the set of all live ranges that lr interferes with.
- $\varphi(lr)$ is the set of all interference edges of lr.
- $G_{R_1 \cup R_2}$ refers to the resulting graph of fusing (or merging) G_{R_1} and G_{R_2}.
- $G^E_{R_1 \cup R_2}$ represents the resulting graph of fusing G_{R_1} and G_{R_2} along the control-flow edge E.

4.1.1 Region formation

In this phase, regions are formed using any number of possible techniques. A region consists of multiple blocks (or one single block) and the edges that connect the blocks within the region. For example, a region can be a single basic block, a trace [47], a superblock [37], a region as defined in [33], the blocks at a particular static loop nesting level [12], or the blocks within a PDG region node [53]. Control flow edges that lie outside of regions are then ordered according to some priority functions consistent with the region formation approach, e.g., edges entering innermost loop regions are placed 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 [23, 60], from static estimates such as loop nesting depth [16], or from static branch estimates [47]. The choice of edge ordering is orthogonal to register allocation but of course influences 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.

There are two issues related to choosing the size of the base region: spill and shuffle costs. The smaller the size of the base region, the more likely the register allocator generates shuffle code and reduces the spill cost because live ranges are allowed to be split at finer granularity (at the cost of shuffle code) to reduce the spill cost. On the other hand, the bigger the size of the base region, the more likely the register allocator generates spill code to reduce the shuffle cost. It is impossible to state that one strategy is better than the other because the actual outcome depends on characteristics of the programs. Chapter 5 discusses those two issues in detail.

4.1.2 Graph simplification

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., a live range $lr(y)$ that spans 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 ranges to spill is delayed until more global knowledge about the reference patterns is available; this phase determines only *how many* transparent live ranges need to be spilled within each region. The next phase, graph fusion, 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 4.4. There, we also describe what to do if the compiler must spill more live ranges than there are transparent ones.

4.1.3 Graph fusion

The graph fusion phase takes the sequence of control flow edges determined by the region formation phase and fuses interference graphs along each edge. Graph fusion 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_{R_1} and G_{R_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 $G^{E}_{R_1 \cup R_2}$ back to the two original graphs, G_{R_1} and G_{R_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 simplifiability invariant allows us to avoid making any coloring decisions prematurely during the graph merging phase. For the example of Figure 2.2, assume that we choose loops as regions. There is no coloring assignment decision for any live range while graphs of the loop region are fused. All we know is that the interference graph of the loop region is colorable. The actual coloring decision is deferred till the color-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.

4.1.4 Color assignment

In this phase, physical registers are assigned to live ranges. The simplifiability invariant guarantees that once all interference graphs have been fused, the resulting interference graph is colorable and out-of-color spilling or splitting decisions have already been made. Maintaining the simplifiability invariant is necessary, otherwise we are back to the original problem of making splitting and spilling decisions based on a whole interference graph. There exist further opportunities for improving the code in this phase: biased coloring [7] may eliminate shuffle code, optimistic coloring [9] may assign registers to live ranges that have been spilled by simplification, and the enhancements discussed in Chapter 6 may decide to spill live ranges to reduce call cost (call-cost spilling).

4.1.5 Properties of graph fusion

Initially, each interference graph is sparser (i.e. less constrained) than the function-wide interference graph $G_{function}$ because each graph is only a portion of $G_{function}$. As graph fusion proceeds further, more interference graphs are fused, resulting in denser interference graphs (i.e. more constrained) than the ones that are fused earlier. Thus, the earlier an edge is considered by the fusion process, the less likely it is that live ranges are split along that edge. Fusing graphs based on an edge ordering 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. Register allocation now becomes an problem of edge ordering and forming regions. Compiler writers can determine the size of regions and the edge ordering to tune the quality of register allocation. Moreover, maintaining the simplifiability invariant allows register allocation to avoid committing any register binding decision during graph fusion. As a result, no early binding is made.

4.1.6 Example

We illustrate the above steps with an example. Consider assigning registers to the program fragment shown in Figure 4.2(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 4.2 (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 graphs along edges 1, 3, and 2, we obtain the interference graph of Figure 4.2 (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, graph fusion 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

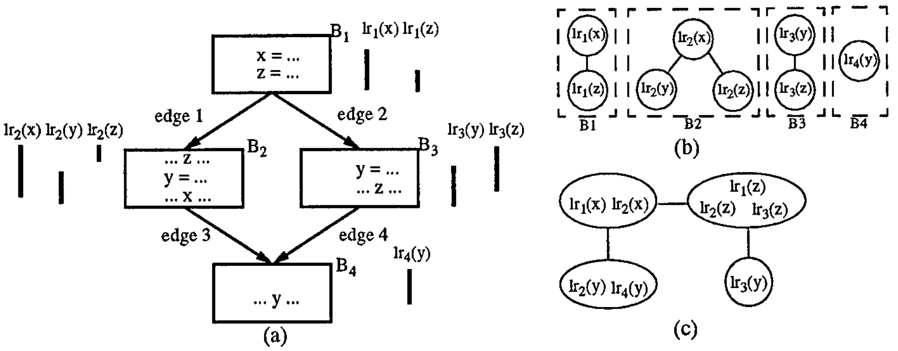


Figure 4.2: Simple code fragment for a world with $N = 2$.

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 Fusion operation

This section describes the essential core of the fusion-based approach, *fusion operation*. The fusion operation fuses two interference graphs into one and maintains the simplifiability invariant. Figure 4.3 depicts several important steps of the fusion operation: finding live ranges that span the edge E, unioning non-splitable live ranges, merging cliques, taking a snapshot of the interference graph, unioning splitable live ranges, and maintaining simplifiability.

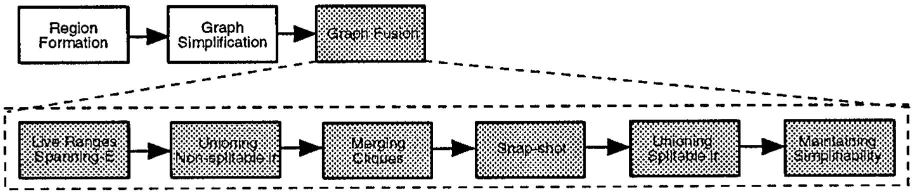


Figure 4.3: Fusion operation.

4.2.1 Finding spanning-E live ranges

For each basic block B, two sets of live ranges are maintained: $ReachIn(B)$ and $LiveOut(B)$. Given a live range $lr(x)$, $lr(x) \in ReachIn(B)$ if lr_x is not defined in B (definitions of lr_x reach the entry of B), and $lr(x) \in LiveOut(B)$ if $lr(x)$ is live at the exit of B. 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. The $ReachIn$ and $LiveOut$ sets indicate which live ranges span edge E. That is, if $lr_1(x) \in LiveOut(B_1)$ and $lr_2(x) \in ReachIn(B_2)$, one definition of $lr_2(x)$ comes from $lr_1(x)$ (there may be other definitions reaching $lr_2(x)$ from other paths to B_2). To facilitate applying the fusion operation along edge E, we define an internal structure called $span\_e$ that is composed of two live-range pointers, *from* and *to*; "from" points to $lr_1(x)$ and "to" points to $lr_2(x)$.

The first step of the fusion operation traverses $LiveOut(B_1)$ and $ReachIn(B_2)$ to find out which live ranges span edge E. The fusion operation attempts to coalesce the live range segments $lr_1(x)$ and $lr_2(x)$ in the interference graph $G^E_{R_1 \cup R_2}$ so that no shuffle code is needed for x along E. This step returns a working list, ω, of $span\_e$.

The example of Figure 4.4 shows the relationship among the structures of $span\_e$, $ReachIn$, and $LiveOut$. $LiveOut(B_1)$ has three live ranges, $lr_1(x)$, $lr_1(y)$, and $lr_1(z)$ because the virtual registers, x, y, and z are defined in B_2 and their values are live at the exit of B_1. $ReachIn(B_2)$ contains $lr_2(x)$ and $lr_2(y)$ because x and y are defined in B_1. The working list ω for edge E records that live ranges of x and y span E.

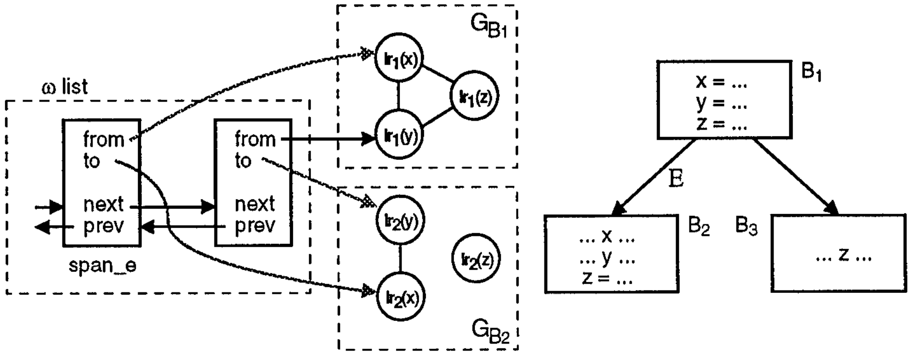


Figure 4.4: Example of finding spanning-E live ranges.

		$lr_2(x)$	
$lr_1(x)$	Spilled	Transparent	Non-transparent
Spilled	Spilled Non-splitable	Spill (Section 4.4) Non-splitable	Split (Section 4.4) Splitable
Transparent		Coalesced, transparent Non-splitable	Coalesced, non-transparent Splitable
Non-transparent			Coalesced, non-transparent Splitable

Figure 4.5: Different possibilities when coalescing $lr_1(x)$ and $lr_2(x)$.

4.2.2 Unioning non-splitable live ranges

There are two places (unioning non-splitable and splitable live ranges as shown in Figure 4.3) where the fusion operation combines live ranges that span edge E. Basically, both steps combine "from" and "to" live ranges of span_e into one. A live range segment 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). If both segments, "from" and "to", are spilled, no shuffle code is needed. If $lr_1(x)$ is spilled and $lr_2(x)$ is transparent, we always spill the combined live range $lr_{12}(x)$ in $G^{E}_{R_1 \cup R_2}$ to eliminate the shuffle code that is needed otherwise. If $lr_1(x)$ is spilled and $lr_2(y)$ is 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). If both $lr_1(x)$ and $lr_2(x)$ are transparent, then $lr_{12}(x)$ in $G^{E}_{R_1 \cup R_2}$ is non-splitable because spilling those live ranges do not introduce spill cost. Figure 4.5 enumerates all possible cases of combining two live ranges. Empty entries in this table are symmetric cases.

This step traverses the ω list and coalesces the live ranges whose combined live ranges are not splitable. Live ranges that are coalesced in this step are removed from ω. Combining two live ranges "from" and "to" consists of three major steps, unioning two live ranges, merging interference edges, and propagating attributes as shown in Figure 4.6. The small diamonds in the figure are condition checkers. For instance, the checker on the left-hand side checks if the combined live range of the current span_e is splitable or not.

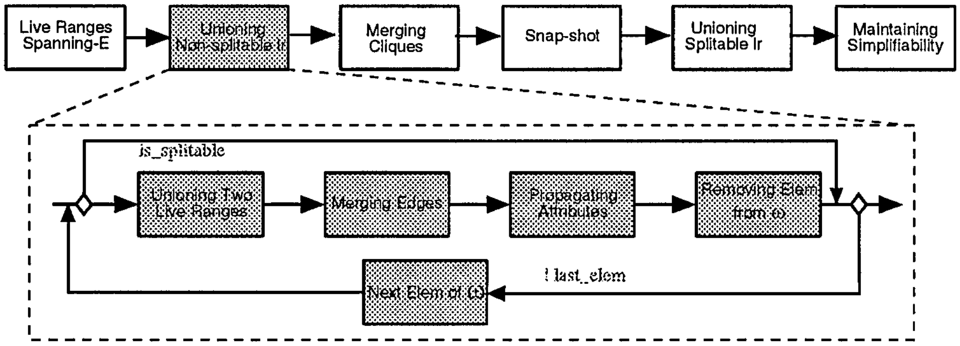


Figure 4.6: Structure of unioning 2 non-splitable live ranges.

Unioning two live ranges

How we union two live ranges depends entirely on the data structures of live range and instruction. A typical machine instruction is composed of three parts: opcode, source, and destination operands. Prior to register allocation, operands may access virtual registers because the code-selection phase of cmcc selects code using virtual registers. Two essential aspects are kept inside the operand accessing a virtual register, (1) a pointer pointing to the virtual register, and (2) a pointer pointing to the corresponding live range of the virtual register at the instruction. At the time of code emission, the code emitter then gets the physical register of the operand via the live-range pointer. During region formation of the graph construction phase, an interference graph is built for each region and the live-range pointers of operands are set properly.

When two live ranges are unioned into one, the operands having pointers pointing to the two live ranges must be able to acknowledge the change. That is, the operands must access the combined live range as opposed to the two old live ranges. One solution is changing the live-range pointers of the operands to point to the combined one. However, updating the live-range pointers is not elegant because it is very inefficient to update pointers every time when two live ranges are unioned. Another way to acknowledge that two live ranges are unioned is through manipulating the data structure of live ranges. The algorithm, *disjoint-set forests* [19], with *union-by-rank* and *path-compression* heuristics is used for unioning two live ranges. Each disjoint set represents one live range. Each live range contains one *parent* pointer and *rank* for the disjoint set. The live range whose parent pointer points to itself (root) is the representative of all live ranges that are in the same set. Namely, the root live range keeps all information regarding color assignment, interference, ..., etc. All other live ranges of the same set need to follow their parent pointers to get to the root live range to retrieve all information. As a result, the live-range pointers of operands are never required to be updated using the disjoint-set algorithm.

Figure 4.7(a) illustrates an example of unioning two live ranges of x using the disjoint-set algorithm. Each shaded region represents one live range. One live range comprises lr_1 and lr_2, and the other consists of lr_3, lr_4, and lr_5. Arrows indicate the *parent* pointers. The node whose parent points to itself is the root that represents the live range, e.g. lr_1 and lr_3. Unioning the two live ranges is done by simply changing lr_3's parent pointer to point to lr_1. lr_1 is now the one who represents the union of the two live ranges as shown in Figure 4.7(b). Finding the root live range for lr_4 traverses the parent pointers to get to the root. The traversing path from lr_4 to the root is compressed by changing the parent pointer of each node on the path to point directly to the root (the dotted line illustrated in Figure 4.7(c)). Changing the parent pointer of each node on the traversing path is known as *path compression* (the path from lr_4 to lr_1 is now shorter than the one of Figure 4.7(b)).

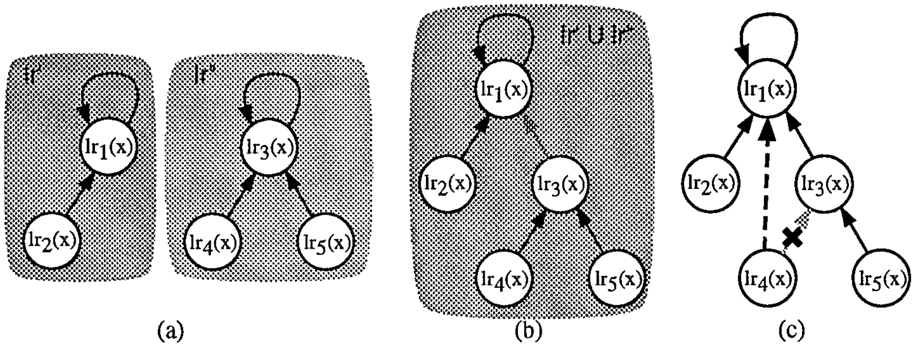


Figure 4.7: Unioning two live ranges using disjoint-set.

There is one price to be paid for using the disjoint-set algorithm— superfluous memory. The size of the data structure for a live range is large. The cmcc compiler is developed on DEC Alpha machines on which each pointer requires 64 bits (8 bytes). Using virtual functions in C++ costs each object whose class declares virtual functions a pointer that points to the virtual function table for that class [59, 49]. The size of our data structure for a live range is 184 bytes. What a non-root live range needs is the parent pointer to proceed to the root to retrieve live-range information. The other fields of the structure are never used and therefore wasted. To alleviate the problem, another structure, *live-range union*, is created. This structure sits between operands and live ranges. Now, an operand that accesses a virtual register has a live-range union pointer rather than a live-range pointer. A live-range union contains a parent pointer, rank as well as a live-range pointer. Only root live-range unions retain live ranges. As two live ranges are unioned, the combined live range, lr_{12}, of lr_1 and lr_2 is either lr_1 or lr_2. The other one can later be destroyed and recycled. Live-range union allows the internal representation of register allocation to use memory not more than necessary.

Figure 4.8 sketches the structures for an instruction, a live-range union, and a live range. Only the root live-range union, which is shaded, retains the live-range information. The live ranges of the other two live-range unions have been freed up in the process of unioning live ranges. To access the physical register that the `src1` operand uses, we need to go through `lr_ptr` to get to its corresponding live-range union, follow the parent pointer of the live-range union up to the root, and then query the live range to return the physical register (see Figure 4.8).

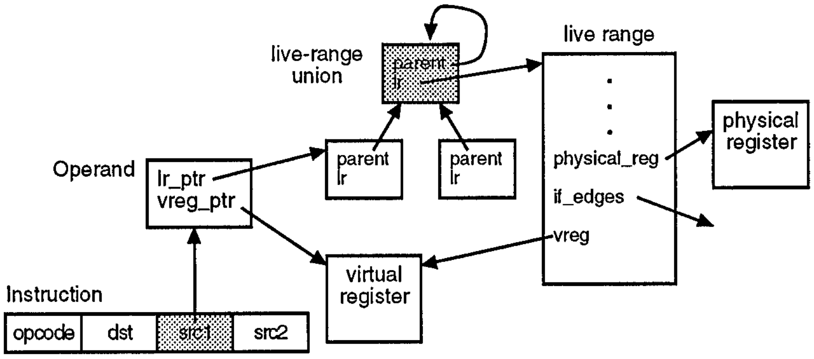


Figure 4.8: Live-range union.

Merging interference edges

The two live ranges have their own interference edges, $\varphi(lr_1)$ and $\varphi(lr_2)$, prior to uniting. In this step, $\varphi(lr_1)$ and $\varphi(lr_2)$ are merged. If lr_{12} is lr_1, then each interference edge, $[lr_2, X]$, of lr_2 is updated by substituting lr_2 with lr_{12} ($[lr_{12}, X]$) where $X \in ncighbor(lr_2)$. Likewise, each edge of $\varphi(lr_1)$ is also updated in a similar fashion if lr_{12} is lr_2. Duplicate edges occur when lr_1 and lr_2 interfere with the same live range prior to uniting. In such a case, duplicate edges are not added to the interference graph and therefore freed up.

The example in Figure 4.9 illustrates merging interference graphs. The interference graph prior to unioning lr_1 and lr_2 is given in Figure 4.9(a). $\varphi(lr_1)$ has two edges: $[lr_1,lr_x]$ and $[lr_1,lr_y]$. $\varphi(lr_2)$ has also two edges: $[lr_2,lr_y]$ and $[lr_2,lr_z]$. Both lr_1 and lr_2 interfere with lr_y. After we combine lr_1 and lr_2 into lr_{12}, we update $\varphi(lr_1)$ and $\varphi(lr_2)$. The two edges of $\varphi(lr_1)$ now become $[lr_{12},lr_x]$ and $[lr_{12},lr_y]$; the two edges of $\varphi(lr_2)$ become $[lr_{12},lr_y]$ and $[lr_{12},lr_z]$. Merging $\varphi(lr_1)$ and $\varphi(lr_2)$ eliminates one duplicate edge, $[lr_{12},lr_y]$. The resulting interference graph is depicted in Figure 4.9(b).

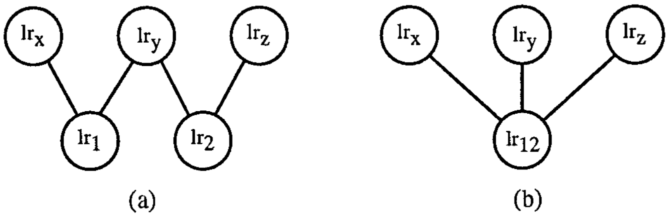


Figure 4.9: Example of merging interference edges.

Propagating attributes

Each live range has several attributes that are propagated during graph fusion: *caller_cost*, *spill_cost*, *has_def*, and *area*. The *caller_cost* attribute is the estimated cost of assigning a caller-saved register to the live range—the number of saves and restores that must be executed around function calls if the 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 color assignment to decide whether a live range is assigned a 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 $lr(x)$. The has_def attribute indicates whether x's value needs to be stored to memory at exits from $lr(x)$; if a live range does not modify the value of a virtual register, then no stores are required at exits from the live range. The *area* attribute indicates the size of the live range and is used for making out-of-color spilling decisions [5].

These attributes are propagated as follows:

- $has\_dcf(lr_{12}) = has\_dcf(lr_1) \vee has\_dcf(lr_2)$
- $callcr\_cost(lr_{12}) = callcr\_cost(lr_1) + callcr\_cost(lr_2)$
- $spill\_cost(lr_{12}) = spill\_cost(lr_1) + spill\_cost(lr_2)$
- $arca(lr_{12}) = arca(lr_1) + arca(lr_2)$

4.2.3 Merging cliques

The clique summary nodes (Section 4.4) are used to delay spilling decisions. Multiple cliques may exist on both ends of the edge E as more graphs are fused. The cliques-merging step deals with how to combine cliques. Details are described in Section 4.4.

4.2.4 Taking a snapshot of the interference graph

As mentioned earlier, combined live ranges may be split to maintain the simplifiability invariant. As two live ranges, lr_1 and lr_2, are combined into lr_{12}, their interference edges, $\varphi(lr_1)$ and $\varphi(lr_2)$, are merged. Duplicate edges may arise and are thrown away. To undo combining, the register allocator must retain the original interference graph. Otherwise, it is hard and very inefficient to recompute interference information. This step takes a partial snapshot of the current interference graph before combining any splitable live ranges. Because only splitable live ranges need their original interference information, the origial interference edges of these live ranges are stored. Non-splitable live ranges were already removed from ω in the step of unioning non-splitable live ranges. Now all live ranges of ω are splitable. This step, therefore, stores the interference edges of the live ranges of ω.

4.2.5 Unioning splitable live ranges

This step traverses ω and combines each pair of live ranges (all resulting live ranges are splitable). Unioning two splitable live ranges, lr_1 and lr_2 requires two operations that are similar but not identical to unioning non-splitable live ranges. First, the two live ranges are combined into one live range, lr_{12}, as described in Section 4.2.2. Nevertheless, the slight distinction is that there is *no* path compression between this step and the step that commits live-range unioning decisions in Figure 4.10. The second operation merges interference edges *without* freeing up duplicate edges. There is another noteworthy point—attributes of live ranges are not propagated at this point of time. All differences of unioning non-splitable and splitable live ranges originate from the fact that the latter wants to retain the old interference information so as to split combined live ranges easily, and the former doesn't. The task of deleting duplicate edges and propagating attributes of splitable live ranges is deferred till the simplifiability invariant is satisfied.

We avoid compressing paths because updating the parent pointer causes a serious problem when we want to split a combined live range. Consider the example of Figure 4.7. Presume that path compression happens prior to splitting combined live ranges. The parent pointer of lr_4 may have been changed to lr_1 (Figure 4.7(c)). It is now very hard to split the combined live range because we have no idea if lr_4 belongs to lr_1 or lr_3 initially, unless there is some extra bookkeeping. On the contrary, if we don't allow path compression to happen, then splitting the combined live range can be done by simply changing lr_3's parent to point to itself.

4.2.6 Maintaining the simplifiability (colorability) invariant

The step that maintains simplifiability ensures that the invariant holds by performing simplification on $G_{R_1 \cup R_2}$. The algorithm described in [52] is used to deal with register pairs. This step consists of four major components as depicted in Figure 4.10: computing degree, removing a LS_N node (degree less than N), lowering register pressure, and committing the live-range unioning decision. The shaded region simplifies the interference graph until an empty graph is obtained.

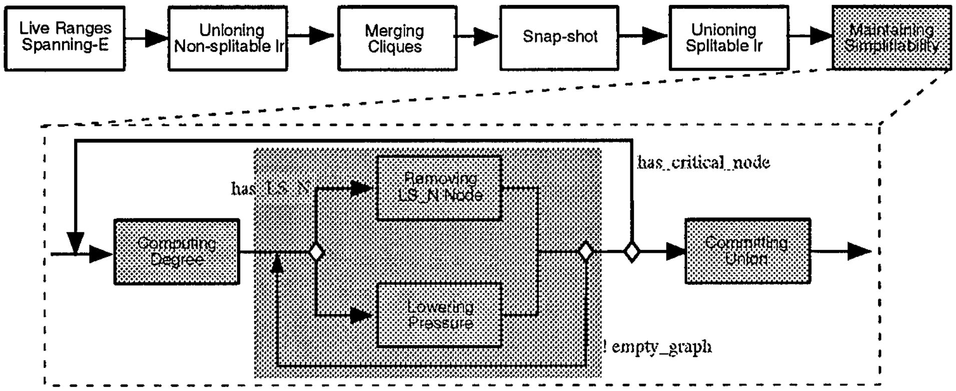


Figure 4.10: Maintaining simplifiability.

Computing degree

There are two working lists, LS_N and GE_N, used by the simplification. The LS_N list keeps live ranges whose degrees are less than N (Max colors). The GE_N list contains live ranges whose degrees are greater than or equal to N. The computing degree step computes the degree of each live range of the interference graph and initializes the two lists.

Removing a LS_N node

Again the small diamonds of Figure 4.10 are condition checkers. The checker between the computing-degree and removing-LS_N steps checks if there are live ranges on the LS_N list. If LS_N is not empty, then the removing-LS_N step removes one live range from the beginning of the LS_N list and then removes the live range from the graph. The degrees of the live range's neighbors are decremented. If any neighbor that was constrained becomes unconstrained, the neighbor is removed from the GE_N list and inserted into the LS_N list.

Lowering register pressure

If all remaining live ranges are on the GE_N list (and LS_N list is empty), then simplification of $G^{E}_{R_1 \cup R_2}$ blocks. In other words, fusing the interference graphs of G_{R_1} and G_{R_2}

along edge E induces high register pressure. Some actions are required to lower the register pressure. In a first attempt, we find a transparent live range to spill, instead of splitting live ranges. Spilling a transparent live range incurs no spill cost (but implies shuffle cost), and the delayed spilling technique may lower the incurred shuffle cost, i.e. push shuffle code onto low frequently executed edges. After spilling a transparent live range, degrees of its neighbors are decreased correspondingly. If there is no transparent live range, then a coalesced (splitable) live range lr_{12} is chosen from the remaining constrained nodes to split. lr_{12} will be split along E back into lr_1 and lr_2. The choice of which live range to split is solely based on a node's degree, because the shuffle costs (move) at edge E for all splitable live ranges are the same. After splitting lr_{12}, if lr_1 (lr_2) becomes unconstrained, lr_1 (lr_2) is inserted into the LS_N list. The degree of the interference graph may be lowered because of splitting, allowing simplification to proceed from the point where it blocked. If simplification blocks again, additional live ranges are split. Experimental results show that choosing the splitable live range that has the *smallest* degree incurs fewer splittings than choosing the *highest* degree. The reason is that lr_1 and lr_2 are more likely to become unconstrained using the heuristic of choosing the smallest degree. Since the original graphs G_{R_1} and G_{R_2} are simplifiable, then at worst all live ranges that span E are split. Figure 4.11 depicts the flow of the steps to lower register pressure. The splitting step may generate *critical nodes* while splitting a combined live range. Section 4.2.7 elucidates how splitting live ranges generates critical nodes. If a critical node arises, then simplification is reiterated from the computing-degree step as illustrated in Figure 4.10.

Splitting a splitable live range lr_{12} consists of three steps (in the shaded region). Unioning the two live ranges, lr_1 and lr_2, causes one of them to become a child of the other using the disjoint-set algorithm. The first step removes lr_{12} from the GE_N list and makes the child to be a root by changing the child's parent pointer to point to itself. The second step removes all edges of $\varphi(lr_{12})$, retrieves the two sets of original interference edges, $\varphi(lr_1)$ and $\varphi(lr_2)$, that were recorded in the taking-snapshot step. If the child is lr_1 then each edge of $\varphi(lr_1)$ is updated by changing $[lr_{12}, X]$ to $[lr_1, X]$ where $X \in neighbor(lr_1)$ (reversing what had been done in the merging-edges step of Section 4.2.5). Each edge of $\varphi(lr_2)$ is updated similarly if the child is lr_2. $\varphi(lr_1)$ and $\varphi(lr_2)$ are hereby added back to the graph again (duplicate edges may still exist and are not put back to the graph). The third step

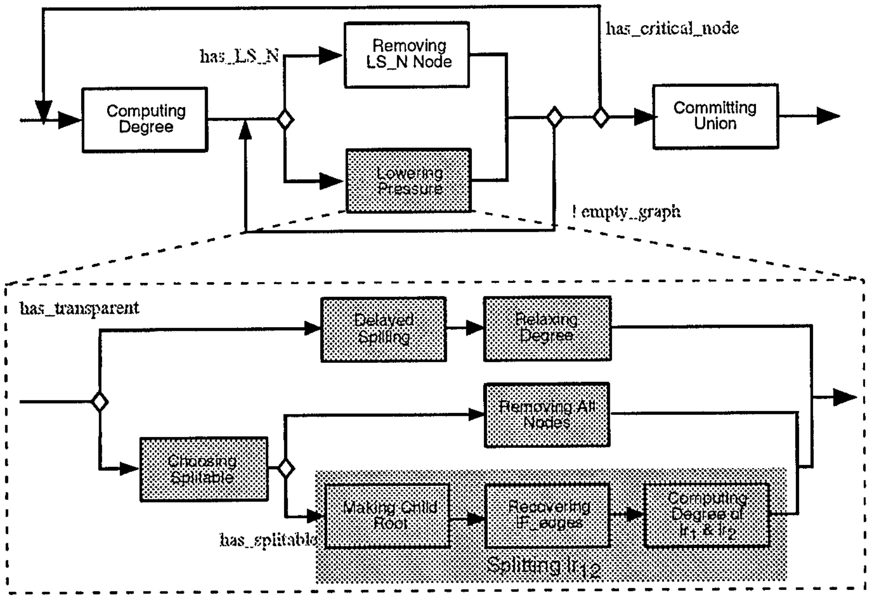


Figure 4.11: Lower register pressure.

recomputes the degrees of lr_1 and lr_2. If lr_1 (lr_2) is unconstrained, lr_1 (lr_2) is inserted into the LS_N list. Otherwise, they are inserted into the GE_N list.

If none of the live ranges of the GE_N list is splitable, then simplification must have produced at least one critical node. Otherwise, simplification should proceed without blocking. Hence, all nodes are simply removed from the graph, and simplification is repeated from the computing-degree step.

Committing live-range unioning decision

By the time we get to the "committing unioning decision" step, the simplifiability constraint is satisfied and all necessary spilling (transparent live ranges) and splitting (coalesced live ranges) decisions are made. Some live ranges are coalesced and some live-range unions are suppressed. The step now commits all coalesced live ranges. Namely, those coalesced live ranges will never be split at edge E. Thus, the attributes of the coalesced live ranges are then propagated. Duplicate interference edges can now be eliminated. Furthermore, parent pointers of live ranges are allowed to be updated freely from now on.

4.2.7 Critical node

When two live ranges lr_1 and lr_2 are combined into lr_{12}, the degrees of their neighbors can only decrease monotonously (the degree will never increase). If one live range lr_x interferes with lr_1 and lr_2, coalescing lr_1 and lr_2 produces one duplicate edge. The duplicate edge is then removed from the graph, as a result, the degree of lr_x is decreased by 1. Conversely, splitting a coalesced live range, lr_{12}, increases its neighbors' degrees monotonously because the removed duplicate edge is added back to the interference graph (the degree of lr_x is increased by 1). One problem arises, due to this property, when splitting is taking place during simplification (Figure 4.11). One live range is removed (simplified) from the interference graph only under the condition that the degree of the live range is less than N. Again, the main observation is that the register allocator is able to find a legal color for the live range regardless how colors are assigned to the residual graph. The observation, nonetheless, may no longer hold once coalesced live ranges are split because the degree of an already-simplified live range may increase and become *constrained*. Those already-simplified live ranges that become constrained are called *critical nodes*. A critical node can happen only when live ranges are split.

Detecting critical nodes is done by keeping track of the degrees of already-simplified live ranges. The notation $\sigma(lr)$ denotes the degree of lr at the time when lr is removed from the graph during simplification. After lr is removed from the graph, $\sigma(lr)$ ($\sigma(lr) < N$ at this point of time) is then frozen and not decreased when subsequent live ranges are removed from the graph. At the time of splitting lr_{12} back to lr_1 and lr_2, $\sigma(lr)$ is increased by 1 if both lr_1 and lr_2 interfere with lr. Whenever $\sigma(lr)$ is equal to N, then lr becomes a critical node (the condition $\sigma(lr) < N$ is violated).

The simplification step of Figure 4.10 never tries to backtrack when critical nodes arise (because it is hard). Instead, simplification continues until an empty graph is obtained. If critical nodes arise, then simplification is reiterated from the computing degree step to make sure that the resulting graph can still be simplified without blocking. The next pass simplifies the interference graph on which some live range are already split at E. Simplification will terminate within a fixed number of iterations because a critical node can happen only when live ranges are split, and there is only a fixed number of splitable live ranges. In reality, critical nodes happen rarely. As I have observed so far, one more iteration of simplification is sufficient to preserve the simplifiability invariant. One implementation could curtail the chance of generating critical nodes. Always simplifying an unconstrained live range that has the smallest degree lowers, in general, the degrees of live ranges at the time when they are simplified. Because critical nodes occur rarely, the technique is not implemented.

4.2.8 Example of critical node

The example of Figure 4.12 shows how splitting live ranges produces a critical node and why re-performing simplification is essential. Figure 4.12 (a) is the interference graph before graph fusion. With N=3, the interference graph can be simplified. One legitimate sequence of removing nodes from the graph during simplification is $lr_1(x)$, $lr_2(x)$, $lr_1(y)$, $lr_2(y)$, $lr(w)$, $lr(z)$, and $lr(t)$. Assume that the fusion operation combines $lr_1(x)$ with $lr_2(x)$ first and $lr_1(y)$ with $lr_2(y)$ next. Figure 4.12(b) portrays the resulting graph after $lr_1(x)$ and $lr_2(x)$ are combined into $lr_{12}(x)$. As $lr_1(y)$ and $lr_2(y)$ are combined, two duplicate edges, [$lr_{12}(y)$,$lr_{12}(x)$] and [$lr_{12}(y)$,$lr(z)$], are taken out from the graph as depicted in Figure 4.12(c).

There are five live ranges in the final graph (Figure 4.12(c)). $lr_{12}(y)$, $lr(w)$, $lr(z)$, and $lr(t)$ form a clique of size 4. Light shaded live ranges and dotted interference edges indicate that the live ranges and edges are simplified and removed from the graph. With $N = 3$, $lr_{12}(x)$ can be simplified because the degree of $lr_{12}(x)$ is 2. Then simplification of the remaining graph blocks because every node has degree = 3 (shown in Figure 4.12(d)). Splitting $lr_{12}(y)$ back to $lr_1(y)$ and $lr_2(y)$ adds the two duplicate back to the interference graph (shown in Figure 4.12 (e)). Consequently, the degree of $lr_{12}(x)$ is increased by 1; $lr_{12}(x)$ becomes a critical node. The degrees of $lr_1(y)$ and $lr_2(y)$ are 2 after splitting $lr_{12}(y)$ (recall that $lr_{12}(x)$ is removed from the graph already). Simplification can proceed from where it blocked by simplifying $lr_1(y)$ and $lr_2(y)$, and an empty graph is obtained.

Even though an empty graph is obtained, the simplifiability invariant doesn't hold due to the critical node $lr_{12}(x)$. If we perform simplification on the interference graph of Figure 4.12(e), we discover that simplification blocks and cannot remove any node from the graph because all nodes are constrained nodes (degree ≥ 3). When critical nodes arise, re-iterating simplification splits live ranges as necessary so as to maintain the simplifiability constraint. That is, if simplification blocks, splitable live ranges are chosen to split. $lr_{12}(x)$

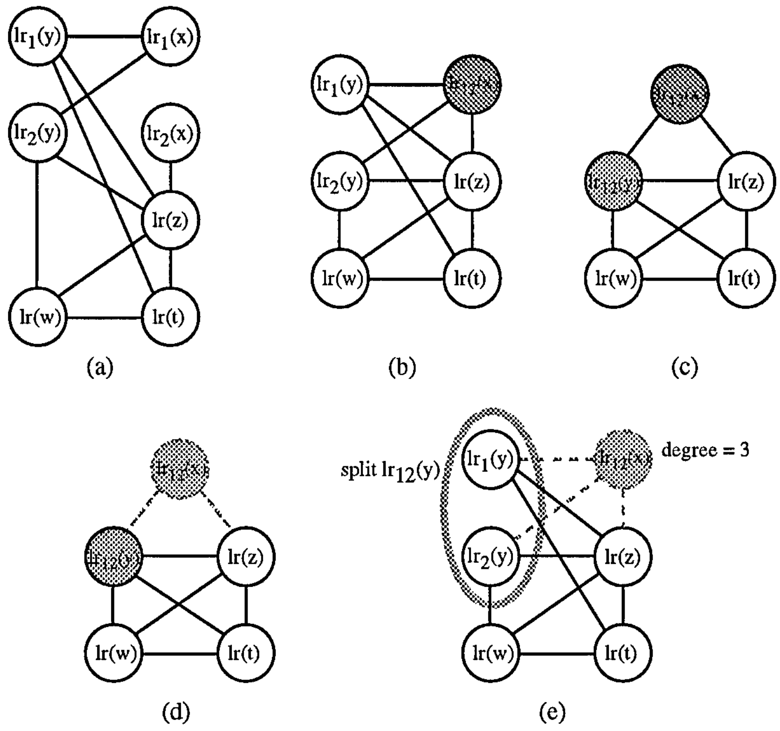


Figure 4.12: Critical node (N=3).

is split in the second pass of simplification on the graph of Figure 4.12(e). As a result, the maintaining-simplifiability step splits the interference graph of Figure 4.12(c) back to the original graph (Figure 4.12(a)).

4.3 Modeling different splitting approaches

Before we describe how the fusion-style approach models various splitting approaches, we need to know how G_R for a region R is constructed because different approaches can be distinguished based on the strategy used to select regions.

4.3.1 Constructing G_R

Constructing G_R, like graph fusion, also relies on the fusion operation to glue small interference graphs into one. Namely, both constructing G_R and graph fusion use the same framework of the fusion operation. The fusion operation that fuses the interference graphs of two regions is called *weak fusion* (presented in Section 4.2) because splitting live ranges is allowed during graph fusion. The fusion operation fusing two graphs that belong to one region is called *strong fusion* because splitting live ranges is forbidden. A region is composed of a collection of basic blocks and set of control-flow edges, R_{edges}, that connect blocks within the region. G_R is constructed in two steps. First, the interference graph is built for each block in the region. Second, the strong fusion operation is applied to every edge of R_{edges}. The two steps are similar to the region formation and graph fusion of Figure 4.1. Blocks are regions, and graphs of regions are fused using the *strong*, rather than *weak*, fusion operation. Because we don't maintain the simplifiability invariant when we build G_R, the graph simplification step of Figure 4.1 is not necessary. No spilling decisions are made during constructing G_R.

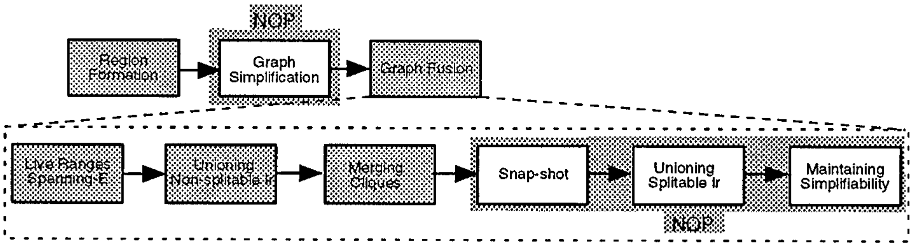


Figure 4.13: Strong fusion operation.

Figure 4.13 depicts the structure of the strong fusion operation. The three steps highlighted by dotted rectangle that deal with splitting live ranges are just *nop*: taking a snapshot of the interference graph, unioning splitable live ranges, and maintaining the simplifiability invariant. All live ranges that span $E \in R_{edges}$ are non-splitable.

4.3.2 Tera

The Tera approach can be modeled by using loops as regions. The edge ordering for the control-flow edges that connect regions are ordered based on the estimated execution frequency using loop depth—the edges in the inner loops appear in front of the edges in the outer loops. We thereby perform graph fusion based on the edge ordering. The way that we model the Tera approach is not identical to what was described in [12] because fusion-style coloring does not construct the tile tree and perform register allocation tile by tile. However, both approaches have the same perspective to register allocation. That is, the inner loops have more freedom in terms of assigning registers. The fusion-based approach should outperform the Tera approach for two reasons. First, no premature binding decisions have been made in the fusion-based approach. Second, the Tera approach does not take into account the edges that connect a parent and a child tile to prevent live ranges from being split on one of the edges that is executed more often than the others. The fusion-based approach can prioritize those edges based on profile information so as to avoid splitting live ranges at frequently executed edges.

4.3.3 PDG

The RAP compiler performs register allocation in a hierarchical fashion based on the PDG. A region node in the PDG is a collection of predicates and statements that are executed under the same control conditions. A region node R may consist of subregion nodes. To model the PDG approach, we can exclude the subregion nodes from R and consider R as a region in fusion-style coloring. We then traverse the PDG from leaves to the root like the way that the RAP compiler performs register allocation. During traversing the PDG, we order control-flow edges by assigning higher priorities to the control-flow edges that connect the current region to its subregions than to the rest of the control-flow edges that are not yet assigned priorities. The PDG approach modeled by fusion-style coloring should outperform the conventional PDG approach for the same two reasons as described in modeling the Tera approach. First, no premature binding decisions have been made in fusion-style coloring. Second, the conventional PDG approach does not take into account the execution frequencies of the edges that connect a parent region and a subregion whereas the fusion-style approach can prioritize the edges based on their estimated execution frequencies.

4.3.4 Multiflow

The Multiflow compiler integrates scheduling and register allocation. Scheduling and register allocation are done on a trace basis. That is, the compiler picks a trace and passes the trace to the code scheduler; the code scheduler then schedules the code of the trace and performs register allocation. Because the cmcc compiler does not have the framework that integrates scheduling and register allocation like the Multiflow compiler, fusion-style coloring can model only the register allocation part of the Multiflow approach. One key idea of the Multiflow approach is that traces chosen first have more freedom in using registers—are not constrained by other traces because other traces are not yet compiled. Fusion-style color has the flexibility to choose traces as regions, maintains the simplifiability invariant to delay color-assignment decisions, and uses the delayed spilling technique to defer spilling decisions. Therefore, unlike the Multiflow approach, traces are never constrained by the color binding decisions of other traces.

There are two counteractive aspects when register allocation chooses traces as regions. The code scheduler requires instructions to exploit ILP. The bigger the trace, the more instructions the code scheduler can exploit. However, the bigger the trace, the more register pressure the trace imposes on register allocation. Once a trace is picked, no live-range splitting could happen using the Multiflow approach. As a result, the spilling decisions made by the register allocator are *all-or-nothing* spilling. The fusion-based approach, on the other hand, has the flexibility to further divide a trace into smaller regions and applies graph fusion to combine the graphs of the smaller regions into the interference graph of the trace. Live-range splitting is therefore possible within the trace.

4.3.5 SSA

With slight modification, the fusion-based framework can model the approach that uses the SSA representation to determine splitting points. Constructing G_{SSA} is similar to constructing G_R where R is the region that covers the whole function. Splitting a live range at a ϕ node is the same as splitting the live range at the entry edges of the block that contains the ϕ node because ϕ nodes are inserted into the entry of blocks [20]. We apply the strong fusion operation on every control-flow edge. During the step of unioning non-splitable live ranges (Figure 4.13), coalescing two live ranges is suppressed if the current edge is a ϕ-node splitting point for the two live ranges. The later color assignment eliminates shuffle moves using biased coloring.

4.4 Delayed spilling

When a region R needs M physical registers to be colored, and $M > N$, then some 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, and K live ranges must be spilled, then there are three cases that must be considered:

$K < T$: 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 high-priority edges.

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

$K > T$: In this case, all transparent live ranges, as well as $K - T$ live ranges with a reference inside R, are spilled. The spilling decisions are selected by a heuristic based on spill cost, area, and the degrees in the interference graph.

Although spilling a transparent live range is very attractive (because it has *no* spill cost), it may not be globally optimal. Figure 4.14 shows a flow graph for which spilling transparent live ranges is not optimal. In Figure 4.14(a), each of B_1 and B_2 has three live

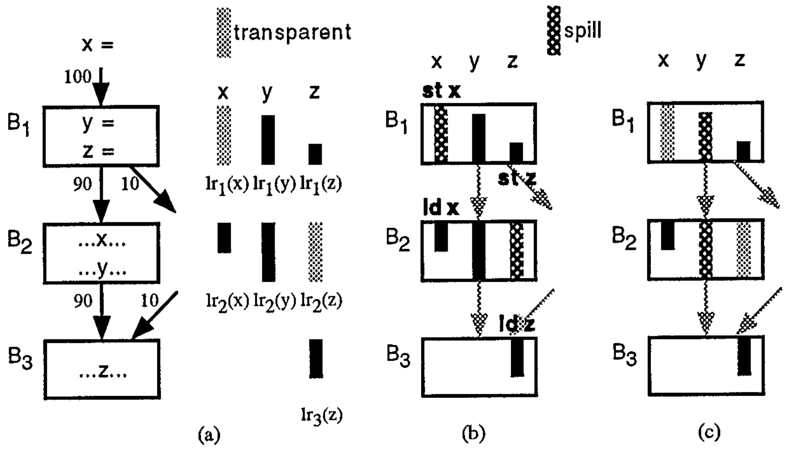


Figure 4.14: Non-optimal spilling (N=2).

ranges simultaneously live. Regions are basic blocks. The vertical lines on the right of the blocks represent the live ranges of the virtual registers x, y, and z (a lightly shaded line indicates that the live range is transparent). The number next to each control-flow edge is the execution frequency of the edge. B_1 and B_3 have the same execution frequency, 100. Three colors are needed to color the interference graph of B_1 and B_2. Since there are only two registers, we must spill one live range from each block. Figures 4.14(b) depicts the case of spilling transparent live ranges (spilled live ranges are drawn with a hatched pattern). $lr_1(x)$ and $lr_2(z)$ are transparent live ranges and spilled to reduce register pressure. Although the two live ranges do not have spill cost, they need shuffle code to move values between registers and memory. The shuffle cost is 370 load/store operations. The optimal result (least overhead operations) in this example is spilling y's live ranges ($lr_1(y)$ and $lr_2(y)$) as depicted in Figure 4.14(c) (only 190 operations of spill code).

I now describe in more detail the delayed spilling technique used to handle the case where $K < T$. Since all transparent live ranges conflict with each other, these live ranges form a clique in the interference graph. The transparent live ranges are therefore collected into a single *clique summary node* C in the interference graph, as depicted in Figure 4.15. The 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

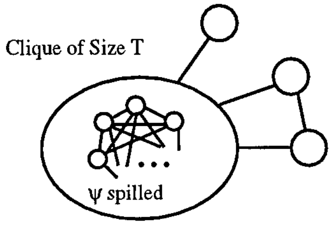


Figure 4.15: Clique summary node.

$\psi(C) = K$ 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* live range(s) to spill until more information is available. The clique representation affects the graph coloring in two aspects: computing the degree and fusing graphs.

4.4.1 Computing degree

The degree of an interference graph node is usually computed by summing the number of interference graph edges that are incident upon the node. With clique summary nodes, however, additional care must be taken. Consider a clique node C and a regular node lr, connected by an interference edge E. The degree that edge E contributes to node lr is $T(C) - \psi(C)$ because lr interferes with every nodes inside clique C. The degree that edge E contributes to clique C is one. In addition to the interference edges of C, the degree of a clique node must add $T(C) - \psi(C) - 1$ because within the clique, each node interferes with $T(C) - \psi(C) - 1$ other nodes.

4.4.2 Fusing graphs

As the interference graphs of larger and larger regions are fused, the compiler acquires a more global understanding of the reference patterns. This knowledge is especially useful if the order of fusing graphs reflects either program structure or execution frequency. Contiguous spilled and non-spilled live range segments do not incur shuffle cost within the segments because no shuffle code is required. Therefore, the tendency of progressively growing live ranges during fusing determines which transparent live ranges need to be spilled eventually. And the expansion of live ranges is based on the edge ordering, which makes the spilling decisions sensitive to the edge priority function, e.g., program structure or execution frequency.

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 because one spilling decision is made. The decision grows a spilled live range, thereby allowing the compiler to avoid shuffle code on the edge E (a contiguous spilled live range crosses the edge E).

2. If lr_1 is a non-transparent live range, lr_2 is in a clique C: The fusion operation makes a non-spilling decision for lr_2 and removes lr_2 from C. Only $T(C)$ is decremented. The decision enlarges a non-spilled live range; since the live range is in a register in an adjacent region, keeping lr_2 in the register (not spilled) eliminates the otherwise needed shuffle code on the edge E. Hence 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 $\psi(C_2)$. There exists several options to determine how many live ranges to spill from C; once $\psi(C)$ is known, $\psi(C_1')$ and $\psi(C_2')$ are computed, $\psi(C_1') = \psi(C_1) - \psi(C)$ and $\psi(C_2') = \psi(C_2) - \psi(C)$:

   (a) $\psi(C) = \min(T(C), \max(\psi(C_1), \psi(C_2)))$

   (b) $\psi(C) = \min(T(C), \min(\psi(C_1), \psi(C_2)))$

   (c) $\psi(C) = \min(T(C), \max(\lceil \frac{T(C)}{T(C_1)} \psi(C_1) \rceil, \lceil \frac{T(C)}{T(C_2)} \psi(C_2) \rceil))$

(d) $\psi(C) = \min(T(C), \min(\lceil \frac{T(C)}{T(C_1)} \psi(C_1) \rceil, \lceil \frac{T(C)}{T(C_2)} \psi(C_2) \rceil))$

$\min(T(C),)$ in the equations is required to make sure that the condition, $\psi(C) \leq T(C)$, of the clique C holds. The first option tends to lower ψ functions of C_1' and C_2' because $\psi(C)$ takes the maximum of $\psi(C_1)$ and $\psi(C_2)$. The third and fourth options attempt to divide the number of spilled live ranges among cliques proportional to their sizes.

If $T(C) = \psi(C)$ for a clique C at any time during this phase, then all live ranges of C must be spilled. At that point, C is an empty clique and removed from the interference graph. When a live range lr is removed from clique C, a copy of $\varphi(C)$ and a new conflicting edge [lr,C] (because lr interferes with every live range in C) are generated for L. The example of Figure 4.16 has 3 live ranges in the clique C. We want to remove lr_x from the clique. A copy of the interference edges (edge 1 and 2) of C is generated (dashed lines). The new generated interference edge [lr_x,C] is added as well.

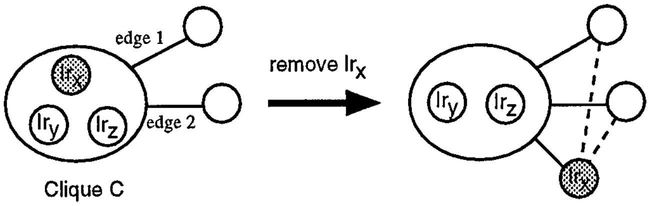


Figure 4.16: Removing a live range from a clique.

4.4.3 Example of delayed spilling

In Figure 4.17(a), each basic block has three live ranges, and they all conflict with each other. Again, the vertical lines on the right of the blocks represent the live ranges of the virtual registers x, y, and z (a lightly shaded line indicates that the live range is transparent). Three colors are needed to color the interference graph of each block. Since we have only two registers, we must spill one live range from each block. Assume that basic blocks are regions. The graph simplification phase (Section 4.1.2) needs to spill one live range from each interference graph of basic blocks so as to maintain the simplifiability constraint.

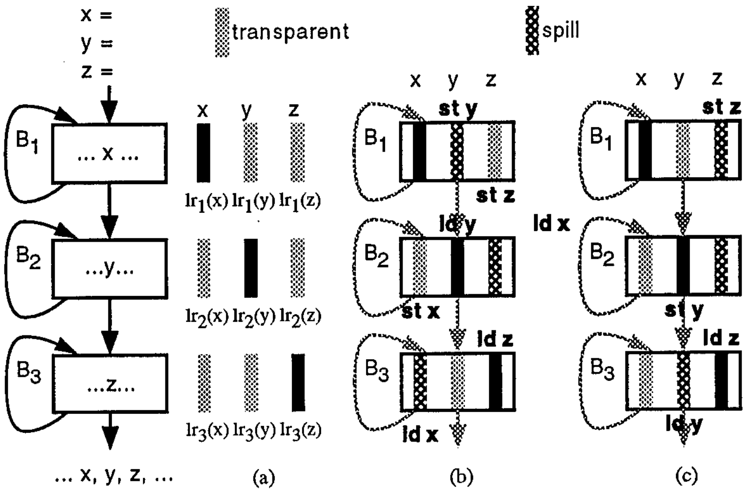


Figure 4.17: Example of delayed spilling (N=2).

Without delayed spilling, spilling decisions need to be made locally. The decisions may not be globally optimal. If $lr_1(y)$, $lr_2(z)$, and $lr_3(x)$ are spilled (no spill cost), six load/store operations (shuffle code) are required to move values between registers and memory, one store and one load operations for each live range as shown in Figure 4.17(b). The delayed spilling technique is able to delay spilling decisions by constructing a clique for each block to collect the two transparent live ranges of the block. The ψ function for each clique is one (one out of two will eventually be spilled). Consider that the interference graphs of B_1 and B_2 are fused. The unioning rules described in Figure 4.5 decide that $lr_2(x)$ and $lr_1(y)$ shouldn't be spilled because $lr_1(x)$ and $lr_2(y)$ are non-transparent live ranges. Therefore, we spill $lr_1(z)$ and $lr_2(z)$ (depicted in Figure 4.17(c)). This spilling decision results in only four load/store operations.

4.5 Biased coloring

$lr_1(x)$ and $lr_2(x)$ are called *adjacent partners* if their values are connected by a splitting point. Two live ranges are *remote partners* if there is no splitting point between them but they are parts of the same live range if we construct the interference graph for Chaitin-style coloring (no live-range splitting). For instance, Figure 4.18(a) depicts an case where the live range $lr(x)$ spans across 3 blocks and is split into 3 smaller live ranges, $lr_1(x)$, $lr_2(x)$, and $lr_3(x)$. $lr_1(x)$ and $lr_2(x)$ are adjacent partners, and $lr_2(x)$ and $lr_3(x)$ are adjacent partners also. $lr_1(x)$ and $lr_3(x)$ are remote partners.

A shuffle move is required if $lr_1(x)$ and $lr_2(x)$ end up in different registers. Biased coloring [7] is a common technique to eliminate a shuffle move between $lr_1(x)$ and $lr_2(x)$ by assigning them the same register. We define some notations that are used in describing biased coloring.

$adjaccnt(lr)$	all adjacent partners of lr
$partncr(lr)$	all adjacent and remote partners of lr
$color(lr)$	the assigned color of lr
$avoid(lr)$	the set of colors that lr would like to avoid using
$avail(lr)$	$\{c \mid c \notin \bigcup_{n \in neighbor(lr)} color(n)\}$

Selecting a color for a live range lr using biased coloring consists of the following steps:

1. choose a color $c \in avail(lr) \wedge c \in \bigcup_{n \in adjacent(lr)} color(n)$.

2. if not found, choose a color $c \in avail(lr) \wedge c \in \bigcap_{n \in uncolored\ adjacent(lr)} avail(n)$.

3. if not found, choose a color $c \in avail(lr)$.

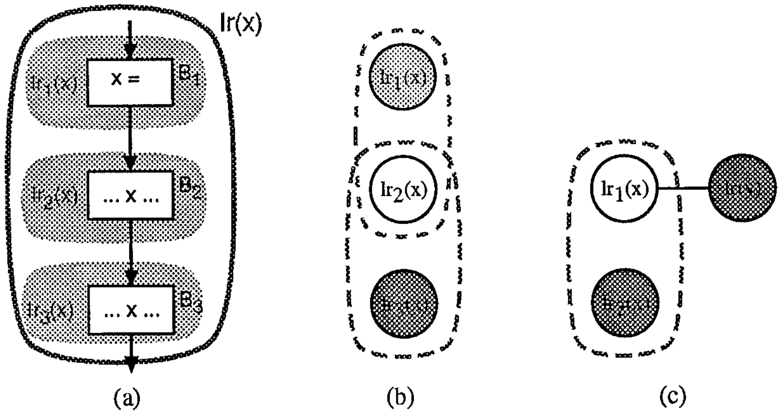


Figure 4.18: Biased coloring

The color ordering phase determines the order in which live ranges are assigned colors. The first step finds an available color matching a colored adjacent partner. The second step tries to avoid using a color that uncolored partners cannot use. The third step of biased coloring picks a color freely for lr if none of the previous two attempts succeeds. Hence, the biased coloring mechanism in [7] has 3 defects.

First, a long live range may be split into several smaller segments. Those smaller segments may be colored in an arbitrary order. Without considering remote partners, biased coloring may easily fail to find a color for all the segments. Assume that the color ordering of the example in Figure 4.18(b) is $lr_1(x)$, $lr_3(x)$, and $lr_2(x)$. Two live ranges circled by a dotted line are adjacent partners. Green color is assigned to $lr_1(x)$. When selecting a color for $lr_3(x)$, the register allocator chooses a color that is still available for $lr_2(x)$ to increase the likelihood of assigning $lr_2(x)$ and $lr_3(x)$ the same color. Because $lr_2(x)$ is not yet colored, and the register allocator doesn't take into account remote partners, it is likely to assign a color to $lr_3(x)$, which is different from $lr_1(x)$. As a result, a shuffle code is inevitable regardless what color is assigned to $lr_2(x)$.

Second, the biased coloring approach does not consider the assigned colors of neighbors' partners ($\bigcup_{n \in neighbor(lr)} \bigcup_{p \in adjacent(n)} color(p)$) while selecting colors. In other words, when choosing a color for lr, the register allocator may assign the color that lr's uncolored neighbors are eager to get even though there are other available colors for lr. In Figure 4.18(c), $lr_1(x)$ interferes with $lr(y)$, and the color ordering is $lr_2(x)$, $lr(y)$, and $lr_1(x)$. Assume that the red color is assigned to $lr_2(x)$. As biased coloring picks a color for $lr(y)$, it has no idea about the assigned color of $lr_1(x)$'s partner. Consequently, the red color may possibly be assigned to $lr(y)$ as well. Now it is impossible for $lr_1(x)$ to have the same color as $lr_2(x)$ because the red color is a conflicting color for $lr_1(x)$. Thus, shuffle code is required.

Third, biased coloring does not prioritize the shuffle moves so as to attempt to eliminate frequently executed shuffle moves.

The biased coloring approach implemented in our register allocator deals with the 3 drawbacks of the classical biased coloring approach adopted by prior reseachers. Partners of a live range in our biased coloring approach consist of adjacent as well as remote partners.

Partners are prioritized in such a way that adjacent partners have higher priorities than remote partners and are prioritized based on their estimated execution frequencies because frequently executed shuffle moves should be eliminated first. The steps are described as follows:

1. $c = color(p)$, where $p = max\_priority(\{x \mid x \in partner(lr) \wedge color(x) \in avail(lr)\})$
2. if not found, choose a color $c \in avail(lr) \wedge c \notin avoid(lr)$.
3. if not found, choose a color $c \in avail(lr)$.
4. if c is found, $avoid(n) += c, \forall n \in neighbor(p), \forall p \in partner(lr)$

The first step chooses one of the assigned colors of lr's partners based on their priorities. The chosen color must be a non-conflicting color. The second step tries to find a non-conflicting color that is not used by the partners of lr's neighbors. The third step chooses a non-conflicting color freely. The fourth step propagates the chosen color to the neighbors of lr's partners.

4.6 Placement of shuffle code

After the color-assignment phase, 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: shuffle move, shuffle store, and shuffle load.

One simple rule of inserting a shuffle store on E is that storing the value of x back to memory when $lr_1(x)$ is in a register, and $lr_2(x)$ is spilled. The rule, however, is too conservative because x's memory location may already contain the up-to-date value. When fusing graphs, the fusion operation propagates the has_def attribute indicating if the value of a live range is defined within the live range. If the has_def attribute of a live range $lr(x)$ is set, then shuffle stores are needed at the edges that exit to a spilled live range $lr'(x)$. For example, in Figure 4.19(a), $lr_1(x)$ is assigned to a register, $r1_x$, and $lr_2(x)$ is spilled.

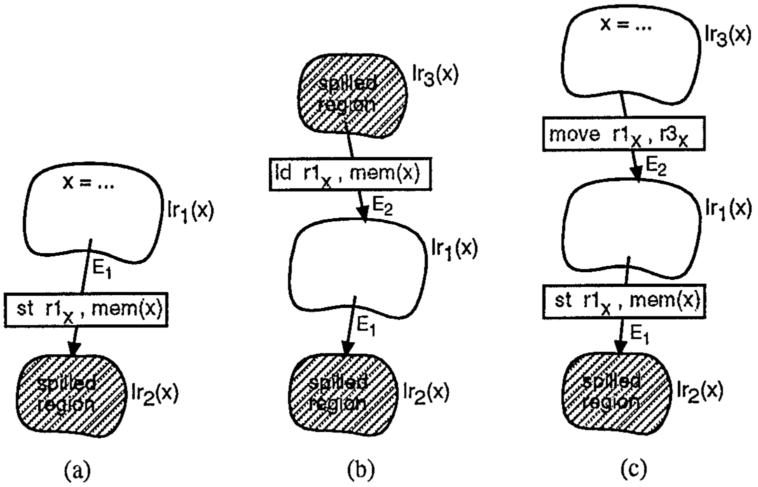


Figure 4.19: Example of inserting shuffle store.

Because x is defined in $lr_1(x)$ (has_def is set), the value of $lr_1(x)$ is stored back to memory by inserting a shuffle store on edge E_1.

When the has_def attribute of a live range is not set (no definition in the live range), shuffle stores may not needed. For instance, in Figure 4.19(b), $lr_1(x)$ is also assigned to register $r1_x$, and $lr_2(x)$ is spilled. However, the value of $lr_1(x)$ is not defined in $lr_1(x)$ (has_def is not set) and comes from a spilled live range, $lr_3(x)$. Because the value of x in its memory location is still up to date, no shuffle store is needed on edge E_1.

We have to pay attention if the value of $lr_1(x)$ comes from another live range that is not spilled and whose has_def is set as shown in Figure 4.19(c). At the exit edge E_1 of $lr_1(x)$, the value of x in its memory location is not up to date so a shuffle store is required to update the memory location of x.

A simple flow-analysis technique is used to compute if the value of a live range $lr(x)$ in its memory location is up to date or not. The flow analysis operates on shuffle moves unlike typical flow analyses in the global optimization phase that operate on control-flow edges. The has_def attribute flows through shuffle moves (forward direction from sources to destinations). Therefore, in the presence of the flow analysis, we know if a shuffle move brings a value that is not yet stored back to its memory location. If the value of $lr(x)$ comes from a shuffle move and the value in memory is not up to date, shuffle stores are required at the edges that exit to spilled live ranges regardless of the has_def attribute of $lr(x)$.

4.6.1 Ordering of shuffle code

Shuffle code on edge $E =< B_1, B_2 >$ may be placed in B_1, B_2, or a newly created block inserted between B_1 and B_2. If B_1 has only one successor, which is B_2, the shuffle code is appended to the end of the schedule of B_1. If B_1 has multiple successors, and B_2 has only one predecessor, the shuffle code is inserted into the beginning of the schedule of B_2. In these two situations, the shuffle code is executed only when the control flow goes through E. If B_1 has more than one successor and B_2 has more than one predecessor, edge E is a critical edge [42]. The shuffle code inserted in B_1 and B_2 may be executed when the control flow does not go through E. Therefore, we perform edge splitting on E [42]; a new block is created and inserted between B_1 and B_2. The shuffle code is then put in the new block.

There is an ordering in which shuffle code is inserted. Shuffle stores are inserted first. Shuffle moves are appended after shuffle stores. Finally shuffle loads are inserted after shuffle moves. A shuffle store for a virtual register x at edge E means that the live range $lr_1(x)$ ends at the exit of B_1 and $lr_2(x)$ is spilled to memory in B_2. In other words, $lr_1(x)$ does not interfere with any live range that starts at the entry of B_2. Likewise, a shuffle load for a virtual register x at edge E indicates that live range $lr_2(x)$ starts at the entry of B_2 and does not interfere with any live range terminating at the exit of B_1. The two conditions imply that shuffle stores should be ahead of shuffle loads. A shuffle move for a virtual register x indicates that $lr_1(x)$ terminates at the end of B_1 and $lr_2(x)$ begins at the entry of B_2. Therefore, a shuffle move has both the conditions of shuffle store and load. A shuffle store ($lr_1(y)$ to memory) should be ahead of a shuffle move ($lr_1(x)$ to $lr_2(x)$). Otherwise, $lr_1(y)$ conflicts with $lr_2(x)$. The value of $lr_1(y)$ is mangled by the shuffle move if the same register is assigned to $lr_1(y)$ and $lr_2(x)$. Similarly, a shuffle move should be in front of a shuffle load.

Multiple shuffle moves may exist on edge E. Special attention must be given to the ordering among the shuffle moves to prevent the source value of a move being targeted by another move. For instance, consider two shuffle moves for x and y ($r_x3 \leftarrow r_x2$ and $r_y4 \leftarrow r_y3$). If the ordering for the two moves is $r_x3 \leftarrow r_x2$ and then $r_y4 \leftarrow r_y3$, the value of y in r_y3 is destroyed by the first move of x. If we swap, however, the two moves so that $r_y4 \leftarrow r_y3$ comes first, the two shuffle moves are able to exchange values safely. The approach of finding the ordering for shuffle moves is done by constructing a directed graph whose nodes are shuffle moves. An arc from a move mv_1 to mv_2 represents that the destination of mv_1 and the source of mv_2 share the same register. Traversing from leaves to roots determines the ordering for shuffle moves. If there is a cycle in the graph, saving/restoring a register to/from a temporary memory location is necessary to replace a move in the cycle so as to break the cycle. To cite an instance, consider two moves, $r_x2 \leftarrow r_x3$ and $r_y3 \leftarrow r_y2$. There is a cycle between the two moves. The code sequence for the two shuffle moves is `st` r_y2`, (temp);` $r_x2 \leftarrow r_x3$`;` `ld` r_y3`,(temp);`. The number of overhead operations is counted as 3 (1 load, 1 store, and 1 move), not 2.

4.6.2 Optimization of placement

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. The code hoisting [41], code sinking [43], and pratial redundancy [15] techniques can be used to optimize the placement of shuffle loads/stores.

However, 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 effective technique is used to optimize the placement of 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 definitions within the live range. If the def-cost is less than the cost of shuffle stores, then all shuffle stores are eliminated by inserting a new shuffle store right after each definition and the 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.

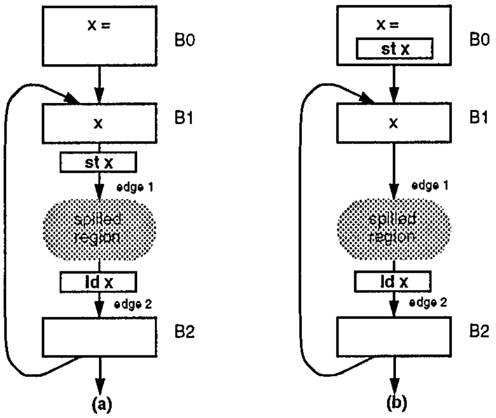


Figure 4.20: Placement of shuffle store.

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. The definitions of those common subexpressions are less frequently executed than the references inside the loops, and they may be spilled inside part of the loops due to the high register pressure. In the absence of the shuffle-code placement optimization, shuffle stores are inserted inside the loops. These shuffle stores are likely to be executed more often than the definitions.

Figure 4.20 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 operations, depicted in Figure 4.20(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 4.20(b)).

4.6.3 A more detailed example

In this section, I provide an example to show the interaction between edge ordering and shuffle code placement. The code fragment of Figure 4.21, is taken from the function update_weights of alvinn. Presumably there are three registers.

Consider an edge ordering based on loop hierarchy:

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

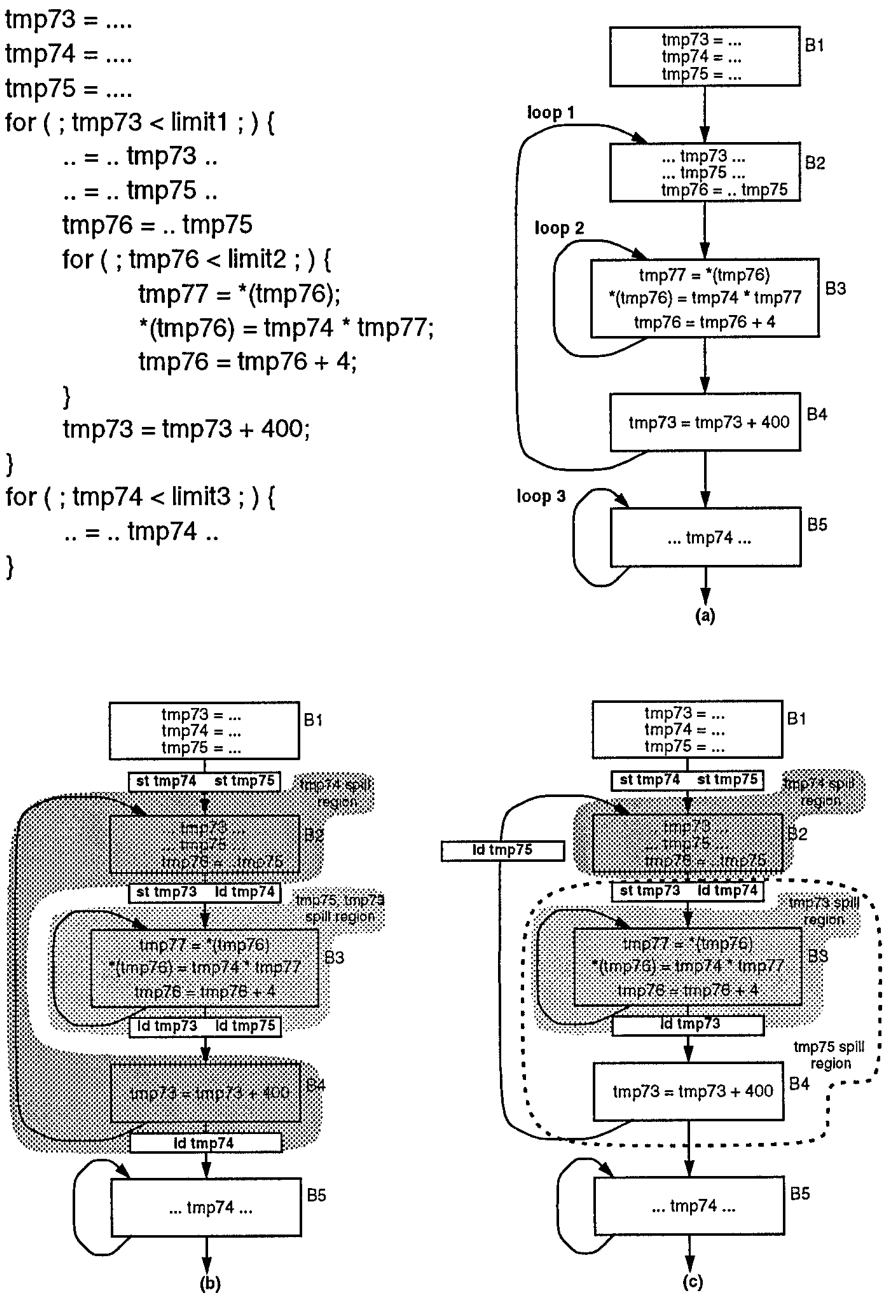
B_3 requires all 3 registers for tmp74, tmp76, and tmp77, which are all referenced within the block. In addition, there are two transparent live ranges induced by tmp73 and tmp75. These two live ranges are spilled when the interference graph G_3 is simplified before graph fusion takes place. B_2 also needs all 3 registers, therefore, the only transparent live range (tmp74) for B_2 is spilled.

As we fuse G_3 along the first edge $\langle B_3, B_3\rangle$, the resulting graph doesn't require any other splitting or spilling decisions since the simplifiability invariant still holds. 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. The shuffle code is highlighted in bold face in Figure 4.21 (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 spilling decision, as seen in Figure 4.21 (c). The shuffle load of tmp74 on $\langle B_4, B_5\rangle$ in Figure 4.21 (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.)

4.7 Evaluation

I measured the impact of the register allocation strategy for various SPEC92 programs (alvinn, compress, ear, eqntott, espresso, gcc, li, sc, doduc, fpppp,



```
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 4.21: Code fragment from `alvinn` with N = 3.

matrix300, nasa7, spice, and tomcatv). I contrast the fusion-based approach with the results using an improved Chaitin-style approach (with the enhancements of Chapter 6). I use all registers on the MIPS, adhering to the standard calling convention. 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, I run two experiments. First, I use only static information to guide register allocation. That is, we use loop depth to estimate the execution frequency of a block. In the fusion-based approach, when considering the basic blocks inside a loop, I use breadth-first order to select edges for fusing. In a second experiment, I use profile information from a prior execution.

As expected, for programs with many small functions, Chaitin's algorithm works fairly well, and the fusion-based approach produces identical results. For most of these programs, use of profile information improves the result, independent of the register allocation strategy. Let us, 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 4.22 shows that 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 using profile information. 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 color-assignment phase (when deciding if a caller-save or a callee-save register should be assigned to a live range, or the live range should 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 28 %; 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 execution frequency provides a good estimation, the profile information does not reduce overhead operations relative to the static case.

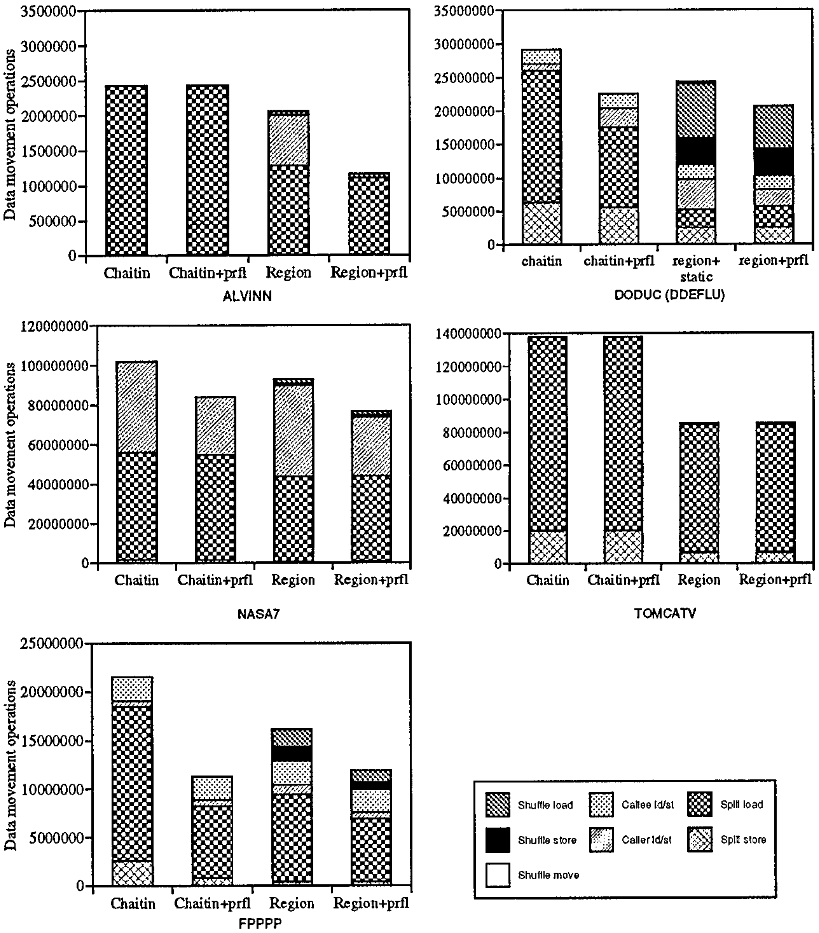


Figure 4.22: All available registers (26 int, 16 double).

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 4.22 the result for ddeflu, where large portions of the spill loads and stores are replaced by shuffle code.

4.8 Summary

In this chapter I present a new approach to register allocation which is fusion-based. The approach deals with spilling, splitting, and color assignment in an integrated fashion. Fusion-based register allocation produces good results in those situations where a conventional Chaitin-style allocator breaks down. 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. In the next chapter, I investigate how the fusion-based approach behaves under the register pressure caused by aggressive code transformations.

Chapter 5

Reconciliation

Nowadays, modern processors have multiple function units as well as the ability to dynamically detect data dependences and to issue multiple independent instructions per cycle [57, 21, 4, 32, 31]. To extract as much instruction-level parallelism (ILP) as possible, modern compilers are looking at increasingly larger regions of a program. Many global optimizations have also been extended to concentrate on frequently executed regions at the potential expense of less optimal code in infrequently executed regions. For example, superblock optimizations [14] extend code motion, dead code elimination, and other optimizations to work on regions comprising extended basic blocks. Regions have also been used to guide predication and if-conversion [22]. This general compilation strategy — wherein the compiler prioritizes frequently executed program regions by concentrating the scope of an optimization phase on such regions — has been termed *region-based compilation* [33]. To exploit the full power of region-based techniques, scope enhancing transformations, such as loop unrolling and function inlining, are necessary to enable the formation of large regions that comprise the most frequently executed blocks of a program. For example, selective unrolling of innermost loops is an attractive scope-expanding optimization for region-based compilers [47].

5.1 Challenges

Region-based compilation is at odds with register allocation: standard register allocation algorithms operate at the scope of an entire function and do not focus their efforts on high priority regions. Moreover, scope enhancing transformations, when coupled with aggressive region-based optimizations (e.g., global instruction scheduling and speculative code motion) severely increase register pressure, creating serious problems for standard function-wide graph-coloring register allocators. For example, loop unrolling may increase the register pressure considerably. After common subexpression elimination, strength reduction, and code scheduling, the live ranges of the different instances of the (former) loop body now overlap, and the register allocator faces a more difficult situation than without unrolling. If the register allocator is unable to handle the unrolled loop, then the benefits of using aggressive region-based optimizations can be negated by the register allocation strategy.

Furthermore, how to reconcile register-allocation decisions among regions is another challenge to register allocation. The IMPACT compiler generates code on a region basis. Reconciliation binding decisions among regions is one big problem for the compiler. To cite from Hank's thesis [34] that implements register allocation in the IMPACT compiler:

> *If the register binding information is not taken into account during the allocation of the current region, separate allocation can quickly produce an explosion of copy operations at region boundaries.*

Consider that $lr(x)$ and $lr'(x)$ are in two adjacent regions, R and R' respectively, and the two live ranges span across an edge E that connects the two regions. Shuffle code is required if register allocation assigns $lr(x)$ and $lr'(x)$ different registers. Assume that R is a higher priority region than R' and is compiled already. The IMPACT region-based register allocator uses an ad hoc approach to reconcile binding decisions between two regions. To avoid an excessive number of copy operations (shuffle moves), the IMPACT compiler attempts to assign $lr'(x)$ the same register as $lr(x)$. If the compiler fails to assign the register of $lr(x)$ to $lr'(x)$, then $lr'(x)$ is spilled. Copy operations are completely eliminated at the expense of shuffle loads/stores. In other words, expensive (usually more than one cycle) load/store operations are used to trade off cheap (one cycle) copy operations. Because the IMPACT region-based register allocator is incapable to reconcile binding decisions among regions, we see that this region-based register allocator produces more overhead operations than a function-based register allocation for 8 of 11 benchmark programs. Moreover, when we see the measurement of the execution time, we notice that 5 of the 11 benchmark programs using region-based register allocation run slower than using function-based register allocation.

In this chapter, I investigate the effectiveness of fusion-based register allocation. There are three topics to gauge the impact of register allocation by graph fusion on region-based compilation:

- What is the impact of the choice of the base region (Section 5.2);
- How sensitive is register allocation to profile information, which determines the various heuristics employed in a register allocator (Section 5.3);
- How sensitive are these approaches to register pressure (Section 5.4).
- How sensitive are these approaches to different region sizes (Section 5.5).
- How sensitive are these approaches to two code transformations, loop unrolling (Section 5.6) and function inlining (Section 5.7).

5.2 Choice of base region

One of the parameters of fusion-style register allocation is the shape of the base regions (for which we compute the initial interference graphs). If we choose a complete function as the base region, then fusion-style register allocation is identical to the (function-wide) Chaitin-style register allocation. The size of the base region affects spill and shuffle costs. In this section, we investigate how the size of the base region has an effect on the quality of register allocation. Section 5.2.1 shows an example that favors a small size of the base region. Section 5.2.2 depicts an example that favors a big size of the base region.

5.2.1 Favor small size of the base region

The example of Figure 5.1 (a) is taken from Figure 4.2(a). This case prefers a small size of the base region. We list the execution frequency beside each edge. The most frequently executed path is $B_1 \rightarrow B_2 \rightarrow B_4$. The edge ordering that the register allocator considers for graph fusion is <B_1,B_2>, <B_2,B_4>, <B_1,B_3>, and <B_3,B_4>. Let us first consider the case that the size of the base region is one single block. Each interference graph for each block is shown in Figure 5.1(b). After graphs are fused along the first three edges, <B_1,B_2>,

<B_2,B_4>, and <B_1,B_3>, the resulting graph can still be simplified with N=2 (shown in Figure 5.1(c)). As the graph is fused along <B_3,B_4> by combining $lr_3(y)$ and $lr_4(y)$, the final graph (shown in Figure 5.1(d)) becomes too constrained; the maintaining-simplifiability step then splits the live range of y on edge <B_3,B_4> (we obtain the interference graph in Figure 5.1(c) by splitting the live range of y). As a result, there are 10 incurred shuffle move operations using one single block as the size of the base region.

Consider that the size of the base region is increased up to 3 blocks. Namely, the size of the base region cannot exceed 3 blocks. Region R_1 contains the most frequently executed path $B_1 \rightarrow B_2 \rightarrow B_4$, and region R_2 contains only one single block, B_3 (shown in Figure 5.1(e)). The interference graph of each region is depicted in Figure 5.1(f). The fusion-based register allocator performs graph fusion along two edges, <B_1,B_3> and <B_3,B_4>. After G_{R_1} and G_{R_2} are fused along edge <B_1,B_3>, we get the resulting graph that is the same as the interference graph in Figure 5.1(c). Like in the previous example using blocks as regions, fusing this graph along edge <B_3,B_4> generates the same graph as the interference graph in Figure 5.1(d). Again splitting the live range of y takes place to maintain the simplifiability invariant. There are also 10 incurred shuffle move operations.

As soon as the size of the base region goes up to 4 blocks, all four blocks are in one region R. All four control-flow edges are in R, no graph fusion is needed. G_R is the same as the interference graph in Figure 5.1(d). Simplification of the interference graph blocks during the graph simplification phase (Section 4.1.2). Spilling one live range is needed. The live range of x has the smallest spill cost (190 spill code operations). The spill cost of the other two live ranges, x and y, are 200. Spilling the live range of x, therefore, incurs 190 spill code operations. The big size of the base region introduces more overhead operations than the small size of the base region.

5.2.2 Favor big size of the base region

Figure 5.2 (a) is the code depicted in Figure 4.14. The interference graph of each block is shown in Figure 5.2(b). The lightly shaded live ranges are transparent live ranges. The register pressure of B_1 and B_2 is 3. Regardless the size of the base region, the register allocator has to make out-of-color spilling decisions for the base regions that consist of B_1 or B_2. If the size of the base region is small, the quality of register allocation may be

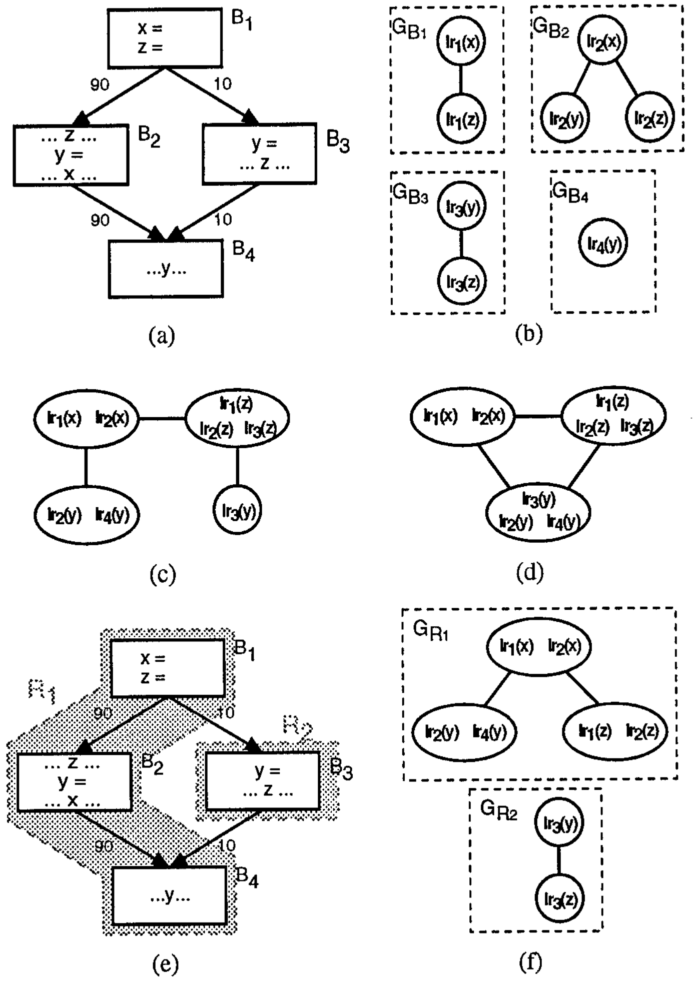


Figure 5.1: Size of base regions (N=2).

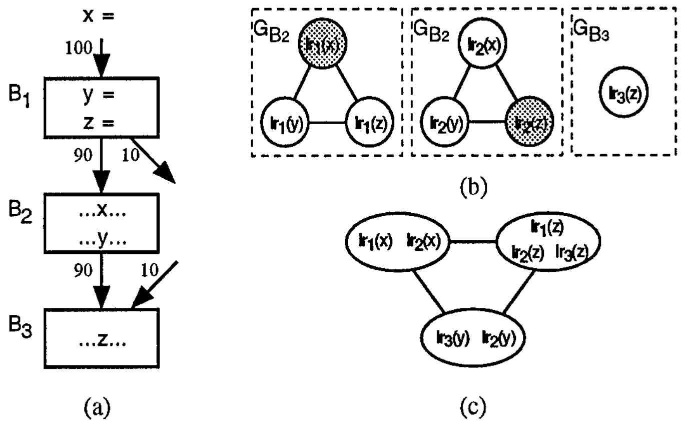


Figure 5.2: Size of base regions (N=2).

hurt by making spilling decisions locally. If we use blocks as the base regions, the graph simplification step spills $lr_1(x)$ and $lr_2(z)$ because the register allocator favors spilling transparent live ranges. Although the register allocator does not produce spill code using the spilling decision (spill transparent live ranges), it pays 370 shuffle load/store operations (100 shuffle stores of x on the edge entering B_1, 90 shuffle loads of x on <B_1,B_2>, 90 shuffle stores of z on <B_1,B_2>, and 90 shuffle loads of z on <B_2,B_3>).

However, if we put all three blocks into one region, the register allocation has a more global view so as to make wise spilling decisions. The interference graph of the region is shown in Figure 5.2(c). The graph simplification step then spills $lr(x)$ which has the smallest spill cost (90) and is composed of $lr_1(x)$ and $lr_2(x)$. The spilling decision generates 90 spill code operations and 100 shuffle store operations on the edge that enters B_1. The big size of the base region introduces fewer overhead operations than the small size of the base region.

5.2.3 Evaluation

To test the hypothesis that fusion-style register allocation handles scope-enlargement gracefully, I increase the scope of the base region by using loop bodies as base regions. In Figure 5.3, I report the performance using basic blocks and loop bodies as the choice of base regions, both with and without loop unrolling. Without loop unrolling, the performance of the basic block regions is identical to the performance of the loop body regions. When loops are unrolled, however, using single basic blocks as the base regions produces fewer overhead operations — using the larger regions formed by loop bodies after loop unrolling incurs a slight penalty. Nevertheless, even the worst case for fusion-style register allocation (i.e., the case where the base regions are the unrolled loop bodies) produces only 50% of the overhead operations generated by a Chaitin-style register allocator. In the best case, using fusion-style register allocation results in only 12% of the transfer operations required by Chaitin-style register allocation.

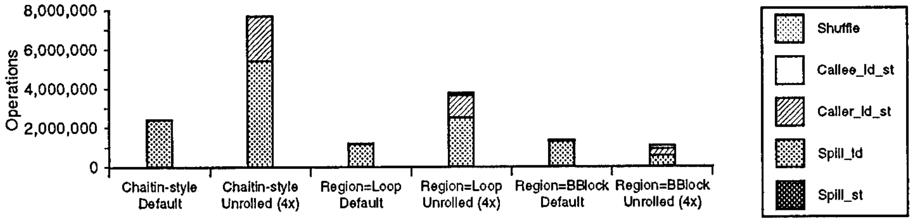


Figure 5.3: Transfer operations in `alvinn`, using all registers, for two different region definitions.

Figure 5.4 depicts the result of compiling for a smaller number of registers (11 integer, 9 floating point registers). Notice the big difference in the number of overhead operations (with unrolling) between the base Chaitin-style register allocator and the fusion-style register allocator (almost a factor of 4). The choice of regions, however, has no noticeable impact on fusion-style register allocation; there just are not enough registers to keep all values in registers. We notice a large number of shuffle operations for loop unrolling code, indicating that the register allocator tries hard to keep values in registers, even if this means that a live range must be broken into smaller segments. From Figures 5.3 and 5.4 we see how poorly the function-wide approach (Chaitin-style) performs compared to a region-based framework:

- without unrolling, the fusion-style approach executes 50% of the operations executed by the Chaitin-style approach;
- with unrolling, using the fusion-style approach results in 25% of the operations required by the Chaitin-style allocator.

Furthermore, when the scope is expanded via loop unrolling, the fusion-style approach, in the small-number-of-registers scenario, executes only 60% of the data movement operations executed by the Chaitin-style approach when given all available registers.

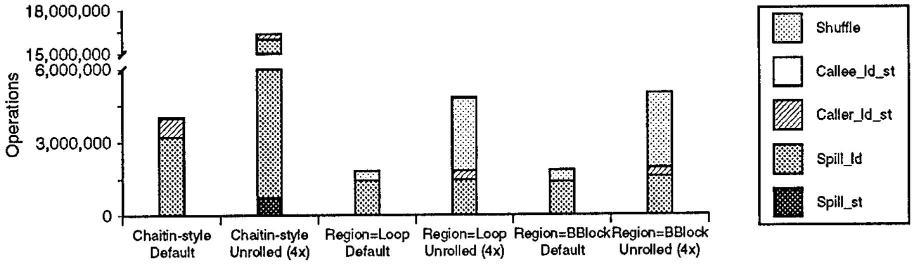


Figure 5.4: Transfer operations in `alvinn`, small number of registers, for two different region definitions.

5.3 Profiling

The figures presented so far in this section were obtained using static information for edge ordering and heuristics. Information obtained from program execution can sometimes improve a register allocator's heuristics, thereby leading to a better register allocation. Figure 5.5 contrasts the performance of a Chaitin-style register allocator for `alvinn` with two variants of the fusion-style register allocator when using profile information.

For all approaches, the results for the unrolled versions of the program are improved noticeably by using profile information. Compare the results for Chaitin-style allocation of Figure 5.3 and 5.5 (the two left-most columns). We see that in the unrolling case, the number of overhead operations is cut in half for Chaitin-style register allocation using profile information as compared to Chaitin-style register allocation using static information. Without unrolling, the improvement is minor. This is not surprising. With unrolling, there are more values that can be put into registers. But without profile information, the heuristics to choose spilling candidates misguides the register allocator. When the compiler uses profile information, better heuristic choices are possible, and we see a clear improvement since there exists many candidates to pick from in the unrolled code.

Fusion-style allocator is able to exploit the same effect (better heuristics). Since fusion-style register allocation is edge-order sensitive, we can expect additional effects from the improved edge ordering that is obtained from the use of profile information. Comparing Figure 5.3 with Figure 5.5, we see that the use of profile information reduces the number of overhead operations by 50% (for the unrolled loops). The number of load operations due to spilling is significantly reduced, and we see also a reduction in the number of operations to save or restore caller-save registers. This reduction of spill overhead indicates that the profile information improved the heuristics that allow the register allocator to make better spilling decisions. The change in caller-save register induced overhead implies that the profile information allowed the register allocator to control the growth of live ranges. If there are sufficient registers, without any checks and balances, fusion grows live ranges until a live range conflicts with other live ranges. As the live range grows, it is more likely to include function calls. These, however, incur a cost if the value is in a caller-save register. If there are only a few references to the value, then it might be better to leave the live range in memory. Access to accurate profile information allows the register allocator to better analyze the tradeoff among caller-save registers, callee-save registers, and spilling (See Chapter 6). This improved decision making accounts for the better results depicted in Figure 5.5. We also see in Figure 5.5 that profile information results only in minor improvements if we use blocks as base regions.

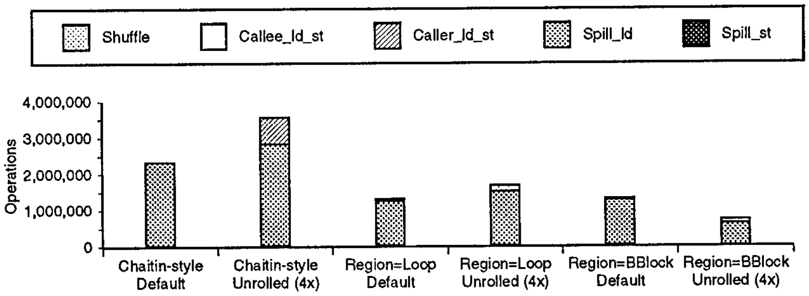


Figure 5.5: Transfer operations in `alvinn`, using all registers and profile information.

For another view, I present in Figure 5.6 the results from compiling `espresso`, using the high-register-pressure scenario. We notice that the Chaitin-style register allocator requires significant spill *loads*, whereas the fusion-style allocator causes significant shuffle activity (some of these operations are stores). Furthermore, looking at the influence of the choice of the base region, we notice that basic blocks as base regions provide only slightly better, but vastly different results. The number of spill loads is reduced dramatically, which indicates that the fine-grain base allows the register allocator to keep values in a register that end up in memory using the slightly coarser-grain base of a loop nest. In other words, the register allocator is able to split live ranges, but moving values from one storage location to another implies shuffle code. About half of the shuffle code are load operations, the other half stores. Register moves play almost no role for this program. The discussion of the differences between basic blocks and loop nests as a base region extends to using complete functions as a base region, as is done in a standard Chaitin-style allocator. We see again that a finer-grain choice of base region (loops vs. functions) reduces the number of overhead operations.

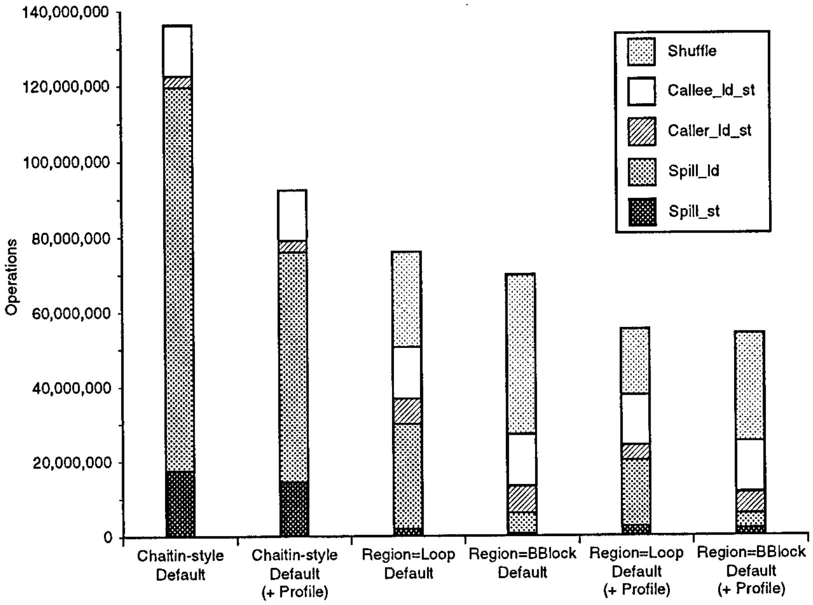


Figure 5.6: Transfer operations in espresso, no unrolling.

Figure 5.7 depicts the result when compiling the programs with loop unrolling. Clearly, expanding the scope implies a penalty for register allocation. The fusion-style approach, however, requires only half of the overhead operations that are inserted by a Chaitin-style allocator. Again, profile information helps although not as much as in the absence of unrolling.

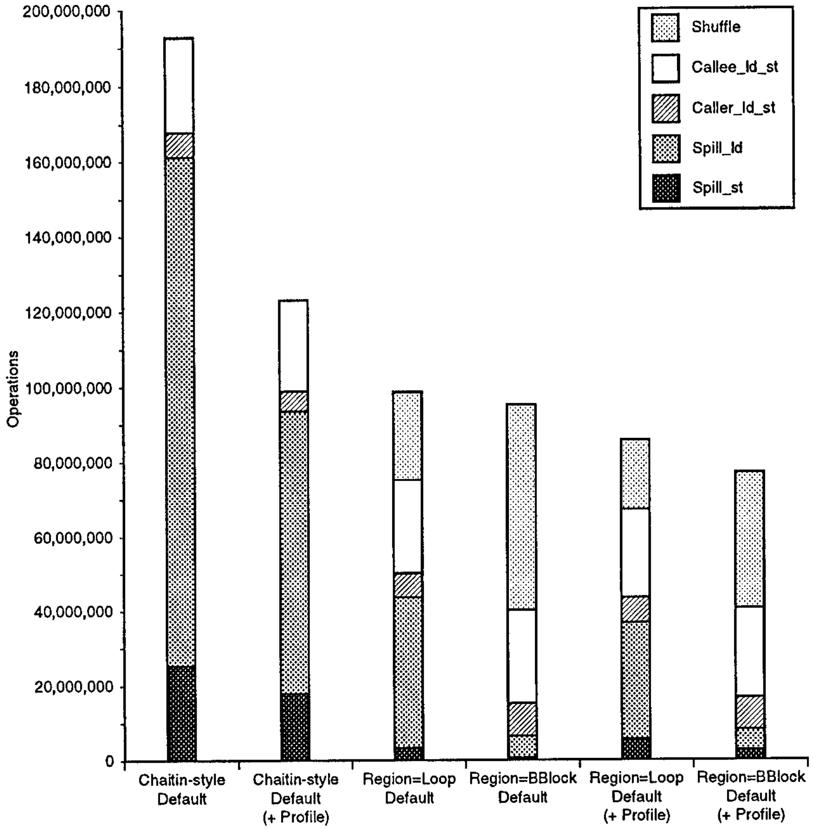


Figure 5.7: Transfer operations in `espresso`, with unrolling.

5.4 Sensitivity to number of registers

The transformations of region-based compilers increase register pressure, and another way to assess how a register allocator deals with register pressure is to investigate how sensitive the allocator is to the number of registers that are available. Clearly, as we increase the number of registers, there comes the point of diminishing return. The next figures depict the number of overhead operations as we allow the register allocator to use more registers (the number of additional registers is shown on the X axis). Recall that the MIPS architecture has separate register banks for integer and floating-point values. The notation (R_i,R_f,E_i,E_f) of the next figures shows the combination of caller-save and callee-save registers for the two register banks. R_i and R_f are the number of caller-save registers for integer and floating-point values, respectively. E_i and E_f are the number of callee-save registers for integer and floating-point values. The standard calling convention of the MIPS machine uses 4 registers of the integer bank to pass arguments and 2 registers to pass function-return values; in addition, 2 floating-point registers pass arguments and 2 floating-point registers can hold a function-return value. All these registers are caller-save registers. Hence, the register allocator uses at least (6,4,0,0).

Looking at the SPEC92 programs, we can classify them roughly into 5 classes:

1. *Fusion-style* $\gg$ *Chaitin-style*: Fusion-style register allocation produces consistently better results than Chaitin-style allocation. Examples are `alvinn`, `espresso`, `nasa7`, and `tomcatv` (Figure 5.8).

2. *Fusion-style* $\approx$ *Chaitin-style (profile sensitive)*: Programs sensitive to getting accurate frequency information (i.e., given the same frequency information, both fusion-style and Chaitin-style register allocation perform about the same). `compress`, `sc`, `li`, and `gcc` are four examples (Figure 5.9).

3. *Fusion-style > Chaitin-style (static case)*: The heuristics that determine which live range to spill are sometimes helped by profile information, and in this case, a Chaitin-style allocator may obtain the same results as fusion-style allocation; see Figure 5.10 for an example (`fpppp`). The results for the Chaitin-style allocator are marginally better than the result for fusion-style allocation. Even with profile information, the

register allocator must make many guesses, and results like those depicted in Figure 5.10 indicate that the spilling decision still depends on some heuristics.

4. *Fusion-style > Chaitin-style (small number of registers)*: Fusion-style register allocation produces fewer overhead operations than Chaitin-style allocation when the register allocator uses a small number of registers. As the number of register increases, the difference of register overhead diminishes. `ear` and `matrix300` are two examples (Figure 5.11).

5. *Fusion-style ≈ Chaitin-style*: Finally, there are programs where neither the choice of register allocation strategy nor the use of profile information seems to make a difference. Examples of this class of programs are `eqntott`, `doduc`, and `spice` shown in Figure 5.12.

5.5 Different region sizes

In this section, we evaluate the influence of different region sizes to fusion-based coloring. Three kinds of region are investigated, blocks, loops, and traces. The region-formation phase determines a trace by selecting a seed block and growing the trace along desirable paths. A trace $\mathcal{T}$ is expanded in the forward as well as the backward direction:

- *forward*: $\mathcal{T}$ is expanded forward by adding block B_k to $\mathcal{T}$ if the condition is satisfied.

$$\exists E =< B_j, B_k > \wedge B_j \in \mathcal{T} \wedge \frac{freq(E)}{freq(B_j)} \geq \beta \wedge \frac{freq(B_k)}{freq(seed)} \geq \gamma$$

- *backward*: $\mathcal{T}$ is expanded backward by adding block B_i to $\mathcal{T}$ if the condition is satisfied.

$$\exists E =< B_i, B_j > \wedge B_j \in \mathcal{T} \wedge \frac{freq(E)}{freq(B_i)} \geq \beta \wedge \frac{freq(B_i)}{freq(seed)} \geq \gamma$$

$freq(B_j)$ and $freq(E)$ are the estimated execution frequencies of block B_j and edge E respectively. The two constants, β and γ, are set to 0.5. These two conditions are the conditions used in the IMPACT compiler to form regions [34].

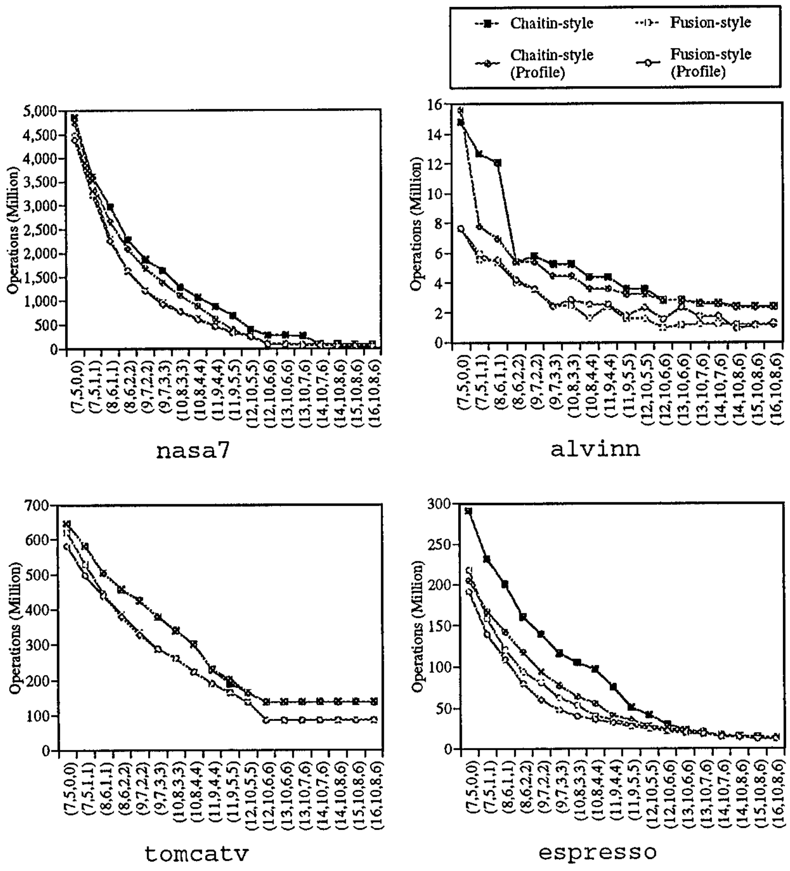
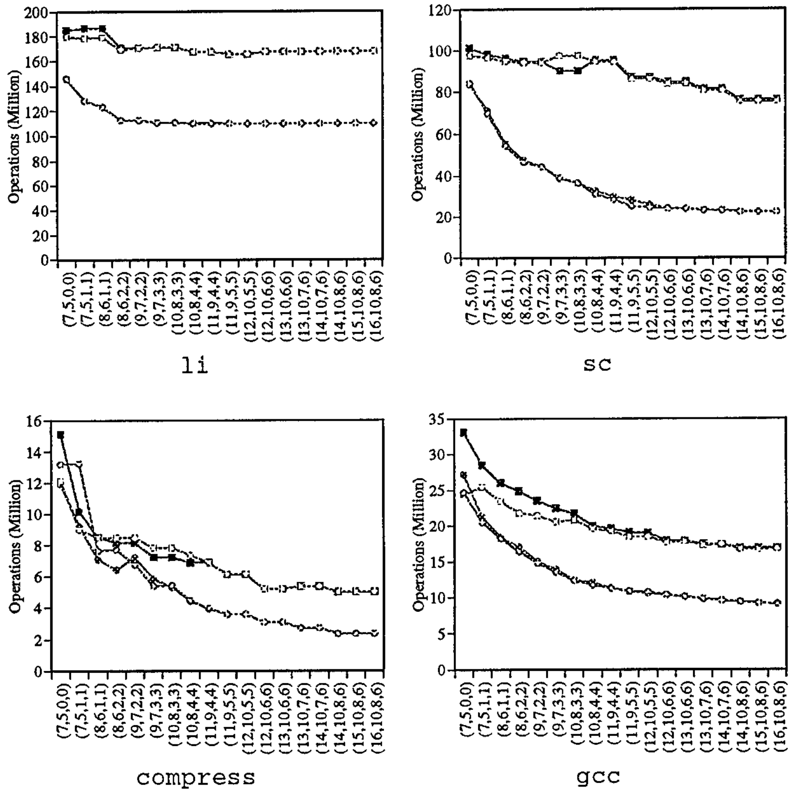


Figure 5.8: Fusion-style ≫ Chaitin-style.



li

sc

compress

gcc

Figure 5.9: Fusion-style ≈ Chaitin-style (profile sensitive).

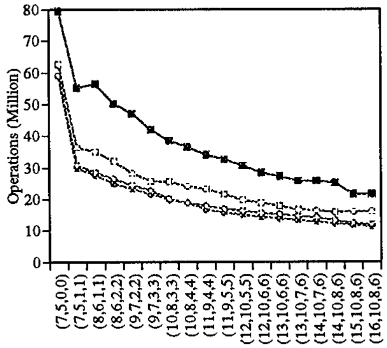
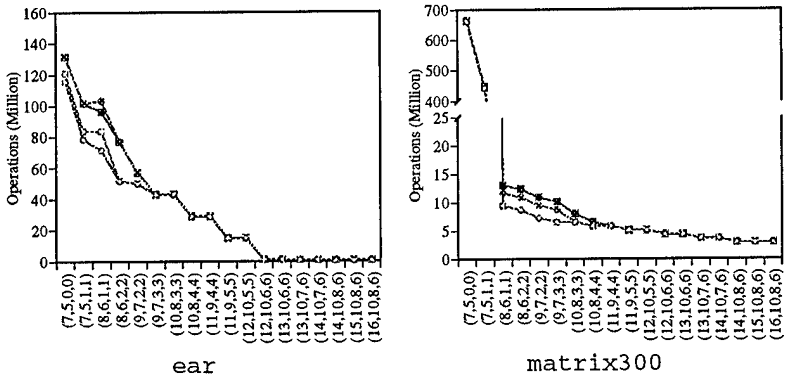


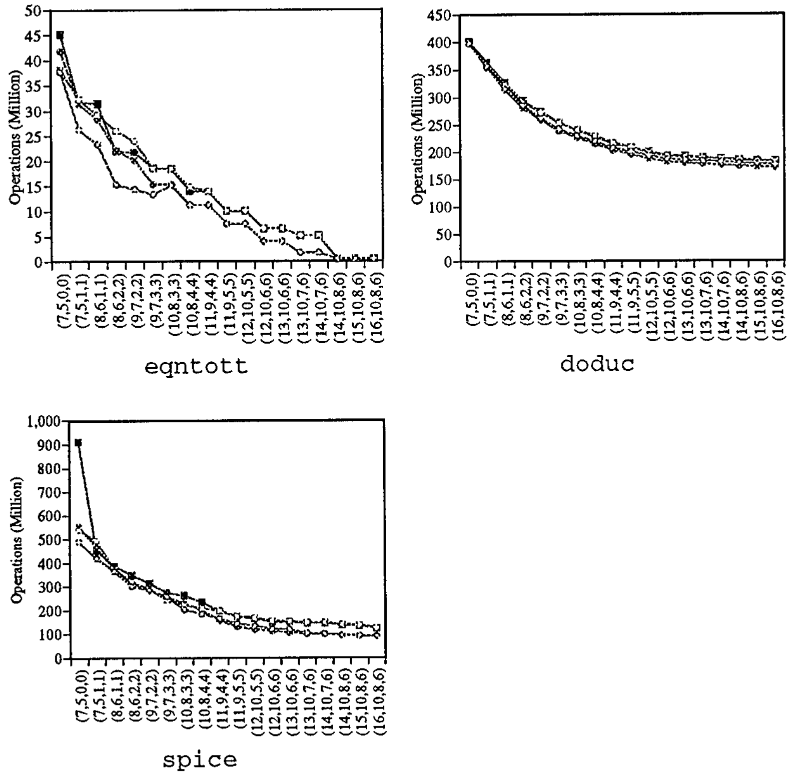
Figure 5.10: Fusion-style > Chaitin-style in the static case (fpppp).



ear

matrix300

Figure 5.11: Fusion-style > Chaitin-style (small number of registers).

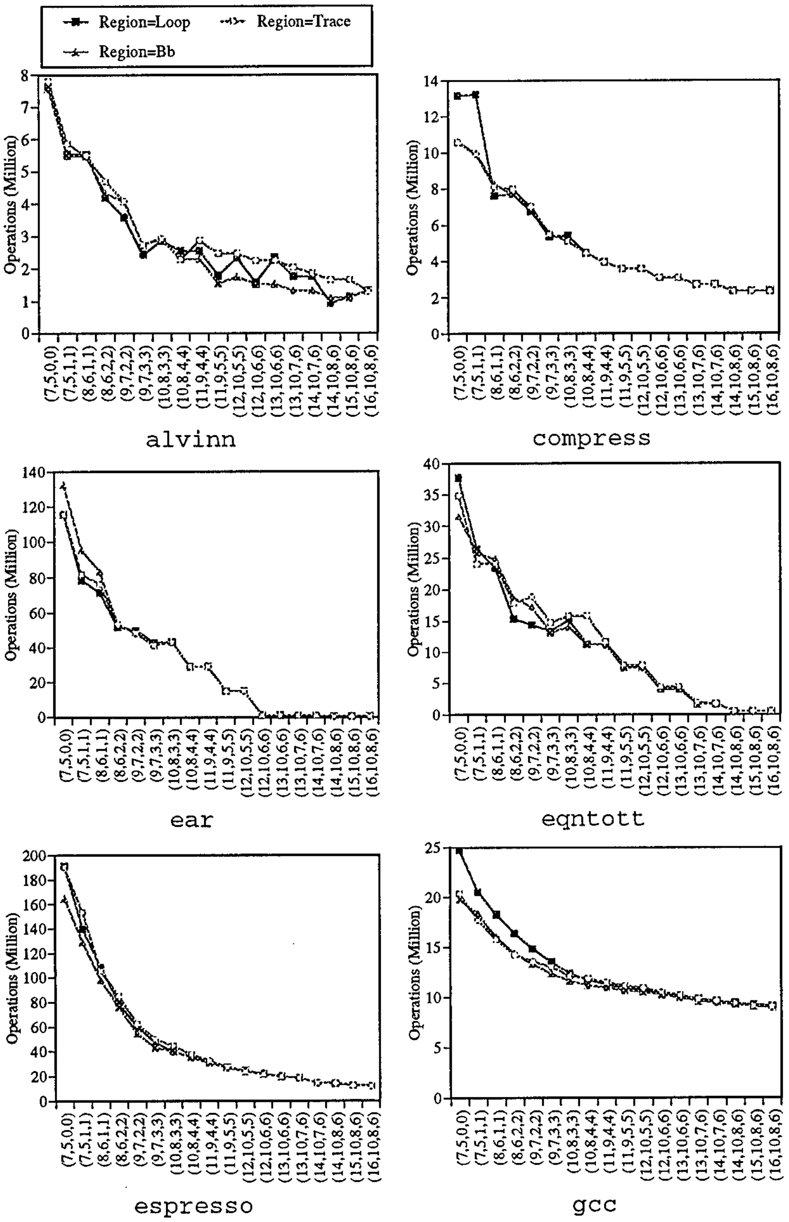


eqntott

doduc

spice

Figure 5.12: Fusion-style ≈ Chaitin-style.



alvinn

compress

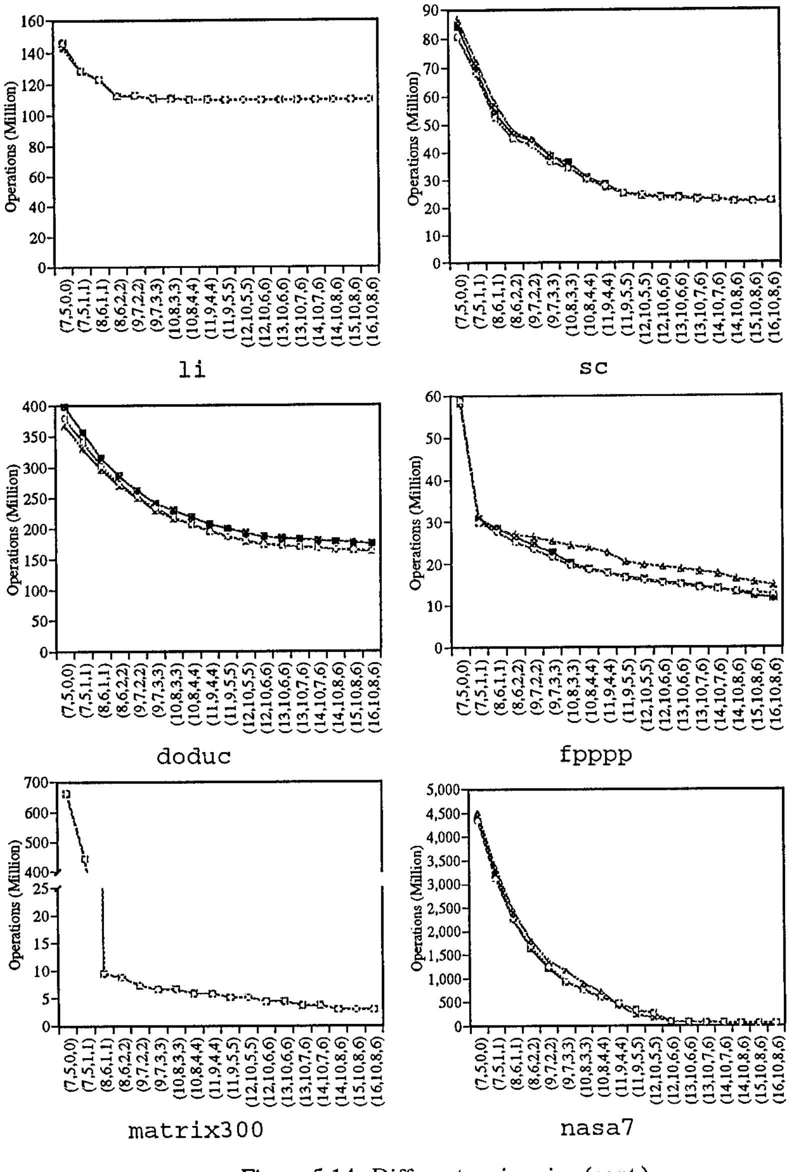
ear

eqntott

espresso

gcc

Figure 5.13: Different region size.



li

sc

doduc

fpppp

matrix300

nasa7

Figure 5.14: Different region size (cont.).

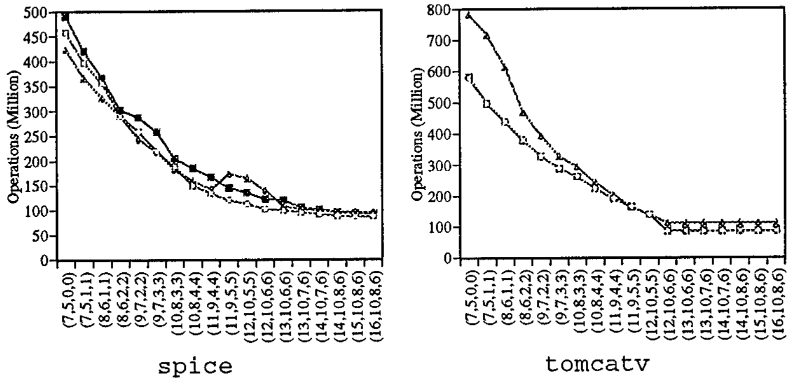


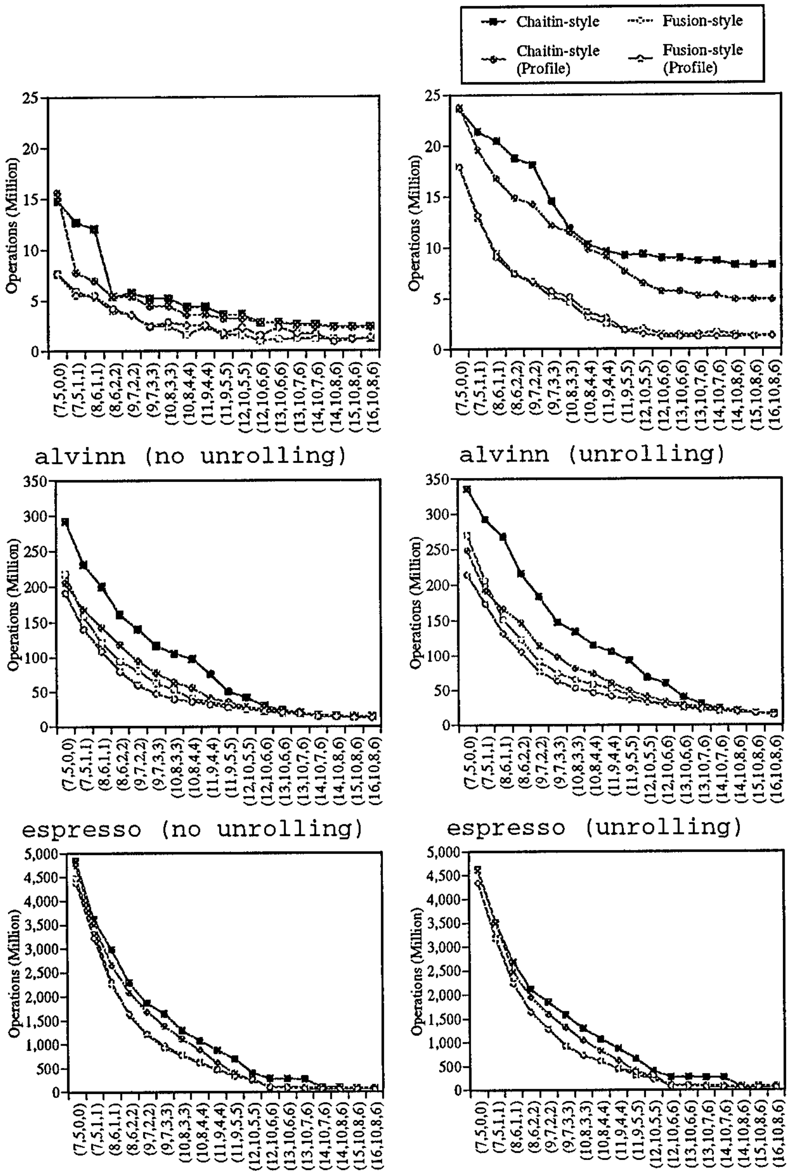
Figure 5.15: Different region size (cont.).

Figures 5.13, 5.14, and 5.15 depict the experimental results of fusion-style coloring with three different region sizes. There are several noticeable aspects. None of the region sizes always outperforms the others because the results highly depend on program characteristics and combination of registers. The influence diminishes as the number of registers increases because the number of registers is then sufficient for all different region sizes. In general, alvinn, espresso, gcc, and doduc prefer a small size of the base region. For gcc and doduc, the results of using traces as regions are as good as using blocks. fpppp, nasa7, and tomcatv prefer a big size of the base region (loops or traces).

5.6 Loop unrolling

Loop unrolling is the code transformation that exposes multiple loop iterations to the compiler. Classical global optimizations such as constant folding, common subexpression elimination, and strength reduction can be applied across different loop iterations. Loop branches and exits inside unrolled loop bodies may be eliminated. Code scheduling can take advantage of the bigger scope of loop bodies and extract ILP among multiple loop iterations. However, the beneficial opportunities to global optimizations and code scheduling do not come for free because of increased register pressure. More common subexpression elimination opportunities imply that more live ranges are needed for the temporaries to hold the increased number of common subexpressions. Strength reduction replaces a more expensive (more cycles) expression by a cheaper (fewer cycles) expression. A temporary that is live throughout the loop is required to hold the value of the expression. Hence, the more expensive expressions are replaced by cheaper expressions, the more temporaries (live ranges) are introduced. All in all, the larger scope of unrolled loop bodies implies that more live ranges are required because there exist more values, variables, and expressions. More aggressive code scheduling implies that the number of live ranges that are simultaneously live increases because more otherwise disjoint live ranges now overlap.

I run experiments with different combinations of register for `alvinn`, `espresso`, `nasa7`, and `tomcatv` with unrolling to see how graciously Chaitin-style and fusion-style coloring behave in the presence of unrolling. To allow an easy comparison, the results of Figure 5.8 are repeated in Figures 5.16 and 5.17. There are two interesting aspects. First, unrolling does not impose serious register pressure for some SPEC92 programs. For `nasa7` and `tomcatv` in both the static and dynamic cases, Chaitin-style and fusion-based register allocation with unrolling produce a similar number of overhead operations as without unrolling. Second, unrolling generates register pressure for `alvinn` and `espresso`. We see that the curves of Chaitin-style and fusion-style register allocation with unrolling diverge further apart than without unrolling. For `alvinn`, fusion-style coloring with unrolling generates twice as many overhead operations as without unrolling (25 % less than Chaitin-style with unrolling) when the number of registers is small. Nonetheless, as the number of registers increases, fusion-style coloring is able to deal with the high pressure caused by unrolling and produces roughly the same number of overhead operations as without unrolling. When analyzing the increase of overhead operations for Chaitin-style with unrolling relative to without unrolling, we see that Chaitin-style coloring is not able to handle the increased register pressure benignly for `alvinn`. The number of overhead operations for unrolling using profile information grows only 50 % at (7,5,0,0) but is tripled for the other combinations of registers. The number of overhead operations in the static case increases less than 80 % when the number of registers is small (prior to (8,6,2,2)) but is quadrupled when the number of registers exceeds (8,6,2,2). For `espresso`, the gap between Chaitin-style and fusion-style enlarges and the merge points of the curves shift to the right. Here we see that fusion-style handles high register pressure graciously.



alvinn (no unrolling)

alvinn (unrolling)

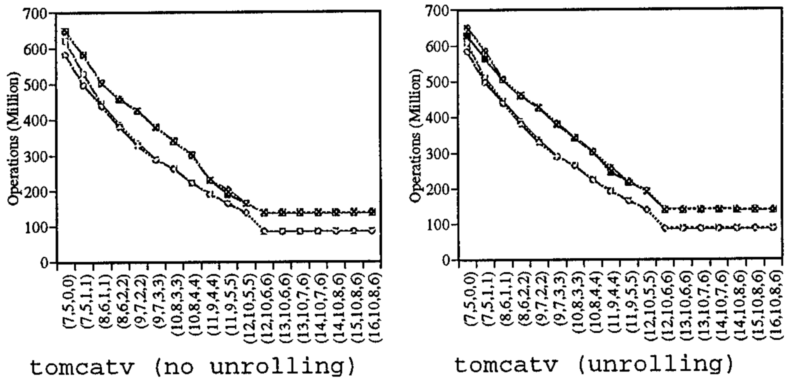
espresso (no unrolling)

espresso (unrolling)

nasa7 (no unrolling)

nasa7 (unrolling)

Figure 5.16: Overhead operations of register allocation for loop unrolling.



tomcatv (no unrolling)

tomcatv (unrolling)

Figure 5.17: Overhead operations of register allocation for loop unrolling (cont.).

5.7 Function inlining

Function inlining is a code transformation that expands the compilation scope beyond function boundaries. Functions are artificial boundaries induced by programmers. Those boundaries may not be the best partitioning borders as far as compilers are concerned. Function inlining affects the compiler in several aspects:

- *global optimization*: Compilers can improve the quality of a program once they know that certain values are constants in the program. Constant folding can calculate the value of an expression at compile time instead of run time [2]. Branch folding eliminates a conditional branch if the result of the comparison of the branch is known at compile time. Loop unrolling prefers constant loop bounds. When a constant is passing to a function call as an argument, without inter-procedural constant propagation [11, 30], the compiler must be conservative and assume that the argument is not a constant. Consequently, some optimization opportunities are lost.

  Function inlining also exposes more opportunities to the global optimizer to find common subexpression between the caller and callee. Moreover, functions are no longer barriers for code motion [36], e.g. a loop invariant of the callee can be moved outside a loop of the caller.

- *memory aliasing*: Without inter-procedural analysis, it is impossible for compilers to get the information regarding incoming arguments. In such a case, compilers usually make conservative assumptions on memory aliasing; that is, two memory accesses always alias if both use incoming arguments as base addresses. Conservative memory aliasing constrains code scheduling to move instructions— two memory accesses must be issued in order. Function inlining replaces formal parameters by their actual parameters. The memory aliasing information is improved (more precise) as a result [3].

- *ILP*: Without function inlining, the compiler is limited by the traditional scope of the compilation unit (function). The bigger the compilation scope, the more likely code scheduling is able to exploit ILP (more likely to find independent instructions). [33] shows that some C programs spend 80 % of their execution time in functions with fewer than 250 operations prior to inlining. After inlining, those C programs spend 50 % of their execution time in functions with more than 1000 operations.

- *calling convention*: Passing parameters to the callee and returning a value to the caller are usually done via argument registers and stack. For example, the caller moves outgoing arguments, if there are any, to argument registers or pushes the arguments onto the local stack if all argument registers are used up by prior arguments. The callee then follows the same convention to get actual parameters and moves them to their corresponding formal parameters. Function inlining eliminates the overhead for moving arguments, returning function values, adjusting the stack frame, and saving/restoring the return address.

- *register allocation*: Two counteractive factors are happening to register allocation when functions are inlined: increment of spill cost and decrement of call cost. When a function call, `bar()`, is replaced by the function body of `bar`, the number of live ranges in the current function increases and all live ranges that contain the call originally interfere with all new live ranges introduced by `bar`. In other words, function inlining increases register pressure and hereby potentially introduces more spill code operations. Nevertheless, if the compiler divides the register file into callee-save and caller-save registers, the call cost may be reduced because the function call boundaries no longer exist. As long as the reduction of the call cost exceeds the increase of the spill cost, register allocation benefits by inlining.

Figure 5.18 sketches an example in which register allocation benefits by inlining. In Figure 5.18 (a), function foo contains a loop body that is executed many times and has a call to a leaf function bar. The shaded regions are the most frequently executed paths of the two functions. Hence, there is high cost to save/restore a callee-save register at the entry/exit of bar; there is high cost to save/restore a caller-save register around the call site bar(). That is, assigning a callee-save register to a live range in bar needs to pay high callee-save cost; assigning a caller-save register to a live range containing the call bar() needs to pay high caller-save cost. After function bar is inlined, the necessity of saving/restoring registers at bar's function boundary disappears (shown in Figure 5.18(b)). foo is now a leaf function (no caller-save cost) and the only function boundary for the code. Therefore, assigning callee-save registers to the live ranges that were in bar pay the callee-save cost at foo's function boundary which is less frequently executed than bar's; assigning caller-save registers to the live ranges that contained the call bar() pays no caller-save cost.

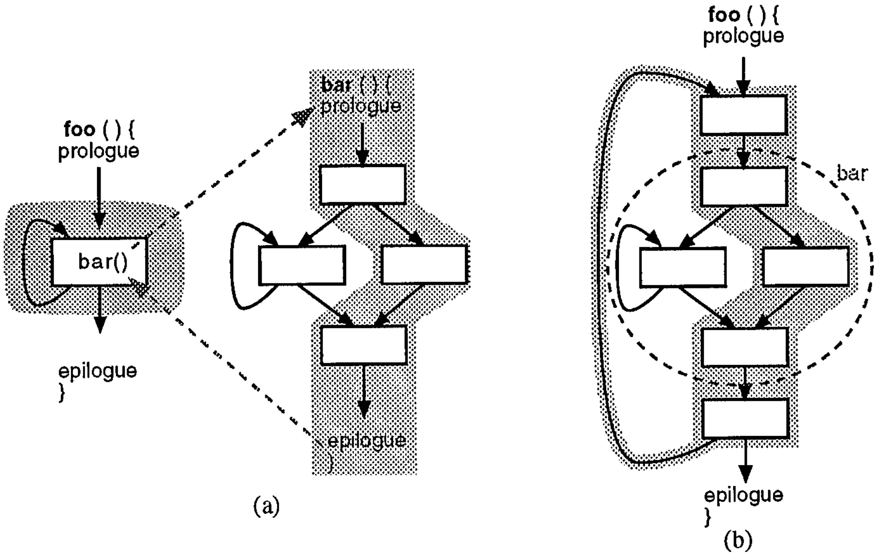


Figure 5.18: Function inlining.

- *compilation*: As functions are inlined, the code size grows. Consequently, the complexity of the compilation time and memory requirement increase as well [33].

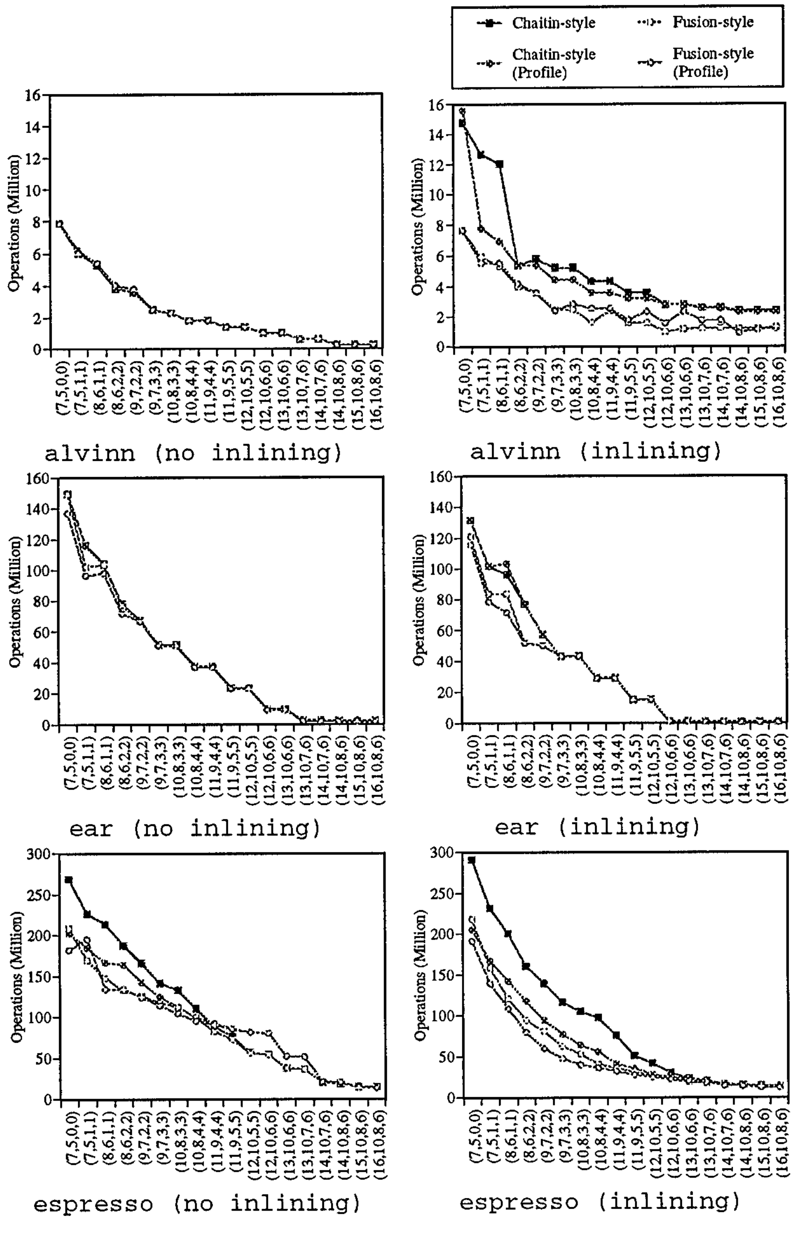
The compiler has to control the complexity so that the compilation time and memory requirement are acceptable and practical.

In this Section, I evaluate the influence of inlining on register allocation. Only data of alvinn, ear, espresso, compress, and sc are presented because the cmcc compiler, presently, does not have the infrastructure for doing function inlining. It would be very tedious and inefficient to analyze and inline functions manually. espresso has one compilation flag "-DNO_INLINE". By default, some frequently executed functions are inlined. To increase register pressure, some functions of alvinn, ear, compress, and sc are inlined manually. All functions (6 functions) of alvinn are inlined into the main function. All performance data of alvinn, ear, and espresso in this thesis are the inlined versions unless it is explicitly stated that the non-inlined versions are used.

Register overhead with and without inlining are given in Figure 5.19 and Figure 5.20. For easy comparison, the register overhead figures of the inlined versions of alvinn, ear, and espresso, and the non-inlined versions of compress and sc are taken from Section 5.4. There are several observable aspects. The curves of Chaitin-style and fusion-style without inlining either merge quickly (espresso) or are tightenly together (alvinn, ear, compress, and sc). Second, ear and espresso with inlining produce fewer overhead operations than without inlining for both Chaitin-style and fusion-style coloring because the reduction of the call cost exceeds the increase of the spill cost. The overhead operations of alvinn, compress, and sc with inlining for both Chaitin-style and fusion-style coloring increase because the increment of the spill cost is bigger than the decrement of the call cost. Third, fusion-style not only benefits from the reduction of call cost, but also handles big compilation scopes well and splits live ranges at low frequently executed paths to reduce spill cost (live-range splitting is very effective in the presence of inlining). For these reasons, the gap between the curves of Chaitin-style and fusion-style register allocation enlarges. Examples are alvinn, espresso, and compress.

5.8 Execution time and compilation time

There are two other dimensions of interest to compiler designers: execution time related to register overhead and compilation time. Table 5.1 shows the speedup due to fusion-style



alvinn (no inlining)

alvinn (inlining)

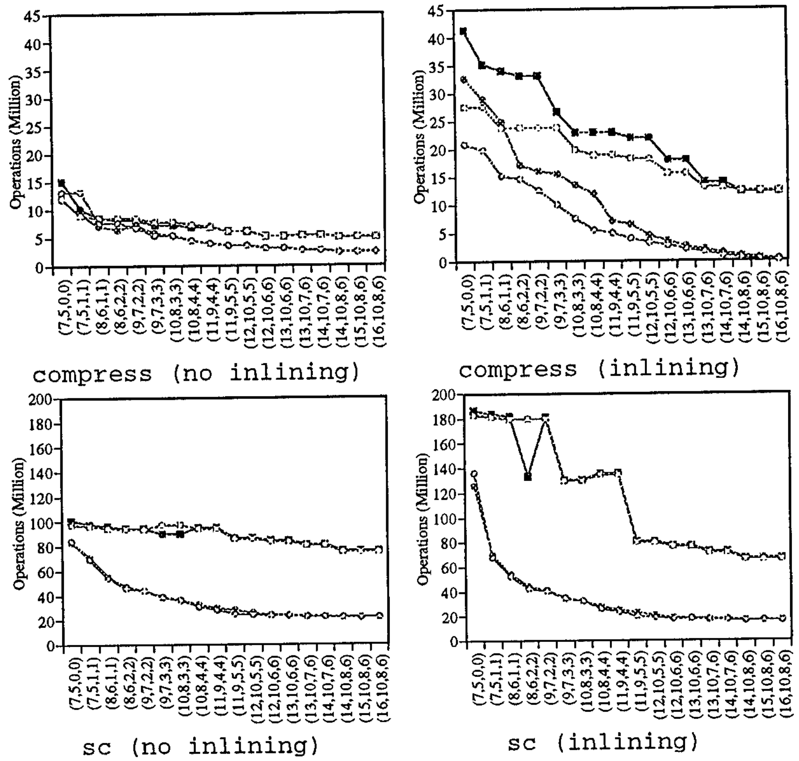
ear (no inlining)

ear (inlining)

espresso (no inlining)

espresso (inlining)

Figure 5.19: Function inlining vs. Without inlining.



compress (no inlining)

compress (inlining)

sc (no inlining)

sc (inlining)

Figure 5.20: Function inlining vs. Without inlining (cont.).

program	`All registers`	`14 int, 14 float`
espresso	0.0	7.5
nasa7	2.4	5.6
tomcatv	8.4	7.0

Table 5.1: Speed-up of execution time over Chaitin-style coloring (percentage).

register allocation (compared to a Chaitin-style register allocator with various enhancements). These data are for the MIPS architecture; the execution time has been measured on a DECstation 5000. The table shows that fusion-style coloring can speed-up execution time by up to 8.4 % over Chaitin-style coloring.

Table 5.2 shows the cost of fusion-style register allocation; this table lists the contribution of register allocation to overall compilation time (seconds) for the `cmcc` compiler. `cmcc` is a research compiler and not fine-tuned for compilation speed; overall `cmcc` takes about twice as long as the native compiler. The data are collected on a DEC AlphaStation (333MHz). We compile the SPEC92 programs and measure the compilation time for Chaitin-style and fusion-style register allocation, using loops as well as single basic blocks as base regions. We also stress register allocation by performing the unrolling transformation (unroll 3 times) to enlarge the code size for some of the programs.

The complexity of simplification is $\mathcal{O}(\mathcal{V}+\mathcal{E})$ where $\mathcal{V}$ is the number of live ranges and $\mathcal{E}$ is the number of interference edges. After fusing graphs along an edge, fusion-based register allocation performs simplification to maintain the simplifiability invariant. Hence, at first sight, we expect to see that the fusion-based approach has a big increase of compilation time because the complexity of maintaining the simplifiability constraint is $\mathcal{O}(\mathcal{F}*(\mathcal{V}+\mathcal{E}))$ where $\mathcal{F}$ is the number of fusion operations. From the table, however, we see that—except for `gcc`—register allocation time increases only by a small constant factor (< 2), even when the base regions are basic blocks and the compiler applies a fusion operation on every control-flow edge. For gcc (c-parse.tab.c), the size of the control-flow graph of the `yylex` routine is tripled by loop unrolling (unroll 3 times). The size grows from (415 blocks, 660 edges) to (1426 blocks, 2246 edges). Even in such a big control-flow graph, the cost of the fusion operations increases only about by a factor of 3 if the base regions are loops. To put this data into perspective, the complete code generator including register allocation is responsible for less than 20% of the total compilation time. So although fusion-style

		Region	
program	Chaitin	LOOP	BB
alvinn inline	1.3	1.6	1.6
alvinn inline unroll	4.8	5.2	5.7
compress	1.7	2.0	4.4
compress unroll	3.9	5.2	11.9
ear	4.2	5.5	8.2
ear unroll	15.7	26.6	49.5
eqntott	4.7	6.1	10.7
eqntott unroll	18.3	29.3	74.4
espresso	29.3	39.6	71.6
espresso unroll	104.0	175.6	417.2
li	5.0	5.9	9.4
li unroll	11.7	16.5	32.8
sc	15.1	25.5	62.8
sc unroll	33.2	71.2	160.1
gcc	148.7	211.7	669.7
gcc (c-parse.tab.c)	7.0	14.7	61.2
gcc (c-parse.tab.c) unroll	22.1	55.0	166.7
doduc	51.4	78.1	128.8
fpppp	60.6	70.8	83.6
matrix300	1.1	1.7	2.5
matrix300 unroll	4.1	7.5	11.0
nasa7	7.4	10.9	13.8
nasa7 unroll	55.2	80.1	100.9
spice	118.2	311.0	392.4
tomcatv	3.4	3.7	3.7
tomcatv unroll	12.7	17.2	30.7

Table 5.2: Compilation time of register allocation (seconds).

allocation incurs a price in compilation time, the increase is acceptable.

There are two major factors that keep the compilation time spent on register allocation low: partial graphs and cliques. When graphs are fused along edge E that connects two regions, R_1 and R_2, the fusion operation needs to deal with only G_{R_1} and G_{R_2} and not all the graphs in the program. Clique summary nodes also help to speed up the compilation time spent on simplification because simplifying a clique removes all transparent live ranges in the clique from the graph. Consider that a clique C contains T transparent live ranges. Each transparent live range in C interferes with $T - 1$ transparent live ranges in C and

$neighbor(C)$. Therefore C represents T live ranges and $\frac{T*(T-1))}{2} + T* \mid neighbor(C) \mid$, where $\mid neighbor(C) \mid$ is the number of neighbors of C. Removing C and $\varphi(C)$ from the interference graph is equivalent to removing T live ranges and $\frac{T*(T-1))}{2} + T* \mid neighbor(C) \mid$ interference edges. These two reasons keep the price of compilation time within a reasonable and acceptable range.

Fusion-based coloring provides a nice framework to tackle the compilation time problem if the compilation time ever becomes unacceptable or is a big concern to the compiler. The fusion operation takes longer and longer as more and more graphs have been fused because the fusion operation gradually deals with bigger graphs, which cover larger regions, and makes all delayed spilling decisions (live ranges are removed from cliques). Graph fusion is based on the edge ordering, so fusing graphs on the edges that are *less* important (less frequently executed) is *more* expensive in term of compilation time, yet has *less* influence on the overall register allocation overhead. To curtail the time spent on graph fusion, we can set a threshold to control the number of control-flow edges along which graphs are fused. The threshold can be based on the time that has already been spent on register allocation or the number of the total control-flow edges that connect regions. As soon as register allocation passes the threshold, the graphs of the regions that are not yet fused with any other graphs are all lumped into one big region $R_{remaining}$ (splitting live ranges is not allowed within the region). Building the graph $G_{R_{remaining}}$ is as fast as building the graph for Chaitin-style coloring. We then apply the fusion operation to the edges that connect $R_{remaining}$ and other regions.

The example in Figure 5.21 is a code fragment of `compress`. The shaded region is the most frequent path. We can apply graph fusion to the region. The fusion-style coloring is able to further divide the region into smaller sub-regions so that finer granularity live-range splitting is thereby possible within the region. The remaining blocks belong to one region, $R_{remaining}$. The register allocator treats $R_{remaining}$ as a typical region. The interference graph, $G_{R_{remaining}}$, is constructed, the graph is simplified so as to make sure that the simplifiability invariant is satisfied. The register allocator can either apply the fusion operation on edges, E_1, E_2, E_3, E_4, E_5, and E_6, to reconcile the two regions, or does nothing and let the color-assignment phase eliminate shuffle code using biased coloring.

Tables 5.3 and 5.4 show experimental results of the compilation time versus register

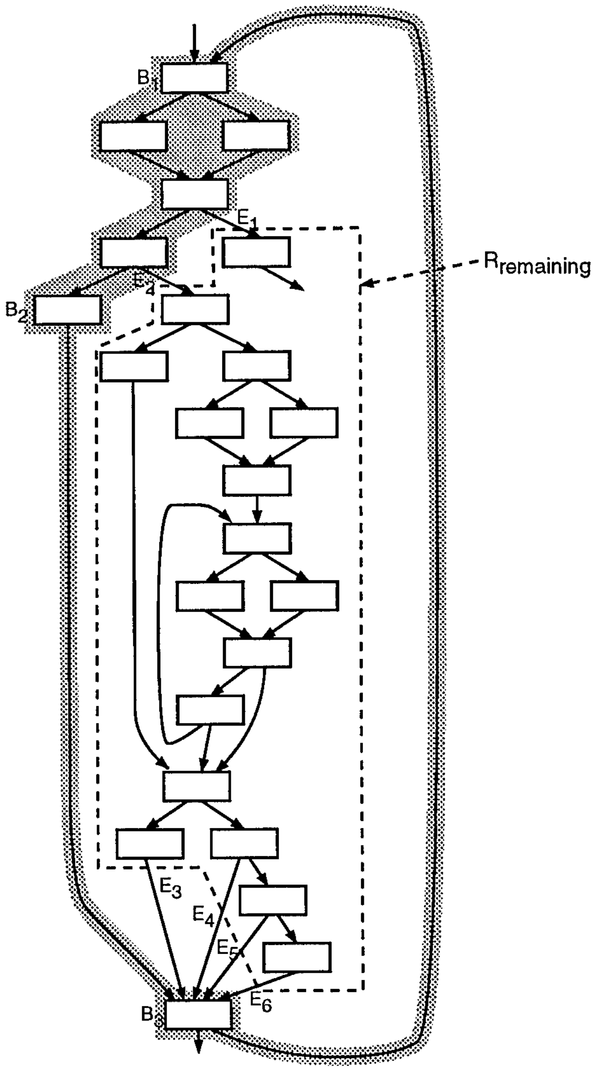


Figure 5.21: Reducing compilation time.

Fusion%	Time		Overhead	
	$(\frac{Fusion}{Chaitin} - 1) * 100$		$(\frac{Chaitin}{Fusion} - 1) * 100$	
	(14i,14f)	(11i,8f)	(14i,14f)	(11i,8f)
100	128.0	120.6	62.6	59.5
90	122.0	114.9	58.4	16.6
80	116.3	112.0	44.2	14.2
70	106.6	100.9	31.9	4.7
60	93.1	87.4	27.3	7.8
50	78.9	72.2	6.3	-4.1
40	66.0	60.7	7.4	-5.8
30	51.2	47.9	6.9	3.8
20	36.4	33.8	12.0	7.7
10	21.4	20.3	2.8	7.8

Table 5.3: Percentage increase of compilation time and improvement of overhead operations (espresso).

Fusion%	Time		Overhead	
	$(\frac{Fusion}{Chaitin} - 1) * 100$		$(\frac{Chaitin}{Fusion} - 1) * 100$	
	(14i,14f)	(11i,8f)	(14i,14f)	(11i,8f)
100	128.0	120.6	62.6	59.5
90	126.8	118.0	62.6	56.6
80	117.2	110.6	59.2	42.5
70	107.5	100.9	45.1	36.0
60	97.0	91.7	31.5	25.4
50	85.2	77.7	26.9	24.3
40	72.0	65.6	24.8	18.4
30	57.2	51.3	16.6	14.9
20	41.9	38.7	16.6	12.7
10	25.6	23.8	11.6	13.0

Table 5.4: Percentage increase of compilation time and improvement of overhead operations (espresso).

overhead operations for `espresso`. For the experiment, fusion-style coloring uses blocks as the base regions and profile information to guide register allocation. Data for two register combinations, 14 integer and 14 floating-point, and 8 integer and 11 floating-point registers, are collected. The left-most column is the percentage of total control-flow edges along which fusion-style coloring applies fusion operations. The "Time" columns indicate the percentage increase of compilation time of fusion-style coloring relative to Chatin-style coloring. The "Overhead" columns are the percentage reduction of overhead operation over Chaitin-style coloring when we use fusion-style coloring.

Table 5.3 is the result of applying fusion operations on the first k percentage of total control-flow edges. During the region formation phase in Figure 4.1, the register allocator collects blocks that are connected by the last $100 - k$ percentage of control-flow edges in the edge ordering into one region R. That is, those edges are in the region R. When looking at the overhead columns, we can observe one noticeable aspect that for a cutoff point of 50% in the (14i,14f) case, the improvement due to fusion drops to a few percentage over Chaitin-style coloring. There is a big overhead improvement drop for the (11i,8f) case even though we apply fusion operations on 90% of edges. We also see cases where fusion-style coloring produces more overhead operations than Chaitin-style coloring. The reason is that the region R contains frequently executed blocks. Consider the example of Figure 5.21. If E_6 is R, then R contains B_3. The register pressure of infrequently parts of R, therefore, may cause shuffle code on frequently executed edges, $<B_2,B_3>$ and $<B_3,B_1>$. Furthermore, collecting infrequently executed edges into R deprives the opportunities of placing shuffle code on those edges. Here we see the case that a bad choice of regions impacts the quality of register allocation badly. Hence, we take another approach that collects the blocks connected by the first k percentage of edges into one region. In other words, the region is frequently executed. The result is shown in Table 5.4. This table indicates that the approach is able to maintain good overhead improvement while reducing compilation time.

5.9 Summary

In this chapter, I have discussed four aspects of register allocation: the size of base region, profile information, sensitivity to register pressure, and sensitivity to unrolling and inlining code transformations. The results confirm that fusion-style register allocation keeps the overhead operations under control. Even in the presence of scope expanding transformations, loop unrolling and function inlining, the number of overhead operations generated by the fusion-based approach either grows slowly or is reduced by splitting live ranges at infrequently executed edges. These results indicate that register allocation is no longer an obstacle to region-based compilation. Compilers that use some form of region for other optimization phases now have a path to reconcile register allocation with region-based optimization.

Furthermore, our results indicate that profile information improves the performance of the register allocator. As gathering profile information becomes more common in optimizing compilers, fusion-style register allocation becomes increasingly attractive, since fusion-style allocation is well-prepared to directly take advantage of profile information. As the implementations of high-performance processors include more and more hardware support for dynamic instruction reordering, optimizers and schedulers in the compiler have to analyze larger and larger regions of a program. Region-based compilation provides an attractive framework to organize these compilations. With fusion-style register allocation, the compiler can now include register allocation in a region-based framework.

Chapter 6

Call-Cost Directed Register Allocation

The register allocation cost is the sum of three components: spill, call, and shuffle cost. The register allocation cost must include the call cost. If the register allocator focuses on the spill cost alone, the register allocator may have an overly optimistic image of the task. The call cost, however, is influenced by the compiler's calling convention. Many compilers divide the registers into two sets, *callee-save* and *caller-save* registers, respectively.

Choosing the right kind of register for live ranges plays a major role in eliminating the register-allocation overhead when the compiled function is frequently executed or function calls are on the most frequently executed paths. Picking the wrong kind of register for a live range incurs a high penalty that may dominate the total overhead of register allocation.

In this chapter, I present three improvements, storage-class analysis, benefit-driven simplification, and preference decision that are effective in selecting the right kind of register for a live range. Also I describe one enhancement to fusion-style coloring that takes call and shuffle cost into account to make splitting decisions. In Section 6.1, I present an allocator based on Chaitin-style coloring with a simple cost model. This register allocator is used as the base for performance comparison. In Section 6.2, I describe the first enhancement, storage-class analysis, that decides the storage class for live ranges. In Section 6.3, I present benefit-driven simplification that determines the color ordering in a novel manner. In Section 6.4, I present the third enhancement, preference decision, that determines the preferable kind of register for live ranges. In Section 6.5, I describe the fourth enhancement, call-cost splitting, which splits live ranges to reduce call cost. In Section 6.6, I evaluate the effectiveness of the first three enhancements. In Section 6.7, I evaluate the influence of the call-cost splitting enhancement.

Figure 6.1 sketches the framework of the register allocator. The shaded regions are the places where the four improvements are implemented. Storage-class analysis and preference decision are implemented in the color-assignment phase; benefit-driven simplification is done in the color-ordering phase; call-cost splitting is implemented in the graph-construction phase. The improvements by no means can only be used by our register allocator. Indeed, because various register allocation approaches have been implemented in this same framework, these approaches may all benefit from the improvements. For instance, fusion-style, Chaitin-style, and optimistic coloring all profit by storage-class analysis, benefit-driven simplification, and preference decision. Priority-based coloring, however, cannot use benefit-driven simplification because it does not use simplification to determine the coloring order.

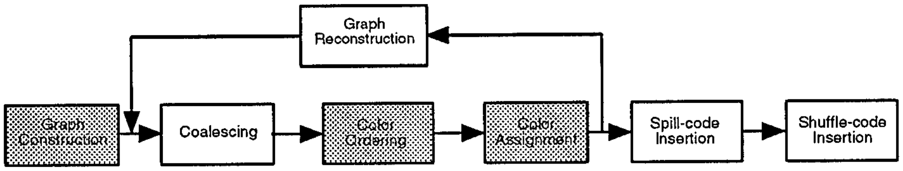


Figure 6.1: Call-cost directed register allocation.

6.1 Chaitin-style base model

Chaitin-style coloring simplifies an interference graph to determine the coloring order and spills live ranges based on spill cost (either static estimated or dynamic spill cost) when simplification blocks. A live range that contains calls prefers using a callee-save register so the color-assignment phase attempts to find a callee-save register. If there is no callee-save register left (taken by all the live range's neighbors), a legal caller-save register is assigned to the live range. A live range containing no calls prefers using a caller-save register so the color-assignment phase attempts to find a caller-save register. If there is no caller-save register left, a legal callee-save register is assigned to the live range.

This simple model is used as the base for comparison. It allows us to access the significance of each improvement in terms of reducing overall overhead operations. Although the base model is simple, it is not unreasonable. In Chapter 7, we compare this base model to a more sophisticated approach to handle call cost which is implemented in various compilers. However, for a significant number experiments, the simple base model outperforms the other approach.

6.1.1 Limitations of the base model

Figure 6.2 depicts the register allocation cost for two programs from the SPEC92 suite, `eqntott` and `ear`, for various combinations of caller-save and callee-save registers. Again, the MIPS architecture has separate register banks for integer and floating-point values. The notation (R_i,R_f,E_i,E_f) of the x-axis of Figure 6.2 shows the combination of caller-save and callee-save registers for the two register banks. R_i and R_f are the number of caller-save registers for integer and floating-point values, respectively. E_i and E_f are the number of callee-save registers for integer and floating-point values.

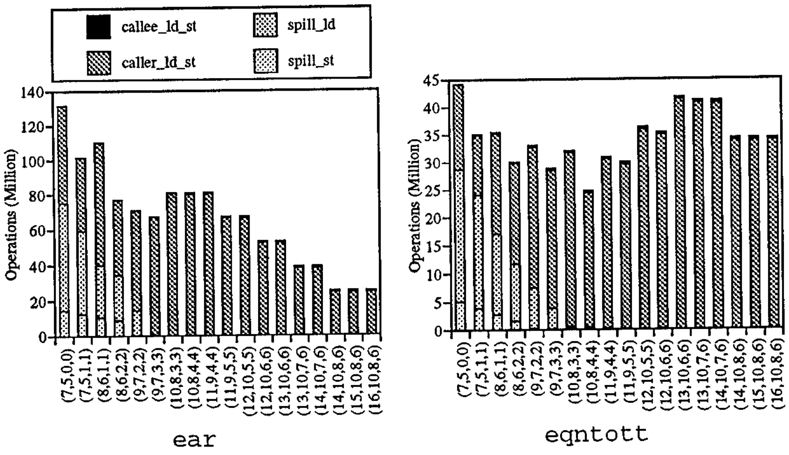


Figure 6.2: Register allocation cost.

From Figure 6.2, we see that giving the register allocator more registers can reduce the *spill* cost dramatically. The spill cost is cut down to 0.8 % with (10,8,4,4) registers for `eqntott` and to 0.5 % with (9,7,3,3) registers for `ear`. However, simply focusing on the spill cost is misleading; the figure shows that spill cost is no longer an issue once a certain number of registers is available. As the figure illustrates, we must take the call cost (caller-save and callee-save cost) into consideration. There are two noteworthy aspects: first, the contribution of the call cost to total register allocation cost is significant. Second, giving the register allocator more registers may actually *worsen* the register allocation cost because some live ranges may now reside in the registers whose call overheads introduce more memory accesses than the spill cost of the live ranges. Cohn and Lowney [18] measured the dynamic call cost for various Windows/NT applications and SPEC95 programs. For more than half of the programs, the call cost is 10% - 25% of their execution time. If the register allocation does not take the call cost into consideration, the call overhead introduced by the register allocation may end up on the *hot* part of the program [18].

6.2 Storage-class analysis

A live range can reside in one of these storage classes (assuming that registers are divided into caller-save and callee-save registers):

- memory,
- a caller-save register, or
- a callee-save register.

Each storage class has an associated cost. Storage-class analysis decides where live ranges reside, based on two functions, $benefit\_caller$ and $benefit\_callee$. The two functions model the benefits provided by these two storage classes over the default location (memory). These functions are defined for each live range lr. For each lr, $benefit\_caller(lr)$ is defined as the weighted reference counts of the spill code minus the weighted caller-save cost. $benefit\_callee(lr)$ is defined as the weighted reference counts of the spill code minus the weighted callee-save cost. The weighted reference counts and call cost are derived using either estimated execution frequency (static) or profiling information of some prior execution (dynamic). That is, these two functions indicate the estimated number of load/store operations that are eliminated if a caller-save (or callee-save) register is assigned to lr. The two functions are similar to the priority function of [16], except the benefits are not normalized by the size of the live ranges.

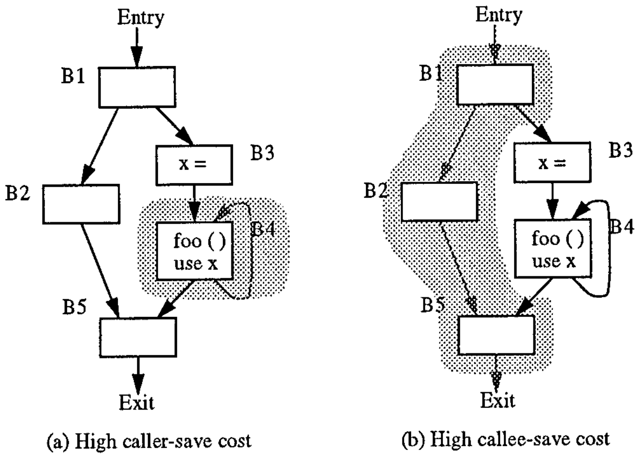
During the color-assignment phase, the selection of the kind of register to use is based on these two functions. If $benefit\_callee(lr) > benefit\_caller(lr)$, finding an available *callee-save* register for lr is attempted prior to finding an available caller-save register. If $benefit\_callee(lr) \leq benefit\_caller(lr)$, it is preferable to put lr into a *caller-save* register over using a callee-save register. The accuracy of the two benefit functions depends on the accuracy of the estimated execution frequency.

The simplification phase guarantees that the color-assignment phase finds registers for live ranges that are on the color stack (C). However, sometimes *not* using a register (i.e., spilling a live range) is better than using the wrong kind of register because the cost of using the register (callee-save or caller-save cost) could be greater than the spill cost of the live range. Figure 6.3 illustrates the effect of assigning the wrong kind of register to live range x. The shaded regions indicate the most frequently executed paths. Using a caller-save register for live range x in Figure 6.3 (a) incurs a high cost; keeping x in a caller-save register can only make the result of register allocation worse because x must be saved and restored once for each use of x in the loop. The two memory accesses required to save/restore the caller-save register are more than the single restore operation, which would be required if x had been spilled.

Likewise, in Figure 6.3 (b), keeping x in a callee-save register results in a higher register-allocation overhead than spilling because saving and restoring the value of the callee-save register happens on the most frequently executed path.

Our algorithm models register assignment as a *possible* improvement. That is, we start out with the assumption that a live range resides in memory (is spilled) and then try to determine if allocating a register reduces the overhead operations. This model allows us to spill live ranges to reduce the overall number of load/store operations even though there are available registers. The two benefit functions give us a good indication if a live range lr is a worthwhile candidate for register residence. If lr gets a caller-save register and $benefit\_caller(lr) < 0$, then spilling lr reduces the load/store counts by $\mid benefit\_caller(lr) \mid$.

While assigning registers to live ranges, the color-assignment phase makes spilling decisions right away for the live ranges that are supposed to get caller-save registers (i.e., spill when $benefit\_caller(lr) < 0$). There are two ways to make spilling decisions for



(a) High caller-save cost

(b) High callee-save cost

Figure 6.3: Storage-class analysis.

live ranges that receive callee-save registers. The first approach models callee-save costs as occuring only once for each callee-save register. For the *first* live range lr that uses a given callee-save register, making the spilling decision then is similar to making the decision to use a caller-save register. That is, if $benefit\_callee(lr) < 0$, then lr is spilled to memory. For a live range that is *not* the first user of a callee-save register, the live range can use the callee-save register for free (since there in no need to spill the register). The second approach views callee-save costs as shared by all live ranges that share a callee-save register. That is, the first live range to use the register does not pay all the costs; the cost is spread over all users. The spilling decisions for live ranges that (potentially) get callee-save registers are not finalized until the color-assignment phase finishes. At that time, it is known which live ranges may use a specific callee-save register. For a callee-save register r, and $\delta(r)$ the set of live ranges that share r, if $\sum_{lr \in \delta(r)} spill\_cost(lr) < callee\_cost(r)$, then all live ranges of $\delta(r)$ are spilled.

Our experimental data indicate that the second approach performs better than the first one for some SPEC92 programs, for others it makes no difference. To illustrate why this is the case, consider two live ranges with spill cost 4000 that share the same callee-save register, and the callee-save cost is 5000. The first approach does not assign any of the two live ranges to the callee-save register because of the high first-use cost. At the end, spilling the two live ranges introduces 8000 load/store operations (assume that they also have high caller-save cost, or all caller-save registers are taken by their neighbors). However, the second approach assigns the two live ranges to the register — a decision that saves 3000 load/store operations over spilling.

The flow of the color-assignment phase using storage-class analysis is depicted in Figure 6.4. This phase pops live ranges out of C and assigns a register to each of them. If a live range lr prefers a callee-save register ($benefit\_callee(lr) > benefit\_caller(lr)$), the register allocator first attempts to find a legal callee-save register. If a callee-save register is found, then the next step, choosing a caller-save register, does nothing because lr is already assigned a register that lr desires. A live range that prefers a caller-save register is also assigned a register in a similar fashion. For simplicity, $benefit\_callee(lr) < 0$ is the heuristic to spill a live range that receives a callee-save register. The condition $benefit \geq 0$ means $benefit\_callee(lr) \geq 0$ for callee-save registers , and $benefit\_caller(lr) \geq 0$ for caller-save registers. The spill step merely adds a live range with $benefit < 0$ to the spill pool (S).

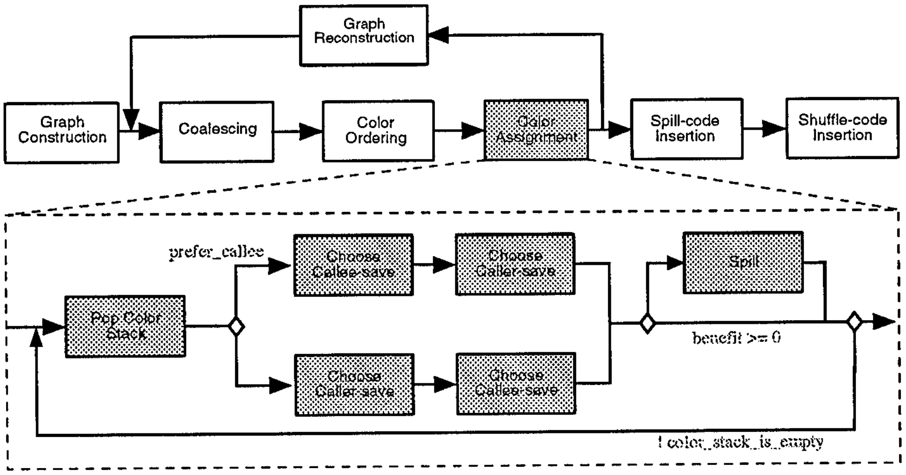


Figure 6.4: Structure of the color assignment using storage-class analysis.

6.3 Benefit-driven simplification

Simplification removes live ranges one by one from the interference graph and pushes them onto $\mathcal{C}$. The reverse ordering in which live ranges are pushed onto $\mathcal{C}$ is the ordering in which they are assigned colors. Simplification is a nice and easy way of packing live ranges into registers but there is no cost model, and therefore, decisions during simplification may turn out to handicap the color-assignment phase. If the target machine has only one kind of register, the order of removing already-unconstrained live ranges does not have any influence on the final register overhead, because using each register incurs the same cost. But for a machine with a mixture of registers of different costs, the position of a live range on $\mathcal{C}$ plays an important role in reducing the register overhead.

Each live range has two benefit functions, $benefit\_caller$ and $benefit\_callee$, which determine the preferred kind of register. Live ranges on top of $\mathcal{C}$ have a higher chance to obtain the preferred kind of register, because fewer registers are already taken by other live ranges. For example, Figure 6.5 (a) depicts an interference graph consisting of 3 live ranges. All prefer using a callee-save register. There are only two callee-save registers available. One live range must use a caller-save register. With $N = 3$, all three live ranges are unconstrained. A legal color stack is shown in Figure 6.5 (b) (obtained by removing lr_y, lr_z, and then lr_x). The top two live ranges, lr_x and lr_z, are assigned callee-save registers during the color-assignment phase and lr_y ends up in the caller-save register. This assignment then saves 3200 load/store operations over spilling. However, the best ordering during simplification is lr_z, lr_y, and then lr_x, so that lr_x and lr_y can obtain the two callee-save registers; this assignment actually saves 4100 load/store operations.

During simplification, if there is more than one unconstrained live range, then the live range that has the smallest priority is removed. Now we can view $\mathcal{C}$ as a priority-based color stack; the higher position a live range on $\mathcal{C}$, the more likely the live range has more freedom with regard to picking registers. Hence register allocation based on this color stack is now similar to priority-based coloring [16]. In other words, benefit-driven simplification unifies the priority-based approach with simplification-based register allocation.

The flow of simplification used by Chaitin-style, optimistic, and fusion-style coloring to determine the color ordering is depicted in Figure 6.6. Moreover, Figure 4.10 and Figure

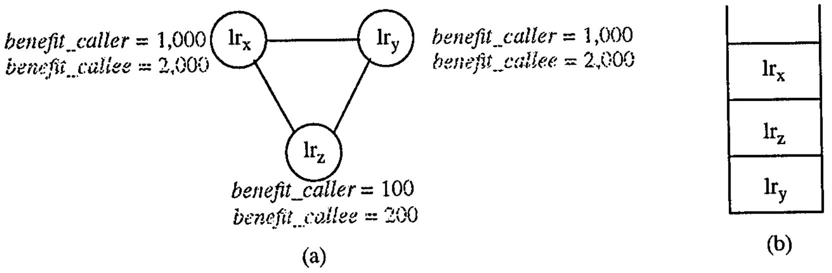


Figure 6.5: Effect of simplification order, with $N = 3$ registers (2 callee-save and 1 caller-save).

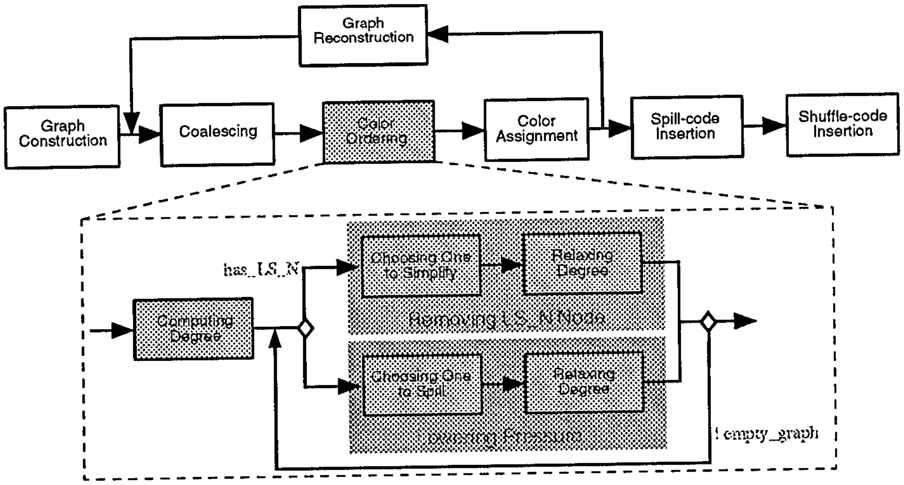


Figure 6.6: Structure of benefit-driven simplification.

6.6 demonstrate that the same simplification framework has been shared by various register allocation approaches. In the color-assignment phase, cliques no longer exist, and no live-range splitting is allowed because fusing graphs is already completed. Hence, the task of the "lowering pressure" step in the color-assignment phase is quite different for and much simpler than that in Figure 4.10. Here the lowering-pressure step simply chooses one live range based on a heuristic to spill and relaxes the degrees of the live range's neighbors. The removing-LS_N step in Figure 6.6 is composed of choosing one unconstrained live range with the *smallest* key to simplify and relaxing the degrees of the live range's neighbors. One implementation of keeping track of the unconstrained live range with the smallest key in LS_N is implementing LS_N using a *heap* structure [19]. In the "relaxing degree" step, as constrained live ranges become unconstrained, the live ranges are inserted into the LS_N heap based on a *key* function. $benefit\_caller$ and $benefit\_callee$ form the foundation of our experiments to investigate different strategies to order unconstrained live ranges during simplification. Specifically, we investigate two keys to order live ranges:

- $\max(benefit\_caller, benefit\_callee)$
- or

$$\begin{cases} \mid benefit\_caller - benefit\_callee \mid & \text{if } benefit\_caller \geq 0 \text{ and} \\ & benefit\_callee \geq 0 \\ \max(benefit\_caller, benefit\_callee) & \text{otherwise} \end{cases}$$

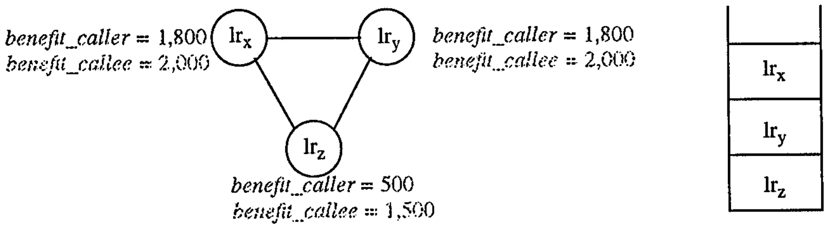
Priority-based coloring uses this strategy to prioritize so as to make sure that the live range with maximum savings has the highest priority to occupy a register. However, this strategy, when implemented as part of benefit-driven simplification, increases the register overhead for some SPEC92 programs, compared to register allocation without using benefit-driven simplification. This strategy is not suitable for Chaitin-style register allocation because simplification guarantees that all live ranges on $\mathcal{C}$ can find registers; in other words, making sure that live ranges with bigger savings own registers is *not* a concern for Chaitin-style coloring. What Chaitin-style coloring cares about is the incurred penalty of using the wrong kind of register — it cares more about the *delta* between the two benefit functions rather than the maximum of the two. Therefore the second strategy is used. The delta is the incurred penalty of assigning the wrong kind of register to a live range. If $benefit\_caller(lr)$ and $benefit\_callee(lr)$ are greater than or equal to zero, the key function is the difference of the two benefit functions. Otherwise, the maximum of the two benefit functions is used as the key function.

The example in Figure 6.7 illustrates how the key functions make a difference in register allocation overhead. The interference graph in Figure 6.7 (a) comprises three live ranges, lr_x, lr_y, and lr_z. Using the first definition, lr_x and lr_y have higher priorities (2000) than lr_z (1500). The order in which live ranges are removed from the graph is lr_z, lr_y, and lr_x. Because all three live ranges prefer a callee-save register, lr_x and lr_y are assigned the two callee-save registers. The total savings amounts to 4500 load/store operations. With the second definition, lr_z has a higher priority than lr_x and lr_y, because for lr_z the penalty of using the wrong kind of register is higher than for the other two. Thus lr_z ends up on top of C. The second definition yields a better allocation than the first one (and results in a total savings of 5300 load/store operations).

6.4 Preference decision

One shortcoming of Chaitin-style register allocation is that low-priority (small-savings) live ranges with high degree may take away the kind of register that high-priority (big-savings) live ranges crave for. Benefit-driven simplification tries to prevent this situation from happening. Nevertheless, there is still no guarantee that live ranges with high priority get the registers they prefer, because simplification is sensitive to the degree of live ranges (a live range can be removed from the interference graph only if the live range's degree is less than N). The color-assignment phase, however, examines only the two benefit functions, $benefit\_callee$ and $benefit\_caller$, to determine what kind of register a live range should get without caring about the needs of other live ranges. This restricted view of the color-assignment phase is the cause of unnecessary overhead operations.

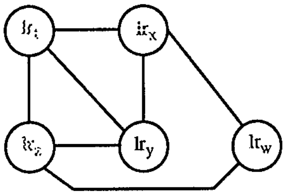
Therefore, prior to the color-assignment phase, we provide a separate phase that predetermines the preferable kind of register for *some* live ranges. The purpose of this phase is to minimize the caller-save overhead. This phase goes through each function call in order of weighted execution frequency and makes the preference decision for live ranges across the function call. For a function call, if the number (L) of the live ranges that contain the call and prefer callee-save registers is less than or equal to the number of available callee-save



(a)

(b)

Figure 6.7: Priority with $N = 3$ registers (2 callee-save and 1 caller-save).



(a)

	lr_t	lr_x	lr_y	lr_z	lr_w
benefit_caller	100	100	100	100	100
benefit_callee	200	400	200	200	4000

(b)

Figure 6.8: Preference decision example with $N = 3$ registers (1 callee-save and 2 caller-save).

registers (M), then there is no preference decision that needs to be made. If L is greater than M, then regardless of how registers are assigned, at least $L - M$ live ranges must use caller-save registers rather than callee-save registers.

We sort those live ranges using a key function. The least $L - M$ live ranges are annotated so that their preferable kind of register is caller-save regardless of the fact that $benefit\_callee > benefit\_caller$. After registers are assigned to those live ranges, just like for regular live ranges, register allocation uses $benefit\_callee$ and $benefit\_caller$ to analyze if they should reside in registers or memory. The key is defined as:

$$\begin{cases} caller\_cost & \text{if } benefit\_caller > 0 \\ spill\_cost & \text{otherwise.} \end{cases}$$

$caller\_cost(lr)$ is the overhead incurred for lr using caller-save registers which is equal to $spill\_cost - benefit\_caller$. For those live ranges with $benefit\_caller > 0$, $caller\_cost$ is used as the key ($caller\_cost$ is the overhead). For the live ranges whose $benefit\_caller \leq 0$, the key is $spill\_cost$ because storage-class analysis spills them if they do not get callee-save registers. In other words, $spill\_cost$ is the incurred penalty for not using a callee-save register.

Figure 6.8 (a) depicts an interference graph with five live ranges. The two benefit functions of each live range are listed in the table of Figure 6.8 (b). Callee-save registers are the precious resources in this example (all live ranges are competing for callee-save registers). The key values of live ranges which are used in benefit-driven simplification are lr_t (100), lr_y (100), lr_z (100), lr_x (300), and lr_w (3900). Only lr_w can be removed at first given $N = 3$, because the degrees of the other live ranges are greater than or equal to 3. Let the subsequent order in which live ranges are removed from the graph be lr_z, lr_t, lr_y, and

lr_x. The ordering in which live ranges are assigned colors is lr_x, lr_y, lr_t, lr_z, and lr_w. lr_x then takes away the callee-save register and leaves no callee-save register but the caller-save registers for lr_t, lr_y and lr_w to use (lr_z gets the same register as lr_x). This assignment saves 900 load/store operations (300 and 600 for using caller-save and callee-save registers, respectively). If live ranges lr_x and lr_w contain the same (high-frequency) function call, then lr_x is forced to reside in a caller-save register instead of the callee-save register. This color assignment then saves 4500 load/store operations (lr_x, lr_z and lr_t get the caller-save registers, and lr_w and lr_y reside in the callee-save register).

6.5 Splitting to reduce call cost

Fusing two interference graphs tends to grow live ranges aggressively to reduce the shuffle cost as long as the colorability invariant is maintained. Storage-class analysis, benefit-driven simplification, and preference decision are the techniques that adjust spill, caller-save, and callee-save costs. Caller-save cost may be reduced at the expense of spill cost. Growing a live range aggressively could potentially include more call sites into the live range. Consequently, the caller-save cost of the live range increases. High caller-save cost of a live range, lr, means two things when a caller-save register is assigned to lr. lr either is spilled or pays high caller-save cost.

One improvement to fusion-based coloring is to limit the growth of live ranges that cross function calls by suppressing coalescing. That is, the technique can be applied only to live-range splitting approaches. The improvement is called "call-cost splitting" that splits live ranges to reduce call cost. By suppressing the growth of a live range containing calls, 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.

The unioning-splitable step (Section 4.2.5) is the place where the call-cost splitting takes place. Prior to unioning two splitable live ranges, lr_1 and lr_2, along edge E, the call-cost splitting determines if suppressing combining lr_1 and lr_2 on edge E could likely reduce overhead operations. There is no guarantee that splitting on edge E is effective because the final result is unknown till the color-assignment phase.

A register allocation heuristic should take every aspect of the cost model into account, and avoid making any decision based on part of the cost model. Hence we discuss call-cost splitting from the viewpoints of shuffle, spill, and call costs.

- *shuffle and spill costs*: Splitting lr_1 and lr_2 on edge $E = \langle B_1, B_2 \rangle$ induces shuffle code on E. The shuffle cost on E should be cheap relative to either lr_1 or lr_2. In other words, one of the live ranges can afford the induced shuffle cost. The condition we check is $freq(E) > spill\_cost(lr_1)/\alpha \vee freq(E) > spill\_cost(lr_2)/\alpha$, where $freq(E)$ is the estimated execution frequency of E and α is a constant that is controlled by the user or the register allocator. "$\alpha = 100$" means that one live range's spill cost is at least 100 times more than the potentially incurred shuffle cost. A big α indicates that live ranges are allowed to grow larger than a small α—split less frequently. The bigger the number of registers, the more likely the color-assignment phase assigns the preferable kind of register to a live range. Therefore, call-cost splitting should happen less frequently (use a big α). If the number of registers is small, live ranges are competing for their preferred kind of registers. Thus, call-cost splitting should be applied more frequently (use a small α).

- *call cost*: The previous condition decides if E is a good splitting point. We want to determine if coalescing lr_1 and lr_2 into lr_{12} may introduce more overhead operations than suppressing lr_1 and lr_2 provided that lr_{12} ends up in a caller-save register. Here we look for the case that one live range (say lr_1) has $benefit\_caller \geq 0$ and the other live range (say lr_2) has $benefit\_caller < 0$, because we get $benefit\_caller(lr_{12}) < benefit\_caller(lr_1)$ if coalescing of the two live ranges is not suppressed. In addition, we check the condition $caller\_cost(lr_2) > 2 * freq(E)$. Splitting lr_1 and lr_2 prevents $caller\_cost(lr_2)$ from being added to $caller\_cost(lr_1)$ at the expense of shuffle cost. Once coalescing of lr_1 and lr_2 is suppressed, not only is shuffle code generated at the entry to lr_2, but also is shuffle code *potentially* introduced at the exit of lr_2 because lr_2 is a stand alone live range. We use $2 * freq(E)$ to estimate the possible incurred shuffle code operations.

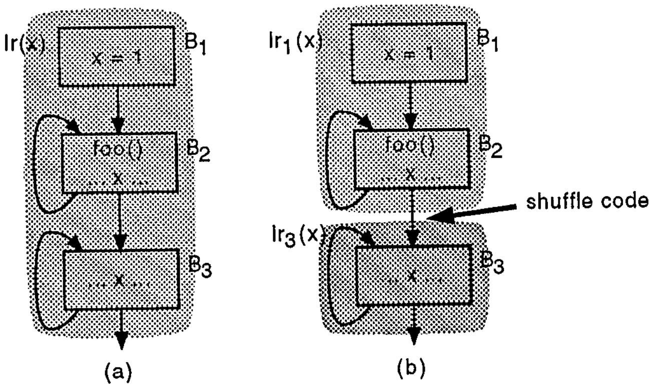


Figure 6.9: Eliminating caller-save cost by splitting.

Figure 6.9 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 6.9 (a). If the 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 6.9 (b). Now $lr_1(x)$ can reside in either memory or a callee-save register (if the callee-save cost is low).

6.6 Evaluation of SC, BS, and PR enhancements

In this section we report on an empirical evaluation. We measure the influence of different combinations of the improving techniques for various SPEC92 programs: (alvinn, compress, ear, eqntott, espresso, gcc, li, sc, doduc, fpppp, matrix300, nasa7, spice, and tomcatv). We pay special attention to the issue of deciding between a caller-save and a callee-save register for a live range. The y-axis of the figures in this section shows the register overhead produced by a base Chaitin-style coloring (with the simple cost model) divided by the overhead of an improved-version Chaitin-style register allocator using various combinations of the enhancements (storage-class analysis, benefit-driven simplification, and preference decision). The bigger this number, the less overhead is there in the improved version. SC, BS, and PR stand for Storage-Class analysis, Benefit-driven Simplification, and PReference, respectively. We can classify the SPEC92 programs (compiled using profile information) into 4 classes.

- Each optimization contributes a significant fraction of improvement. Examples are nasa7 and ear shown in Figure 6.10. For nasa7 and ear, there is not much room for optimizations to reduce overhead operations if only a smaller number of registers is available. With more registers, the optimizations have more freedom to choose the right kinds of storage class (caller-save registers, callee-save registers, or memory) for live ranges.

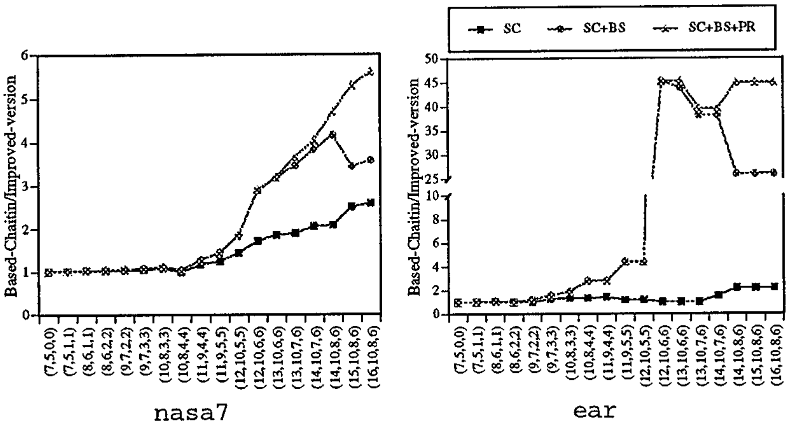


Figure 6.10: SC, BS and PR are effective.

- Only storage-class analysis has a dramatic improvement; spilling live ranges that have the wrong kind of registers helps reducing the overall overhead operations. Examples are li, sc, and matrix300. See Figure 6.11.

- Pre-determining the preferred kind of registers for live ranges does not affect the overall number of overhead operations. Examples of this class are compress, eqntott , espresso, gcc, fpppp, doduc, and spice (shown in Figure 6.12 and Figure 6.13).

- For programs that have low callee-save and caller-save costs, such as tomcatv, which consists of only one big function and no calls, none of the three techniques

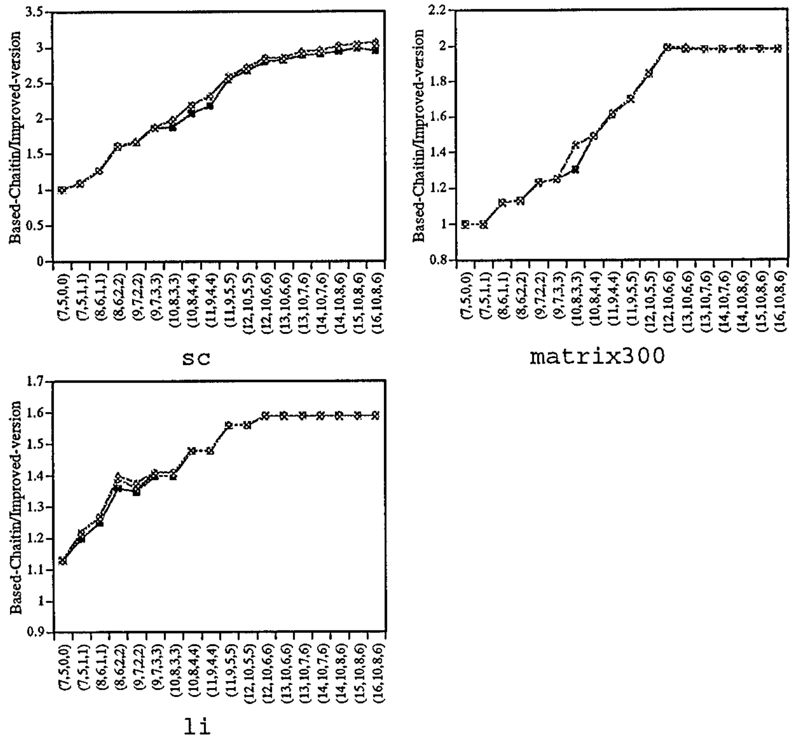
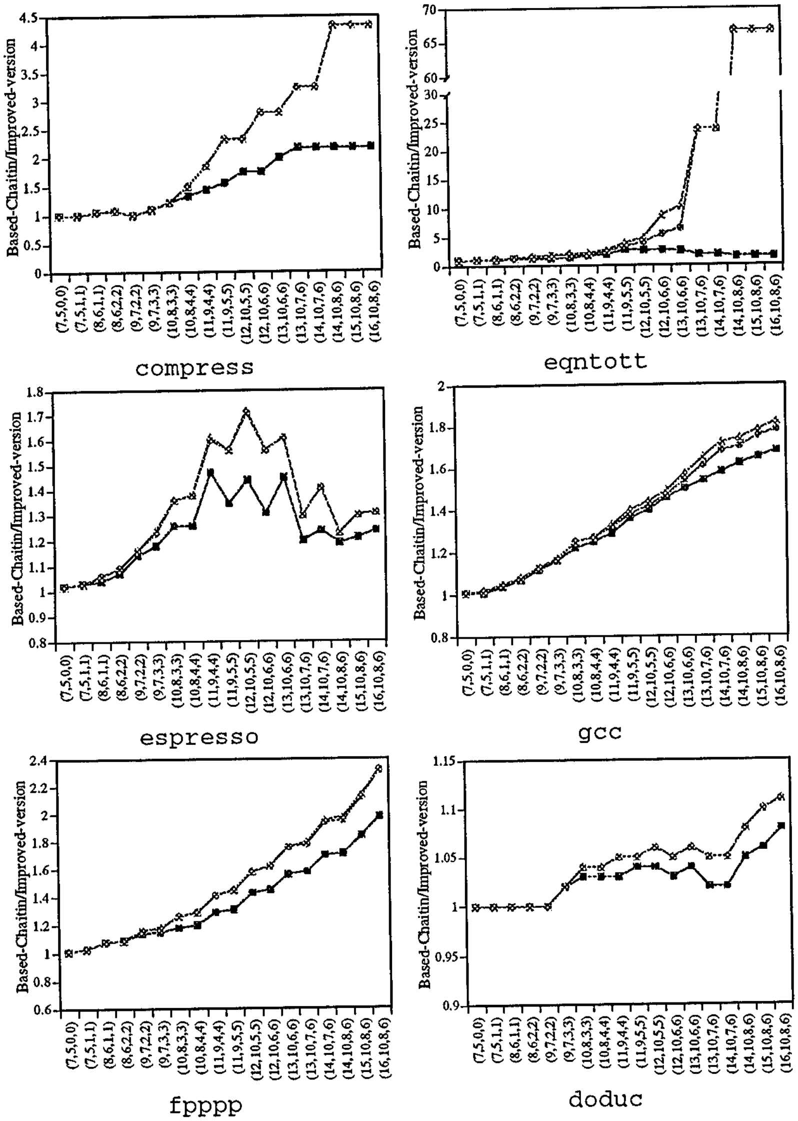


Figure 6.11: Only SC is effective.



compress

eqntott

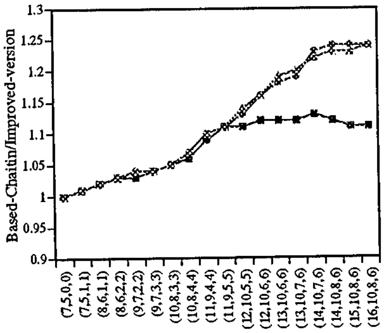
espresso

gcc

fpppp

doduc

Figure 6.12: SC and BS are effective.



`spice`

Figure 6.13: SC and BS are effective (cont.).

makes any difference. All the ratios of Base-Chaitin/Improved-version for the three improvements are 1.0 (Figure 6.14).

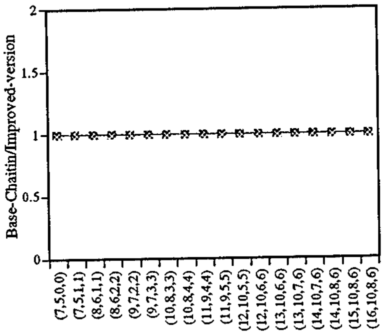


Figure 6.14: No improvement for `tomcatv`.

Figure 6.15 shows the effect of these improvements on the programs from Figure 6.2. This figure depicts the register overhead for `ear` and `eqntott` with the three improvements. For the `ear` and `eqntott` programs, the improved Chaitin-style coloring reduces the register overhead by a factor of 45 and 66 (i.e, with the base register allocator, there are 45 (66) times as many overhead operations as required by the improved Chaitin-style coloring).

I also evaluate how storage-class analysis, benefit-driven simplification, and preference decision reduce register overhead operations using static information. Again static informa-

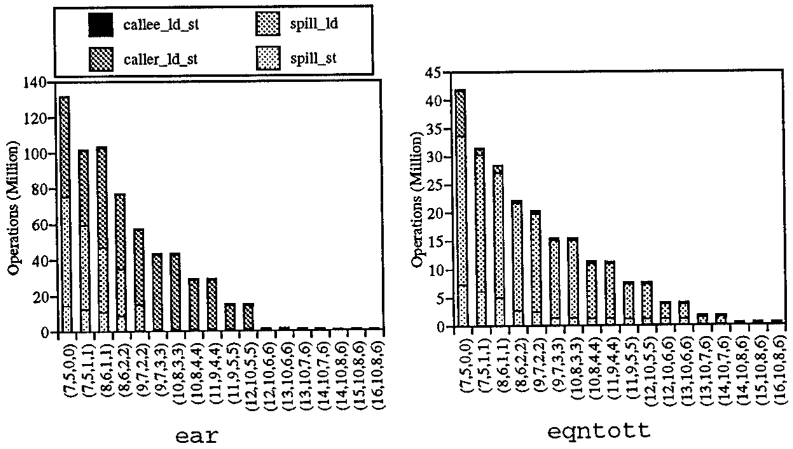


Figure 6.15: Register overhead for improved register allocation.

tion uses the loop hierarchy (nesting levels) as an approximation of the execution frequency of a basic block. The results are shown in Figures 6.16, 6.17, and 6.18. Figures 6.17 and 6.18 show those programs that exhibit the same trend of improvement as using dynamic information. The three improvements, rather than ameliorate, deteriorate the quality of register allocation (generate more operations than the base Chaitin) for the programs in Figure 6.16.

The three improvements are based on the analysis of spill cost, $benefit\_caller$ and $benefit\_callee$ which solely depend on the accuracy of the estimated execution frequency. Using the loop hierarchy to estimate the execution frequency treats every block within the same loop as having the same execution frequency. In practice, this assumption is not always true. When the static information does not capture the actual execution pattern of a program, the techniques may worsen the register-allocation result. For example, the live range lr_x in Figure 6.19 crosses the function call of `foo()`, which is in the loop ($B4$). Based on the static information, the loop is considered to be more frequently executed than the rest of the blocks. $benefit\_callee(lr_x)$, therefore, is greater than $benefit\_caller(lr_x)$. The color-assignment phase, therefore, attempts to put lr_x into a callee-save register at first. There are two outcomes for lr_x of the color-assignment phase: (1) lr_x obtains a callee-save register or, (2) lr_x gets a caller-save register and then is spilled to memory by the storage-class analysis due to the high caller-save cost. If, in fact, the shaded region depicts the most

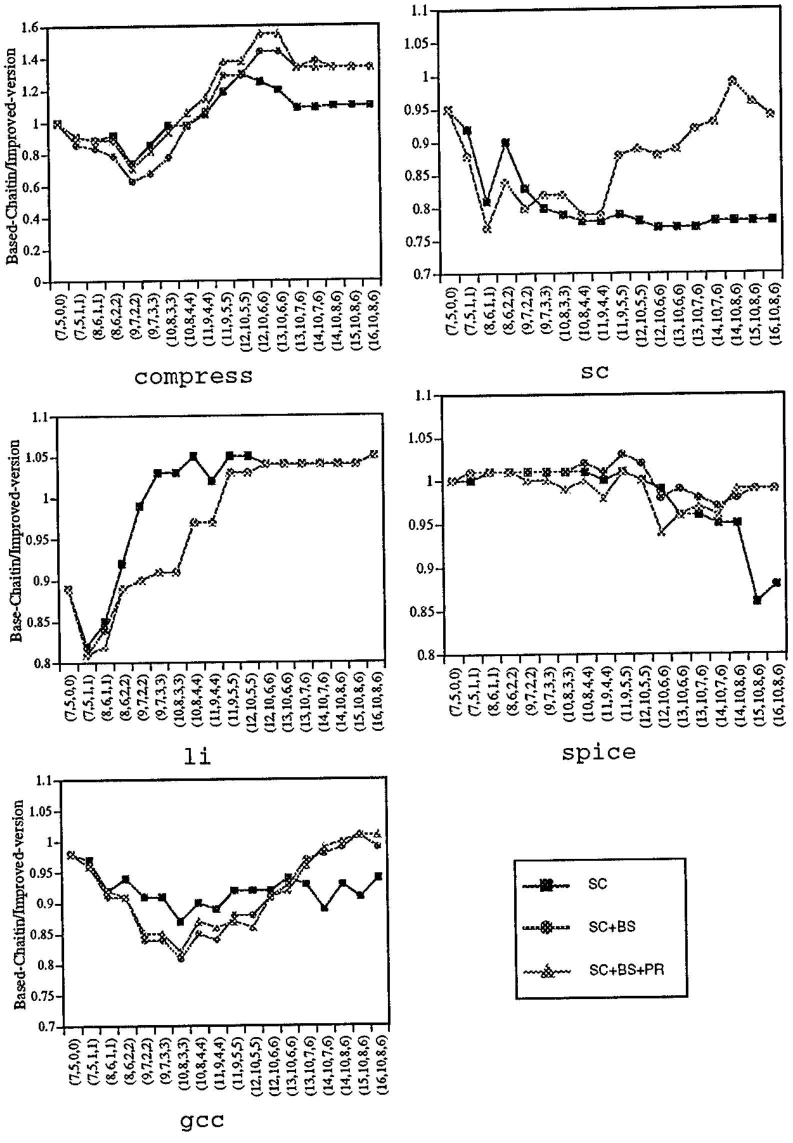
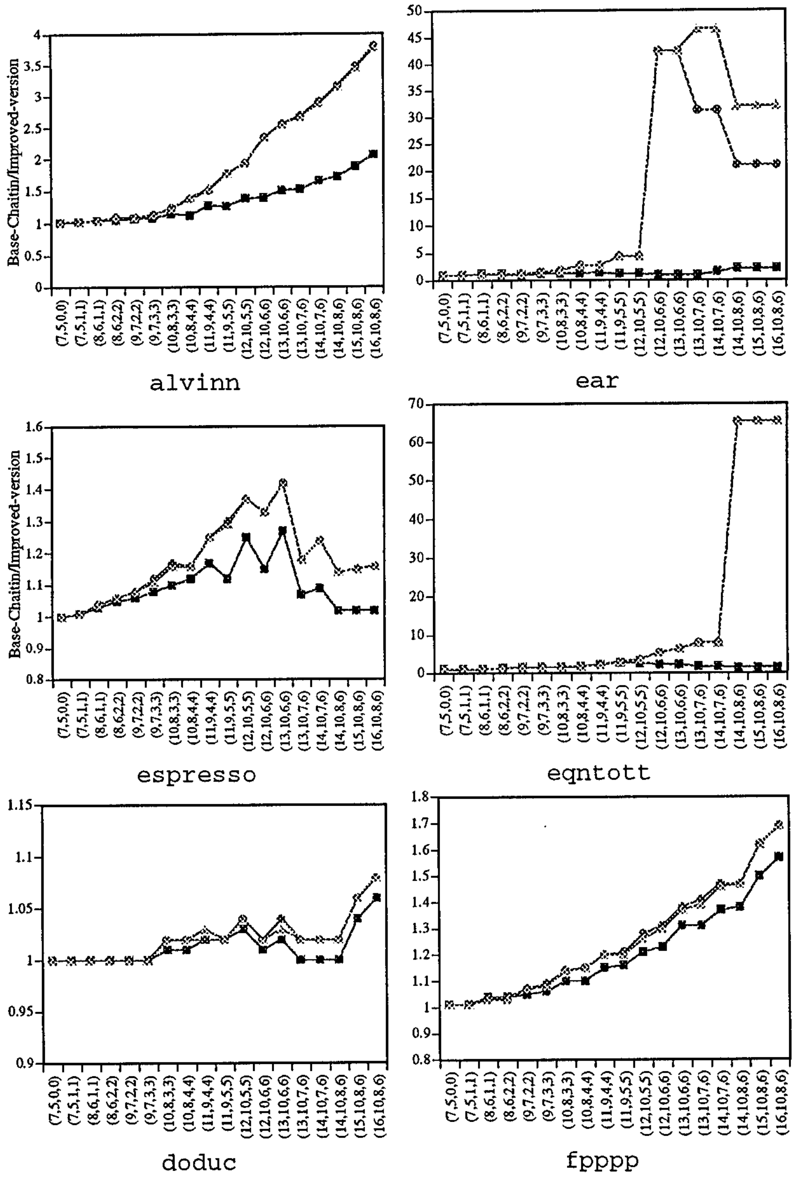


Figure 6.16: Degradation of improvement using static information.



alvinn

ear

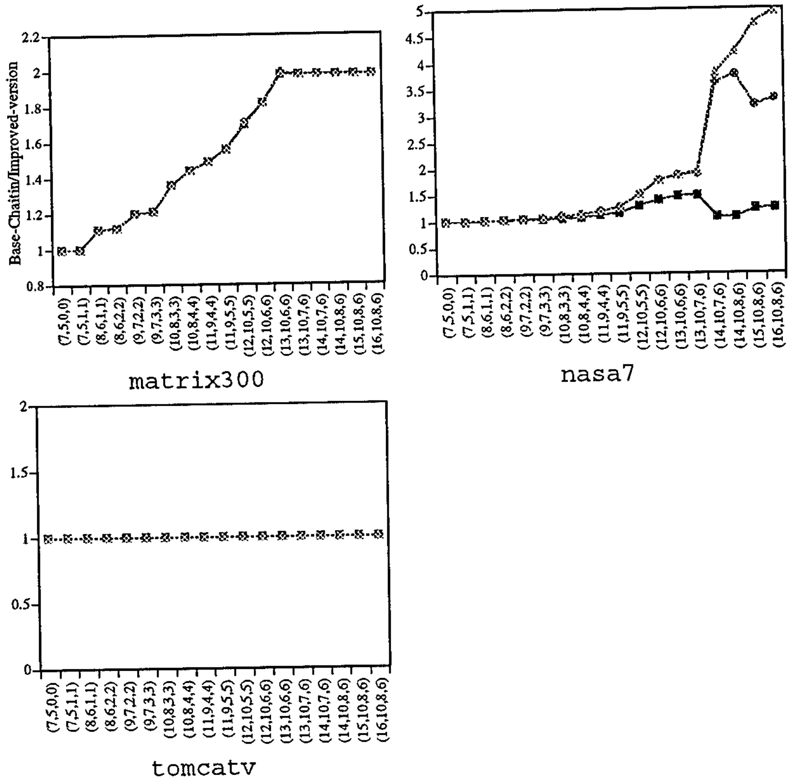
espresso

eqntott

doduc

fpppp

Figure 6.17: Similar trend of improvement as the dynamic case.



matrix300

nasa7

tomcatv

Figure 6.18: Similar trend of improvement as the dynamic case (cont.).

frequently executed path instead, then case (1) pays high callee-save costs, and case (2) incurs high spill costs. The register overheads of `compress`, `sc`, `gcc`, `li`, and `spice` (Figure 6.16) are hurt due to this reason. There is one noticeable aspect. When the number of registers is small, the more improvements are added into register allocation, the worse the register allocation result. As the number of registers increases, we see that the integration of all three improvements can make up the inaccuracy of the static estimated execution frequency. That is, the result of SC+BS+PR is better than SC. If the static information gives us a good indication about the actual execution pattern of a program, we see a similar improvement trend as using dynamic information (as shown in Figures 6.17 and 6.18).

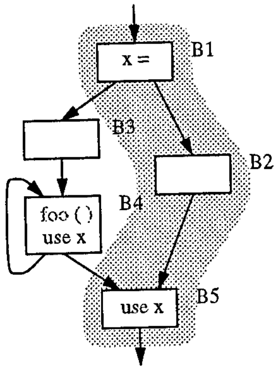


Figure 6.19: Dynamic versus static.

6.7 Evaluation of call-cost splitting

Given the importance of getting the right kind of register in the previous experiment, we take a positive action to limit the growth of live ranges, as described in Section 6.5. Because call-cost splitting can only be applied in fusion-style coloring, the base with which we compare this optimization is fusion-style coloring without call-cost splitting instead of Chaitin-style coloring. Fusion-style coloring with and without call-cost splitting all perform storage-class analysis, benefit-driven simplification, and preference decision. Figure 6.20 presents the results for `alvinn`, `nasa7` and `eqntott` for call-cost splitting. We run the experiment using static and dynamic information. `alvinn` and `nasa7` use all registers; `eqntott`

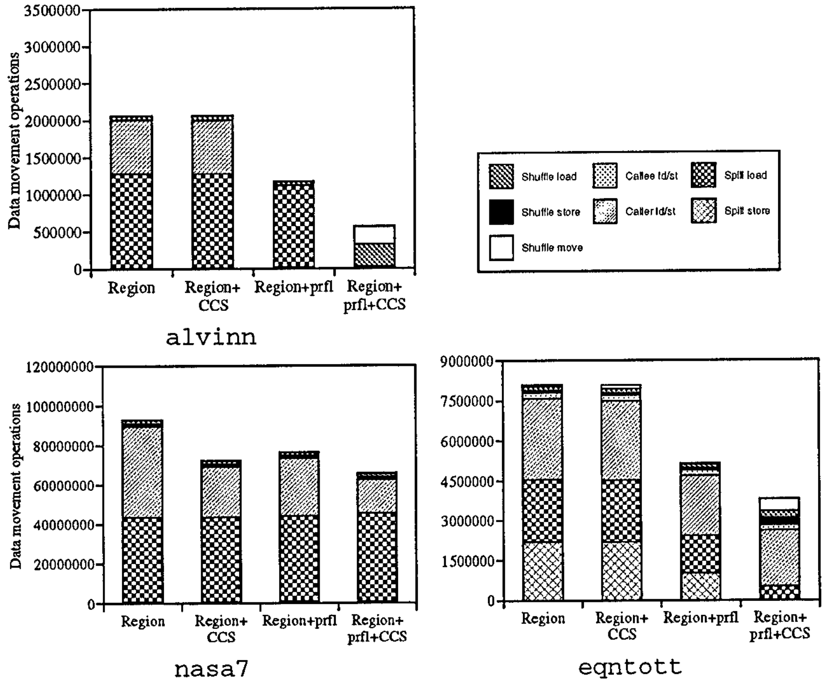


Figure 6.20: Impact of call-cost splitting (CCS).

uses 14 integer and 14 floating-point registers.

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 color-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 6.20 (i.e., for `alvinn` and `eqntott`) is due to such live ranges that we spilled during color assignment. The call-cost splitting suppresses coalescing of two live range segments lr_1 and lr_2 if one of them (say lr_2) has high caller-save cost. As a result, the spill cost of those live ranges stays low. If those live ranges get caller-save registers, they would probably be spilled to memory during color assignment because of a high caller-save cost and a low spill cost. 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 6.20 illustrate the benefits nicely. The call-cost splitting succeeds in isolating high caller-save cost regions at less frequently executed edges. Consequently, the color-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`, the fusion-based approach of dealing with spilling and splitting, including call cost optimization, reduces total data movement overhead by 80 % compared to a Chaitin-style allocator.

6.8 Summary

In this Chapter, I have presented and evaluated the four enhancements, storage-class analysis, benefit-driven simplification, preference decision, and call-cost splitting. The experimental results show that the call cost dominates the register overhead as the number of register increases, because the register allocator is able to assign registers to most live ranges. Ignoring the call cost may lead the register allocator to choose the wrong kind of register for live ranges and introduce, as a result, more overhead operations.

Chaitin-style coloring with all the enhancements (of Sections 6.2 – 6.4) combines the benefits of Chaitin-style allocation (pack more live ranges into registers) with the advantages of priority-based coloring (high priority live ranges have more freedom in choosing registers). The experimental data show that the improvement over ordinary Chaitin-style coloring can be up to a factor of about 50 in the number of overhead operations (55 for `ear` and 66 for `eqntott`).

Moreover, I have presented and evaluated a new technique, call-cost splitting, which constrains the growth of live ranges during graph fusion to reduce caller-save cost.

Chapter 7

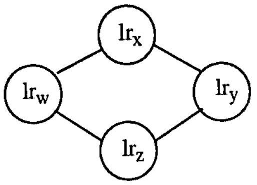
Comparison of Various Approaches

Chapter 6 describes three enhancements, storage-class analysis, benefit-driven simplification, and preference decision. Experimental results have shown that the integration of the three enhancements provides an effective call-cost model to reduce call cost relative to the base Chaitin-style allocator. It is important to evaluate the enhanced model with other existing model that is adopted by prior researchers to see if the enhanced model is practical or not. Various register allocation approaches share the same register allocation framework, e.g., Chaitin-style, optimistic, priority-based, and fusion-based coloring. The framework allows us to compare as well as integrate the enhancements with those approaches. It is interesting to see if the enhancements are effective in approaches other than Chaitin-style.

In this chapter, I compare various different register allocation approaches and discuss their strengths and weaknesses. In Section 7.1, I contrast optimistic coloring with Chaitin-style coloring. In Section 7.2, I discuss the similarity between priority-based and Chaitin-style using the three improvements of Chapter 6. In Section 7.3, I evaluate a register allocator using another call-cost model that has been adopted by prior researchers.

7.1 Optimistic versus non-optimistic

Optimistic coloring [9] delays spilling decisions of live ranges until the live ranges actually fail to find legal colors. Optimistic coloring aggressively tries to find colors for those otherwise spilled live ranges. If we exclude call cost from the overhead operations, optimistic coloring guarantees to deliver a result at least as good as Chaitin-style coloring. If none



	lr_w	lr_x	lr_y	lr_z
benefit_caller	100	200	100	-250
benefit_callee	200	100	200	50

Figure 7.1: Optimistic coloring with N=2 (1 callee-save and 1 caller-save).

of the live ranges spilled by Chaitin-style coloring gets a color, then the register overhead is the same as that of using Chaitin-style coloring. If some otherwise spilled live ranges are assigned colors, then optimistic coloring produces a superior result. However, if we use the actual cost model from Section 2.1, which includes the call cost, trying to squeeze more live ranges into registers may not be a good idea because the call cost of keeping live ranges in registers may be higher than the live ranges' spill cost. Figure 7.1 presents a situation where optimistic coloring generates more register overhead operations. Given two registers, one callee-save and one caller-save, simplification blocks because the degree of all live ranges is equal to 2. Optimistic coloring pushes all live ranges onto C and is able to assign legitimate colors for them. What may happen is that lr_x and lr_z use the callee-save register, and lr_w and lr_y use the caller-save register. In this case, the high caller-save cost of lr_z causes an inferior result.

Table 7.1 compares the overhead of optimistic and base Chaitin-style coloring using static and dynamic information to estimate execution frequencies. The value in each entry is the quotient Base-Chaitin/Optimistic. The darkly shaded regions ($<$ 1.00) highlight the cases where optimistic coloring results in more overhead operations than base Chaitin-style coloring. The lightly shaded regions ($>$ 1.00) indicate that optimistic coloring outperforms base Chaitin-style coloring. The (blank) rest ($=$ 1.00) indicates that optimistic coloring has no influence. Surprisingly, optimistic coloring does not improve overhead operations for most of the cases and, in addition, deteriorates the result of register allocation more often than ameliorates it. There are a couple of reasons for that: (1) if the register allocator assigns registers to the live ranges that have *high* spill cost, assigning registers to the otherwise spilled live ranges that have *low* spill cost makes only a small difference. Therefore we notice only a small improvement. (2) If the live ranges spilled by the base register allocator end up in the wrong kind of registers, the overall effect is *negative*. Optimistic coloring is sometimes effective for a small number of registers because the smaller the number of registers is, the more live ranges are spilled. Yet even in the best cases, the influence of optimistic coloring is small (within ± 6 %) except for fpppp when using static information (up to 36 % improvement in the number of overhead operations with a small number of registers).

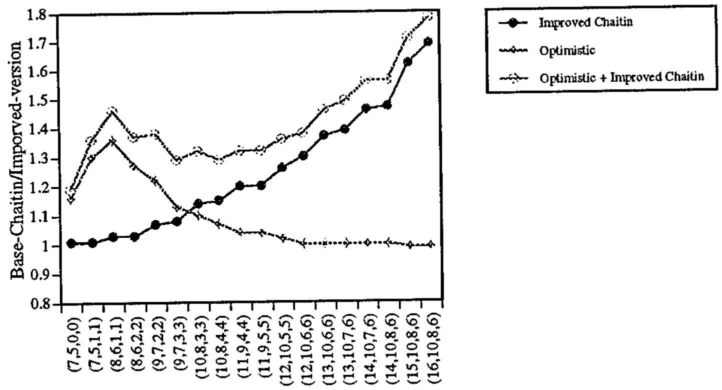


Figure 7.2: Optimistic versus non-optimistic for fpppp (using static information).

I incorporate optimistic coloring into improved Chaitin-style coloring (i.e., including storage-class analysis, benefit-driven simplification, and preference decision enhancements). Except for fpppp when using static information, the results are almost identical to those obtained by improved Chaitin-style coloring alone. This result is no surprise because the improvement of optimistic coloring is small, and the storage-class analysis spills the live ranges that are not worthwhile residing in registers; this optimization may actually undo the color assignment done by optimistic coloring. For instance, consider the example of Figure 7.1 again. The storage-class analysis checks the estimated saving (benefit) of keeping a live range in a register and spills the live range to memory if the saving is negative. lr_z is then spilled to memory in the color assignment phase. The result is the same as Chaitin-style coloring that spills lr_z when simplification blocks. Figure 7.2 shows in more detail the improvements due to optimistic coloring, improved Chaitin-style coloring, and improved Chaitin-style with optimistic coloring for fpppp using static information. Optimistic col-

STATIC	(7,5,0,0)	(7,5,1,1)	(8,6,1,1)	(8,6,2,2)	(9,7,2,2)	(9,7,3,3)	(10,8,3,3)	(10,8,4,4)	(11,9,4,4)	(11,9,5,5)	(12,10,5,5)	(12,10,6,6)	(13,10,6,6)	(13,10,7,6)	(14,10,7,6)	(14,10,8,6)	(15,10,8,6)	(16,10,8,6)
alvinn	0.99	0.99	0.99	0.98	0.98													
compress	1.04	1.06																
ear		0.88	0.89	0.87														
eqntott	0.95	0.93	0.93	0.92														
espresso				1.01	0.99	0.99	0.98	0.99										
gcc	1.01	1.02	1.01	1.01			0.98	0.99	0.98	0.96	0.97	0.97	0.98	0.98	0.99			
li																		
sc	0.94	0.95	0.93	0.99														
doduc	0.99		0.99	0.99	0.99	0.98	0.99	0.98	0.99	0.99	0.99	0.98	0.98	0.96	0.95	0.96	0.98	0.98
fpppp	1.16	1.30	1.36	1.27	1.22	1.13	1.10	1.07	1.04	1.04	1.02						0.99	0.99
matrix300	1.01	1.01	1.05	1.06														
nasa7	1.01	1.01	1.01	1.01	1.01	1.01				1.03			0.99		0.99			1.01
spice	1.02	1.01	1.02	1.02	1.01	1.02	1.02		0.99	0.99								
tomcatv					1.03	1.04	1.04	1.05	1.06	1.07								

DYNAMIC	(7,5,0,0)	(7,5,1,1)	(8,6,1,1)	(8,6,2,2)	(9,7,2,2)	(9,7,3,3)	(10,8,3,3)	(10,8,4,4)	(11,9,4,4)	(11,9,5,5)	(12,10,5,5)	(12,10,6,6)	(13,10,6,6)	(13,10,7,6)	(14,10,7,6)	(14,10,8,6)	(15,10,8,6)	(16,10,8,6)
alvinn	0.94	0.98	0.97	0.98	0.98	0.98												
compress	1.03	1.06																
ear		0.91	0.89	0.74														
eqntott		0.99	0.98	0.98														
espresso	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.98										
gcc	1.02	1.07	1.03	1.01	1.01	0.99	0.99	0.98	0.97	0.98	0.98	0.97	0.97	0.97	0.98	0.98	0.98	0.98
li				0.95														
sc	0.96	0.96	0.95	1.01	1.01	1.01	1.01	1.01										
doduc	0.99	0.99	0.99	0.98	0.98	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.98	0.96	0.99	0.99	0.99
fpppp	0.98	0.95	0.95	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.95
matrix300			0.95	0.95														
nasa7	1.02		1.02								0.99		0.99		0.99	0.99		
spice				1.01	0.99			0.99										
tomcatv					1.03	1.04	1.04	1.05	1.06	1.07								

Table 7.1: Optimistic coloring versus Chaitin-style.

oring performs well with a small number of registers, but the enhancement drops as the number of registers increases. Improved Chaitin-style coloring, on the other hand, performs well as the number of registers increases, because the more registers are available, the more freedom the register allocator has to choose between caller-save and callee-save registers. We see the improvements of optimistic coloring plus improved Chaitin-style coloring as the two are integrated. When the number of registers is small, improved Chaitin-style coloring is not very effective, so we see only the contribution of optimistic coloring. As the number of registers increases, optimistic coloring has less influence and improved Chaitin-style coloring picks up, so we see the enhancement due to improved Chaitin-style coloring.

7.2 Priority-based vs. Chaitin-style

The use of a benefit function to sort live ranges invites a comparison with priority-based coloring [16]. At first sight, Chaitin-style coloring and priority-based coloring appear to have little in common. Priority-based coloring assigns registers to live ranges based on a priority function and splits a live range lr when no legal register exists for lr.

Chaitin-style and priority-based coloring share a major core if priority-based coloring simply spills rather than splits live ranges when it runs out of colors. That is, both approaches try to determine the color ordering in which live ranges are assigned colors (registers). Chaitin-style coloring simplifies the interference graph to decide the ordering in which live ranges are assigned colors. The color-assignment phase then guarantees to assign registers for all non-spilled live ranges based on the ordering. Priority-based coloring uses cost analysis as the priority to determine the ordering and guarantees that the color-assignment phase assigns registers to the most important (high savings of memory accesses) live ranges. The two approaches aim at two different directions: (1) Chaitin-style coloring finds an ordering that packs live ranges into registers—potentially it uses fewer colors than priority-based coloring, and (2) priority-based coloring wants to make sure that the most important live ranges are in registers even though it may require more colors (or spill more unimportant live ranges). Both approaches have their own strengths and weaknesses. Chaitin-style coloring with all three enhancements is a hybrid of the two approaches that uses Chaitin-style coloring as the framework for packing live ranges and uses storage-class analysis and benefit-driven simplification to achieve the same effect as the priority-based approach.

7.2.1 Priority functions

Reducing register overhead operations is the primary goal of the priority function. Hence, the more memory accesses saved by assigning a register to a live range, the *higher* the priority of the live ranges is. The bigger a live range, the more conflicting live ranges cannot use the same register as the live range. That is, higher register pressure is introduced if bigger live ranges reside in registers. Higher register pressure is more likely to cause overhead operations. Therefore, the bigger a live range, the *lower* the priority of the live range is. Based on the two conditions, we use $\frac{max(benefit\_caller(lr),benefit\_callee(lr))}{size(lr)}$ as the priority function for priority-based coloring. $sizc(lr)$ is the number of basic blocks that lr contains. This priority function is the same as the one used in [16]. The spilling heuristic of Chaitin-style coloring and optimistic coloring is $\frac{spill\_cost(lr)}{degree(lr)}$. The heuristic tends to spill live ranges that have low spill cost and high degree. The priority function and the spilling heuristic are alike. The numerator of the priority function reflects the spill cost because the bigger the spill cost, the bigger the two benefit functions are. The denominator reflects the degree of live ranges because the bigger the live ranges, the more likely the live ranges have higher degree.

There are a few ways to determine the color ordering for priority-based coloring:

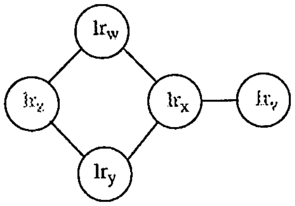
- *removing unconstrained*: Unconstrained live ranges are removed from the interference graph and pushed onto C. The remaining live ranges are pushed onto C from the least priority to the highest priority. Chow uses the same approach in [16].

- *sorting unconstrained*: Unconstrained live ranges are not pushed onto C in a priority fashion if we simply remove unconstrained live ranges. That is, the kind of register that a higher-priority live range wants is possibly taken away by lower-priority live ranges. Sorting unconstrained live ranges alleviates the problem by pushing unconstrained live ranges onto C also in a priority manner.

- *sorting*: All live ranges are purely sorted in terms of their priorities. The sorting approach guarantees that the higher the priority, the more likely that live ranges are assigned their preferred kind of register.

All three heuristics produce nearly identical register overhead (± 10%) for most cases. A detailed comparison is given in Tables 7.2 and 7.3. RU, SU, and S indicate Removing Unconstrained, Sorting Unconstrained, and Sorting, respectively. There are two rows for each SPEC92 program, RU/SU and RU/S. The entry is the ratio of the register overhead of the two heuristics, e.g., 1.04 for RU/SU means that the RU heuristic introduces 4% more overhead operations than the SU heuristic. The empty entries in the tables indicate the two heuristics generate the same register overhead (ratio is 1.00). The experimental results show that the RU and SU heuristics give identical results except in a few cases. S(orting approach) yields much better results (less overhead) for `ear` (both static and dynamic) and `espresso` (dynamic).

Figure 7.3 shows an example where S yields a better result than RU and SU. Given N=2 (1 callee-save and 1 caller-save), lr_v is the only unconstrained live range (degree = 1). RU and SU then remove lr_v and assign colors to the rest live ranges in the priority order, lr_x, lr_z, lr_w and lr_y. The callee-save register is assigned to lr_x because lr_x is considered the highest priority live range. As a result, lr_v can reside only in the caller-save register. The RU and SU approaches save 5900 load/store operations. S treats lr_v as the highest priority live range and assigns the caller-save register to lr_v. This assignment saves 8500 load/store operations.



(a)

	lr_w	lr_x	lr_y	lr_z	lr_v
benefit_caller	1500	2000	1000	1500	4000
benefit_callee	1000	1000	400	1000	1000
size	4	3	4	3	2
degree	2	3	2	2	1
priority	350	667	250	500	2000

(b)

Figure 7.3: Priority functions with N=2 (1 callee-save and 1 caller-save).

7.2.2 Evaluation

We compare improved Chaitin-style coloring with priority-based coloring using the sorting heuristic. We can classify the SPEC92 programs based on the results into 3 classes. Figures 7.4, 7.5, 7.6, and 7.7 show the results.

- Both priority-based and improved Chaitin-style coloring are doing equally well. Examples are `alvinn`, `eqntott`, `gcc`, and `li` (Figure 7.4). For `alvinn`, packing live ranges is important for small numbers of registers. When the number of registers increases, packing live ranges becomes less important (because high spill-cost live ranges are most likely to find registers) so we see that the two approaches yield similar results.

- Improved Chaitin-style coloring is superior to priority-based coloring. `compress`, `ear`, `sc`, `doduc`, `nasa7`, `spice`, and `tomcatv` fall into this category (Figures 7.5 and 7.6). For `nasa7`, improved Chaitin-style coloring produces slightly fewer overhead operations than priority-based coloring when the number of registers is small. When the two approaches have more registers, the results start to diverge. Chaitin-style coloring produces a similar improvement trend in both the dynamic and static cases. Priority-based coloring, nevertheless, does not improve overhead operations a lot over the base in the static case. In this case, priority-based coloring introduces 4 times more overhead operations than improved Chaitin-style coloring. The same scenario happens to `ear` as well. For `tomcatv`, the priority-based approach spills more live ranges than the improved Chaitin-style approach, both in the static and dynamic cases, due to the inability of packing live ranges densely. For `sc`, improved Chaitin-style coloring yields bigger improvement than priority-based coloring in the dynamic case. We, therefore, classify it into this class although priority-based coloring using static information has slightly fewer overhead operations than improved Chaitin-style coloring.

- There is no clear winner between the two approaches. Examples of this class are `espresso`, `fpppp` and `matrix300` (Figure 7.7). For `espresso`, priority-based coloring is clearly superior to improved Chaitin-style coloring in the dynamic case but not in the static case. If we compare the results for `fpppp` (static) (Figure 7.7)

STATIC		(7,5,0,0)	(7,5,1,1)	(8,6,1,1)	(8,6,2,2)	(9,7,2,2)	(9,7,3,3)	(10,8,3,3)	(10,8,4,4)	(11,9,4,4)	(11,9,5,5)	(12,10,5,5)	(12,10,6,6)	(13,10,6,6)	(13,10,7,6)	(14,10,7,6)	(14,10,8,6)	(15,10,8,6)	(16,10,8,6)
alvinn	RU/SU										1.02	1.02							
	RU/S							1.02	1.08	1.03	1.02	1.02							
compress	RU/SU																		
	RU/S		1.03																
ear	RU/SU																		
	RU/S	1.04						0.65	0.61	2.55	3.69	3.69	3.55	3.55	2.62	1.71	0.77	0.77	0.77
eqntott	RU/SU											1.02	1.03	1.03					
	RU/S				1.01		0.99	0.99			1.02	1.03	1.05	1.05	1.02	1.02			
espresso	RU/SU																		
	RU/S	0.98	0.99			1.04	0.97	0.92	1.01	1.01	0.85	1.01	1.03	0.94	0.96	0.99	1.02	1.03	1.03
gcc	RU/SU											0.99							
	RU/S				0.99		0.99	0.98	0.99	0.98	0.98	0.98	0.99	0.98		0.98	0.97	0.97	0.97
li	RU/SU																		
	RU/S							0.96	0.97	0.97	0.98	0.98							
sc	RU/SU																		
	RU/S			0.99	1.01				1.02	1.02			1.06	1.06	1.07	1.07			
doduc	RU/SU																		
	RU/S	0.93	0.98		0.98	0.98		1.01	1.01		0.97	0.99	0.99	0.98	0.98	0.98	0.98	0.98	0.98
fpppp	RU/SU																		
	RU/S	1.21		0.99		0.99	0.90	0.97	0.98	0.99	0.99	0.99	0.98	0.99	1.01	1.01	1.01	1.01	1.01
matrix300	RU/SU																		
	RU/S					0.91	0.90												
nasa7	RU/SU																		
	RU/S			0.97	0.97			0.99											
spice	RU/SU																		
	RU/S	0.98	0.98	0.96	0.96	0.98	0.98	0.96	0.98	0.92	0.97	0.96	0.93	0.94	0.92	0.90	0.85	0.85	0.85
tomcatv	RU/SU																		
	RU/S	0.89	0.94	0.98	0.94	0.96	0.98	1.04	1.04	1.04	1.05	1.05							

RU: removing unconstrained SU: sorting unconstrained S: sorting

Table 7.2: Comparison of priority-based heuristics (using static information).

DYNAMIC		(7,5,0,0)	(7,5,1,1)	(8,6,1,1)	(8,6,2,2)	(9,7,2,2)	(9,7,3,3)	(10,8,3,3)	(10,8,4,4)	(11,9,4,4)	(11,9,5,5)	(12,10,5,5)	(12,10,6,6)	(13,10,6,6)	(13,10,7,6)	(14,10,7,6)	(14,10,8,6)	(15,10,8,6)	(16,10,8,6)
alvinn	RU/SU																		
	RU/S						1.02	1.02	1.08	1.03	1.02	1.02							
compress	RU/SU								1.02	1.08	1.03								
	RU/S	0.97	1.04	0.93	0.95				1.08		1.10								
ear	RU/SU																		
	RU/S							0.65	0.61	2.54	3.47	3.48	45.04	45.04	52.17	34.00	20.34	20.34	20.34
eqntott	RU/SU											1.02	1.03	1.03					
	RU/S									1.01	1.01	1.03	1.05	1.05	1.02	1.02			
espresso	RU/SU									1.03	1.04		0.93	0.94	0.90	0.93	0.94		1.01
	RU/S	0.99	1.01	1.01	1.02	1.03	1.05	1.03	1.04	1.12	1.22	1.20	1.20	1.26	1.29	1.17	1.12	1.07	1.07
gcc	RU/SU																		
	RU/S	0.99	1.02	1.01	1.02	1.02	1.02	1.02	1.02	1.01	1.01	1.01	1.02	1.02	1.02	1.03	1.03	1.01	1.01
li	RU/SU																		
	RU/S						0.99	0.99											
sc	RU/SU																		
	RU/S		0.99						1.04	1.05									
doduc	RU/SU																		
	RU/S	0.93	0.97	0.98	0.99				0.97	0.94	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96
fpppp	RU/SU																		
	RU/S	0.95								0.98	0.99						0.97	0.97	
matrix300	RU/SU																		
	RU/S																		
nasa7	RU/SU																		
	RU/S			0.97	0.97			0.99											
spice	RU/SU																		
	RU/S	0.97	0.96	0.96	0.97		0.99	0.99	1.01			0.94	0.99	0.99	0.99	0.99	0.99	0.99	0.99
tomcatv	RU/SU																		
	RU/S	0.89	1.09	1.17	0.94	0.96	0.95			1.05	1.06	10.6							

RU: removing unconstrained SU: sorting unconstrained S: sorting

Table 7.3: Comparison of priority-based heuristics (using dynamic information).

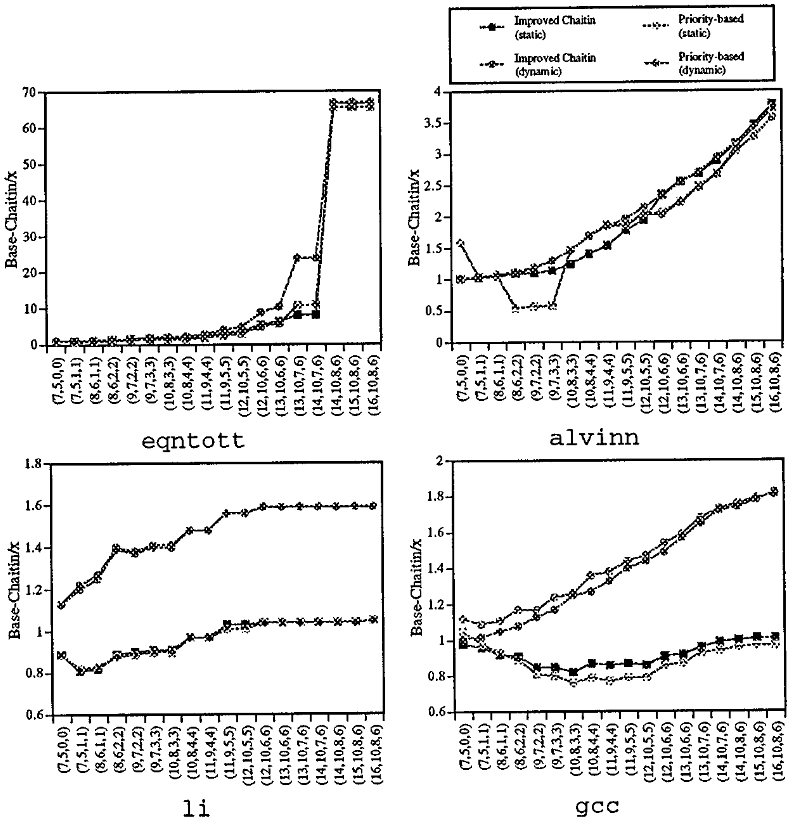


Figure 7.4: Priority-based ≈ Chaitin-style.

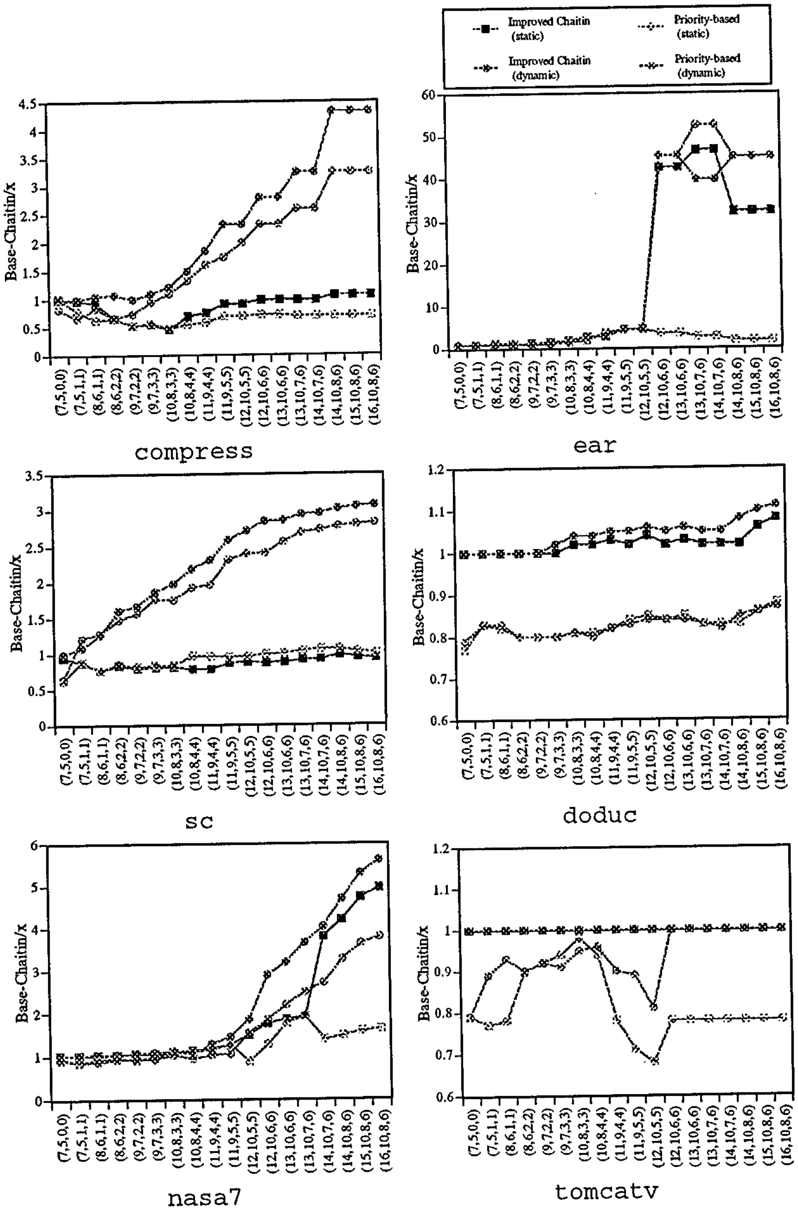


Figure 7.5: Priority-based < Chaitin-style.

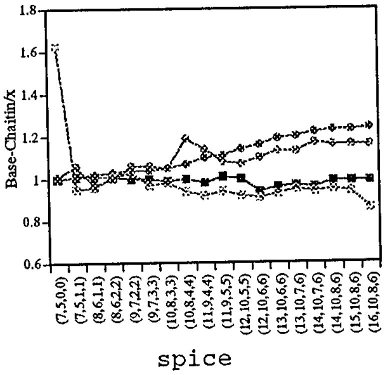
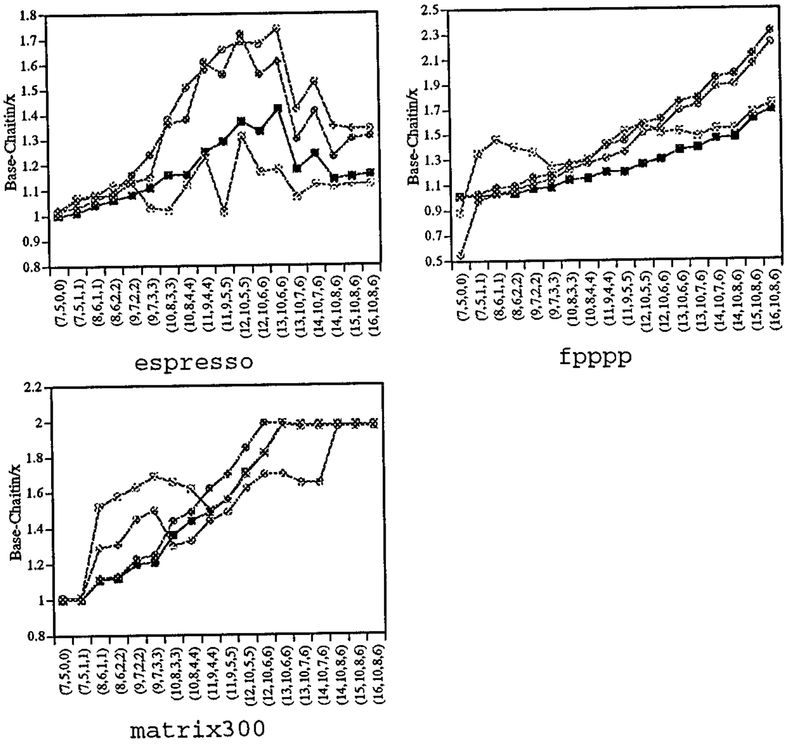


Figure 7.6: Priority-based < Chaitin-style (cont.).

with the results shown in Figure 7.2, we notice a similarity between priority-based coloring and the integration of improved Chaitin-style and optimistic coloring. When the number of registers is small, we see the trend of optimistic coloring. As the number of registers increases, we see the trend of improved Chaitin-style coloring.

From the experimental results, improved Chaitin-style coloring is superior to priority-based coloring in two ways: (1) Chaitin-style coloring is able to pack more live ranges into a set of registers, which potentially introduces fewer spill code operations, (2) the priority function used in priority-based coloring may cause low spill-cost live ranges to take away the preferable registers of high spill-cost live ranges.

The example in Figure 7.8 illustrates how the second case happens with N=3 (1 callee-save and 2 caller-save). The two benefit functions, $benefit\_caller$ and $benefit\_callee$, of each live range are shown in the table. Priority-based coloring considers lr_y to have the highest priority because of high spill cost. Subsequently, the callee-save register is assigned to lr_y, which leaves the two caller-save registers for lr_w, lr_x and lr_z. This color assignment results in a bad register assignment because the penalty of picking the wrong kind of register for lr_w is high. The assignment ends up with saving 8300 memory accesses. Benefit-driven simplification, on the other hand, can first simplifies either lr_x or lr_z. Consequently, all remaining nodes are unconstrained (degree = 2). Benefit-driven simplification using the second strategy (Section 6.3) puts lr_w on the top of the color stack because lr_w has the highest penalty of picking the wrong kind of register. As a result, improved Chaitin-style



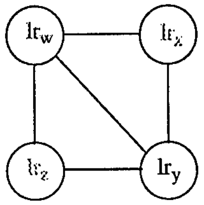
espresso

fpppp

matrix300

Figure 7.7: Priority-based >< Chaitin-style.

coloring saves 11000 memory accesses. This is the reason why priority-based coloring yields much more overhead operations for `nasa7` and `ear` in the static case.



(a)

	lr_w	lr_x	lr_y	lr_z
benefit_caller	300	2000	4000	2000
benefit_callee	3000	2400	4500	2400
size	4	3	4	3
degree	3	2	3	2
priority	750	800	1125	800

(b)

Figure 7.8: Priority-based coloring with N=3 (1 callee-save and 2 caller-save).

7.3 Chaitin-style versus CBH

In this section, we compare improved Chaitin-style coloring with another base model that is implemented in various compilers. This model extends Chaitin-style coloring to explicitly model the calling convention. According to Briggs [8], this model also extends hierarchical coloring (as described for the Tera compiler [12]). Based on the main components, we name this cost model "CBH" (for Chaitin/Briggs–Hierarchical coloring). Although the CBH cost model has been adopted by prior researchers, there is no published report of the model's effectiveness.

The comparison of this section serves three purposes. First, it allows us to know if our base Chaitin-style coloring is a reasonable base or not, relative to a model adopted by other compilers. Second, the comparison presents the effectiveness of the CBH cost model. Third, the comparison also shows if the call-cost directed Chaitin-style model outperforms CBH.

In the CBH approach, live ranges that cross calls interfere with all caller-save registers. In other words, none of the caller-save registers can be assigned to live ranges that contain calls. A live range is introduced for each callee-save register extending from the entrance of a routine to the exit of the routine. The live range is called a *callee-save-register* live range. Each callee-save-register live range has two references, one at the entry and the other at the exit (these represent save/restore of the callee-save register at the entry/exit). With the execution frequency of the routine, we can get the spill cost for the live range. The call cost is then modeled by CBH just like regular spill cost. Register allocation performs simplification to determine if it is worthwhile using a callee-save register. When simplification blocks, one live range with the least spill cost is chosen from the remaining live ranges, including the callee-save-register live ranges. Choosing a callee-save-register live range as opposed to a regular live range to spill means spilling each remaining ordinary live range is more expensive than saving/restoring the callee-save register at the entry/exit. Once a callee-save register is spilled, the callee-save register can then be assigned to live ranges. This is similar to the way Chow handles callee-save registers [16]. That is, the *first* live range that uses a callee-save register pays the callee-save cost.

I compare CBH with improved Chaitin-style coloring. To allow an easy comparison of the new figures with the figures in Section 6.6, I use the same base line. The data for improved Chaitin-style coloring in Figures 7.9, 7.10, and 7.11 are the data (SC+BS+PR) from Section 6.6. Again the y-axis of all figures shows the ratio of register overhead operations relative to the register overhead produced by the base Chaitin-style. The bigger the number on the y-axis, the more overhead operations are removed.

We can analyze the data from several different perspectives. CBH represents a model of call cost that is conceptually simple (always a good property). CBH does not allow live ranges that cross calls to use caller-save registers because they interfere with all caller-save registers. Those extra interference edges, however, may overly constrain register allocation, especially when there is not a sufficient number of callee-save registers because all live ranges crossing calls are competing for callee-save registers. Examples of programs that exhibit this behavior are `alvinn`, `compress`, `ear`, `espresso`, `gcc`, `li`, `sc`, `doduc`, `matrix300`, and `spice`. For those SPEC92 programs, *high* spill-cost live ranges that cross calls are spilled because there are only a small number of callee-save registers. As the number of callee-save registers increases, the constraints become less critical, and CBH is able to assign registers effectively for `alvinn`, `ear`, and `matrix300`.

Then there are cases where improved Chaitin coloring and the CBH approach produce almost identical results, i.e. the same number of overhead operations. fpppp and tomcatv fall into this category.

Sometimes the CBH approach does not improve the quality of register allocation using profile information. Examples are compress, li, sc, gcc, doduc, nasa7 and spice. For those programs, we see that CBH cannot catch up with improved Chaitin-style coloring. The main reason for this situation is that many live ranges on the most frequently executed paths also cross call sites that are on less frequently executed paths. All those live ranges are competing for callee-save registers regardless the fact that the call sites contained in the live range are not frequently executed. Consequently, those live ranges are either assigned callee-save registers or spilled. As a result, CBH spills more than necessary and incurs high register overhead (spill code) for such programs. On the other hand, improved Chaitin-style coloring pays caller-save cost at occasionally executed paths and assigns caller-save registers to live ranges that do not cross calls on frequently executed paths.

Another noteworthy aspect is that CBH requires a fair number of registers to make up the deficiency of its call cost model for matrix300 and nasa7. The data for improved Chaitin-style coloring and CBH of matrix300 (dynamic) show that improved Chaitin-style coloring continues improve beyond using (9,7,3,3) registers, whereas CBH still suffers from the lack of callee-save registers until it has 4 additional integer and 3 additional floating-point callee-save registers. nasa7 follows a similar pattern. For ear and eqntott, we see a similar trend although it is less pronounced. However, the situation is not as severe as for matrix300 or nasa7, and CBH needs only a few more registers to compensate the limitation of its call cost model.

For ear and nasa7, improved Chaitin-style coloring achieves a big improvement for both the static and dynamic cases. CBH, however, produces roughly identical overhead operations as the base Chaitin-style coloring in the static case. From those figures, we can derive the conclusion that improved Chaitin-style coloring is superior to CBH and the "base model" is actually reasonable after all. For a lot of register ranges, the base Chaitin-style allocator is actually superior to the CBH-style allocator.

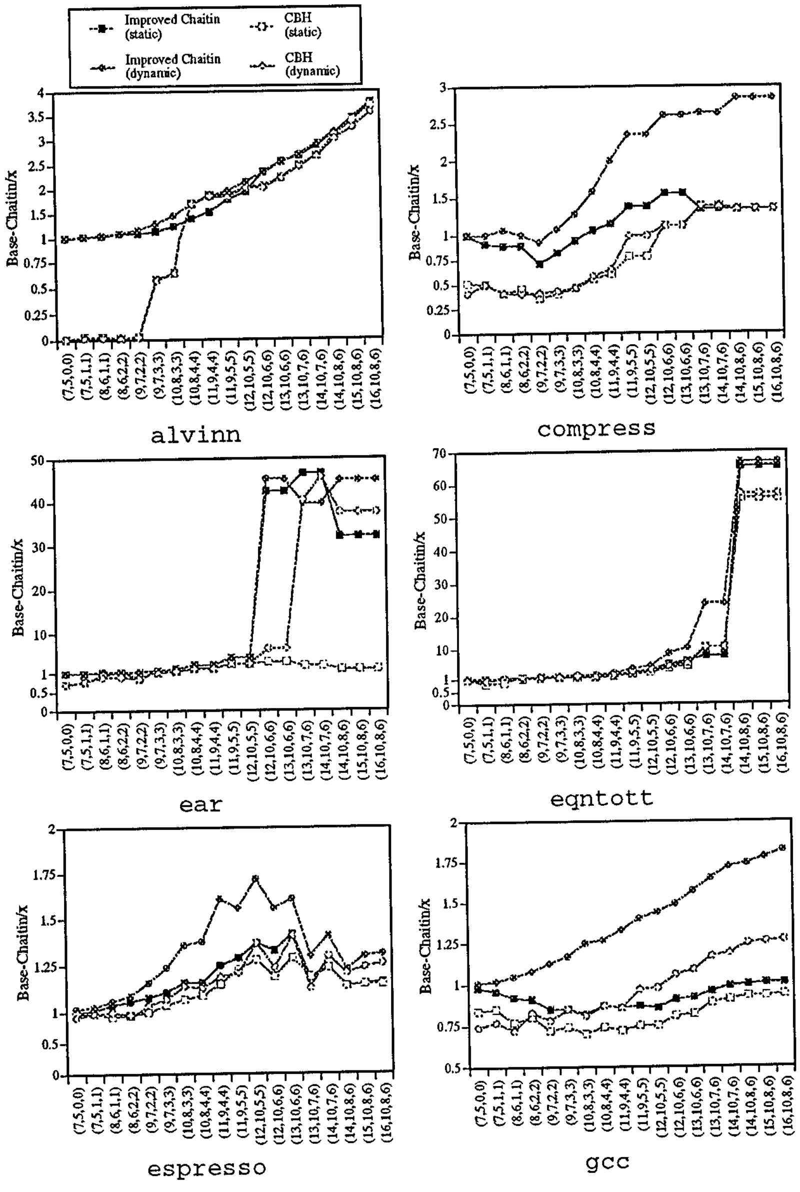
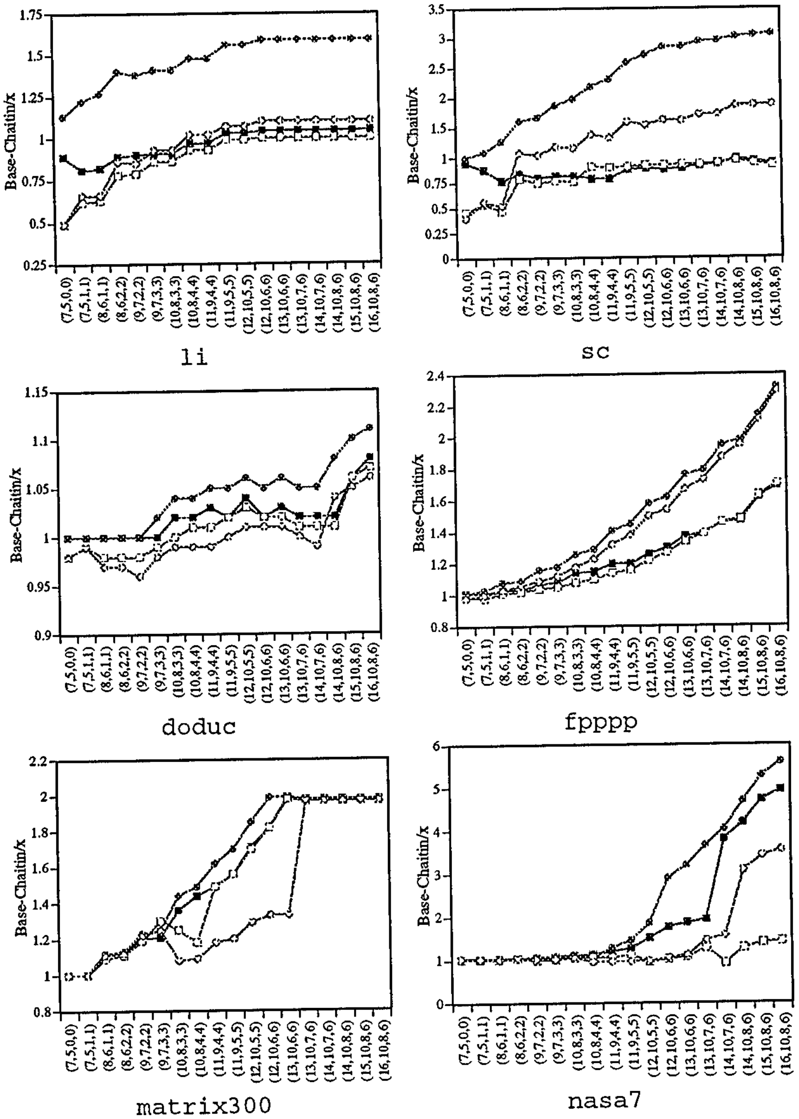


Figure 7.9: Improved Chaitin-style versus CBH.



li

sc

doduc

fpppp

matrix300

nasa7

Figure 7.10: Improved Chaitin-style versus CBH (Cont.).

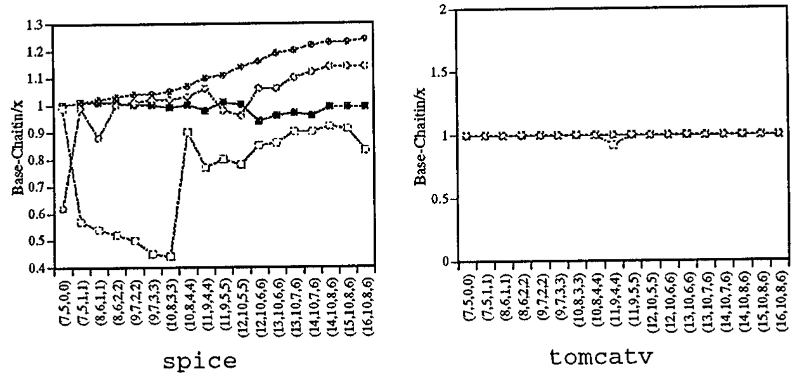


Figure 7.11: Improved Chaitin-style versus CBH (Cont.).

7.4 Summary

In this chapter, I evaluate the effectiveness of optimistic coloring. The results indicate that the influence of optimistic coloring is small in general. In addition, optimistic coloring needs to take call cost into account. Otherwise, the overall effect may be negative.

From the experimental results, improved Chaitin-style coloring is superior to priority-based coloring. Chaitin-style coloring is able to pack more live ranges into a set of registers. The enhancements described here allow Chaitin-style allocation to order live ranges, thereby achieving similar benefits as can be obtained from the priority function of priority-based coloring. As a result, low spill-cost live ranges do not take away the registers preferred by high spill-cost live ranges.

Coloring based on the CBH cost model imposes the constraint of not assigning caller-save registers to live ranges containing calls. Due to this constraint, callee-save registers then become the most critical resources in a lot of cases. As a result, the caller-save registers may be underutilized; live ranges containing calls are spilled although the calls may be executed only infrequently.

One interesting aspect of register allocation is that each model hits some aspects of dealing with call cost. There are some programs (or some register ranges for some programs) where the priority-based and/or CBH-style approaches perform reasonably well, as there are situations where optimistic allocation performs well. One of the challenges is to devise an approach that combines the benefits of all models, and I demonstrated that storage-class analysis, benefit-driven simplification, and preference decision can be integrated into Chaitin-style coloring. This improved Chaitin-style register allocator provides an effective way to deal with modern calling conventions for a wide range of programs.

Chapter 8

Extensions

There are two important techniques of register allocation that have not yet been covered so far: coalescing and rematerialization. Both techniques improves the quality of register allocation. Coalescing tries to eliminate unnecessary copy instructions. Rematerialization replaces spill code by instructions that recompute the values. It is important to see how coalescing and rematerialization interact with fusion-based register allocation. There are a couple of interesting questions that we would like to be able to answer. Can fusion-based register allocation implement rematerialization (Section 8.1)? Can fusion-based register allocation improve coalescing quality (Section 8.2)? In this chapter, we discuss each of these aspects in detail.

8.1 Rematerialization

Most modern processors require multiple cycles to access the cache, i.e., a memory access needs more than one cycle; on the other hand, several values can often be recomputed using a single instruction. *Rematerialization* recognizes that some values are cheaper (require fewer cycles) to recompute than to spill to memory and uses the instructions that recompute the values to replace the spill code. Because the register allocator knows the value of a rematerializable live range, to spill a live range, it is not necessary to insert a store after each definition of the live range. Then instead of inserting a load before each use of the live range, register allocation places there the instruction that recomputes the value. [10] lists opportunities for rematerialization:

- *immediate loads of constants*: For a machine that has the immediate addressing mode, immediate loads of integer constants can be accomplished, e.g., using `li` on MIPS, `set` on SPARC, or `ldil` on Alpha processors.

- *computing effective address*: The constant offset of the stack frame can be computed using `add` with immediate addressing mode, e.g. "`add srl, fp, 64`".

- *loads*: The value of a load from a known constant location in either the stack frame or the static data area can be reloaded using one load instruction, e.g., "`load srl, 64(fp)`". Although the load instruction is as expensive as the spill load, we save/eliminate spill stores that write the value to the spill location.

- loading non-local frame pointer from a display.

The cost of spilling a rematerializable live range ought to be modeled differently from a live range whose value can not be rematerialized. For instance, fusion-based coloring uses one of the three heuristics in [5] to choose one live range from the set of constrained live ranges. The heuristic is $\frac{cost(lr)}{degree(lr)*area(lr)}$, where $cost(lr)$ is a function of $spill\_cost(lr)$ (the weighted reference count). Assume that a memory access takes two cycles. The $cost(lr)$ is redefined as follow:

$$cost(lr) = \begin{cases} rcmatcrialization\_cost(lr)) & \text{if } lr \text{ is rematerializable} \\ spill\_cost(lr)*2 & \text{otherwise} \end{cases}$$

$rcmatcrialization\_cost(lr)$ is the weighted *cycle* count of recomputing the value of lr.

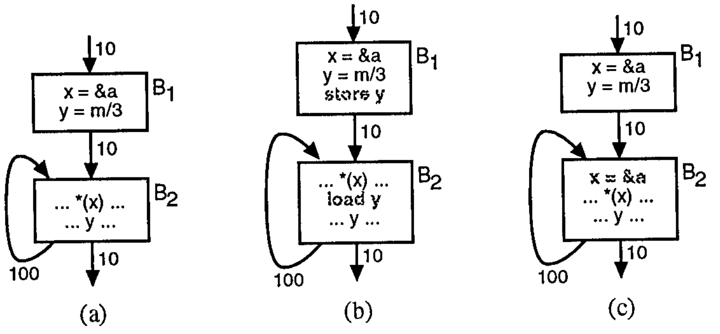


Figure 8.1: Spilling a rematerializable live range.

Consider the two live ranges, lr_x and lr_y, in Figure 8.1(a). Both x and y are defined in B_1 and referenced once in B_2. lr_x and lr_y have the same spill_cost. The numbers next to control-flow edges are execution frequencies. Spilling x or y to memory incurs 220 cycles. Figure 8.1(b) shows the code after we spill y. lr_y can not be rematerialized because the cost of recomputing "y=m/3" is expensive. lr_x can be rematerialized because recomputing address a can be done using one instruction. The rematerialization_cost of lr_x is 100 cycles. Assume that the register allocator requires to spill either lr_x or lr_y. The register allocator recognizes that rematerializing lr_x is cheaper than spilling lr_y using the redefined spilling heuristic (depicted in Figure 8.1(c)).

8.1.1 Prior approaches

Chaitin et al. proposed and implemented the rematerialization technique in the IBM PL.8 compiler [13]. Briggs et al. [10] show that there are cases where the value of a live range is rematerializable in some parts of the live range, but not the whole live range. The rematerialization technique in the IBM PL.8 compiler was not integrated with live-range splitting. Hence, once the rematerializable value of a live range is killed within the live range, they give up potential opportunities of rematerializing some parts of the live range and spilling the rest.

Briggs et al. [10] extend Chaitin's approach by splitting each live range into its component values using the SSA representation [20] so that each live range has exactly one definition. The goal is to isolate rematerializable values so as to expose all possible opportunities of rematerialization. Wegman and Zadeck's sparse simple constant algorithm [62] is modified to propagate rematerialization tags to each value. Because splitting each live range into its component values may cause excessive shuffle code (moves), conservative coalescing and biased coloring techniques are used to eliminate unproductive moves. Again the shortcoming of their approach is that splitting decisions are made prematurely. Conservative coalescing and biased coloring techniques do not guarantee to remove all unproductive moves. In addition, splitting points may be on frequently executed paths. Failing to eliminate those splitting points causes a lot of excessive shuffle code.

8.1.2 Rematerialization in fusion-style coloring

This section presents how rematerialization can be implemented in fusion-style register allocation. There are 3 phases in which fusion-style coloring deals with rematerization: propagating rematerialization tags, graph construction, and shuffle code insertion. A new phase is inserted before the graph construction phase to propagate rematerialization tags. Therefore, when constructing a live range, we know if the live range is rematerializable or not.

Propagating rematerialization tags

The modified Wegman and Zadeck's algorithm used in [10] does not fit into our model because the actual splitting decisions are not yet made and live ranges are not yet built. We can use data-flow analysis to propagate rematerialization tags. The data-flow equations are defined as follows:

$$\begin{aligned}
RcmIn(B) &= \bigcup_{J \in Pred(B)} RcmOut(J)\\
RcmOut(B) &= (RcmIn(B) \cdot \neg RcmKill(B)) + RcmGcn(B)\\
RcmGcn(B) &= \{inst_x \mid inst_x \in B \wedge \text{the value of } inst_x \text{ is never killed in } B\}\\
RcmKill(B) &= \{inst_x \mid dcf_x \in B\}
\end{aligned}$$

The notation $inst_x$ represents an instruction whose target is virtual register x and the value is rematerializable, e.g. "add x, fp, 64". $inst_x \in B$ indicates that the instruction $inst_x$ shows up in the schedule of block B. $dcf_x \in B$ denotes that there is an instruction in B whose target is virtual register x. Hence, $RcmGcn(B)$ is the set of all instructions that appear in block B and whose values are rematerializable and not killed/modified. $RcmKill(B)$ is the set of all instructions whose values are rematerializable and targets are defined in B.

Graph construction

Once we have computed the data-flow analysis for rematerialization, we can set the rematerialization attribute for each live ranges. The attribute for a live range that does not span across blocks (local) can simply be determined by checking if the value of the live range is rematerializable or not. If a live range $lr_1(x)$ is live at the entry of a block B_1, and there is only one definition $inst_x$ in $RemIn(B_1)$, the rematerialization attribute of $lr_1(x)$ is set. Likewise, for a live range $lr_1(x)$ that is defined in a block B_1 and live at the exit of a block B_1, the rematerialization attribute of $lr_1(x)$ is set if $inst_x$ is in $RemOut(B_1)$ (there can only be at most one $inst_x$ from B_1 in $RemOut(B_1)$).

The rematerialization attribute is propagated, like has_def, during the graph fusion phase. When two live ranges, lr_1 and lr_2 are combined, the rematerialization attribute of the combined live range lr_{12} is non-rematerializable if one of them is non-rematerializable.

$$rematerialization(lr_{12}) = rematerialization(lr_1) \wedge rematerialization(lr_2)$$

Shuffle code insertion

On an edge $E = \langle B_1, B_2 \rangle$, shuffle code is inserted for a virtual register x that has been split on E, i.e., if $lr_1(x)$ and $lr_2(x)$ have not been assigned the same storage location. Again we use the notations $r1_x$ and $r2_x$ to indicate the two registers that are assigned to $lr_1(x)$ and $lr_2(x)$ respectively. There are three cases to consider when register allocation inserts shuffle code for rematerializable live ranges.

- *shuffle store* ($lr_2(x)$ is spilled): A shuffle store was supposed to be inserted on edge E to store the value of $lr_1(x)$ as shown in Figure 8.2(a). If $lr_2(x)$ is rematerialized, then the shuffle store is not necessary and removed (as depicted in Figure 8.2(d)) because $lr_2(x)$ does not access memory.
- *shuffle load* ($lr_1(x)$ is spilled): A shuffle load was supposed to be inserted on edge E to load the value of $lr_1(x)$ as shown in Figure 8.2(b). If $lr_1(x)$ is rematerialized, the shuffle load is then replaced with the instruction that recomputes the value of $lr_1(x)$ as depicted in Figure 8.2(e).
- *shuffle move*: A shuffle move was supposed to be inserted on edge E to copy the value of $lr_1(x)$ to $lr_2(x)$ as shown in Figure 8.2(c). If $lr_1(x)$ is rematerializable and the definitions of $lr_1(x)$ can be eliminated, the shuffle move is then replaced with the instruction that recomputes the value of $lr_1(x)$ (as depicted in Figure 8.2(f)). Section 8.1.3 lists the conditions for eliminating definitions of a rematerializable live range.

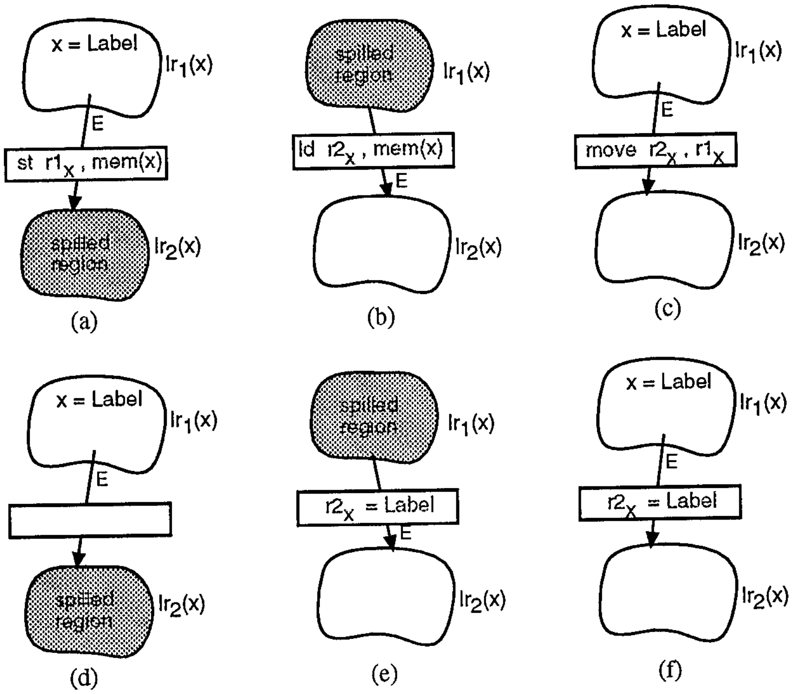


Figure 8.2: Shuffle code for rematerialization.

8.1.3 Optimization of rematerialization

There is one optimization that we can do to improve the result of rematerialization by eliminating the instruction $inst_x$ of a live range lr_x. Removing $inst_x$ of lr_x must meet three conditions:

- lr_x is rematerializable.
- lr_x contains no use (only the definitions).
- Every spilled adjacent live range of lr_x that uses lr_x's value is rematerialized (no shuffle store is inserted).

8.1.4 Examples of rematerialization

The example of Figure 8.3(a) is taken from [10]. Assume that we use loops (or blocks) as regions. Register allocation first performs the data-flow analysis of propagating rematerialization tags; then it identifies that $lr_1(p)$ and $lr_2(p)$ are rematerializable, but not $lr_3(p)$ (two distinct definitions reach B_3). Consider the case where B_2 has a lot of register pressure, but B_1 and B_3 don't. The register allocator chooses $lr_2(p)$ to spill and rematerializes it during the graph simplification phase. After fusing graphs, we obtain the result in Figure 8.3(b). Live ranges of p are split on <B_1,B_2> and <B_2,B_3>. The shuffle store on <B_1,B_2> is eliminated; the shuffle load on <B_2,B_3> is replaced with "p=Label" (in B') which recomputes the value of p. The instruction "p=Label" in B_1 satisfies all three conditions and is removed. Consequently, fusion-based register allocation with the rematerialization analysis is able to obtain the result in Figure 8.3(c) which is the "ideal" result described in [10].

Moreover, if the register allocator keeps track of the three conditions, the register allocator can perform the optimization and removes lr_x while fusing graphs rather than after fusing all graphs. Removing lr_x during the graph fusion phase alleviates register pressure because lr_x does not occupy one register resource. As a result, subsequent graph fusion could benefit from the optimization (the resulting graph contains one live range fewer than otherwise).

The example in Figure 8.4(a) shows a case where the fusion-style approach provides a better framework than Briggs et al. The code consists of two loops. The number next to

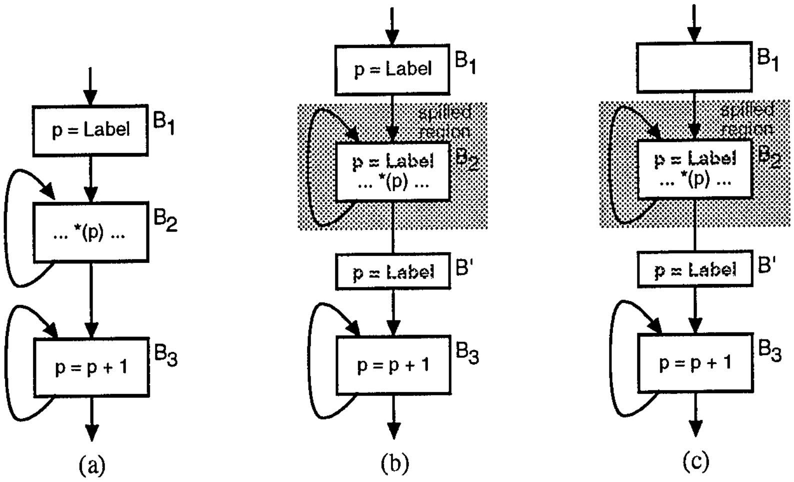


Figure 8.3: Example of rematerialization.

a control-flow edge is execution frequency. We notice that the code is not loop intensive. On average, the trip count of the first loop is 2; the trip count of the second loop is 4. The shaded region is the most frequently executed path. Assume that the code in B_3 causes high register pressure, and register allocation needs to spill one live range. Briggs et al. split the live range of p on $<B_5,B_6>$ and spill p to memory in B_1, B_2, B_3, B_4, and B_5. The splitting and spilling decisions result in 110 overhead cycles (100 cycles of shuffle cost on $<B_5,B_6>$ and 10 cycles for "p=Label" of B_3) as shown in Figure 8.4(b).

If fusion-style coloring chooses loops as regions, shuffle code on $<B_5,B_6>$ is inevitable because register pressure of B_3 causes the live range of p to be spilled to memory in the first loop, like the result of Figure 8.4(b). Hence, there are 100 overhead cycles (100 for shuffle code, 10 for "p=Label" of B_3, and -10 for eliminating "p=Label" of B_1). However, if fusion-style coloring chooses blocks as regions, the graph-fusion framework is able to identify and isolate the troublesome block B_3, and therefore fusion-style coloring spills p only in B_3. Consequently, there are only 20 overhead cycles (10 for "p=Label" of B_3 and 10 for shuffle cost on $<B_3,B_5>$ as shown in Figure 8.4(c)).

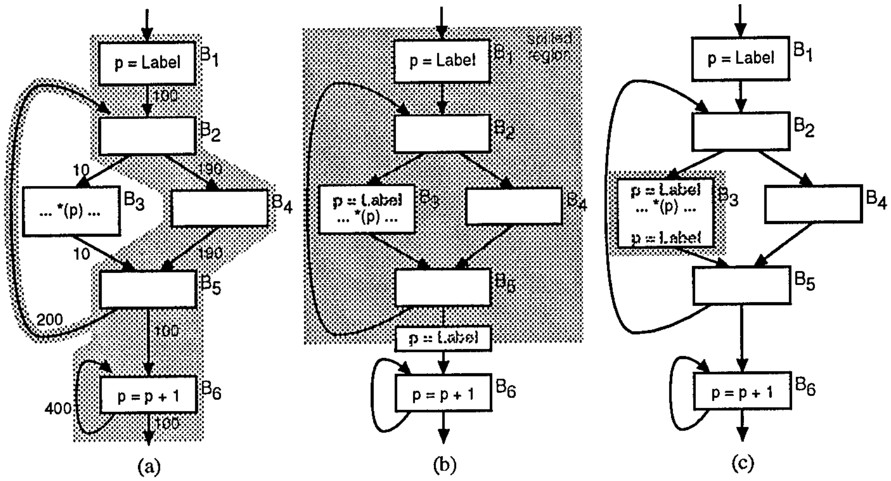


Figure 8.4: Example of shuffle code.

8.2 Coalescing

During compilation, a compiler may generate unnecessary copy/move instructions. The compiler can eliminate a copy instruction that moves a data value between two non-conflicting live ranges by coalescing the two live ranges into one. In general, copy/move instructions originate from 4 different sources:

- *users' code*: Users may write an assignment "x = y" which copies y's value to x. Global optimization eliminates the copy assignment by performing copy propagation and dead code elimination. Copy propagation first propagates y to every use of x if possible. Then dead code elimination eliminates the assignment if all uses of x are replaced by y. The assignment remains if not all uses of x are replaced by y.

- *global optimization*: Global optimization may produce copy assignments. For instance, the "a+b" expression of statement "x = a+b" may be a common subexpression. The statement is then replaced by a copy assignment "x = tmp" where tmp is a compiler generated temporary to hold the value a+b.

- *calling convention*: The protocol of passing parameters and function return values between a caller and callee is called calling convention. That is, with a given calling convention, a caller knows where to save outgoing parameters and retrieve function return values. A callee knows where to retrieve incoming parameters and save its return value. Most modern processors have a few registers reserved for the calling convention (argument and function return registers). If an outgoing or incoming parameter resides in a register, moving the outgoing parameter to an argument register or moving an argument register to the incoming parameter is a copy instruction. Likewise, passing function return values may also require copy instructions.

- *live-range splitting*: If register allocation splits live ranges, shuffle moves may be needed to move values at splitting points.

When two live ranges, lr_1 and lr_2, are coalesced into one, the two sets of interference edges, $\varphi(lr_1)$ and $\varphi(lr_2)$, are combined as well. Apparently, the interference graph after coalescing may have different register pressure. Coalescing may reduce register pressure as well as increase register pressure.

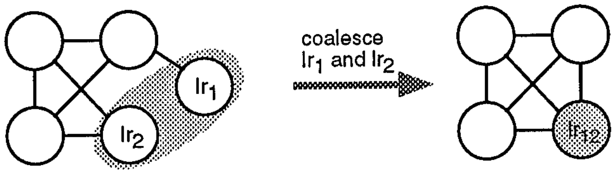


Figure 8.5: Coalescing increases register pressure (N=3).

Figure 8.5 shows a case where coalescing two live ranges increases register pressure. With N=3, we can simplify the interference graph on the left hand side by first removing lr_1 and lr_2. Then the remaining three live ranges become unconstrained and are removed from the graph. The graph can be colored using 3 colors. Coalescing lr_1 and lr_2 produces a clique of size 4, which cannot be colored using 3 colors.

Figure 8.6 shows a case where coalescing two live ranges decreases register pressure. There is a 3-color solution for the interference graph (red color for lr_x and lr_z, green color for lr_y, and blue color for lr_1, lr_2, lr_t, and lr_w). Before coalescing, the graph can not be simplified because simplification blocks as soon as lr_t is removed from the graph (all remaining nodes' degrees $\geq$ 3). Thus, register allocators that rely on simplification to determine the color ordering fail to color the graph with N=3. Optimistic coloring does

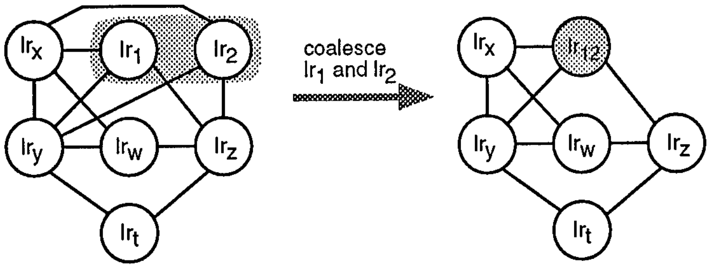


Figure 8.6: Coalescing decreases register pressure (N=3).

not guarantee to find a 3-color solution. For example, simplification of the graph before coalescing blocks as soon as lr_t is removed from the graph. Optimistic coloring may decide to defer spilling lr_x and remove it from the graph and tries to find an available color for lr_x later. One legitimate ordering of simplifying the remaining graph is lr_1, lr_2, lr_w, lr_y, and lr_z. When assigning colors to live ranges in the reverse order, optimistic coloring may assign the red color to lr_z and lr_y, the green color to lr_2, and the blue color to lr_t, lr_w, and lr_1. As a result, there is no available color for lr_x. After coalescing lr_1 and lr_2, 3 duplicate edges, [lr_{12},lr_x], [lr_{12},lr_y], and [lr_{12},lr_x] are eliminated, and a graph G which is less constrained than the original graph is obtained. The graph G can be simplified with N=3 by removing nodes in the order of lr_t, lr_z, lr_{12}, lr_w, lr_x, and lr_y.

8.2.1 Prior approaches

Chaitin et al. eliminate a move instruction when the live ranges of the source and destination do not conflict [13]. After all possible desirable copy instructions are eliminated, the entire interference graph is rebuilt from scratch. Then Chaitin et al. rerun the coalescing module because some copy instructions that at first were impossible to be eliminated may now be redundant. The coalescing module is reiterated until no more copy instruction is eliminated. The problem with the approach is that coalescing live ranges may increase register pressure as shown in Figure 8.5. Therefore, coalescing may render the final resulting graph more constrained, and register allocation results in more spill code than without coalescing.

Briggs uses coalescing, *conservative* coalescing, and biased coloring to eliminate copy instructions [7]. All possible copy instructions except shuffle moves are coalesced using the approach of Chaitin et al. The main goal of splitting live ranges is reducing register pressure. Shuffle moves are coalesced conservatively so as to prevent introducing extra spill code. The conservative coalescing approach coalesces two live ranges under the constraint that the resulting live range has less than N neighbors whose degree ≥ N. The constraint makes sure that the resulting live range is not spilled. Biased coloring is a final attempt to clean up unproductive shuffle moves by assigning the same colors to live ranges that the moves connect. Conservative coalescing may suppress coalescing two live ranges under the condition that the resulting live range may be spilled. In fact, the resulting live range is not spilled. Namely, conservative coalescing is too conservative. Consider the example of Figure 8.6. After we coalesce lr_1 and lr_2 into lr_{12}, we can simplify the resulting without blocking. Conservative coalescing, on the other hand, believes that lr_{12} will be spilled because there are 3 neighbors, lr_x, lr_y, and lr_z, whose degrees ≥ 3.

George and Appel take an *iterated* coalescing approach that integrates coalescing into simplification [28]. Live ranges are categorized as either *move-related* or *non-move-related*. A move-related live range copies a value to/from other live ranges. Simplification removes only non-move-related live ranges. Simplification blocks when the residual graph consists of only move-related live ranges or non-move-related ones whose degrees ≥ N. George and Appel then perform conservative coalescing on the residual graph. The iterated coalescing approach is likely to coalesce more live ranges than the conservative approach because the residual graph is less constrained than the entire graph. A move-related live range may become non-move-related after coalescing. The iterated coalescing approach reiterates from simplification to remove non-move-related live ranges. If no live ranges can be coalesced, and all non-move-related live ranges have degrees ≥ N, then a move-related live range of low degree is chosen to be considered as non-move-related. That is, the iterated approach gives up the hope of coalescing the live range with others and then reiterates from simplification again.

8.2.2 Coalescing in fusion-style coloring

Presently, our register allocator uses different approaches to eliminate copy instructions for Chaitin-style and fusion-style coloring. For Chaitin-style coloring, we coalesce two live ranges connected by a move if they don't interfere. Fusion-style coloring splits live ranges only when necessary to reduce spill code. Hence, coalescing live ranges at splitting points is unnecessary because coalescing them increases spill code (undoes splitting decisions). Therefore, fusion-style coloring uses biased coloring to eliminate copy instructions.

Copy instructions originating from users' code and global optimization are actually *artificial* splitting points and may be on frequently executed paths. Using biased coloring to eliminate these two kinds of copy instructions may not be ideal because biased coloring may fail to eliminate a frequently executed copy instruction. Fusion-style coloring is sensitive to edge ordering and therefore able to handle high register pressure well by splitting live range at infrequently executed edges. These nice properties allow us to have a different viewpoint on coalescing the two kinds of copy instructions. We can aggressively coalesce live ranges without worrying much about the incurred register pressure because fusion-style coloring is able to alleviate the pressure by spilling transparent or splitting live ranges. One feasible implementation of this aggressive coalescing approach is performing Chaitin-style coalescing prior to fusion-style coloring as shown in Figure 8.7. Chaitin-style coalescing contains two steps, graph construction and aggressive coalescing. The graph construction step builds the function-wide graph $G_{function}$. If live ranges are coalesced in the second step, then $G_{function}$ is thrown away, and the two steps are repeated until no more live ranges can be coalesced.

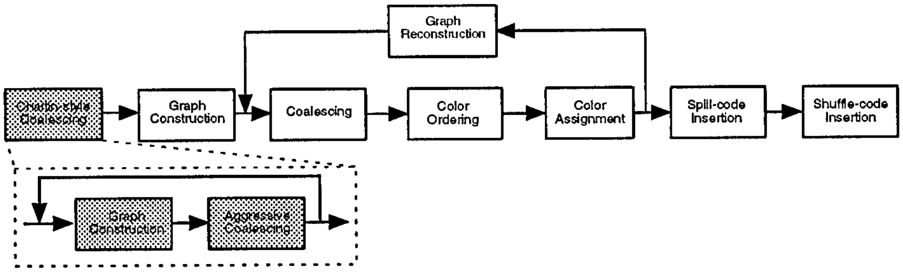


Figure 8.7: Structure of coalescing.

8.3 Summary

In this chapter, I present how to integrate rematerialization and coalescing with fusion-style coloring. Rematerialization and fusion-style coloring cooperate with each other seamlessly. Fusion-style coloring is able to isolate high register-pressure regions so as to avoid rematerialization happening on frequent executed paths. Rematerization helps fusion-style coloring by pruning useless rematerializable live ranges out of the interference graph (alleviates register pressure). Because of the ability of handling high register pressure in fusion-style coloring, it is feasible to eliminate frequently executed copy instructions by spilling transparent live ranges or splitting live ranges at infrequently executed places.

Chapter 9

Conclusions

9.1 Summary

This dissertation has proposed a novel register-allocation approach which is fusion-based. Fusion-style register allocation starts off with constructing regions and applies graph fusion along control-flow edges to combine the interference graphs of regions into the interference graph for the whole function. Graph fusion integrates spilling, splitting, and color binding in a seamless fashion. The delayed spilling technique avoids making premature spilling decisions. Maintaining the colorability invariant splits live ranges only when necessary. No color binding decisions are ever made during graph fusion. More importantly, fusion-style coloring is sensitive to the ordering of control-flow edges that connect regions. Splitting is unlikely to happen at high priority edges (frequently executed) and likely to happen at low priority edges (infrequently executed). As to fusion-style coloring, the register-allocation problem becomes a problem of determining the edge ordering and selecting regions. Fusion-style coloring provides a nice framework for register allocation which models various live-range splitting approaches such as the Tera, SSA, PDG, and Multiflow approaches.

This dissertation has presented how rematerialization and coalescing can be integrated with fusion-style coloring. Fusion-style coloring is able to leverage rematerialization by isolating the high register pressure region and avoiding inserting rematerialization code into frequently executed paths. Rematerialization is also able to leverage fusion-style coloring by pruning unnecessary rematerializable live ranges during graph fusion to alleviate register pressure so as to benefit subsequent graph fusion by lowering register pressure. Fusion-style coloring allows register allocation to split live ranges at infrequently executed edges or spill low-cost live ranges to trade off against frequently executed copy instructions.

It is very important to look at register allocation from the compiler's perspective because the compiler is more interested in how the register allocator interacts with the rest of the compiler. This dissertation has shown that fusion-style register allocation provides a framework that is well-prepared for region-based compilation. Fusion-style coloring is able to deal with the register pressure caused by scope expanding transformations such as unrolling and inlining, whereas the traditional Chaitin-style coloring breaks down. In other words, fusion-style coloring provides a solution to region-based compilation so that register allocation is no longer an obstacle to region-based compilation.

This dissertation has presented and evaluated several enhancements to register allocation. Chaitin-style register allocation with these enhancements integrates both Chaitin-style and priority-based coloring into one so that one's strengths compensates the other's weaknesses. The experimental results have shown that Chaitin-style register allocation with the enhancements actually outperforms some existing approaches that have been adopted by researchers.

9.2 Future work

Even though this dissertation has demonstrated that fusion-style coloring is superior to other register-allocation approaches, there still remain some future work to be done. Two important areas require investigations and evaluations:

First, integrating fusion-style coloring into interprocedural register allocation is possible and may provide pleasing results. Consider a call graph with 3 functions, f, g, and h. f calls g; g calls h. Assume that h is compiled and requires 2 callee-save registers, pr_1 and pr_2. Interprocedural register allocation can treat pr_1 and pr_2 as *caller-save* register instead of callee-save. If g uses the two registers, then save/restore code are have to be inserted around the call site $h()$. However, the call site $h()$ may be frequently executed. Because fusion-style coloring is edge-ordering sensitive, it provides a nice framework to deploy register save/restore code on less frequently executed paths. The places of saving/restoring

pr_1 and pr_2 can be moved further away from the call site $h()$ as graphs are fused.

Second, another important area is to integrate fusion-style coloring with code scheduling. If there are more than N non-transparent live ranges simultaneous live at one point, we know that the register allocator must spill some live ranges to lower the register pressure. If the code scheduler is sensitive to register allocation in the first place, then register allocation can avoid spill and shuffle code. For instance, when scheduling code, the scheduler makes sure that there are no more than N simultaneously live non-transparent live ranges at any given point. Whole or partial live ranges are allowed to be spilled to satisfy the condition during code scheduling. Because spill code is produced during code scheduling, the spill code can be scheduled right away instead of after register allocation.

Bibliography

[1] A. Adl-Tabatabai, T. Gross, and G. Y. Lueh. Code reuse in an optimizing compiler. In *Proc. SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 51–68. ACM, October 1996.

[2] Jeffrey D. Ullman Alfred V. Aho. *Principles of Compiler Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[3] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 241–249. ACM, June 1988.

[4] D. Alpert and D. Avnon. Architecture of the pentium microprocessor. *IEEE MICRO*, pages 11–21, June 1993.

[5] 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 *Proc. ACM SIGPLAN '89 Conf. on Prog. Language Design and Implementation*, pages 258–263. ACM, July 1989.

[6] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iwarp: An integrated solution to high-speed parallel computing. In *Proc. Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

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

[8] P. Briggs. Personal communication, 1996.

[9] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN'89 Conf. on Prog. Language Design and Implementation*, pages 275–284. ACM, July 1989.

[10] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proc. ACM SIGPLAN '92 Conf. on Prog. Language Design and Implementation*, pages 311–321. ACM, June 1992.

[11] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proc. ACM SIGPLAN Sym. on Compiler Construction*, pages 152–161. ACM, July 1986.

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

[13] 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.

[14] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.

[15] F. Chow. *A Portable, Machine-independent Global Optimizer – Design and Measurements*. PhD thesis, Stanford University, 1984.

[16] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM Trans. on Prog. Lang. Syst.*, 12:501–535, Oct. 1990.

[17] F.C. Chow. Minimizing register usage penalty at procedure calls. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 85–94. ACM, June 1988.

[18] R. Cohn and P.G. Lowney. Hot cold optimization of large windows/nt applications. In *Proc. Micro 29*, pages 80–89. ACM, 1996.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT-Press, 1990.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

[21] Richard Sites (editor). *Alpha Architecture Reference Manual*. Digital Equipment Corp., Burlington MA, 1992. EY-L520E-DP.

[22] J. Z. Fang. Compiler algorithms for if-conversion, speculative predicate assignment and predicated code optimizations. In *9th International Workshop, Languages and Compilers for Parallel Computing*. Springer, August 1996.

[23] 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 Prog. Languages and Operating Systems (ASPLOS V)*, pages 85–97. ACM, October 1992.

[24] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[25] 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.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[27] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[28] L. George and A. W. Appel. Iterated register coalescing. In *Conf. Record of the 23th Annual ACM Symp. on Principles of Prog. Lang.*, pages 208–218. ACM, January 1996.

[29] D. W. Goodwin and K. D. Wilken. Optimal and Near-Optimal Global Register Allocation Using 0-1 Integer Programming. *Software-Practice and Experience*, 26:930–965, August 1996.

[30] D. Grove and L. Torczon. A study of jump function implementations. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 90–99. ACM, June 1993.

[31] T. Halfhill. Amd vs. superman. *Byte*, pages 95–104, November 1994.

[32] T. Halfhill. Intel's p6. *Byte*, pages 42–58, April 1995.

[33] R. Hank, W. Hwu, and B. Rau. Region-based compilation: An introduction and motivation. In *Proc. 28th Annual ACM/IEEE Intl. Symp. on Microarchitecture*, pages 158–168, Ann Arbor, Nov 1995. ACM/IEEE.

[34] R. E. Hank. *Region-based compilation.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[35] J. L Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufman, 1990.

[36] W. W. Hwu and P. P. Chang. Inline function expression for compiling realistic C programs. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 246–257. ACM, June 1989.

[37] 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.

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

[39] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[40] G. Kane and J Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[41] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN'92 Conf. on Prog. Language Design and Implementation*, pages 224–234. ACM, June 1992.

[42] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Prog. Lang. Syst.*, 16(4):1117–1155, July 1994.

[43] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN'94 Conf. on Prog. Language Design and Implementation*, pages 147–158. ACM, June 1994.

[44] P. Kolte and M. J. Harrold. Load/store range analysis for global register allocation. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Language Design and Implementation*, pages 268–277. ACM, June 1993.

[45] S. M. Kurlander and C. N. Fischer. Zero-cost range splitting. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Language Design and Implementation*, pages 257–265. ACM, June 1994.

[46] J. R. Larus and P. N. Hilfinger. Register allocation in the spur lisp compiler. In *Proc. ACM SIGPLAN Sym. on Compiler Construction*, pages 255–263. ACM, July 1986.

[47] 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.

[48] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. Technical Report 96-106, Carnegie Mellon University, School of Computer Science, March 1996.

[49] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[50] W.G. Morris. Ccg: A prototype coagulating code generator. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 45–58. ACM, June 1991.

[51] Masood Namjoo and Anant Agrawal. Implementing sparc: A high-performance 32-bit risc microprocessor. *SunTechnology*, Winter, 1988.

[52] B. R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 40–52. ACM, June 1990.

[53] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Language Design and Implementation*, pages 266–277. ACM, June 1994.

[54] T. A. Proebsting and C. N. Fischer. Probablistic register allocation. In *Proc. ACM SIGPLAN '92 Conf. on Prog. Language Design and Implementation*, pages 300–310. ACM, June 1992.

[55] O. Waddell R.G. Burger and R.K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 130–138. ACM, June 1995.

[56] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pages 28–39. ACM, June 1990.

[57] S. Song, M. Denman, and J. Chang. The powerpc 604 risc microprocessor. *IEEE MICRO*, pages 8–17, October 1994.

[58] P.A. Steenkiste and J.L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for lisp. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[59] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

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

[61] D. W. Wall. Global register allocation at link time. In *Proc. ACM SIGPLAN '86 Symp. on Compiler Construction*, pages 264–275, Palo Alto, June 1986. ACM.

[62] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, April. 1991.