Issues in Register Allocation by Graph Coloring

Guei-Yuan Lueh

November 1996
CMU-CS-96-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This technical report addresses some issues in register allocation by graph coloring and presents three improvements, storage-class analysis, priority-based simplification and preference decision. The influence of the three improvements to graph coloring is discussed in this report. Comparisons of various register allocations are discussed as well.

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

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

1 Introduction

Graph coloring is a common approach to model register allocation. Register allocator assigns registers (colors) to live ranges in such a manner that *conflicting* live ranges — i.e., live ranges that are simultaneously live in the program — are kept in different registers. The compiler constructs an *interference graph* whose nodes represents live ranges and whose edges connect live ranges that conflict. This graph captures the information about conflicts between live ranges; the register allocator attempts to find a legal N-coloring of the graph, with N the number of physical registers.

Chaitin's graph-coloring algorithm forms the basis of many practical register allocators [6]. Chaitin's algorithm consists of two phases, *simplification* and *register assignment*. Simplification is based on the observation that if a vertex V has degree $< N$, then the register assignment phase guarantees to find a color for V, because at least one legal color remains regardless what colors are assigned to V's neighbors. The key idea then is that the graph is *N-colorable* if the residual graph obtained by deleting V and all edges that are incident upon V is also N-colorable. This process of removing *unconstrained* live ranges from the graph, i.e., vertices that have degree less than N, is called simplification; it attempts to reduce the interference graph to an empty graph. Simplification *blocks* when all live ranges of the graph have degree $\geq N$. In this case, a live range is picked to spill, i.e. the corresponding node and all its edges are removed from the graph, until simplification can proceed further. The choice of live range depends on some heuristic estimate of the cost of moving this live range to memory. Simplification terminates when an empty graph is obtained.

Register assignment follows the simplification phase and picks physical registers (colors) for live ranges. It assigns colors to the live ranges in the *reverse* order in which the live ranges were removed during simplification. Chaitin's algorithm provides a conceptually simple way to pack live ranges into registers.

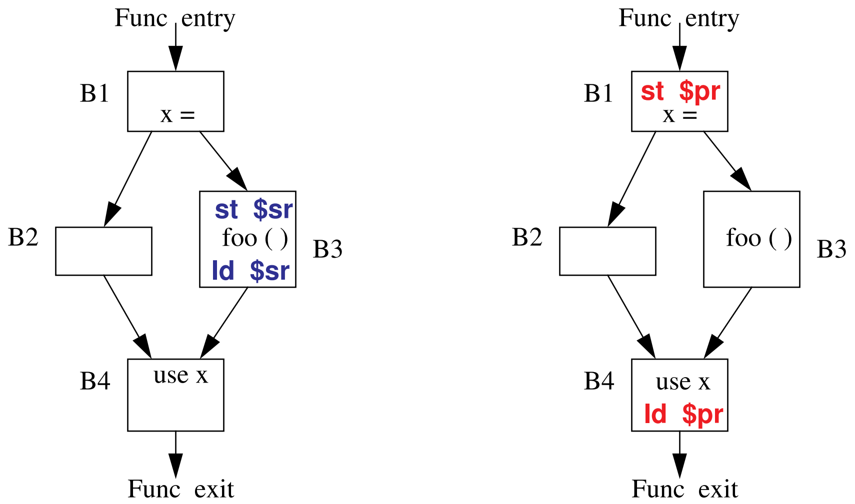
In this paper, we report on an empirical evaluation of Chaitin-style approaches to register allocation which are implemented in the `cmcc` optimizing C compiler [1]. A number of improvements have been suggested in the literature (see Section 2), but no comprehensive evaluation is available.

1.1 Cost model for register allocation

Our basic machine model is a RISC processor that requires that the operands of all operations reside in registers. (All our measurements are done for the MIPS 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; no register has been assigned to this value), (ii) call cost (to free up/restore registers upon procedure entry/exit), and 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 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 [7]. There is a distinct cost associated with each kind of register assigned to a live range. When a live range lr ends up in a caller-save register, we must pay the cost of saving and restoring lr's value at all function calls that are crossed by lr. For instance, x in Figure 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 to be saved (restored) at the entry to (exit from) the function (Figure 1 (b)), because the register allocator assumes that all callees of the current function demand all callee-save registers after the exit of the function.



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

Figure 1: Caller-save and callee-save costs.

Figure 2 depicts the register allocation cost for 2 programs from the SPEC92 suite, `eqntott` and `ear`, for various combinations of caller-save and callee-save registers. The experimental data shown in this report are based on the `cmcc` code generator for the MIPS which produces code that is competitive with the native MIPS compiler and the GNU gcc compiler for this machine. 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 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. 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, by default, the register allocator uses (6,4,0,0).

From Figure 2, we see that giving the register allocator more register 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.

In this report, we discuss three improvements to Chaitin-style register allocation which consider not only spill cost but also call cost. Figure 3 shows the effect of these improvements on the programs from Figure 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).

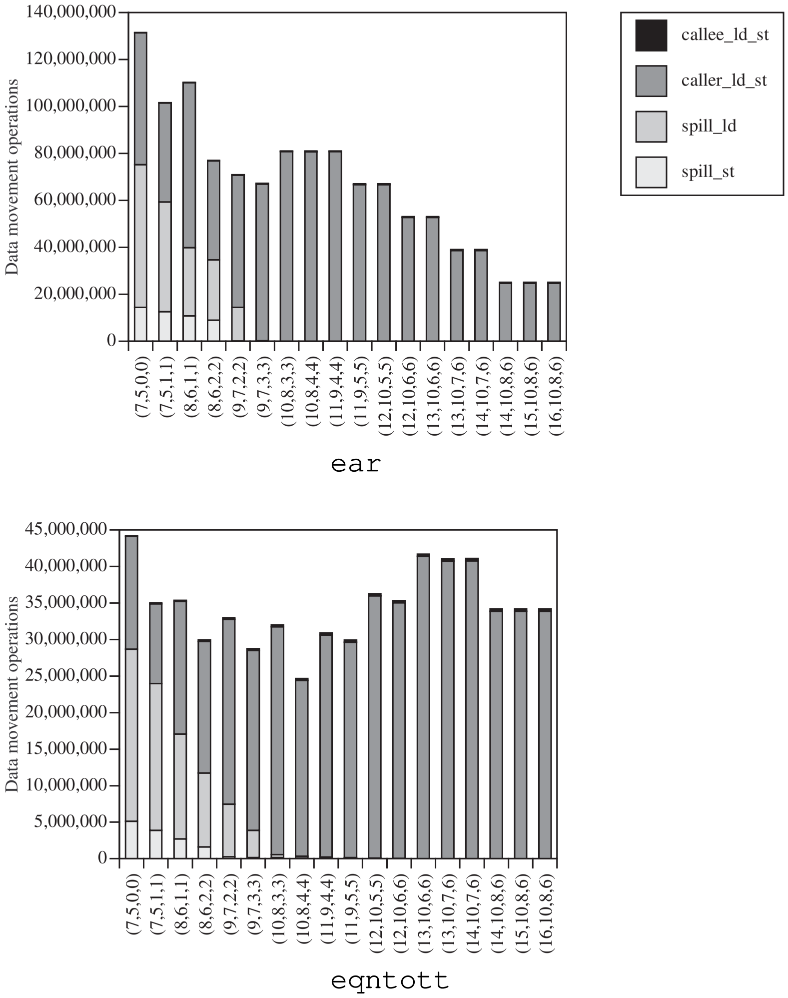


Figure 2: Register allocation cost.

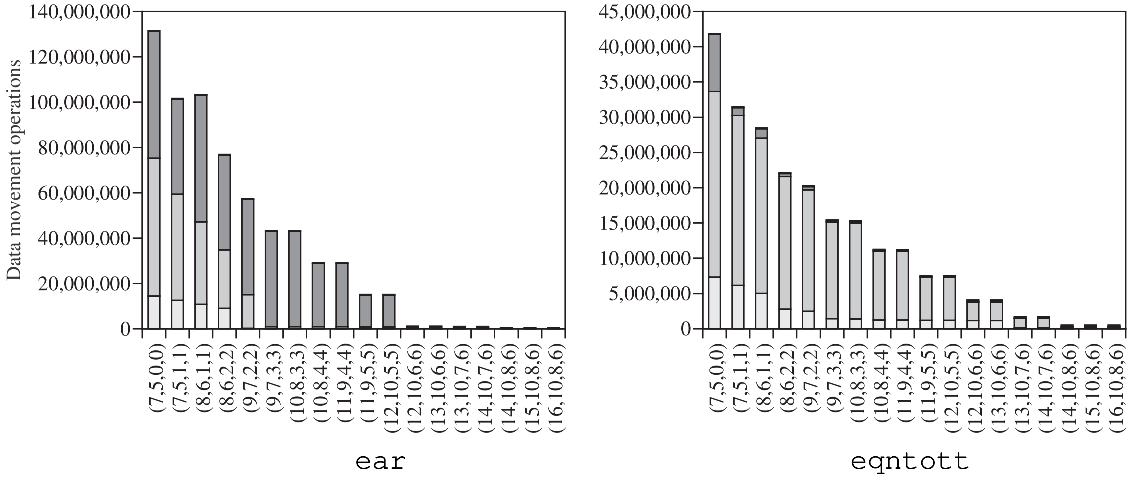


Figure 3: Register overhead for improved register alloction.

2 Prior work

A number of improvements over Chaitin-style register allocation have been suggested. In this section, we discuss the major known register-allocation strategies, with a focus on which kind of register overhead they attempt to reduce.

2.1 Reducing spill cost

In eseence, there exist two ways to reduce the 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).

2.1.1 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. The heuristic that chooses which live ranges to spill has a direct effect on the quality of the register allocation, so various heuristics have been investigated.

- Simplification is a heuristic approach to coloring, and as such may miss legal coloring opportunities. Optimistic coloring [3] improves simplification by attempting to assign colors to live ranges that would have been spilled by the basic algorithm. Optimistic coloring delays spill decisions until the register assignment phase. Spilling decisions are made during the register assignment phase: when no legal color exists for the next live range to be colored (instead of when the simplification blocks), this live range is spilled.

- Priority-based coloring [7] and probabilistic register allocation [13] assign registers to live ranges based on priority functions. The priority-based approach uses a priority function that captures the savings in the number of memory accesses of a live range. The priority function used in the probabilistic approach is the probability that a variable reference is allocated to a register. Both approaches make sure to assign registers to the most important live ranges and spill the least important ones if necessary.

- The register allocation algorithm used in the RS/6000 compiler [2] 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 L is selected for spilling, instead of inserting a load before each use and a store after each definition of L, the register allocation uses the *cleaning* technique that attempts to insert at most a single load and store inside each basic block. In effect, L is split into segments that span at most a basic block.

- Rematerialization [4] spills live ranges that are cheaper to recompute than to store/load them back/from memory, e.g., loading constant values or computing addresses.

2.1.2 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 benefit of splitting is that it reduces the degree of the interference graph. Furthermore, splitting can take the structure of the program into account; it allows the spilling of only those live range segments that span program regions of high register pressure.

- The priority-based coloring [7] splits a live range L when no legal color exists for L, i.e., when all N colors have been taken up by L's neighbors. Splitting takes place during the coloring process.
- Register allocation for the Tera compiler (as described in [5]) and probabilistic register allocation [13] perform register allocation hierarchically (from the innermost loops to the outermost loops). Loops are the natural splitting points of live ranges. As coloring is done hierarchically, shuffle code tends to be outside of the innermost loops.
- The RAP compiler [12] 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 [9]. The trace scheduler picks a trace and then passes it to the code scheduler; the code scheduler then performs register allocation and scheduling together. 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.
- Coagulation code generation [11] 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 [9].
- Kurlander and Fischer [10] 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.

2.2 Reducing call cost

A register may not be free at procedural boundaries, or at call sites (caller-save) resp. 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.

2.2.1 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.

- Priority-based register allocation [7] uses a priority function, which takes caller-save and callee-save cost into consideration. The register allocation assigns the kind of register to a live range which yields the maximum savings.

- The generic callee-save convention saves/restores the values of the 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 the callee-save registers are never used during the execution of the function. Chow uses *shrink-wrapping* [8] 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 [14]. Measurements show that over two thirds of procedural activations actually make no calls [14]. 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. A shuffle code (register move) is needed to move a value between two different classes of register.

2.2.2 Inter-procedural allocation

Inter-procedural register allocation usually assigns registers in a hierarchical manner over the call graph such that the 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 [17]. 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 those of f 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 reallocation once the register allocator assigns a register to a variable.

- Chow extends the priority-based approach to an inter-procedural register allocation that constructs a call graph and compiles functions from leaves to the root [8]. The usage information of callee-save registers is propagated to the upper regions of the call graph. A function tries to avoid using the same callee-save registers used by its lower regions on the call graph such that save/restore operations of the registers become redundant and can therefore be eliminated.

- Steenkiste develops an inter-procedural register allocation in the context of a LISP compiler[16]. 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 [8]. 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 [15] 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 spill code for callee-save registers to less frequently-executed functions.

3 Improvements

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 (i.e., resulting in high callee-save cost) or function calls are on the most frequently executed paths (i.e., causing high caller-save cost). 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 section, we discuss three optimizations that are effective on selecting the right kind of register for a live range.

3.1 Storage-class analysis

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

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

Each storage class has an associated cost. Storage-class analysis decides where live ranges reside, based on two functions, $f_{benefit\_caller}$ and $f_{benefit\_callee}$ that 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, $f_{benefit\_caller}(lr)$ (resp. $f_{benefit\_callee}(lr)$) is defined as the weighted reference counts of the spill code minus the weighted caller-save (resp. callee-save) cost. That is, these two functions indicate the estimated number of 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 priority-based approach [7], except the benefits are not normalized by the size of live ranges.

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

The simplification phase guarantees that the register assignment phase finds registers for live ranges that are on the color stack. However, sometimes *not* using a register (i.e., spilling a live range) is better than using the wrong kind of register. Figure 4 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 4 (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 4 (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 $f_{benefit\_caller}(lr) < 0$, then spilling lr reduces load/store counts by $\mid f_{benefit\_caller}(lr) \mid$. While assigning registers to live ranges, the register assignment phase makes spill decisions right away for the live ranges that are supposed to get caller-save registers (i.e., spill when $f_{benefit\_caller}(lr) < 0$).

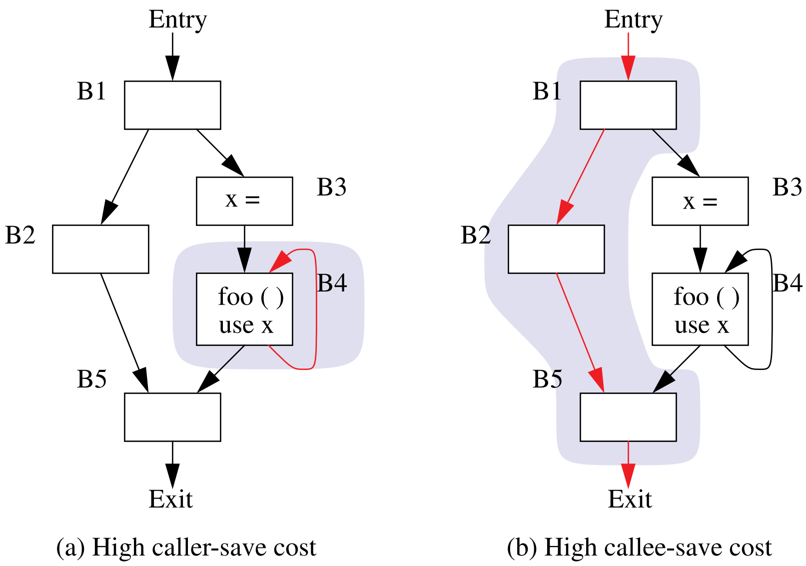


Figure 4: Storage-class analysis.

There are two ways to make spill decisions for 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 spill decision then is similar to making the decision to use a caller-save register. That is, if $f_{benefit\_callee}(lr) < 0$, 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). Priority-based coloring [7] uses the same approach to compute the priority function of a live range for callee-save registers. 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 costs are spread over all users. Spill decisions for live ranges that (potentially) get a callee-save registers are not finalized until the register 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 SPEC 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 spill cost for each live range 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.

3.2 Priority-based simplification

Simplification removes live ranges one by one from the interference graph and pushes them to the color stack. The reverse ordering in which live ranges are pushed to the stack 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 register assignment. If the target machine has only one kind of registers, 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 the color stack plays an important role in reducing the register overhead.

Recall that each live range has two benefit functions, $f_{benefit\_caller}$ and $f_{benefit\_callee}$, which determine the preferred kind of register. Live ranges on top of the color stack have a higher chance to obtain the register of their choice, because fewer registers are already taken by other live ranges. For example, Figure 5 (a) depicts an interference graph consisting of 3 live ranges. All of them prefer using callee-save registers. 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 5 (b) (obtained by removing lr_y, lr_z, and then lr_x). The top two live ranges, lr_x and lr_z, get callee-save registers during the coloring 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_x, lr_y, and then lr_z, so that lr_x and lr_y can obtain the two callee-save registers; this assignment actually saves 4100 load/store operations.

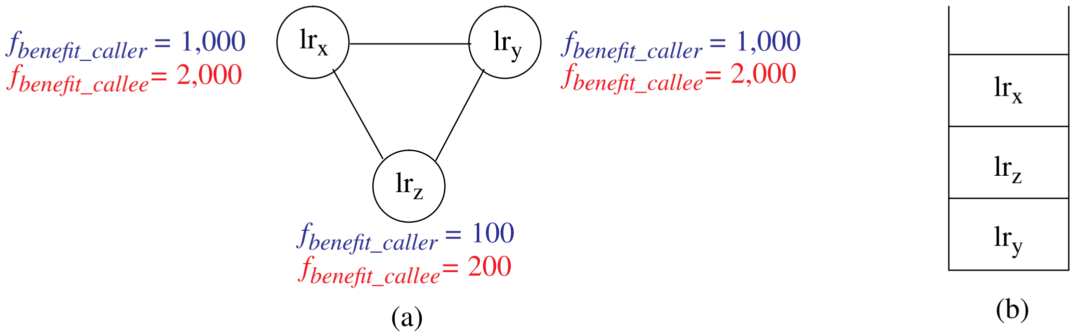


Figure 5: Effect of simplification order, with $N = 3$ registers (2 callee-save and 1 caller-save).

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 the color stack as a priority-based color stack; the higher the position on the stack, the higher the priority with regard to picking registers. Hence register allocation based on this color stack is now similar to priority-based coloring [7]. Chaitin-style coloring and priority-based coloring, however, differ in two ways: (1) Chaitin's color stack guarantees colorability (every one node on the stack gets a color) but priority-based coloring does not (however, it makes sure that the higher-priority live ranges get registers), and (2) priority-based coloring can split live ranges but Chaitin's algorithm cannot. We implement priority-based coloring that does not split live ranges because we want to measure the influence of the two distinct color orderings on the register overhead. In Section 5, we provide a comparison between the two approaches.

The priority, which determines the ordering of unconstrained live ranges during simplification, is based on the two benefit functions. The priority functions that we choose to experiment are defined as follows:

- max$(f_{benefit_caller}, f_{benefit_callee})$
-

$$\begin{cases} \mid f_{benefit\_caller} - f_{benefit\_callee} \mid & \text{if } f_{benefit\_caller} \geq 0 \text{ and } f_{benefit\_callee} \geq 0 \\ f_{benefit\_caller} & \text{if } f_{benefit\_caller} \geq f_{benefit\_callee} \text{ and } f_{benefit\_callee} < 0 \\ f_{benefit\_callee} & \text{if } f_{benefit\_callee} > f_{benefit\_caller} \text{ and } f_{benefit\_caller} < 0 \end{cases}$$

Priority-based coloring uses the first approach as the priority function so as to make sure that the live range with maximum savings has the highest priority to own a register. However, this approach when implemented

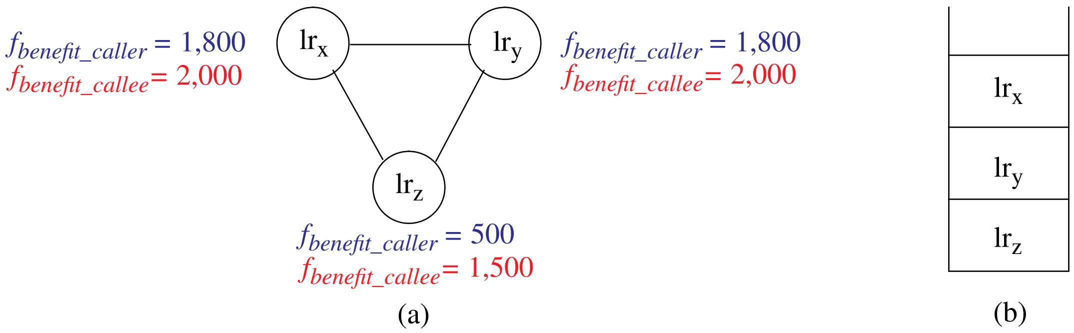


Figure 6: Priority with $N = 3$ registers (2 callee-save and 1 caller-save).

as part of priority-based simplification, increases the register overhead for some SPEC programs, compared to register allocation without using priority-based simplification. This approach is not suitable for Chaitin-style register allocation because simplification guarantees that all live ranges on the color stack can find registers; in other words, making sure that live ranges with bigger savings own registers is *not* a concern for Chaitin-style register allocation. 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 definition is used.

Consider the example in Figure 6 that illustrates how the priority functions make a difference in register allocation overhead. The interference graph in Figure 6 (a) comprises 3 live ranges, lr_x, lr_y, and lr_z. Using the first approach, lr_x and lr_y have higher priorities (2000) than lr_z (1500). The color stack after simplification is shown in Figure 6 (b). Because all three live ranges prefer using callee-save registers, lr_x and lr_y are assigned the two callee-save registers. The total savings of load/store operations are 4500. The second approach discovers that lr_z has a higher priority than lr_x and lr_y, because the loss of using the wrong kind of register is higher. Thus lr_z ends up on top of the color stack. The second approach yields a better allocation, which results in a total savings of 5300 load/store operations.

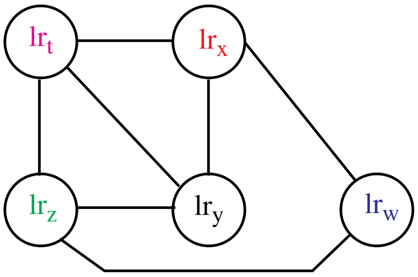
3.3 Preference decision

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 analysis tries to prevent this situation from happening. Albeit 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 register assignment phase, however, examines only the two benefit functions, $f_{benefit\_callee}$ and $f_{benefit\_caller}$, 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 register assignment phase is the cause of unnecessary overhead operations.

Therefore, we provide a separate phase prior to register assignment that pre-determines 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. If the number of the live ranges (L) is less than or equal to the number of available callee-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. The $L - M$ live ranges with the lowest priority are annotated so that the register assignment phase can attempt to find caller-save registers before considering callee-save registers. The priority is defined as:

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

$caller\_cost(lr)$ is the incurred overhead for lr using caller-save registers which is equal to $spill\_cost - f_{benefit\_caller}$. For those live ranges with $f_{benefit\_caller} > 0$, $caller\_cost$ is used as priority ($caller\_cost$ is the overhead). For the live ranges whose $f_{benefit\_caller} \leq 0$, $spill\_cost$ is used as priority, 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.



(a)

	lr_t	lr_x	lr_y	lr_z	lr_w
$f_{benefit\_caller}$	100	100	100	100	100
$f_{benefit\_callee}$	200	400	200	200	4000

(b)

Figure 7: Preference decision example with $N = 3$ registers (2 callee-save and 1 caller-save).

Figure 7 (a) depicts an interference graph with 5 live ranges. The two benefit functions of each live range are listed in the table of Figure 7 (b). Callee-save registers are the precious resources in this example (all live ranges are competing for callee-save registers). Based on the simplification priority defined in Section 3.2, the priority order is 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_w, lr_z, lr_t, lr_y, and lr_x. Then, lr_x and lr_y take away the two callee-save registers; this register assignment leaves no callee-save register but the caller-save register for lr_t and lr_w to use (lr_z gets the same register as lr_x). This assignment saves 1000 load/store operations (200 and 800 for using caller-save and callee-save registers, respectively). If live ranges lr_x and lr_w contain the same function call, then lr_x is forced to reside in a caller-save register instead of a callee-save register. This allocation then saves 4600 load/store operations (lr_x and lr_z get the caller-save register, and the rest gets callee-save registers).

4 Evaluation

Section 3 describes a number of enhancements and improvements to coloring-based register allocation. In this section we report on an empirical evaluation. We measure the influence of different combinations of the improving techniques for various SPEC programs: (`alvinn`, `compress`, `ear`, `eqntott`, `espresso`, `li`, `sc`, `doduc`, `fpppp`, `matrix300`, `nasa7`, `spice` and `tomcatv`). We pay special attention to the issue of deciding between caller-save and callee-save register for a live range. That is, for live ranges that do not cross function calls, the register allocator attempts to find caller-save registers before looking for a callee-save registers, because caller-save registers incur no overhead cost. Likewise, live ranges that contain function calls prefer callee-save over caller-save registers. The y-axis of the figures in this section shows the register overhead produced by a base Chaitin-style register allocator divided by the overhead of an improved-version Chaitin-style register allocator using various combinations of the enhancements (storage-class analysis, priority-based simplification and preference decision). The bigger this number, the less overhead is there in the improved version. SC, PS and PR stand for Storage-Class analysis, Priority-based Simplification and PReference, respectively.

All experiments use dynamic execution-frequency information. The issue of dynamic versus static information is discussed in Section 4.1. Looking at the SPEC programs, we can classify them into 4 classes.

- Each optimization contributes a significant fraction of improvement. Examples are `nasa7` and `ear` shown in Figure 8. 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.

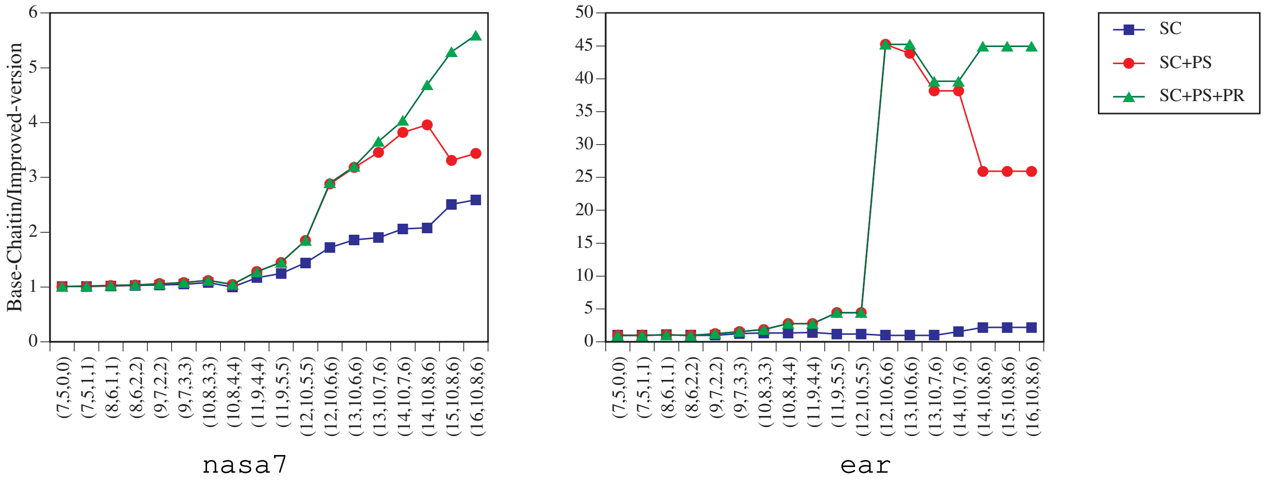


Figure 8: Improvement as function of register pressure.

- 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 9.

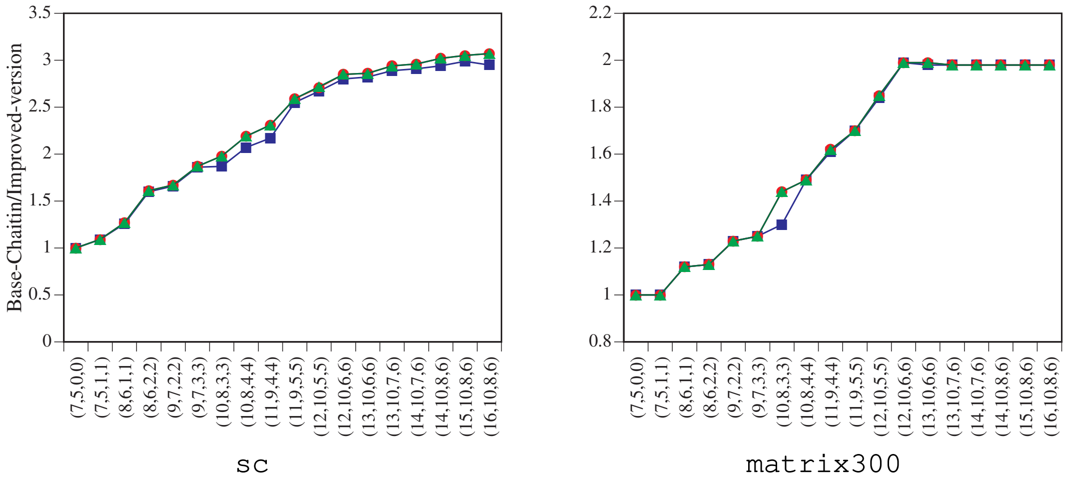


Figure 9: Improvement as function of register pressure.

- Pre-determining the preferred kind of registers for live ranges does not effect the overall number of overhead operations. Examples of this class are `compress`, `eqntott` (shown in Figure 10), `espresso`, `fpppp`, `doduc`, and `spice`.

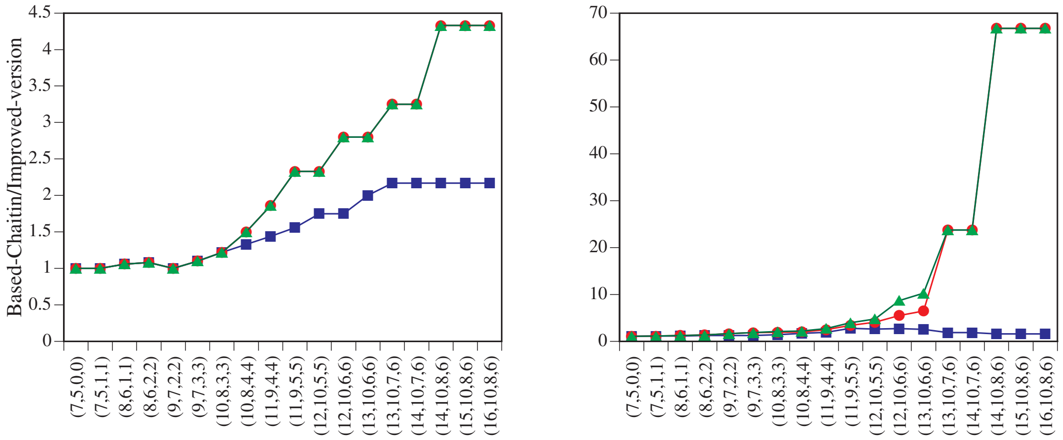


Figure 10: Improvement as function of register pressure.

- For programs that have low callee-save and caller-save costs, such as `tomcatv` (Figure 11), which consists of only one big function and no calls, none of the three techniques makes any difference.

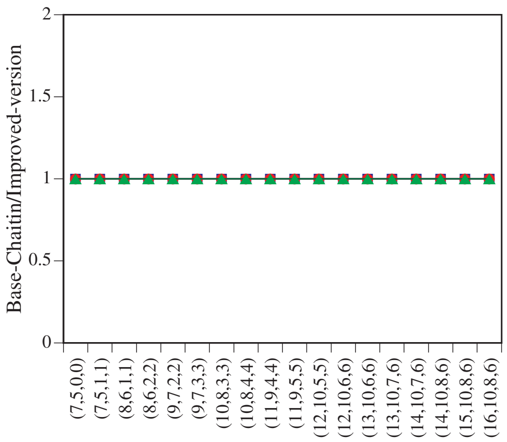


Figure 11: Improvement as function of register pressure for `tomcatv`.

The data for the rest of the programs can be found in Figure 22 of the Appendix.

4.1 Static versus dynamic

The three improvement described in this report are based solely on the analysis of overhead operations. This analysis relies on the execution frequency of basic blocks to estimate the spill cost as well as the call cost. To assess the effect of the frequency information, we measured the SPEC programs using static and dynamic execution-frequency information. Static information uses the loop hierarchy (nesting levels) as an approximation of the execution frequency of a basic block [3, 7]. The execution frequency of a block B is defined as $10^{depth(B)}$. The model treats every block within the same loop as having the same execution frequency. In practice, this

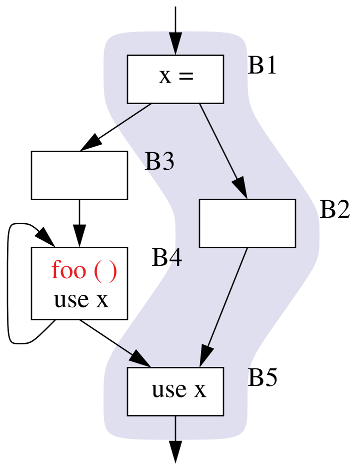


Figure 12: Dynamic versus static.

assumption is not always true. When the static information does not capture the actual execution frequency, the techniques may worsen the register-allocation result. For example, the live range lr_x in Figure 12 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. $f_{benefit\_callee}(x)$, therefore, is greater than $f_{benefit\_caller}(x)$. Register assignment attempts therefore to put lr_x into a callee-save register at first. There are two outcomes for lr_x of the register 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 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` (Figure 13), `li`, and `spice` (Figure 23 in Appendix) are hurt due to this reason.

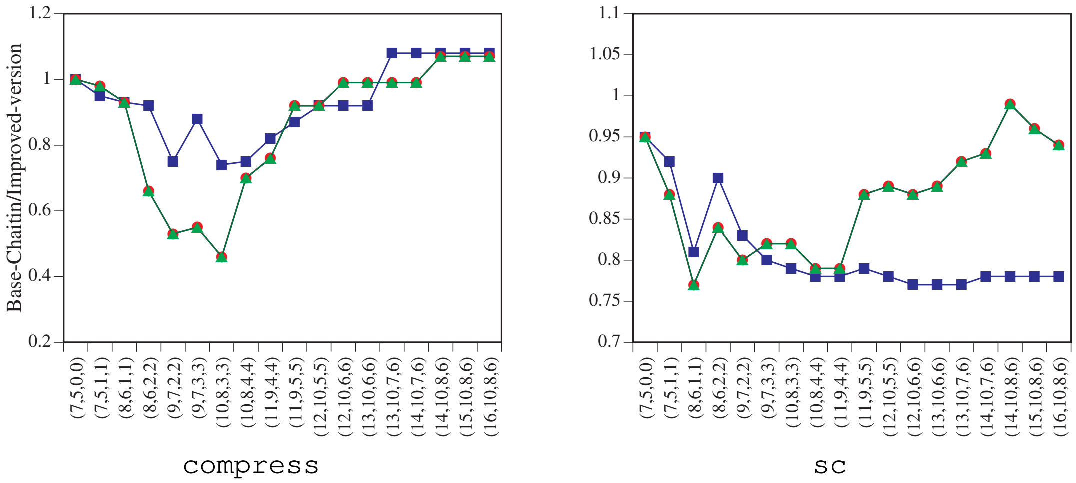
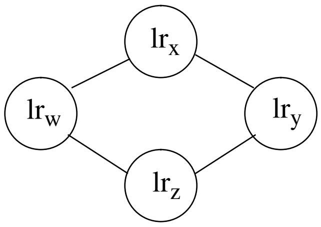


Figure 13: Improvement as function of register pressure (using static information).

If the static information gives us a good indication about the actual execution frequency, we see a similar improvement trend as using the dynamic information. `alvinn`, `ear`, `eqntott`, `espresso`, `doduc`, `fpppp`, `matrix300`, `nasa7` and `tomcatv` fall into this category (Figure 24 in Appendix).

4.2 Optimistic versus non-optimistic



	lr_w	lr_x	lr_y	lr_z
$f_{benefit\_caller}$	100	200	100	**-250**
$f_{benefit\_callee}$	200	100	200	50

Figure 14: Optimistic coloring with N=2 (1 callee-save and 1 caller-save).

Optimistic coloring [3] delays spill decision 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 of 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 range gets a color, then optimistic coloring produces a superior results. However, if we use the actual cost model, 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 14 presents a situation that 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 the color stack 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_y causes the inferior result.

We consider optimistic coloring for SPEC programs using static and dynamic information. Table 1 and Table 2 in the appendix compare the overhead of optimistic and Chaitin-style coloring. The value in each entry is Base-Chaitin/Optimistic. The darkly shaded regions ($<$ 1.00) highlight the cases where optimistic coloring results in more overhead operations. The lightly shaded regions ($>$ 1.00) indicate that optimistic coloring outperforms Chaitin-style coloring (with no enhancements). 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 with a small number of registers because the smaller the number of registers is, the more live ranges are spilled. And even in the best cases, the influence of optimistic coloring is negligible (within $\pm$ 6 %) except for `fpppp` when using static information (up to 36 % improvement of overhead operations with a small number of registers).

We incorporate optimistic coloring into improved Chaitin-style coloring (i.e., including all three improving techniques). Except for `fpppp` when using static information, the results are almost identical to those obtained by the improved Chaitin-style coloring alone. This result is no surprise because the improvement of optimistic coloring is negligible, 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. Figure 15 shows in more detail the improvements due to optimistic coloring, improved Chaitin-style coloring, and improved

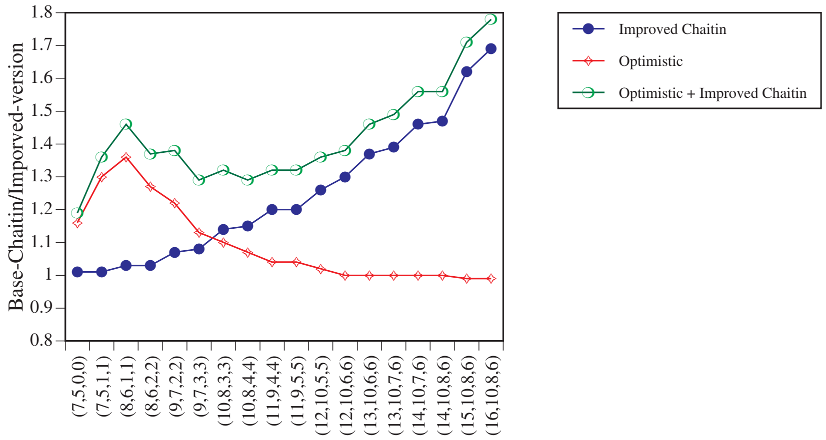


Figure 15: Optimistic versus non-optimistic for `fpppp` (using static information).

Chaitin-style with optimistic coloring for `fpppp` using static information. Optimistic coloring performs well with a small number of registers, but the enhancement drops as the number of register 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 improvement 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 trend of optimistic-coloring improvement. 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.

5 Priority-based vs. Chaitin

At first sight, Chaitin-style coloring and priority-based coloring [7] appear to have very little in common. Priority-based coloring assigns registers to live ranges based on a priority function and splits a live range L that when no legal register exists for L. Optimistic coloring is similar to priority-based coloring in the sense that optimistic coloring spills, instead of splitting, a live range L when all available colors are already assigned to L's neighbors.

Chaitin-style coloring and priority-based coloring share a major core in common 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 register assignment 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 register 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 uses less colors, 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 ones). 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 priority-based simplification to achieve the same effect as priority-based approach. The main goal is to let one's strengths compensate the other's weaknesses.

5.1 Structure of the register allocator

We 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 16). The graph-construction phase builds the interference graph for the input instruction sequence. The color-ordering phase heuristically determines the order in which live ranges are to be assigned colors. The color-assignment phase assigns colors to live ranges based on the ordering computed by the previous phase; this phase uses the interference graph to make sure that conflicting live ranges are assigned different colors. If either of the color-ordering or color-assignment phases spills a live range, the register allocator rebuilds the interference graph and restarts from the color-ordering phase. Our register allocator reserves no registers for spill code; that is, the register space is not divided into "local" and "global"registers.

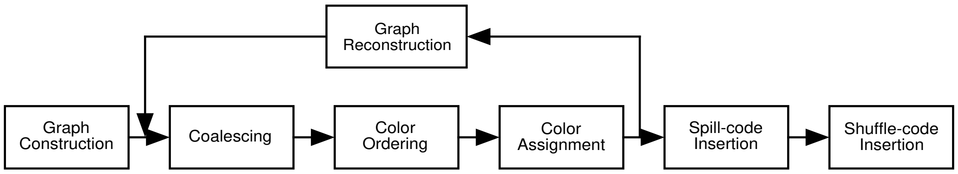


Figure 16: Structure of the register allocator.

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 coloring proceeds with the modified graph. 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. Our register allocator takes the second approach (i.e., it reuses the graph, to save compilation time).

Spill code and shuffle code, respectively, are inserted into the final schedule by the last two phases, spill-code insertion and shuffle-code insertion. Shuffle code moves a data value between the different storage locations assigned to a split live range. The shuffle-code insertion phase is just a *nop* because both approaches do not split live ranges. This register-allocation structure also models other live-range splitting approaches [1].

There are two stacks that are used as the interface between the color-ordering and color-assignment phases: (1) the `color` stack (C), and (2) the `spill` stack (S). The color-ordering phase determines the order in which live ranges are assigned colors and pushes live ranges onto the C stack based on the ordering — the higher the position within C, the higher the priority. The color-assignment phase pops live ranges from C and finds legal colors (i.e., non-conflicting colors) for them. The S stack is a pool for keeping spilled live ranges. If the color-ordering phase decides to spill live ranges, or the color-assignment phase fails to find legal colors for live ranges, those spilled live ranges are pushed onto S.

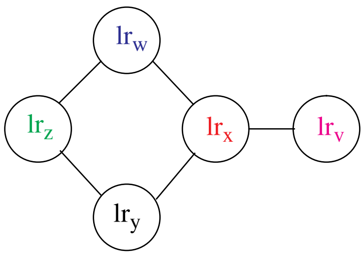
5.2 Priority functions

We use $\frac{max(f_{benefit\_caller}(lr), f_{benefit\_callee}(lr))}{size(lr)}$ as the priority function for priority-based coloring. $size(lr)$ is the number of basic blocks that lr contains. The priority function is the same as the one used in [7]. 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 the `color` stack. The remaining live ranges are pushed onto the `color` stack from the least priority to the highest priority. Chow uses the same approach in [7].
- **sorting unconstrained** Unconstrained live ranges are not pushed onto the `color` stack in a priority fashion in the removing-unconstrained approach, which means the kind of register that higher-priority live ranges covet for could possibly be taken away by lower-priority live ranges. This approach alleviates the problem by pushing unconstrained live ranges onto the `color` stack also in a priority manner.
- **sorting** All live ranges are purely sorted in terms of their priorities. This approach guarantees that the higher the priority, the more likely the live ranges are assigned their preferable kind of register.

All three heuristics produce nearly identical register overhead ($\pm$ 10%) for most cases. A detailed comparison is given in Table 3 and 4 of the appendix. RU, SU and S indicate Removing Unconstrained, Sorting Unconstrained and Sorting, respectively. There are two rows for each SPEC 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. 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 17 shows a case that S yields a better result. Given N=2, 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 as the highest priority live range. As a result, lr_v can only reside in the caller-save register. 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
$f_{benefit\_caller}$	1500	2000	1000	1500	4000
$f_{benefit\_callee}$	1000	1000	400	1000	1000
Size	4	3	4	3	2
Degree	2	3	2	2	1

(b)

Figure 17: Priority functions with N=2 (1 callee-save and 1 caller-save).

5.3 Evaluation

We compare improved Chaitin-style coloring with priority-based coloring using the sorting heuristic. We classify the SPEC programs based on the results into 3 classes. Figure 18, 19 and 20 show some of the results. Figure 25 in the appendix shows the rest of figures.

- Both priority-based coloring and Chaitin-style coloring are doing equally well. Examples are `alvinn`, `eqntott` (Figure 18), and `li`. 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 more likely to find registers) so we see that the two approaches yield similar results.

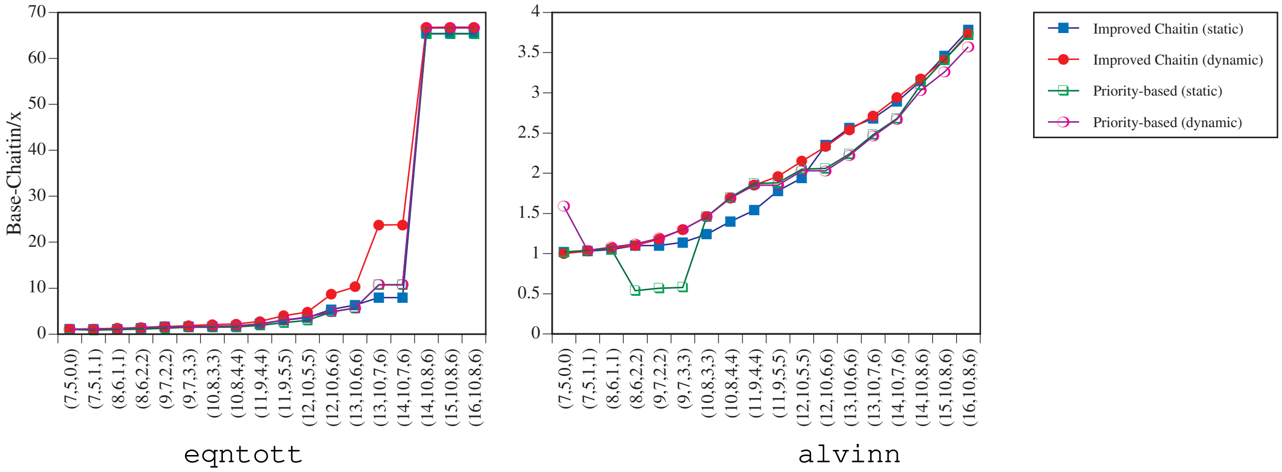


Figure 18: Priority-based coloring versus Chaitin-style coloring (having similar results).

- Improved Chaitin-style coloring is superior to priority-based coloring. `compress`, `ear`, `sc`, `doduc`, `nasa7`, `spice`, and `tomcatv` fall into this category. 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 trend of improvement in both 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 that improved Chaitin-style's. We also see the same scenario for `ear`. For `tomcatv`, priority-based approach is hurt by spilling more live ranges than improved Chaitin-style approach, both in static and dynamic cases, due to the inability of packing live ranges densely. For `sc` (Figure 25 in the appendix), improved Chaitin-style coloring has a 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 less register overhead than improved Chaitin-style coloring.

- There is no clear winner between the two approaches. Examples of this class are `espresso`, `fpppp` (Figure 20) and `matrix300`. For `espresso`, priority-based coloring is clearly superior to improved Chaitin-style coloring in the dynamic case but not in the static case. For `fpppp` in the static case, there is a similarity between priority-based coloring and the integration of improved Chaitin-style and optimistic coloring (shown in Figure 15). 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 less spill code, (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 21 illustrates how the second case happens. The two benefit functions of each live range are shown in the table. Priority-based

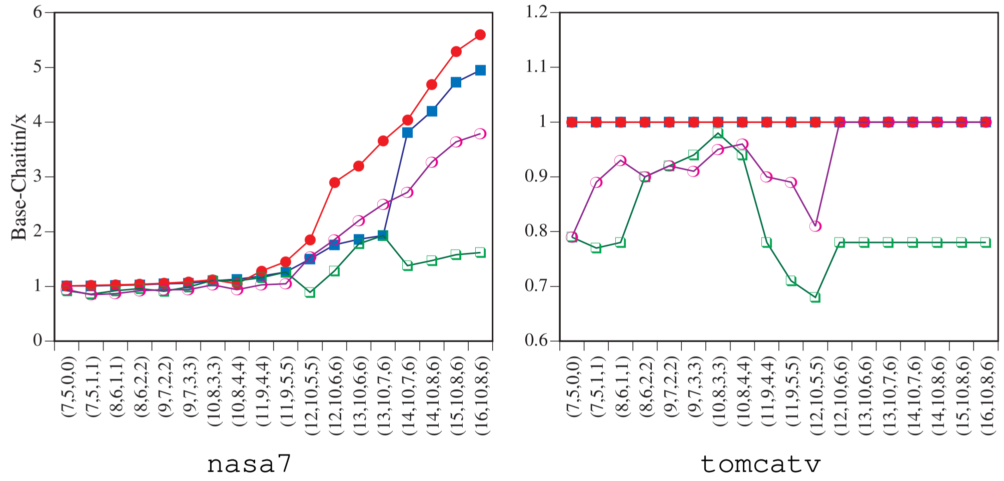


Figure 19: Priority-based coloring versus Chaitin-style coloring (Chaitin-style has less overhead).

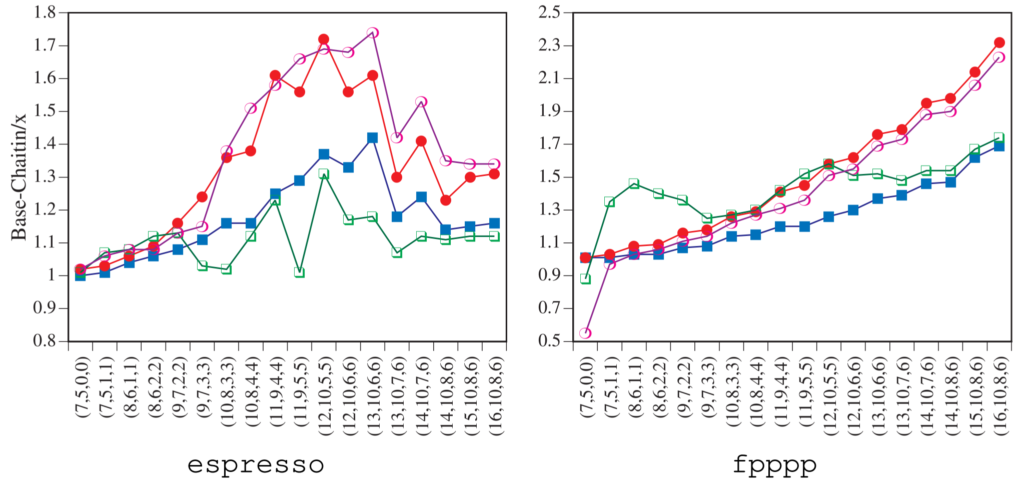
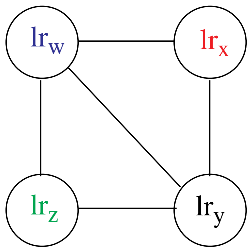


Figure 20: Priority-based coloring versus Chaitin-style coloring.

coloring considers lr_y to have the highest priority. Subsequently, the callee-save register is assigned to lr_y, which leaves only the caller-save registers for lr_w. This results in a bad register assignment because the penalty of picking the wrong kind of register for lr_w is high. 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
$f_{benefit\_caller}$	300	2000	4000	2000
$f_{benefit\_callee}$	3000	2400	4500	2400
Size	4	3	4	3
Degree	3	2	3	2

(b)

Figure 21: Priority-based coloring with N=3 (1 callee-save and 2 caller-save).

6 Conclusion

The experimental results of this report show that the call cost dominates the register overhead as the number of register increases because the register allocation is able to assign registers to most live ranges. Ignoring the call cost could possibly lead the register allocator to choose the wrong kind of register for live ranges and introduce, as a result, more overhead operations.

The report discovers that optimistic coloring in the real world (we cannot ignore the call cost) has only negligible improvement (within a few percent) and moreover has negative influence in many cases.

Chaitin-style coloring reduces overhead operations by simplifying the interference graph to find the color ordering that allows to assign more live ranges registers. Priority-based coloring reduces overhead operations by assigning registers based on the priority function. Chaitin-style coloring with all the enhancements integrates both Chaitin-style and priority-based coloring. The experimental data show that improved Chaitin-style coloring is superior to priority-based coloring and the improvement over ordinary Chaitin-style coloring can be a factor of 55 for `ear` (static case) and 66 for `eqntott` (both static and dynamic cases).

References

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

Appendix

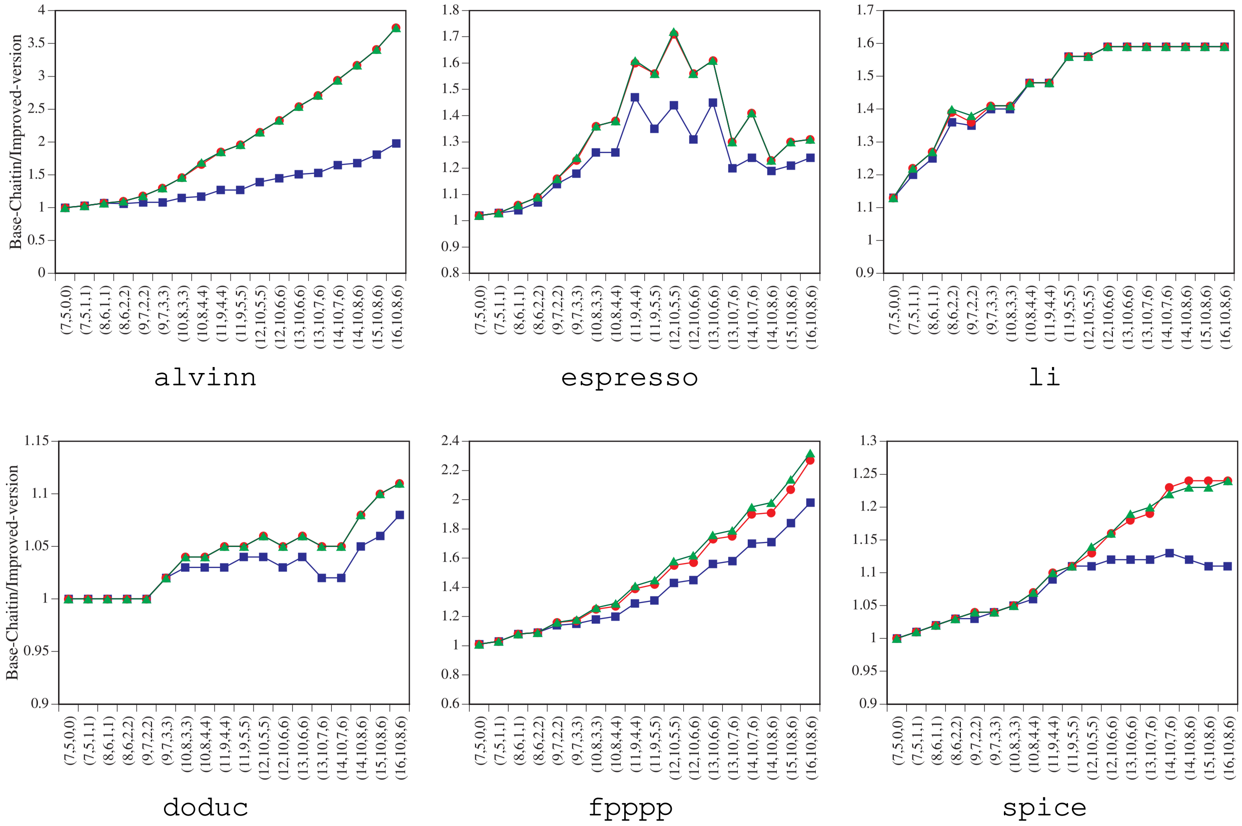


Figure 22: Improvement as function of register pressure (using dynamic information).

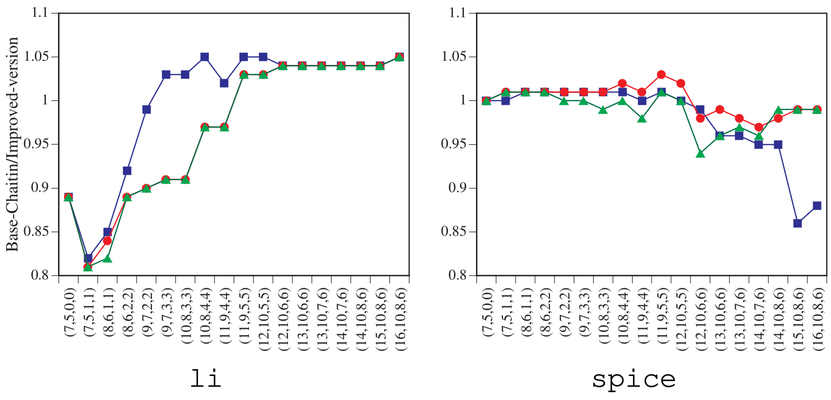


Figure 23: Improvement as function of register pressure (using static information).

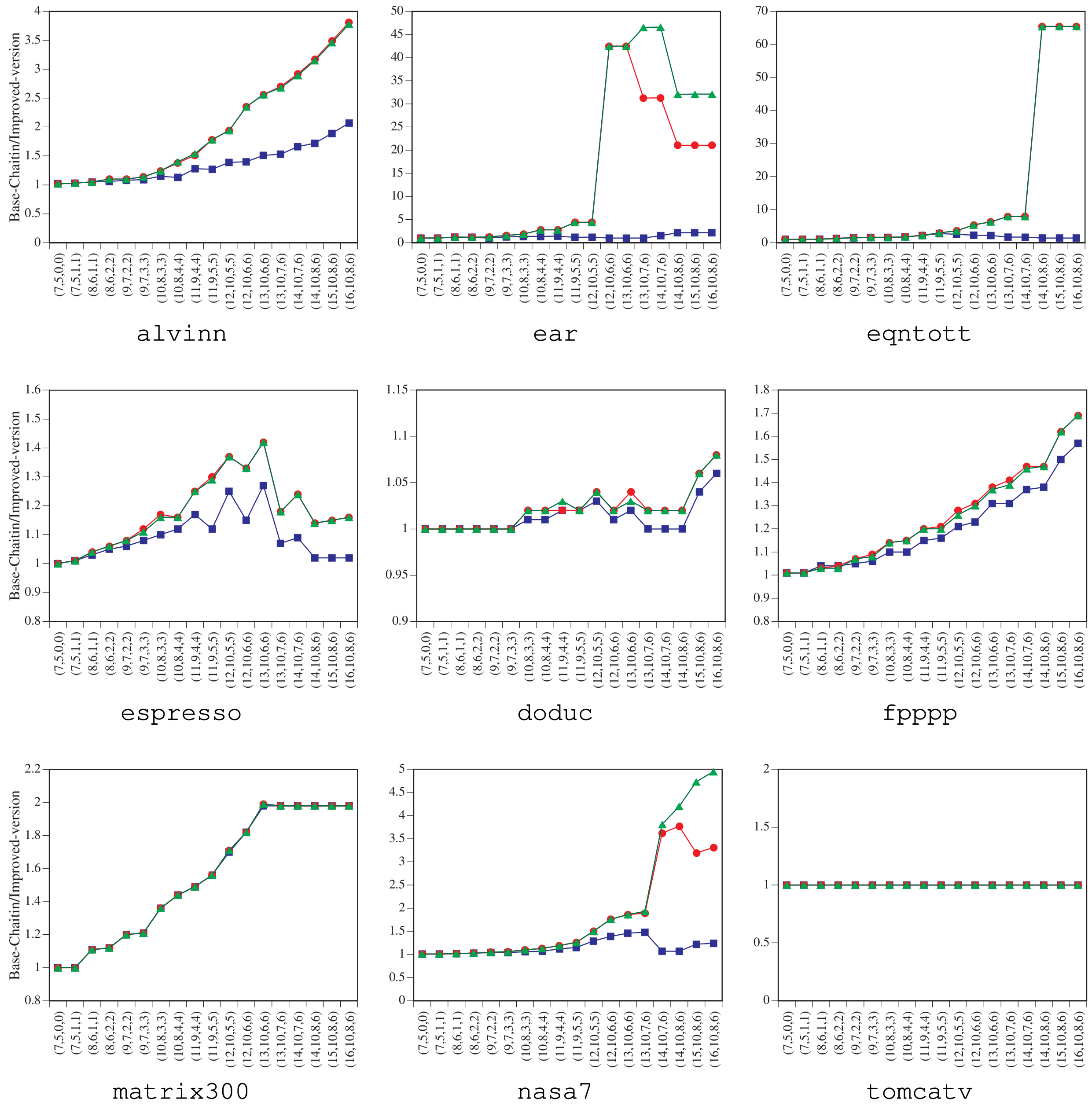


Figure 24: Improvement as function of register pressure (using static information).

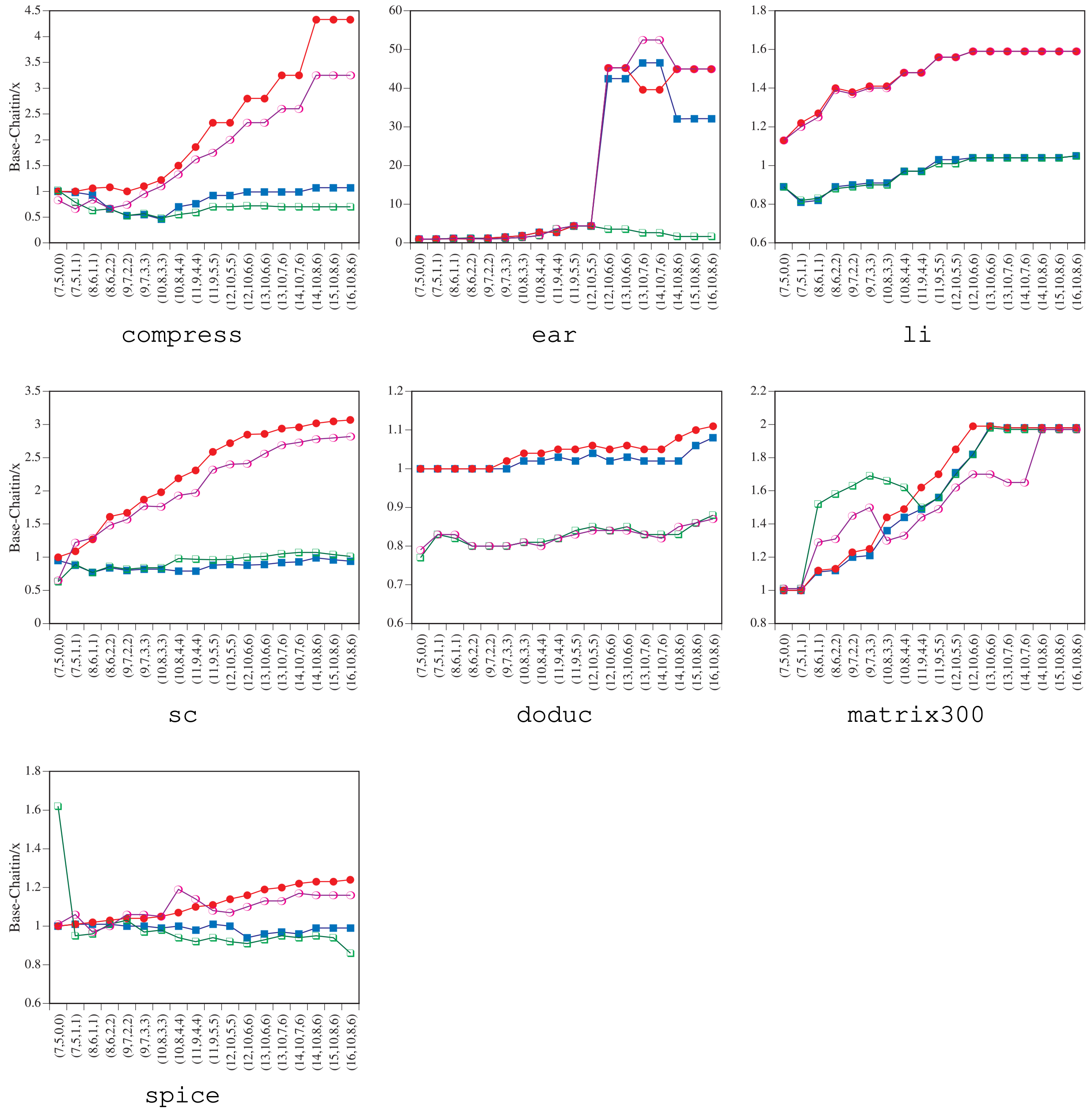


Figure 25: Priority-based coloring versus Chaitin-style coloring.

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	0.94	0.98	0.97	0.98	0.98	0.98												
compress	1.03	1.04																
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										
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	0.99	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 1: Optimistic coloring versus Chaitin (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	0.99	0.99	0.99	0.98	0.98													
compress	1.02	1.03																
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										
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								

Table 2: Optimistic coloring versus Chaitin (using dynamic information).

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									0.98	1.03	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
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	0.99	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 3: 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																		
	RU/S				0.96	0.78	0.91	0.90											
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		0.95	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
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								0.99	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			0.99	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 4: Comparison of priority-based heuristics (using dynamic information).