The Priority-Based Coloring Approach to Register Allocation

FRED C. CHOW
MIPS Computer Systems, Inc.
and
JOHN L. HENNESSY
Stanford University

Global register allocation plays a major role in determining the efficacy of an optimizing compiler. Graph coloring has been used as the central paradigm for register allocation in modern compilers. A straightforward coloring approach can suffer from several shortcomings. These shortcomings are addressed in this paper by coloring the graph using a priority ordering. A natural method for dealing with the spilling emerges from this approach. The detailed algorithms for a priority-based coloring approach are presented and are contrasted with the basic graph-coloring algorithm. Various extensions to the basic algorithms are also presented. Measurements of large programs are used to determine the effectiveness of the algorithm and its extensions, as well as the causes of an imperfect allocation. Running time of the allocator and the impact of heuristics aimed at reducing that time are also measured.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*hardware/software interfaces*; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*; *performance attributes*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*

General Terms: Algorithms, Design, Languages, Measurement, Performance

Additional Key Words and Phrases: Aliases, chromatic number, graph coloring, live ranges, spilling, register allocation

1. INTRODUCTION

A register allocator determines the contents of the limited number of hardware registers during the course of program execution. Register allocation is usually performed at the end of global optimization, when the final structure of the code to be emitted has been determined and all potential register usages are exposed. The register allocator attempts to map the registers in such a way as to minimize the number of memory references. Register allocation may lower the instruction

Authors' current addresses: F. C. Chow, MIPS Computer Systems, Inc., 928 Arques Avenue, Sunnyvale, CA 94086; J. L. Hennessy, Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

count, for example, on a load/store architecture, and may reduce the execution time per instruction, by changing memory operands to register operands. These improvements also reduce code size, which may lead to other, secondary performance improvements.

Register allocation also plays a significant role in determining the effectiveness of other optimizations [4]. Many optimizations create temporaries, and the improvement from the optimization depends on the cost of access to the temporaries. For example, in common subexpression elimination, if the subexpression needs to be stored in memory and retrieved, the optimization may buy little. In some cases, it may actually slow the program down.

This paper addresses the problems of global register allocation on a per-procedure basis, in which the content of an entire procedure is taken into account in mapping register contents, but the contents of other procedures are not examined. Section 2 presents the problem and discusses the shortcomings of a straightforward coloring approach. Section 3 describes the parameters used by the algorithm. Section 4 defines live ranges as they apply to the algorithm. Section 5 presents the priority-based coloring algorithm. Section 6 discusses the differences between our approach and the previous approach. Section 7 describes how the algorithm can cater to special register linkage conventions. Section 8 describes some post-allocation optimizations. Section 9 describes how the algorithm treats different register types. Section 10 describes further refinements to the algorithm using some heuristics. In the last few sections, measurements are provided of both the performance and the running time of the algorithm.

2. THE GLOBAL REGISTER ALLOCATION PROBLEM

Previous researchers have shown that global register allocation corresponds closely to the graph-coloring problem [2, 3]. Chaitin et al. were the first to apply the coloring approach to register allocation in the experimental PL.8 compiler at IBM [1]. A *coloring* of a graph is an assignment of a color to each node of the graph in such a manner that any two nodes connected by an edge do not have the same color. For register allocation, the graph, called the *interference* or a *conflict* graph, is constructed from the program. Each node in the interference graph represents a live range of a program data value that is a candidate to reside in register. Informally, the live range is a collection of basic blocks where a particular definition of that variable is live. Two nodes in the graph are connected if the two data values corresponding to those nodes interfere with each other in such a way that they must not reside in the same register. In coloring the interference graph, the number of colors used for coloring, *r*, corresponds to the number of registers available for use in register allocation. A register allocator wants to find an assignment of the program variables to registers that minimizes the execution time. In global register allocation, we take into account the entire procedure in deciding on the best assignment.

The *chromatic number* of a graph is the minimum number of colors required to successfully color the graph. Chaitin's approach to register allocation attempts to map a fixed set of colors to all the nodes of the interference graph. Whenever the chromatic number is greater than the number of available registers, the algorithm is blocked, and it is necessary to add spill code. Spilling a variable means keeping it in storage rather than in a register, and the node corresponding to the variable is then deleted from the interference graph. Such deletions simplify the interference graph and will reduce its chromatic number until it is no greater than the number of available registers, at which point the coloring algorithm can proceed to completion.

Earlier descriptions of the algorithm have not addressed some other practical issues that arise. We see three key issues.

First, the allocation should use a cost and benefit analysis. The value of assigning a given variable to a register depends on the cost of the allocation and the resultant savings. The cost comes from the possible introduction of register-memory transfer operations to put the variable in a register and later to update its home location. The savings results from the gain in execution speed due to the eliminated memory accesses. Estimating the cost and savings depends on the hardware characteristics and addressing capabilities of the machine. The estimates need to be parameterized by machine and supplied to the register allocation algorithm. A straightforward algorithm that does not assess the cost of allocation may overallocate the registers. Although this overallocation is unlikely to arise on a load/store architecture, it can arise on a machine with memory addressing modes.

Second, the algorithm must handle spilling gracefully. When the coloring algorithm is blocked, the decision regarding which variables to spill is difficult to make. For more optimal allocation results, we must consider spilling a variable during a limited range of its use. The spilling decisions should not be separated from the coloring decisions, since this makes it more difficult to predict the effect of a given spill on the effectiveness of the final allocation.

Third, the running time of the algorithm must be reasonable. The determination of whether a graph is r-colorable is NP-complete. A heuristic linear-time algorithm can fail, and a backtracking algorithm may require exponential computation time, whether or not the graph is colorable.

Once we have decided to use some estimates to guide our coloring, we must consider various factors to determine the importance of allocating different variables. There are three major factors that determine the cost and savings estimates:

(1) *Execution frequencies.* The loop structure of a program influences the savings estimates. Allocation of the more frequently used variables to registers increases the savings. The positions for inserting register-memory transfer operations should also be optimized with respect to the control flow of the program.

(2) *Clustering.* Variables occur with different degrees of localization or clustering. The register allocator should take into account the extent of the region over which a variable appears, since allocating a variable over a region ties up the register throughout that region. The allocator should prefer dense over sparse occurrences, even if the number of occurrences is the same.

(3) *Procedure linkage convention.* Depending on the linkage convention, registers may need to be saved and restored at call boundaries. Special considerations are also required when parameters are passed in registers. The cost of

saving and restoring registers must be considered; otherwise, we may overallocate the registers, which can result in additional register-memory transfers.

Finally, allocation in large procedures must be considered. A practical register allocation method should be able to adequately handle procedures of arbitrary size. Large procedures have more register-residing candidates with larger ranges of occurrences. Thus, in a large procedure, the chance of the graph being r-colorable is less, and the coloring process is more complex and costly. Even if the underlying machine has many registers available, when the size of the procedure increases, a point can be reached when it becomes impractical to try to allocate registers by pure coloring.

A practical register allocator should strive to produce good results despite any adverse situation that may arise in practice. It does not need to look for the optimal solution, but correct and efficient ones. The compile-time cost attributed to register allocation should be reasonable compared to the time taken to perform other global optimizations. The priority-based register allocation scheme, first presented in [6], was developed to satisfy these goals while addressing the problems mentioned above.

Generally, there are two possible models for performing register allocation. The first model assumes that all variables reside in memory initially, and the register allocator chooses individual live ranges to allocate to registers. The second model regards all live ranges as symbolic registers and spills a live range to memory only when the symbolic register cannot be mapped to a real register. Figure 1 illustrates these two models. The difference between these two approaches is how they deal with live ranges that are not allocated. The first method (Figure 1a) treats the unallocated variables as memory resident and requires that the code generator include extra code to access them. The second approach (Figure 1b) generates code to spill the registers into memory locations to lower the chromatic number; when the coloring is finished, every variable has been allocated to a register, possibly with spill code. Although these unallocated ranges may arise because the chromatic number of the graph exceeds the available register count, on an architecture with memory referencing instructions they are more likely to occur when the cost of allocating a range exceeds the savings from allocating the range. In such cases, the savings from allocating the variable to register is not as high, and the first approach will simply leave the variable in memory for the entire range. In the second approach, allocation of the range with spill code will result in extra load and store instructions that lower the program performance.

We use the first model for allocation and assume that all variables and expression temporaries have home locations. We also assume that a code generator will use a local register allocator to handle accesses to variables that have not been assigned to a register. This requires reserving at most four registers for use by the code generator. This might lead to inefficiency, but the inefficiency is significant only if the number of available registers is small.

The priority-based register allocation algorithm determines priorities for register-residing candidates using estimates derived from the program and from machine parameters, and then assigns the candidates to registers according to

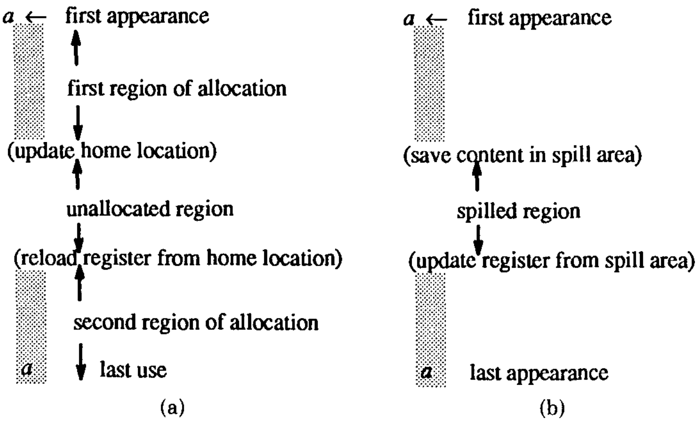
a ← first appearance
first region of allocation
(update home location)
unallocated region
(reload register from home location)
second region of allocation
a
last use
(a)
a ← first appearance
(save content in spill area)
spilled region
(update register from spill area)
a
last appearance
(b)

Fig. 1. Two models of register allocation. (a) Starting out in memory. (b) Starting out in register.

the priority ordering. By not backtracking, the algorithm achieves good running time regardless of colorability. It yields similar results as the pure coloring method if the interference graph is colorable. Otherwise, it produces good, though not necessarily optimal, results. When the algorithm is blocked in the coloring process, that is, if it cannot assign a color to the next highest priority node, it splits the live range that cannot be colored and continues. The following sections present the algorithm in greater detail.

3. ALLOCATION PARAMETERS

In this section we identify the parameters that play important roles in our global register allocation algorithm.

3.1 Machine Parameters

The most important parameters affecting register allocation are the execution-time costs and savings due to register assignments. If the first occurrence of a variable is a use, the variable must first be loaded from memory to the register before it can be referenced there in the subsequent code. We call this loading operation RLOD. If the value of the variable is changed in the intervening code where it resides in register, the home memory location of the variable has to be updated with the register content at the end of the code segment, unless it is dead at that point. We call this updating operation RSTR. These transfer operations between registers and memory represent the execution-time cost of the register assignment. The execution-time savings indicates how much faster the given code segment will execute when the given variable is put in register. We define the following three parameters used to determine these costs and savings:

(1) *MOVCOST*. The cost of a memory-to-register or register-to-memory move, which, in practice, is the execution time in the target machine corresponding to RLOD and RSTR. If the execution times for loads and stores are different in the target machine, they have to be distinguished further.

(2) *LODSAVE.* The amount of execution time saved for each reference of a variable residing in a register rather than in memory.
(3) *STRSAVE.* The amount of execution time saved for each definition of a variable residing in register compared with the corresponding store to memory being replaced.

Since only the relative values between *MOVCOST* and the two kinds of savings are important, *MOVCOST* can be assumed to be 1, and values can be found for *LODSAVE* and *STRSAVE* relative to 1.

These parameters vary widely among machines. In load/store machines, all arithmetic and logic operations can only be performed on registers, and the difference between being allocated and unallocated is the extra load or store instruction that accompanies each occurrence of the variable. Thus, *LODSAVE* and *STRSAVE* are equal to *MOVCOST* and are given the value 1.

Other machines have the capability of directly fetching operands from memory while performing operations on them, although such operations require longer execution time. In these machines, *LODSAVE* and *STRSAVE* may be less than 1, since register residence is optional for a value to be operated on and since being preallocated in register may save less than the time for one instruction.

Within the same machine, *LODSAVE* and *STRSAVE* may not be constant for all loads and stores, since they depend on the actual machine instructions and addressing modes being used. The addressing mechanism used also depends on whether a given variable is local, global, or an up-level reference. In such cases, we use average values for *LODSAVE* and *STRSAVE.*

3.2 Program Parameters

The usage of register resources may be in multiple regions of code space, so the estimates of costs and savings should be weighted by the execution frequencies in those regions. Live ranges extending over loops have greater priority than those not over a loop. The frequency weights can be estimated by analyzing the control flow structure of the procedure and computing the loop-nesting depths.

An alternative method to obtain execution frequency estimates is to use profile information from earlier execution runs. By arranging to feed the profile information to the register allocator, much more accurate execution frequencies can be obtained. In the case of two-way branches that are not parts of loops, the program trace can provide exact information as to which branch is more likely. Such details are not possible with static analysis of the program. We are currently using static analysis; we plan to add the capability to use profile information in the future.

3.3 Linkage Parameters

Different run-time conventions can apply to the different registers. Often, a set of registers is reserved for passing parameters in procedure calls, and another set is used for passing back the return values. Both of these classes can be used for keeping variables between calls and returns. The linkage convention also classifies each register as caller-saved or callee-saved. With caller-saved registers, the caller is responsible for saving their contents before procedure calls and restoring their contents after returns from calls, if necessary. With callee-saved registers, the caller regards the registers' contents as being undisturbed across a procedure call. A procedure that uses a callee-saved register must save its contents at procedure entry and restore it before exit. The parameter registers and function return registers must be caller-saved when they are used to keep variables between calls. The classes of registers have to be treated differently in the register allocation process if the allocation is to be optimal.

The register file in the underlying machine architecture offers another variation in linkage support. A register allocator for the SPUR machine with register windows is described in [11]. The availability of register windows reduces the cost of register usage by eliminating the needs to save and restore used registers at procedure calls or procedure entries and exists, since a new window is allocated at the invocation of each procedure. The registers that overlap more than one window are parameter registers and must be regarded as caller-saved if they are used internal to the procedure.

3.4 Hardware Parameters

The hardware architecture can place limitations on register usages. The architecture may have different classes of registers: integer registers, address registers, and floating-point registers. Different register sizes may also exist, with possible overlap between them. For example, a double-word register may alternatively be used as two single-word registers. Depending on the width of the data path in the machine, the cost and savings may need to be scaled. For example, in a machine with only a word-sized data path, the savings for a double-word variable residing in a register can be twice as much as that for a single-word variable, since two load instructions are required to reference the double-word quantity from memory.

4. LIVE RANGES

A live range is an isolated and contiguous group of nodes in the control flow graph in which a program data value is live. Each node in the control flow graph corresponds to a basic block. Procedure calls also delimit basic blocks because of the register save operations and side effects incidental to procedure calls.

By restricting register assignment to only one program quantity in each basic block, we treat the basic block as the unit in the usage of register resources. With large basic blocks, this restriction could limit the performance of the register allocator by disallowing the same register to hold different quantities in different parts of the basic block. To avoid this limitation, our compiler forces the creation of a new basic block whenever the number of variable occurrences reaches a limit. The SPUR allocator referred to earlier forces a new basic block when the number of statements exceeds a limit. Since basic blocks are usually small, this need to force new basic blocks is infrequent in practice.

Live ranges are computed for both program variables and compiler-generated temporaries. The presence of the global optimization phase exposes many more temporary references whose allocation to registers is necessary to maximize the benefits of the optimizations. In general, any program quantity whose usage requires loading a register is a register-residing candidate and has as associated live range. In load/store machines, even a constant can be a register-residing candidate if the constant cannot be put in an instruction as an immediate value. In the rest of this paper, the term *variable* will be used to include all such register-residing candidates. Our register allocator will not allocate a register to hold a variable that appears only once in a procedure, since the later code generation phase can handle these variables equally well.

The live range is the basic unit for register assignment. Each live range is assigned to a single register. In the course of allocation, a live range may be split into two or more smaller live ranges. Subsequently, each of the smaller live ranges can be assigned to a different register. In the rest of this section, we describe how the initial live ranges are computed and the information associated with them.

4.1 Computing Live Ranges

A live range typically begins at definition points of a variable and terminates at its last uses. None of the uses inside the live range have a definition outside the live range. Furthermore, no definition of the variable inside the live range may reach a reference point outside the live range. Thus, these live ranges usually correspond to def-use chains. However, there are situations where this is not true. For a nonlocal variable, a definition may not precede the first use. The live range starts at the first use and ends at the last appearance, which may be either a definition or a reference.

Procedure calls affect the definition of live ranges, depending on the register linkage convention. In the caller-saved convention, all registers need to be freed at a procedure call, and a live range is not allowed to extend over a call. Instead, the live range ends at the last appearance before the call. This will free the register up before the call. After the call, the live range should begin at the first appearance of the variable. In the callee-saved environment, live ranges are allowed to extend over procedure calls, since the callee-saved registers can be allocated across the calls. Figure 2 gives examples of live range limits under the two different register linkage conventions.

We compute live ranges by data-flow analysis [9] at the beginning of the register allocation phase. The process involves solving separately for the *live* and *reaching* data-flow attributes. The live attribute is solved by backward iteration through the control flow graph. A variable is live at block i if there is a direct reference of the variable at block i or at some point leading from block i not preceded by a redefinition. The reaching attribute is solved by forward iteration through the control flow graph. A variable is *reaching* in block i if a definition or use of the variable reaches block i. The live range is the intersection of the set of program graph nodes in which the variable is live and the set of nodes in which it is reaching.

The above computation yields a set of nodes that constitutes the live range of each variable. The live range is exact; regions where there is no benefit in putting the candidate in register are excluded. These include gaps that exist between the occurrences of the def-use chains. Because of the existence of these gaps, the nodes in a live range are not necessarily connected. We describe a live range as *contiguous* if it consists of only one connected component, and as *noncontiguous* otherwise. Each connected subcomponent of a live range can be assigned to a

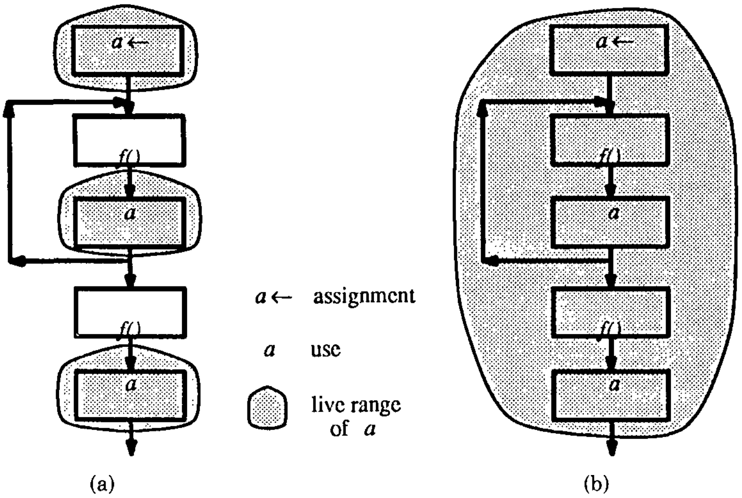

Fig. 2. Live range of a local variable a. (a) Caller-saved. (b) Callee-saved.

different register. To allow this to happen, a potentially costly separation process is needed to identify the connected components in each live range and to make them separate live ranges. However, the splitting mechanism built into our allocation algorithm can accomplish the same effect as this separation process, though driven by a different cause. Our splitting mechanism also caters to situations where we only want to put part of the live range in a register, even if the live range is contiguous. Thus, the splitting mechanism attacks a larger problem than just the separation of unconnected components, and we omit the separation in favor of splitting. We regard each register-residing candidate as initially corresponding to only one live range, even if there are multiple connected components within it.

Treating multiple connected components as one live range and assigning them to the same register may occasionally yield suboptimal allocation results. This occurs in situations where the chromatic number of the interference graph can be reduced by separating out the connected components. An example can actually be found later in Figure 8, where the best result could have been achieved by treating variable a initially as two live ranges. This is not an important practical consideration; as we will show later that our splitting process leaves most of the final live ranges as contiguous. By having fewer live ranges at the beginning, the interference graph has fewer nodes and is much simpler, allowing for faster allocation. The interference graph grows only when mandated by circumstances.

4.2 Information Associated with Live Ranges

The representation of a live range contains all of the information needed in the course of register allocation. Among other information, it gives the basic blocks contained in the live range in the form of a bit vector called **livebbs**. This bit-vector representation facilitates the operations that check whether a given block is in a live range or whether two live ranges overlap. There is also an interference

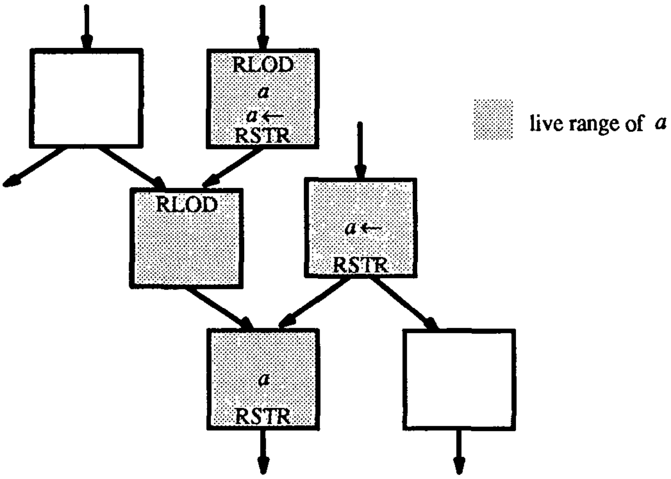

Fig. 3. Insertions of RLODs and RSTRs at live range boundaries.

list (**intflist**) in the live range representation that gives the list of the live ranges that interfere with it. For our purpose, two live ranges interfere if their intersection is not empty. Once a live range has been assigned a register, that register will not be available for any other live range that interferes with the assigned live range. Accordingly, each live range has a set of registers called **forbidden** that it must not use as its resident register.

Each register allocation candidate in conjunction with a basic book during which it will occupy a register is referred to as a *live unit.* Thus, each live range representation has a field called **liveunits** that points to a list of live unit nodes. The live unit representation describes the number of loads and stores occurring in that basic block, whether the first appearance in the basic block is a store and whether the object is dead on exit from the basic block. The live unit nodes are linked in two different ways. The first link forms the list of live units that comprise the live range. The second link forms the list of live units occupying a given basic block. Because the number of different live ranges spanning a given basic block is generally small, it is usually faster to get to a live unit node using the second list.

By virtue of the connectivity among the blocks in a live range, when the live range is assigned to a register, RLODs need to be inserted only at entry points to the live range, and RSTRs need to be inserted only at the exit points (Figure 3). We decide whether these RLODs and RSTRs are really needed based on the information contained in the live range and live unit representations. In a later section, we look into determining these register moves. For now, we assume that two flags, **needrlod** and **needrstr**, tell whether RLOD and RSTR are required at the entry and exit, respectively, of the basic block.

4.3 Handling Aliases

An alias occurs if the same memory location is both directly and indirectly accessed. This can occur in association with pointer operations or parameters passed by reference. If allocated in a register, the content of the home location may not be up-to-date with the register containing the current value, and any indirect access will fetch noncurrent values.

We solve the alias problem by not allowing live ranges to extend over points in the program where aliases to the objects occur. A live unit in which there is an alias to the live range's variable is called an *aliased live unit.* The aliased live units are removed from the live range. This may require the insertion of RLODs and RSTRs at the boundary. Within the aliased live units, the aliased objects are treated as if they are spilled. We present some data on aliasing and discuss its impact on allocation results in Section 12.

Aliases may also occur at a procedure call, when the called procedure or any other procedure activated by the call has access to the variable. But, since procedure calls are at basic block boundaries, they do not cause any aliased live unit to be created. They merely result in additional boundaries to the live range. Before the call, the home location of the variable has to be updated, and after the call, the content of the home location has to be reloaded to the assigned register.

5. THE COLORING ALGORITHM

In the process of coloring, we distinguish between *constrained* live ranges and *unconstrained* live ranges. Unconstrained live ranges have a number of neighbors in the interference graph less than the original number of registers available. These live ranges are not colored until the very end, since it is certain that some unused color can be found for them. The rest of the live ranges are constrained live ranges. As we shall see in Section 5.3, live range splitting can cause the constrained and unconstrained pools to change.

5.1 Priority-Based Coloring

For now, assume that the target machine has a single set of homogeneous general-purpose registers. The coloring algorithm consists of the following steps:

(1) Separate the unconstrained live ranges.

(2) Repeat steps (a) to (c), each time assigning a color to a live range until all constrained live ranges have been assigned a color or until there is no register left that can be assinged to any live range in any basic block.

(a) Compute the priority function $P(lr)$ for each constrained live range lr if it has not been computed. If $P(lr)$ is negative, or if lr is uncolorable, which occurs when all available registers have been used throughout its region, mark lr as a noncandidate, and leave it unassigned. Delete lr from the interference graph so that live ranges interfering with it will have one less interference.

(b) Find the live range with the highest priority function, lr^*, and assign a color to it. The color assigned must not be in the **forbidden** set for lr^*. For each live range that lr^* interferes with, update the **forbidden** information.

(c) Check each live range interfering with lr^* to see if it needs to be split. A live range needs to be split if all of its target registers have been assigned to one of its neighbors in the interference graph. This fact is conveyed by its **forbidden** set being equal to the set of available registers. If splitting is necessary, apply the splitting algorithm given in Section 5.3.

(3) Assign colors to the unconstrained live ranges, each time choosing a color not belonging to their **forbidden** set.

There can be variations to the above strategy. In step (2a) one may choose to split a live range whose priority function is of only a small negative value instead of declaring it a noncandidate, since a smaller live range with a positive priority function can possibly be carved out of it. This will yield better results at the cost of more allocation time. In step (2b), rather than selecting a color randomly, a more refined strategy that takes more computation time is to select a color that affects the least number of lr^*'s neighbors that have not been colored. This involves finding the color that is already in the **forbidden** set of the most number of neighbors.

5.2 Priority Function

The priority function $P(lr)$ is an empirical measure of the relative benefits of assigning a given live range to register. It is proportional to the total amount of execution-time savings, $S(lr)$, gained due to the live range's residing in register. To compute $S(lr)$, evaluate the savings due to individual live units, s_i. For a live unit i, let u denote the number of uses, and let d denote the number of definitions [8]. Let n denote the number of register moves needed in that live unit, determined by looking at the flags **needrlod** and **needrstr** in the live unit representation. Thus, n ranges from 0 to 2. Then,

$$s_i = \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVCOST} \times n. \qquad (1)$$

$S(lr)$ is computed by summing s_i over all the live units i in the live range weighted by the execution frequency estimates w_i in the individual basic blocks:

$$S(lr) = \sum_{i \in lr} s_i \times w_i. \qquad (2)$$

An additional factor to consider before we arrive at $P(lr)$ is the size of the live range, which is approximated as the number of live units, N, in the range. A live range occupying a larger region of code takes up more register resource if allocated in register. The total savings should be normalized by N so that smaller live ranges with the same amunt of total savings will have higher priority. Thus, the priority function $P(lr)$ is computed as

$$P(lr) = \frac{S(lr)}{N}. \qquad (3)$$

The priority function is an estimate of the total register savings normalized by the size of the region occupied. The priority function does not take into account the degree of clustering in the live range. In general, a live range in which the occurrences are spread evenly should be favored over one in which the occurrences are mostly concentrated in a small region, given that their range and number of occurrences are the same. This is because, in a live range with uneven occurrences, we want to delay assigning it a register so as to leave open the possibility that only the heavily clustered portion be allocated to register, in case we run out of registers in the later part of the coloring process. The closeness of occurrences is difficult to express quantitatively. One possibility is to compute the standard deviation of the average number of occurrences in each live unit over the live range, also weighted by execution frequencies. Higher standard deviation then implies greater clustering. The priority function $P(lr)$ can then be adjusted by the standard deviation so that greater clustering causes lower priority. We do not currently take this clustering into account, since it would probably not make a difference in the results, especially when the number of registers available is large.

5.3 Splitting Live Ranges

In splitting a live range, we separate out a component of the original live range that is as large as possible to the extent that its basic blocks are connected. This has the effect of avoiding the creation of too small live range fragments. Whenever possible, well-formed live ranges (i.e., those that originate at definition points) are given higher preference. The steps to split out the new live range *lr′* from the original live range *lr* are as follows (a more concise description of this basic algorithm can be found in [11]):

(1) Find a live unit in *lr* in which the first appearance is a definition, preferably one at an entry point to *lr*. If this cannot be found, then start with the first live unit that contains a use. The basic block for this first live unit of *lr′* must have unused registers. This guarantees that the new live range *lr′* will be colorable when formed. Initialize the **forbidden** set of *lr′* to be the set of used registers in the lone basic block.

(2) For each successor of the live units in *lr′* that belongs to *lr*, check whether it can be added to *lr′*. A live unit can be added if it does not cause the forbidden set of *lr′* to be full. On each addition of a live unit, this **forbidden** set is updated by computing its union with the set of used registers in the new basic block.

(3) Now that the final shape of *lr′* (and thus that of *lr*) has been determined, update the interferences. Apart from *lr* and *lr′*, the live ranges whose interferences need to be updated are exactly those live ranges that interfere with the original *lr*. After the split, these live ranges may interfere with *lr*, *lr′*, or both, depending on their spans.

(4) Update other information in the live ranges *lr′* and *lr*. For example, the flags **needrlod** and **needrstr** in the live unit representation are no longer current, since they are dependent on the actual live range boundary.

(5) If *lr*, *lr′*, or both become unconstrained after the split, remove them from the constrained pool and add them to the unconstrained pool of live ranges. Because of the introduction of new live ranges, it is also possible that some unconstrained live range is made constrained. This must arise out of the live ranges that interfere with both *lr* and *lr′* after the split, since they are the only ones whose number of interferences increase due to the split. Thus, only these live ranges need be checked, and the constrained pool and unconstrained pool are updated.

The addition of live units follows a breadth-first traversal of the nodes of the original live range. Each addition of a live unit to *lr′* will bring in additional

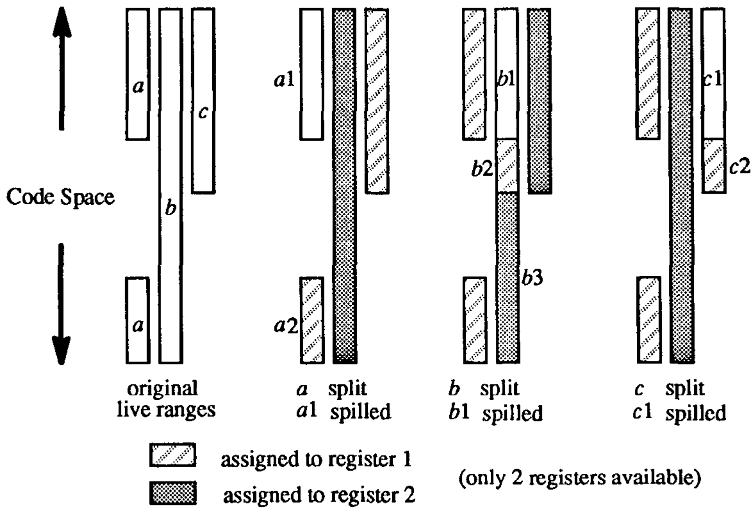

Fig. 4. Different allocation results under different priorities.

adjacent candidates to be considered for adding to *lr'*. Step (2) is best done by manipulating the linear list of live units for *lr'*. Starting at the beginning of this list, it works by going down the list one live unit at a time, checking its successors for possible inclusion. New live units are always added to the end of the list. Thus, the list of live units for *lr'* is growing dynamically while the algorithm iterates down the list. As a result, the live units in *lr'* are always contiguous, and the list of live units is always maintained in breadth-first ordering.

After the colorable *lr'* is created, the live range left behind, *lr*, may still require further splitting in order to become colorable. Thus, the above algorithm is repeated until *lr* is colorable. Splitting is also discontinued if the resultant *lr* is found to be uncolorable, which occurs when there is no unused register in all of the basic blocks in *lr*.

Figure 4 gives examples of allocation results due to different splitting operations under different relative priorities among three variables, *a*, *b*, and *c*. Assuming that there are only two registers available, one variable has to be spilled when all three variables are simultaneously live. Initially, variable *a* is only one live range, even though it consists of noncontiguous parts.

In our implementation, live ranges are identified by unique numbers corresponding to the positions in the bit vectors used in data-flow analysis to determine the extents of live ranges. These same bit vectors are used in other data-flow analyses in the earlier global optimization phases. When the live range *lr* is split, *lr'* is assigned the original bit position that *lr* has, and the new *lr* is assigned the next unused bit number. Step (2c) of the coloring algorithm in Section 5.1 iterates through all the assigned bit numbers starting from zero, so the algorithm will automatically take care of *lr* as a separate live range when it gets to its bit number.

It is anticipated that splitting will occur more frequently as we approach the end of the priority-based coloring, when most registers have been used up. The live ranges still competing for registers have to be twisted to fit the registers still remaining. The splitting process will carve out parts of the live ranges that are not colorable and parts that fit the remaining registers. The process of splitting out a live range part and not coloring it is equivalent to spilling it. Because of the priority coloring order, splitting and spilling will most likely occur on low-priority live ranges.

5.4 Termination of the Algorithm

The inner loop of the algorithm assigns one live range to a register during each iteration. The loop terminates when (1) no candidate is left (all candidates have been assigned to registers) or (2) all registers are used up in all basic blocks that have candidates. The events that happen during each iteration are (a) a candidate is assigned to a register, (b) candidates are removed because they are not colorable or because of negative priority functions, and (c) live range candidates are split. Events (a) and (b) always reduce the number of candidates, which in due course will lead to termination condition (1). Event (a) also makes unavailable the assigned register over the regions occupied by the candidate, which causes the loop to approach closer to termination condition (2). Event (c) has the opposite effect of event (b), by causing a net increase in the number of candidates. But, since we require the live range to be at least as large as a basic block, splitting must end when each live range becomes a single block. Thus, even though event (c) causes the loop to iterate more times, the algorithm will eventually terminate.

The two termination conditions of the algorithm also ensure that the algorithm will assign as many candidates to registers as possible—a requirement for a good allocator.

6. DIFFERENCES FROM CHAITIN'S APPROACH

Apart from using coloring as the central paradigm, our register allocation approach has very little in common with Chaitin's approach, as described in [2], and [3]. In this section we contrast the two approaches. The most important differences are those arising from the timing of register allocation with respect to code generation. Chaitin's scheme allocates registers after the final form of the code is known, whereas our scheme allocates registers using an intermediate representation and before code generation. First, we discuss the differences between the algorithms arising from this choice of where register allocation should be performed:

Unit of allocation. Chaitin uses the machine-level instruction as the unit of coloring, whereas we use the basic block as the unit of coloring. Since Chaitin's register allocator is applied on the machine instruction level, within each basic block, there are more candidates to be colored, and many of these live ranges span only part of the basic block. By using smaller units of coloring, Chaitin's algorithm can have a lower chromatic number. Because we color at the intermediate code level, there are less candidates to be colored. Our interference graph is smaller, and the allocation will be faster, though the coarser interference graph may result in less effective allocation. We do not believe, however, that the difference in allocation results is significant. The effect of the size of coloring units on interferences is discussed in Section 11.

Use of registers. On the machine instruction level, all register usages are visible, and Chaitin can make all machine registers participate in coloring. On the intermediate code level, we have to exclude special-purpose registers from the allocation and also set aside enough registers to be used as scratch registers in later code generation.

Allocation models. The input to Chaitin's allocator is written in a language that assumes an unlimited number of symbolic registers. Chaitin's allocator maps these symbolic registers to the limited number of machine registers, possibly spilling some of them to memory. We use the model that assumes that all variables have home locations, and the allocator's task is to identify regions where the variables should be promoted to registers. In our approach, the program remains executable even if the register allocation phase is omitted. This difference extends to where variables are spilled to: Chaitin's allocator always spills to local memory in the current stack frame; by leaving variables unallocated, our allocator, in effect, always spills them to their home locations.

Register-residing candidates. Our register allocation applies to all program variables and globally occurring expression temporaries and constants. Local temporaries for translation purposes are handled by the code generator. Chaitin does not distinguish the local translation temporaries from the global temporaries, and all of them are allocated together. However, his approach excludes the allocation of global variables, since the model he uses assumes all live ranges start out in registers, and live ranges are always spilled to the local stack memory.

A second set of differences arises from the way in which coloring proceeds once the interference graph has been computed. These decisions are largely independent of the previous differences, and another algorithm might mix characteristics from both approaches.

Splitting of live ranges. Chaitin's algorithm never splits live ranges. A live range is either colored or spilled. Our algorithm can split a live range and either assign different colors to them or leave one of them unallocated. Likewise, the algorithms handle blockage differently. When the allocation process is blocked due to insufficient registers, Chaitin lowers the chromatic number of the interference graph by selectively spilling live ranges to memory. Instead of spilling, we split the live ranges that cannot be colored to reduce the number of interferences and to allow the allocation to continue.

Use of priority estimates. Both approaches use some cost–savings estimates to help select candidates or noncandidates. Chaitin's allocator finds the variables that are least costly to spill and spills them. We order the candidates according to their benefits of residing in registers and allocate them in that order.

Coloring passes. In our approach, we perform the coloring in one single forward pass. In Chaitin's approach, a forward pass is conducted to reduce the interference graph and to determine what spilling needs to be done, and then colors are assigned in a reverse-order backward pass.

Spill handling. When spilling is required, our method just stops coloring and leaves the variables in their home locations. When Chaitin spills a node, that node is eliminated from the interference graph, but it is still necessary to reload the variable at each use and to store into it at each definition. Thus, what actually happens is that a global node of higher degree is replaced by several local nodes of lower degree. Although this replacement will often reduce the chromatic number of the graph, this is not guaranteed. Sometimes, Chaitin's algorithm will have to iterate through the spilling process multiple times before the graph can finally be colored.

Overall, because Chaitin colors at the instruction level and includes all quantities visible at the machine code level in his allocation, his implementation requires much more memory and processing overhead than ours. However, the finer granularity of his interference graph could result in better allocation in some circumstances. We believe those circumstances are rare, and the advantage, small; the discussions in Sections 11 and 12 serve to confirm this view.

Chaitin also uses his coloring technique to eliminate unnecessary register-to-register copy operations. This is done by coalescing the nodes in the interference graph that are the sources and targets of copy operations. This optimization does not apply equally well at the intermediate code level, where earlier global optimization has already gotten rid of most of such redundancies.

7. REGISTER LINKAGE

The procedure call linkage convention affects register usage patterns between procedure boundaries. In this section we examine how our register allocation algorithm can be adapted to take advantage of opportunities arising from different linkage conventions.

7.1 Parameter Passing

A standard mechanism for passing parameters copies the outgoing parameters to the argument area in the activation record of the called procedure. Since the activation records are ordinarily allocated in the run-time stack, the passing process involves memory stores and the subsequent memory references of the incoming parameters in the called procedure. To reduce the number of memory references during procedure calls, parameters can be arranged to be passed via registers. The linkage convention usually sets aside a number of registers to hold the parameter list. The caller will put the ith parameter in the ith parameter register before the call, and the called procedure will fetch the ith parameter from the ith parameter register. To take full advantage of this parameter-passing convention, the register allocator should recognize these parameter registers, know their contents on both sides of a call, and use this information to reduce the memory references around the call.

At procedure entry, the parameters already reside in their corresponding parameter registers, and so it is cheapest for them to be referenced out of their incoming registers. Thus, at the procedure entry, the parameters are preassigned to their parameter registers (*precolored*). When their live ranges extend beyond the first basic block, the preassignments can extend with them. The parameters do not need to be homed unless the same parameter registers need to be made available for subsequent calls in the procedure body.

The same applies to the outgoing parameters at a procedure call. At the basic block of the call, the variables and expressions being passed are preassigned to their parameter registers. The preassignments apply to their live ranges, so that the parameter registers may be used to contain them at blocks preceding the calls. Since the parameter registers are caller-saved, the variables may need to be homed just before the call if they are live after the call.

There may be conflicts among such preassignments. For example, there may be a procedure call immediately at the entry block, so that different variables may be preassigned to the same parameter register. In such cases, the preassignment for the outgoing call preempts the preassignment at call entry. This will require the parameter to be transferred to another register immediately on entry so as to make room for the same parameter register to contain the parameter for the next call.

When a variable is used as a different parameter number over a region containing more than one call, the same live range may be preassigned to different parameter registers at the same time. We allow the same live range to be preassigned to more than one register. In the live range representation, a field called **precolors** gives the set of registers that the live range has been preassigned to. When the live range is to be assigned a register (step (2b) of Section 5.1), preference is given to one among its **precolors** set. In between calls, the parameter registers are used as ordinary caller-saved registers if they are not preassigned to contain the parameters in the upcoming calls. At each call, if a parameter is resident in a different register, a register-to-register move is issued to move it to the correct parameter register.

7.2 Caller-Saved and Callee-Saved Registers

One linkage convention used to provide extra opportunities to minimize call overhead is the simultaneous use of caller-saved and callee-saved registers. The machine's registers are divided into the two fixed sets. Because the two sets of registers are subject to different saving rules in their usages, the register allocator can choose the class that is best for the situation. We assume no knowledge of the register usage patterns between individual procedures. Thus, a procedure that does not make any other call (a *leaf* procedure) should favor the use of caller-saved registers, since their previous contents do not have to be saved before they can be used. If the body of a procedure contains many calls, callee-saved registers are more preferable because saving and restoring once at the procedure entry and exit, respectively, are cheaper than saving before each procedure call and restoring after the call. Caller-saved registers can be used for live ranges that do not extend across calls.

The main difficulty in supporting the simultaneous existence of these two classes of registers is in the treatment of live ranges. As is evident from Figure 2, the extents of live ranges for caller-saved registers are different from those for callee-saved registers. Computing two live ranges for each variable could potentially double the complexities of the register allocation. It is possible, however, to use only one live range by sacrificing some accuracy for the caller-saved live ranges. As seen in Figure 2, caller-saved live ranges are always smaller than callee-saved live ranges. We can use the callee-saved live ranges for both classes of registers as long as we also make explicit the need to insert register save code around procedure calls when using a caller-saved register.

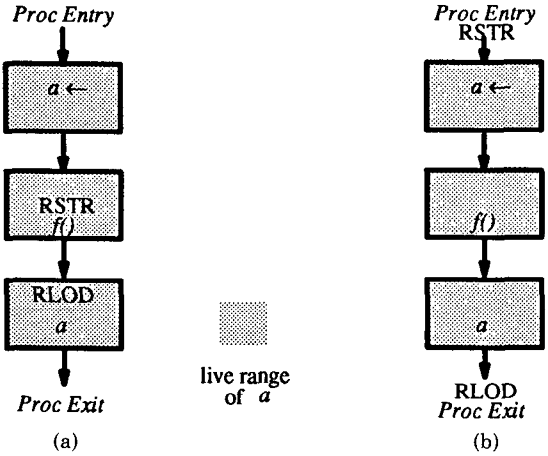

Fig. 5. Cost configurations in the two register classes. (a) Using caller-saved. (b) Using callee-saved.

The two register classes require different priority functions. For each live range, we call the caller-saved priority function $P_r(lr)$ and the callee-saved priority function $P_e(lr)$. $P_r(lr)$ and $P_e(lr)$ are different in the following respects:

(1) In using a caller-saved register for a live unit that contains a procedure call, an extra RSTR before the call and an extra RLOD following it may be required, depending on whether the home location is up-to-date and whether the variable is live. Thus, eq. (1) of Section 5.2 has a different value for n in computing $P_r(lr)$.

(2) $P_e(lr)$ should reflect the extra costs due to the requirement that any callee-saved register must be saved on procedure entry before they can be used, and its original content must be restored on exit. These extra costs occur only once for each callee-saved register. Thus, only the first live range that uses a given callee-saved register needs to account for these savings and restoring costs. Once these costs have been accounted for, the same callee-saved register can be used to contain other live ranges for free.

As the result of (2), the value of $P_e(lr)$ for a given live range alternates between two different values as coloring progresses. When a used callee-saved register is available to the live range, $P_e(lr)$ does not include the extra *first-use* costs. When all used callee-saved registers are in the live range's **forbidden** set, it will incur the first-use costs when it uses any callee-saved register. When someone else has used that register first, $P_e(lr)$ will fluctuate back to the former value.

By the above arrangement, the coloring algorithm will choose the particular class of registers according to whether the highest priority function $P^*(lr)$ is caller-saved or callee-saved (step (2b), Section 5.1). Figure 5 gives an example of the different cost configurations when using the two different register classes for a live range that spans a procedure call.

This scheme was found to distinguish between the two classes of registers quite effectively. In a leaf procedure, the algorithm uses up all caller-saved registers before it starts to use the callee-saved ones. In a nonleaf procedure, the callee-saved registers are much more in demand. By the time that all callee-saved registers are used up, the remaining live ranges are often found to be inappropriate for caller-saved registers because there are too many calls within the live ranges. Under such circumstances, we apply the splitting algorithm to the live ranges with special flags to tell it to regard live units that are delimited by calls as nonadjacent. This splits up the live ranges according to call boundaries to form the caller-saved live ranges, as depicted in Figure 2a, thus allowing the remaining caller-saved registers to be used effectively.

The priority-based coloring approach can also be extended to support interprocedural register allocation. This is discussed in detail in [5].

8. PLACEMENT OF REGISTER MOVES

As mentioned earlier, in each live unit, the flags **needrlod** and **needrstr** determine whether an RLOD and RSTR are required at the entry and exit, respectively, to the basic block. These flags do not include the effects of the linkage conventions at points of calls. We now look at how these two flags are determined.

8.1 Initial Estimates of RLODs and RSTRs

When the first occurrence of a variable in a basic book is at the left-hand side of an assignment, an RLOD is unnecessary, since its original value is destroyed and thus not referenced. If the first occurrence in the basic block is a reference, then we look at two different situations:

Case 1. The basic block is an entry point to the live range. This means that an immediate predecessor of the basic block does not belong to the live range. In this case, an RLOD is always needed.

Case 2. The basic block is not an entry point to the live range. In this case, an RLOD is needed only if the preceding basic block ends with a procedure call that has access to the variable. Otherwise, the register already contains the value of the variable on entry to the current basic block.

The **needrstr** flag depends on the **needrlod** flag of the adjacent basic blocks and thus is determined only after the **needrlod** flag has been determined for all the live units in the live range. If the whole live range does not contain any assignment to the variable, then RSTR is unnecessary in any live unit belonging to the live range, since the value at the home location is current with the value in the register. If the live range does contain assignments to the variable, then RSTR needs to be considered for insertion only at basic block exits that the assignments reach. RSTR is inserted only if the variable is not dead on exit, which only happens when one of the following condition applies:

Condition 1. The basic block is an exit point of the current procedure, and the variable is nonlocal.

Condition 2. The current basic block ends with a procedure call to a procedure that has access to the variable.

Condition 3. At any immediate successor block that belongs to the same live range, **needrlod** is true; this is because, if RSTR is not generated at the current basic block, the RLOD at the successor will not load the current value to the register.

Condition 4. The variable is live at the entry point of any immediate successor block that does not belong to the same live range; this only occurs when the live range has been split.

8.2 Optimizing RLODs and RSTRs

The above computation of **needrlod** and **needrstr** is not optimal for a given live range. First, there may be redundancies among the inserted RLODs or RSTRs. Second, it assumes that RLODs are always generated at basic block entry points, and RSTRs, at basic block exit points. RLODs can, in fact, be inserted at the exits of preceding basic blocks right after the RSTRs native to them. Similarly, RSTRs can be inserted at the entries of the succeeding basic blocks right before the native RLODs. These are the furthest extents that we can move the RLODs and RSTRs without changing the boundaries of the live ranges. Applying the following transformations will optimize the positions of RLODs and RSTRs.

RLOD optimization. When an RLOD is at the entry of a block with at least one immediate predecessor belonging to the same live range, then the RLOD can be deleted and inserted at the exits of all the immediate predecessors that are outside the current live range. When inserting the RLOD at the exit of a predecessor, if the current basic block is not its only immediate successor, then create a new block as a bridge between that predecessor and the current block, and insert the RLOD in the new block.

After the above optimization of RLODs, some RSTRs can be deleted. These are RSTRs internal to the live range that do not reach any RLOD within the live range, originally inserted due to Condition (3) of Section 8.1. These RSTRs are redundant with respect to the anticipated RSTRs inserted at the live range's exit points. However, any RSTR inserted because of accesses from procedure calls cannot be removed.

The remaining RSTRs can further be optimized by applying the following transformation:

RSTR optimization. When an RSTR is at the exit of a block with at least one immediate successor belonging to the same live range and when that successor does not have an RLOD at its entry point, then the RSTR can be deleted and inserted at the entries of all the immediate successors that are outside the current live range. When inserting the RSTR at the exit of a successor, if the current basic block is not its only immediate predecessor, then create a new block as a bridge between the current block and that successor, and insert the RSTR in the new block.

Figure 6 shows the results after the above optimizations are applied to the live range in Figure 3. The creation of bridging insertion blocks allows movements of the RLODs and RSTRs to occur. Figure 7 gives an example of the optimization applied to a loop. It can be seen that the effect is the code motion of the RLODs and RSTRs from within the loop to outside the loop. These optimizations never move them into loops because the body of a loop always terminates with a

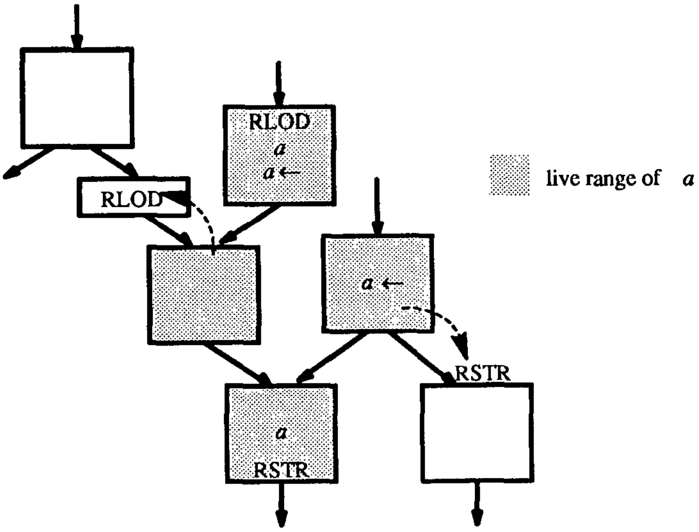

Fig. 6. Optimization of the placements of RLODs and RSTRs.

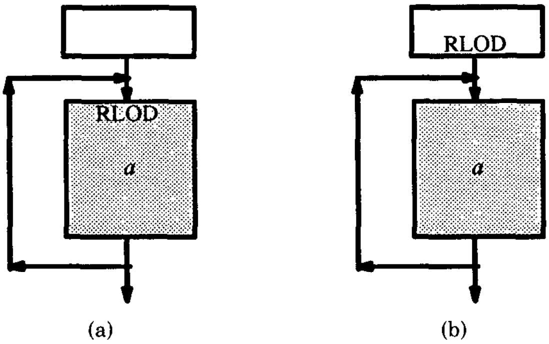

Fig. 7. Single reference of a global variable inside a loop.
(a) Zero savings. (b) Positive savings.

conditional branch and thus has more than one successor, and this backward branch also causes the loop body to have more than one predecessor.

The above optimizations of the positions of RLODs and RSTRs are actually examples of partial redundancy suppressions [13], applied in both the forward and backward directions with respect to the control flow. It is simplified because we never need to move across the body of a block. Also, we allow the creation of insertion blocks to enable more redundancy suppression that could have been possible otherwise.

By using the optimized positions of RLODs and RSTRs to compute register allocation costs, we can arrive at more exact estimates of the benefits of allocating items in registers. One example is in the case of a single reference occurring inside a loop (Figure 7). In this case, the live range consists of only the body of the loop. If the RLOD is kept inside the loop, there can be no run-time savings incurred by allocating the item in register. But, if the RLOD is inserted at the loop header, the advantage is obvious, and the live range will be accorded high priority. In computing the priority function at a loop header, we specifically recognize the RLODs that can be moved out to reduce the frequency weights accorded to them.

9. REGISTER TYPES

Depending on the architecture, the target machine may provide different register types to hold different types of data: integer values, floating-point values, addresses, etc. Under such circumstances, the coloring algorithm can be applied separately for each class of data types. Conflicts among live ranges designated for different register types can be ignored. Thus, there is a separate interference graph for each register data type.

A situation that may arise is the existence of overlaps among the registers. Within the same register type, different register sizes may be provided to hold different sizes of data. For example, a word-sized register may alternatively be used as two half-word-sized registers. In such a situation, we use the smallest-sized registers as the register set. A large data item will then simultaneously occupy more than one register. For a variable that requires two small registers to hold it, it is regarded as constrained if its number of neighbors reaches half the total number of small registers. This is because its neighbors could use up alternate registers so that no contiguous register pair is left to guarantee the colorability of the variable.

When overlap exists between the different register types, the different data-type classes can no longer be allocated separately. It is necessary to construct a single interference graph for all of them and to perform coloring over all the live ranges at the same time.

10. SPLITTING STRATEGIES

The basic splitting algorithm is described in Section 5.3. The splitting algorithm is conservative in that it tries to form the largest live range that is still colorable (step (2) of the algorithm). In circumstances when a large number of register-residing candidates yields an interference graph with a chromatic number much larger than the number of registers, many splits are required to obtain a completely colorable graph. In such cases, heuristics that encourage smaller live ranges to be formed earlier can speed up the algorithm by reducing the number of intermediate splits.

The basic purpose of splitting is to reduce the number of interferences for the split live ranges. Although the number of live ranges is increased, each smaller live range usually interferes with fewer other candidates. This is because the live ranges occupy smaller regions, so that the probability of overlaps with other live ranges is reduced. This is illustrated in Figure 8. The following two heuristics for splitting are based on this fact:

Heuristic A: Split when there are too many neighbors. If the number of uncolored neighbors in the interference graph is too many compared to the number of colors left, the live range can be split earlier. The reason is that, even if a color

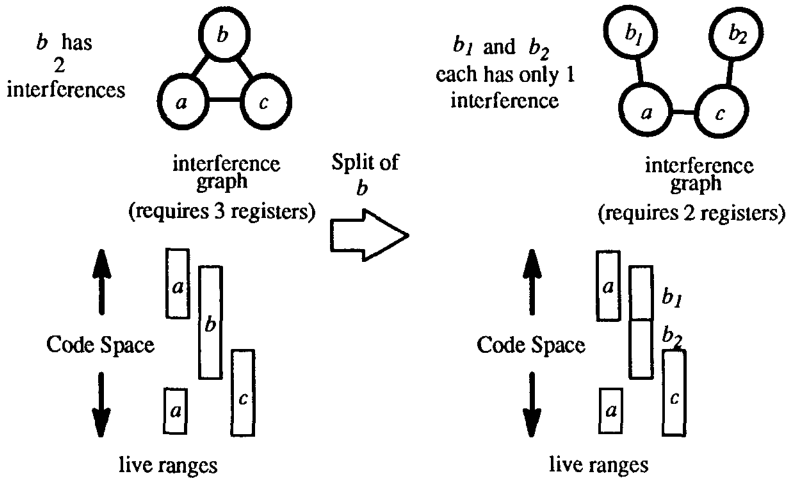

Fig. 8. Effect of splitting on colorability.

is assigned to it, it may seriously affect the ability to color its neighbors. This heuristic is only useful when the live range's neighbors in turn have numerous interferences among themselves. We find it useful to initiate a split when the number of neighbors is greater than twice the number of colors left; in such cases, the probability of successfully assigning colors without requiring any split is small.

Heuristic B: Do not add a basic block if it brings in too many new neighbors (used when finding the new live ranges during splitting). As described in Section 5.3, in splitting a live range, we start with an initial basic block and add adjacent blocks to it one-by-one. Each addition of a basic block brings in new neighbors for the live range being formed because there are live ranges that start in that basic block. If the number of new neighbors is too many, then it is not worthwhile to include that basic block, since the colorability of the new live range would be seriously affected. To avoid overfragmentation, we do not add a basic block if it causes an increment in the number of neighbors that is greater than or equal to the number of colors left for the live range before the addition.

Heuristic A is used in step (2c) of Section 5.1 in checking which live range should be split. Heuristics A and B combined can be used in step (2) of Section 5.3. This implies that a basic block is added only if both the absolute total and the net increase in the number of neighbors are not too great. We will show the effectiveness of these heuristics on the allocation time later on. In general, they improve running time, while their effect on the results is negligible.

11. PERFORMANCE MEASUREMENTS

Earlier papers [4, 6, 11] have provided measurements and evaluations on the behavior of the priority-based coloring algorithm under different parameters and various settings. Before presenting additional data, we summarize these results:

Number of registers. Increasing the number of registers is always beneficial, since it allows more variables to be assigned to registers. Beyond a certain point,

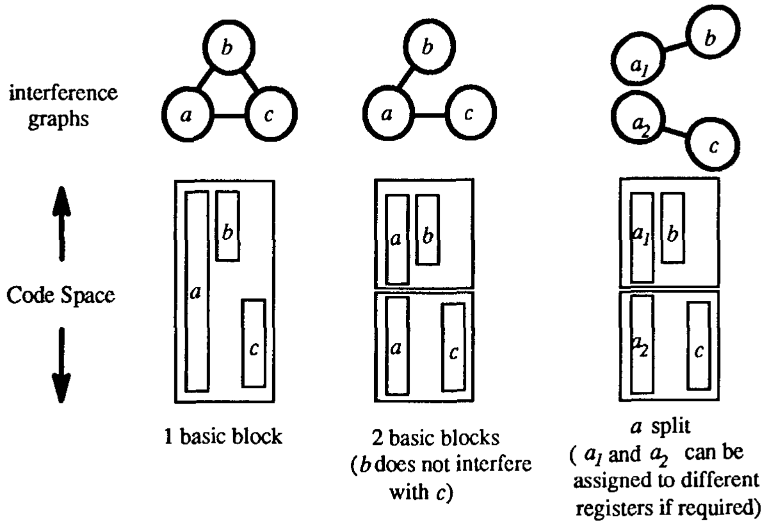
interference graphs
Code Space
b
a
c
a
b
c
1 basic block
2 basic blocks
(b does not interfere with c)
a split
(a_1 and a_2 can be assigned to different registers if required)
a_1
a_2

Fig. 9. Effects of smaller basic blocks.

each additional register yields a diminishing return, since it is not as heavily demanded as the first few. Data from [6] confirm that increasing the number of registers beyond the chromatic number does not improve the running time of the program. This is especially evident in programs with no large procedures, where the maximum chromatic number among all the procedures is usually small. When the number of registers is small and less than the chromatic number, however, the algorithm can still make effective use of them, as data from both [6] and [11] show. Measurements from [11] also show that the algorithm runs faster when there are more registers, since less splitting is needed to arrive at the results.

Basic block size. The allocation results tend to be more optimal as the limit on basic block sizes is decreased. This is because the finer division of code space allows greater flexibility in the drawing of register usage boundaries. Figure 9 shows how a smaller basic block size can result in fewer live variables in each basic block, and thus fewer edges in the interference graph and easier coloring. This effect is especially evident when there are many candidates whose live ranges are smaller than the basic block size limit. Data from [11] show that, when basic blocks are made smaller, less spill code is needed, which leads to better allocation results. Although smaller basic block size leads to simpler interference graphs (in terms of the number of edges), it also increases the number of nodes in the control flow graph and in the live ranges. These always increase the time needed for data-flow analysis and live-range operations (like splitting). Because of these two opposing effects, Larus and Hilfinger [11] found that the processing time for register allocation is roughly constant. Hence, smaller-sized blocks can be used to increase the allocator's performance. If the blocks are too small, however, many small live ranges have to span more than one basic block, and the interference graph cannot be further simplified. At this point, the processing time will only increase.

Register usage advantage. The algorithm will allocate more variables in registers when the values of *LODSAVE* and *STRSAVE* are higher. Data in [4] show that, for a given machine, values for *LODSAVE* and *STRSAVE* exist for best performance, although these values are not always obvious and may be best determined by experimentation. For register-memory machines, the optimal allocation may not put all candidates into registers, but may choose to leave some in memory and access them with memory addressing modes instead.

In this section we supplement the above results by providing dynamic measurement data based on some large realistic programs. Some of the data also serve to validate the merits of the techniques we have discussed in this paper.

The measurements are generated by the MIPS optimizing compilers [7], which implemented the techniques we have presented.[1] The target machine is the MIPS R2000 processor,® a RISC machine with a load/store architecture in which most instructions execute in a single clock cycle [14]. The run-time data are generated using the MIPS instruction tracing facility, *pixie*, a part of the profiling tools provided by the compiler suite [12]. This tool gives an exact count of the total number of instruction cycles needed to execute the program, and the total number of loads and stores. Thus, the data give an indication of the program running time independent of the other hardware effects, such as caches and processor clock frequency.

Our measurement data are based on seven benchmarks. The first four are UNIX® utilities that represent different nontrivial tasks: *bm* is the Boyer–Moore fast pattern matcher, *diff* is the file comparison utility, *yacc* is the general-purpose parser generator, and *nroff* is the UNIX document processor. The remaining three are components of the MIPS compiler suite: *ccom* is the MIPS C front end, *upas* is the MIPS Pascal front end and *as1* is the MIPS assembler/reorganizer. All seven programs use only integer data. Only a small portion of the library routines used by these programs are written in assembler language. The nonassembler portion of the libraries were recompiled in similar mode each time, so that the numbers shown represent the full effects. More data about the benchmarks are given in Table I.

11.1 Overall Performance

The MIPS R2000 processor provides 31 general-purpose registers. Excluding those designated for system or code generation/assembler usages, there are altogether 21 registers available for use by the global register allocator. Among these 21 registers, 12 are caller-saved, and 9 are callee-saved. Among the caller-saved registers, 4 are used for parameter passing, and 1 is designated as the function return register. Between calls, the parameter registers and function return register can be used by the register allocator. Register allocation reduces the total running time of programs by eliminating load and store instructions. In addition, pipeline constraints require an extra cycle of latency for each load instruction, for which the reorganizer in the final assembly phase is able to schedule other useful operations 65 percent of the time. The net effect is that each load instruction costs about 1.35 cycles. On many processors, memory

[1] All the measurements were generated using Release 1.31 of the compiler suite.

® R2000 and R3000 are Trademarks of MIPS Computer Systems, Inc.

® UNIX is a trademark of AT&T Bell Laboratories.

Table I. Benchmark Data

Program	Language	Source line count	Static code size (bytes)	Cycles executed/call
bm	C	431	20,480	273
diff	C	667	28,672	148
yacc	C	2,257	49,152	200
nroff	C	7,258	61,440	55
ccom	C	13,748	188,416	57
upas	Pascal	16,487	274,432	48
as1	Pascal	11,260	196,608	66

references are much more expensive, and register allocation can have an even greater effect.

Table II shows the overall performance of the register allocator by comparing the dynamic benchmark execution data without and with register allocation. The running times of the benchmarks in cycles are reduced by a geometric mean of 28 percent when all the capabilities of the register allocator are enabled. The immediate effect of performing register allocation shows up in the reduction of the total number of load and store instructions executed by the programs, also displayed in the table. Part of the total load and store instructions are due to array and pointer accesses, which cannot be eliminated by register allocation. The remaining loads and stores are attributed to scalar variables, saved common subexpressions, and register saves and restores (at procedure calls, procedure entries and exits), which we collectively refer to as *scalar* loads and stores. Scalar loads and stores are removable by register allocation. Table II shows that our register allocator removes a mean of 75 percent of all scalar loads and stores. Part of the remaining scalar loads and stores are not removed because of aliasing.

11.2 Effects of RLOD and RSTR Optimizations

Table III shows the effects of the RLOD and RSTR optimizations we discussed in Section 8.2. Using the default register allocation data as reference, Table III shows the percent increase in execution times and loads and stores if the RLOD and RSTR optimizations are disabled. These optimizations always reduce the total number of loads and stores. The same optimizations may introduce additional branches into the program (as shown in Figure 6). The additional branch instructions, together with their associated branch delay slots, which may or may not be filled with useful operations, can offset the gain in execution speed brought about by the reduction in the number of loads and stores. But in our benchmarks there is net reduction in the total number of cycles executed. The execution times improve by a mean of more than 1 percent due to these optimizations.

11.3 Effects of Precoloring Parameters

In Section 7.1 we discussed the preassignment of parameters to the parameter registers to reduce the movement of register contents during procedure calls. Without this precoloring, additional register move instructions or load and store instructions will be incurred at calls. Table IV shows the effects of the precoloring technique and the usefulness of designating four registers as parameter registers.

Table IIa. Overall Effects of Global Register Allocation

	Without register allocation			With register allocation		
Program	Cycles	Total loads/stores	Scalar loads/stores	Cycles	Total loads/stores	Scalar loads/stores
bm	67,914	37,029	29,607	42,401	8,542	1,120
diff	5,425,351	2,881,526	2,162,270	3,220,852	881,807	162,551
yacc	6,656,398	3,387,930	2,723,907	4,579,316	1,088,748	424,725
nroff	24,581,785	9,749,653	8,757,185	20,564,118	4,962,657	3,970,940
ccom	11,652,148	5,488,283	4,341,541	8,738,096	2,573,021	1,426,136
upas	26,247,734	11,892,372	8,084,750	19,598,648	6,163,554	2,355,493
as1	8,263,939	4,261,986	3,289,195	5,746,910	1,933,127	960,517

Table IIb. Overall Performance of Global Register Allocation

	Percent reduction in		
Program	Cycles	Total loads/stores	Scalar loads/stores
bm	37.6	76.9	96.2
diff	40.6	69.4	92.5
yacc	31.2	67.9	84.4
nroff	16.3	49.0	54.7
ccom	25.0	53.1	67.2
upas	25.3	48.2	70.9
as1	30.5	54.6	70.8
Geometric mean	28.4	59.0	75.4

Table III. Effects of Disabling RLOD and RSTR Optimizations

	Percent increase in		
Program	Cycles	Total loads/stores	Scalar loads/stores
bm	4.2	21.5	163.0
diff	0.1	0.3	1.5
yacc	2.6	3.9	10.0
nroff	0.4	1.8	2.2
ccom	2.1	2.1	3.7
upas	2.8	3.0	7.9
as1	1.3	1.7	3.4
Geometric mean	1.07	2.41	6.6

If precoloring is disabled, the running time increases in all the benchmarks. The changes in the number of loads and stores are always less than the changes in the number of cycles. This is because the gain in execution speed is also due to the elimination of extra move instructions between parameter registers and nonparameter registers at the calls.

Since precoloring is applicable only at procedure calls, its effect is greater in call-intensive programs. As the data from Table I show, *bm*, *diff*, and *yacc* have greater average cycles executed per procedure call compared with the rest

Table IV. Effects of Disabling the Precoloring of Parameters

	Percent increase in		
Program	Cycles	Total loads/stores	Scalar loads/stores
bm	0.3	0.8	5.9
diff	0.3	0.0	0.2
yacc	0.2	0.1	0.2
nroff	1.5	2.2	2.8
ccom	0.8	0.4	0.7
upas	0.6	0.0	0.1
as1	0.9	0.2	0.3
Geometric mean	0.55	0.20	0.55

of the benchmarks, and this accounts for the diminished effect of precoloring on their total execution cycles (0.3 percent, 0.3 percent, and 0.2 percent, respectively), versus a mean of almost 1 percent for the other four programs.

11.4 Effects of Allocating Constants

Table V shows the effects of the allocation of constants to registers. This allocation does not reduce the number of load and store instructions, since constants are not loaded to registers from memory, but via the immediate field in the instruction, followed possibly by additional arithmetic operations for long constants. In the benchmarks, when constants are allocated to registers, the number of loads and stores actually increases due to register saves and restores at procedure entries and exits. With the exception of *ccom*, all the benchmarks benefit from the allocation of constants. The result for *ccom* demonstrates the shortcomings of relying on static program information in performing optimizations. In *ccom*, there are a number of program loops that seldom iterate more than once. Our register allocator arbitrarily assumes each loop is executed ten times, so it uses callee-saved registers to keep constants occurring inside the loops. Because of the lower number of iterations, the gain due to the allocation is not sufficient to offset the costs of saving and restoring the callee-saved registers. This effect can be prevented by the use of profile information in performing register allocation.

11.5 Effects of Calling Conventions

In this section we investigate the effects of calling conventions on program running time. We consider three different calling conventions. The first is the pure caller-saved calling convention, when all registers are used in the caller-saved mode. The second is the pure callee-saved calling convention, when all registers are used in the callee-saved mode. The third is the mixed calling convention, when part of the registers are used caller-saved and the rest are used callee-saved. In the last case, the technique discussed in Section 7.2 is used in the allocation algorithm.

At the current stage of development, different parts of the MIPS compiler software have been hard-coded according to the native calling convention of twelve caller-saved and nine callee-saved registers. To test the above three

Table V. Effects of Disabling the Register Allocation of Constants

	Percent increase in		
Program	Cycles	Total loads/stores	Scalar loads/stores
bm	0.1	0.0	−1.8
diff	1.2	−1.8	−9.8
yacc	1.6	−2.5	−6.4
nroff	0.8	−0.1	−0.1
ccom	−0.7	−3.0	−5.5
upas	1.5	−1.6	−4.2
as1	0.9	−0.6	−1.1

different calling conventions and to make a fair comparison, it is necessary to limit the number of registers used. In each of the three different setups, the register allocator is only allowed to use a total of eight registers. The four parameter registers are disabled. In Table VI we show the performance of the register allocator under these three situations. The first column of Table VI shows the results for the native configuration, reproduced from Table II for comparison.

Table VI shows that the mixed calling convention is the best among the three. When only eight registers are available, the four-caller-saved and four-callee-saved setup outperforms the other two. As explained in Section 7.2, the availability of both kinds of registers provides an extra degree of freedom in the optimization of register usages. The eight-caller-saved and zero-callee-saved setup is a close second in performance; it is actually the best in the case of *yacc*. The zero-caller-saved and eight-callee-saved setup offers the worst results.

The poor performance of the pure callee-saved convention can be explained as follows: A typical procedure, especially a large one, consists of many different execution paths, not all of which are exercised for each invocation. By saving registers at the entry and restoring at the exit, some redundancy is introduced because the saved registers may not be used for the execution path of that invocation. Thus, for callee-saved registers, the real costs relative to savings cannot be accurately evaluated at compile time, and there is a tendency to overutilize them. This problem does not occur for the caller-saved registers, since the saves/restores occur at call sites within the live ranges, and these registers are saved/restored only when they are being used. Our conclusion is contrary to the traditional acceptance of the callee-saved convention as the better protocol. The callee-saved convention does lead to smaller code size. However, previous measurements that show the superiority of the callee-saved convention are not based on compilers performing global register allocation [10]. When the compiler makes use of complete program information, the caller-saved convention can provide better results.

Table VI provides another glimpse at the performance of our register allocator. With four-caller-saved and four-callee-saved registers, the register allocator reduces the execution times by a mean of 26 percent, 2 percent less than that under the native register configuration of twelve-caller-saved and

Table VI. Effects of Different Calling Conventions

	Percent reduction in cycles due to register allocation			
Program	12 caller-saved, 9 callee-saved	8 caller-saved, 0 callee-saved	0 caller-saved, 8 callee-saved	4 caller-saved, 4 callee-saved
bm	37.6	32.1	31.7	32.6
diff	40.6	34.6	34.2	36.0
yacc	31.2	28.3	26.8	28.0
nroff	16.3	13.2	9.0	14.8
ccom	25.0	23.1	18.2	23.2
upas	25.3	20.5	17.4	24.9
as1	30.5	27.7	22.3	28.3
Geometric mean	28.4	24.6	21.1	26.0

nine-callee-saved registers. Thus, the algorithm is effective at making use of a small number of registers.

12. STATIC MEASUREMENTS AND ALLOCATION TIME

One aspect of the algorithm that has not been adequately investigated in earlier papers is the effect of procedure size. In Section 2 we stated that a large procedure yields more candidates with large live ranges and thus more interferences. Here we present some static allocation data and allocation time measurements to show how the algorithm behaves on progressively larger procedure sizes.

Table VII gives the characteristics of the register allocation candidates in a sample of subroutines of different sizes. Since it is difficult to find large procedures written in C or Pascal, we change our measurement setting to FORTRAN code. To avoid irregularity in code style and to make a fair comparison, we limit our subroutine selection to a single program: *hspice*, a popular circuit simulation program. The program has extensive use of floating-point data. Thus, the floating-point registers of the MIPS R2010® coprocessor also come into play. Among the floating-point registers of the R2010, there are two parameter registers, two function return registers, two caller-saved registers, and six callee-saved registers available for use by the register allocator. Since the parameter registers and function return registers are in fact caller-saved, the number of caller-saved registers actually totals six, the same as the number of callee-saved registers.

In Table VII, the procedure sizes can be represented either by the number of source lines that the procedures are written in or by the number of basic blocks in the procedure, although the latter is more appropriate in the context of register allocation. The basic block counts appear large compared with the source line counts because procedure calls delimit basic block boundaries. The number of procedure calls in each procedure is also shown.

Next, Table VII shows details about the live ranges and live units in each procedure. The original number of live ranges corresponds to the number of register-residing candidates before the application of the coloring algorithm. The next entry shows the number of aliased live ranges; a live range is aliased if the

® R2010 is a trademark of MIPS Computer Systems, Inc.

Table VII. Characteristics of Allocation Candidates in Procedure Samples

	Subroutine name								
	moseql	reordx	moseq3	jfet	modml	mos5a	tmpupd	mosfet	disto
Source line count	42	186	244	313	573	346	723	1,030	484
Basic block count	31	154	82	119	599	244	669	480	474
Number of procedure calls	2	39	10	31	70	56	370	81	215
Number of original live ranges	32	93	156	132	244	155	341	357	389
Number of aliased live ranges	0	19	0	20	109	0	11	74	9
Number of live units (unaliased)	195	1,573	1,866	5,665	55,243	6,560	31,983	79,246	51,005
Number of aliased live units	0	722	0	100	222	0	1,793	243	180
Total unaliased occurrences	117	510	716	714	1,433	1,646	2,252	2,474	3,362
Total aliased occurrences	0	66	0	10	0	0	5	12	22

variable it represents is ever affected by aliasing over any part of the procedure. The next entries show the total number of unaliased live units that can be register-allocated, followed by the total number of aliased live units. The aliased live units were removed from the original live ranges at the beginning of register allocation. In our sample, only 13 percent of the original live ranges and 1.4 percent of the live units are affected by aliasing. But the actual percentage varies widely between procedures. This percentage also depends highly on the source language, and it is usually lower for FORTRAN and Pascal programs and higher for C. Dividing the number of live ranges into the number of live units gives the average size in basic blocks of the live ranges in each procedure, and this, in general, increases as the size of the procedure increases.

The next to last entry of Table VII shows the total static unaliased occurrences of all the allocation candidates in the live units. Dividing the total number of live units into the total occurrences gives the average static density of occurrences. This density is higher for small procedures and lower for large procedures, ranging from 0.6 to 0.03 occurrences per live unit. This is due to the fact that in larger procedures live ranges are larger and the occurrences of each variable are farther apart. The last entry of Table VII shows the total static aliased occurrences. On average, only 0.9 percent of the occurrences are affected by aliasing. Since aliasing is independent of occurrences, this ratio does not vary significantly from the above ratio of 1.4 percent in terms of live units.

Table VIII presents data on the application of the coloring algorithm. The splitting performed by the algorithm on the original live ranges yields the final number of live ranges shown. The next six rows show a breakdown of these final live ranges. Rows (a) and (b) show the number successfully assigned a color and the number not assigned because of insufficient registers, respectively. Row (c) shows the number of live ranges that have no occurrence of a variable, representing useless regions being split out from other live ranges and thus not

Table VIII. Register Allocation Results and Timing in Procedure Samples

	Subroutine name								
	moseq1	reordx	moseq3	jfet	modml	mos5a	tmpupd	mosfet	disto
Number of original live ranges	32	93	156	132	244	155	341	357	389
Number of final live ranges	38	99	275	218	411	554	638	751	1,478
(a) Colored live ranges	19	59	83	70	81	135	137	163	538
(b) Live ranges cannot color	0	0	0	0	0	0	0	0	36
(c) Zero-occurrence live ranges	3	3	6	19	77	4	67	70	163
(d) Too much call overhead	0	0	12	0	0	7	0	0	0
(e) Shapes not well formed	0	0	0	3	1	3	0	6	5
(f) Sparse occurrences	16	37	174	126	252	405	434	512	736
Contiguous final live ranges	30	74	240	191	334	463	552	675	1,331
Total allocated occurrences	86	435	403	510	867	814	1,499	1,428	2,236
Allocation time (s)	0.04	0.28	0.92	0.78	4.80	4.94	8.60	11.81	31.32

assigned to a register. Row (d) shows the number of live ranges that are not allocated because of save/restore overhead associated with procedure calls; without such overhead, they could have benefited from being put in registers instead of being left for the code generator to manage locally. Row (e) shows the number of live ranges not allocated because their shapes are not well formed; they have too many entry and exit points. We classify a live range this way when the number of inserted RLODs and RSTRs is more than three-quarters of the total number of nodes in the live range. Row (f) shows the number of live ranges not allocated because the number of occurrences of the variables is too low, taking loop structures into account; the occurrences are either too sparse or are not high enough to justify the RLODs and RSTRs that need to be inserted. The live ranges belonging to rows (e) and (f) have negative priority functions; that is, the benefits of assigning them to registers do not justify the costs.

Due to the need to accommodate caller-saved registers, as discussed in Section 7.2, many live ranges are split a number of times before they are assigned to registers or are left unassigned. This accounts for the difference between the final and the original numbers of live ranges shown in Table VIII. As the size of the procedure increases, the number of splits increases at a higher than linear rate, due to the higher degree of interferences. The number of colored live ranges is only a fraction of the final live ranges. Most of the live ranges are left uncolored due to sparse occurrences, which often apply to arithmetic and address constants. The algorithm is effective in leaving only a few live ranges for which it cannot find a color, as shown in row (b).

Continuing down Table VIII, the next entry shows the number of live ranges among the final live ranges that are contiguous. The data show that 87 percent

Table IX. Register Allocation Efficiencies

	Subroutine name								
	moseql	reordx	moseq3	jfet	modml	mos5a	tmpupd	mosfet	disto
Efficiency quotient	29.5	52.6	24.2	32.8	20.3	67.6	48.4	30.1	22.3

$$\text{Efficiency quotient} = \frac{\text{Basic block count} \times \text{Number of final live ranges}}{\text{Allocation time (in ms)}}$$

of the final live ranges are contiguous. This justifies the omission of a separate step at the beginning of register allocation to separate out the connected components of each live range, as discussed in Section 4.1.

The next to last entry of Table VIII shows the static count of the occurrences that are allocated to registers. The ratio of the allocated occurrences to the total unaliased occurrences shown in the last row of Table VII gives the percentage of allocated static occurrences. It ranges from 85 percent to 50 percent, and is lower for larger procedures where there are more sparse occurrences. Comparison of this ratio with the reduction in the number of dynamic scalar loads and stores in Table IIb shows that the static and dynamic measures of allocation fall into roughly the same range. From row (f) of Table VIII, we know that most of the unallocated references are due to sparse occurrences.

The allocation times shown in the last row of Table VIII are the running times of the coloring algorithm on the MIPS M/1000® system and do not include the time for data-flow analysis prior to register allocation. The subroutines are arranged in order of increasing allocation time.

Since the priority-based allocation algorithm allocates variables to register without backtracking, the amount of work in the allocation is roughly proportional to the number of live ranges to be allocated. For each live range, the amount of processing is roughly proportional to the number of basic blocks it spans. Because splitting and the use of heuristics are involved, it is hard to give an exact formulation of the complexity of the algorithm. The complexity of the allocation problem, however, can be estimated as the product of the number of basic blocks and the final number of live ranges. The efficiency of the allocation can be measured by the ratio of the allocation time to this measure of complexity. As shown in Table IX, the value of this efficiency quotient varies quite irregularly across different procedures. For the range of procedure sizes ordinarily encountered in practical programs, represented by the above samples, the efficiency quotient does not vary by more than a factor of 3.

When Heuristics A and B, discussed in Section 10, are not used, Table X shows that the allocation time increases by an average of about 50 percent. This speedup is not due to the net reduction in the number of splits, which actually increases in all the subroutines shown. Instead, the speedup is due to the reduction in the number of intermediate splits needed to arrive at the final colored live ranges. The larger number of splits when Heuristics A and B are used can be

® M/1000 is a trademark of MIPS Computer Systems, Inc.

Table X. Allocation Results with Heuristics A and B Disabled

	Subroutine name								
	moseq1	reordx	moseq3	jfet	modm1	mos5a	tmpupd	mosfet	disto
Number of original live ranges	32	93	156	132	244	155	341	357	389
Number of final live ranges	37	103	233	182	369	499	595	581	1,390
Colored live ranges	18	57	76	63	80	135	135	131	423
Live ranges cannot color	0	0	0	0	0	0	0	0	68
Allocation time (s)	0.04	0.33	0.89	0.91	12.38	4.65	11.88	18.03	36.58

attributed to the uncolored live ranges. Because they are immediately removed from the picture once they are determined unsuitable for assignment to registers, these extra splits do not slow down the allocation process. Separate measurements of the running times and the number of loads and stores show that they are not noticeably affected by the use of Heuristics A and B.

13. CONCLUSIONS

To improve program performance effectively, modern compilers must make good use of the fastest memory resource available to it: the set of hardware registers provided by the target machine. The kind of performance improvement that global register allocation can bring about has made it a discipline impossible to ignore. The priority-based coloring scheme presented in this paper provides a well-balanced compromise between good global allocation and reasonable compile-time cost. The algorithm is also flexible enough to be adapted to different architectures, language environments, and program characteristics.

Most of what is discussed in this paper has been implemented in the MIPS Optimizing Compiler Suite [7]. As of this writing, this compiler has been in production releases since September 1986, being shipped with R2000/R3000-based systems built by different computer manufacturers. During this period, it has compiled and optimized thousands of nontrivial programs, including the RISC/OS UNIX kernel® and numerous third-party software packages, giving further proof of the robustness of this methodology.

ACKNOWLEDGMENTS

The authors wish to thank members of the MIPS compiler team for their support throughout implementation and testing. In addition, Steve Correll, Steve Hanson, Earl Killian, John Mashey, and Larry Weber have provided helpful ideas and suggestions. This final paper also incorporates useful comments by the referees, especially Gregory Chaitin; information he provided about his algorithm has helped us to significantly improve this paper.

® RISC/OS is a trademark of MIPS Computer Systems, Inc.

REFERENCES

1. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction* (Boston, June 1982). ACM, New York, 1982, pp. 22–31.
2. CHAITIN, G. J. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction* (Boston, June 1982). ACM, New York, 1982, pp. 98–105.
3. CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. Register allocation via coloring. *Comput. Lang. 6* (1981), 47–57.
4. CHOW, F. A portable machine-independent global optimizer—Design and measurements. Ph.D. thesis and Tech. Rep. 83-254, Computer System Lab, Stanford Univ., Stanford, Calif., Dec. 1983.
5. CHOW, F. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, June 1988). ACM, New York, 1988, pp. 85–94.
6. CHOW, F., AND HENNESSY, J. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction* (Montreal, June 1984). ACM, New York, 1984, pp. 222–232.
7. CHOW, F., HIMELSTEIN, M., KILLIAN, E., AND WEBER, L. Engineering a RISC compiler system. In *Proceedings of COMPCON* (San Francisco, Mar. 4–6, 1986). IEEE, New York, 1986, pp. 132–137.
8. FREIBURGHOUSE, R. A. Register allocation via usage counts. *Commun. ACM 17*, 11 (Nov. 1974), 638–642.
9. HECHT, M. S. *Data Flow Analysis of Computer Programs.* North-Holland Elsevier, New York, 1977.
10. LANG, T., AND HUGUET, M. Reduced register saving/restoring in single-window register file. *Comput. Arch. News 14*, 3 (June 1986), 17–26.
11. LARUS, J. R., AND HILFINGER, P. N. Register allocation in the SPUR Lisp compiler. In *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction* (Palo Alto, June 1986). ACM, New York, 1986, pp. 255–263.
12. MIPS COMPUTER SYSTEMS, INC. *MIPS Language Programmer's Guide.* MIPS Computer Systems, Inc., Sunnyvale, Calif., 1986.
13. MOREL, E., AND RENVOISE, C. Global optimization by suppression of partial redundancies. *Commun. ACM 22*, 2 (Feb. 1979), 96–103.
14. MOUSSOURIS, J., CRUDELE, L., FREITAS, D., HANSEN, C., HUDSON, E., PRZYBYLSKI, S., RIORDAN, T., AND ROWEN, C. A CMOS RISC processor with integrated system functions. In *Proceedings of COMPCON* (San Francisco, Mar. 4–6, 1986). IEEE, New York, 1986, pp. 126–137.

Received January 1988; revised May 1989; accepted November 1989