Optimal Local Register Allocation for a Multiple-Issue Machine

Waleed M. Meleis Edward S. Davidson

Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
waleed@fiddler.eecs.umich.edu davidson@eecs.umich.edu

ABSTRACT

This paper presents an algorithm that allocates registers optimally for straight-line code running on a generic multi-issue computer. On such a machine, an optimal register allocation is one that minimizes the number of issue slots that the code requires. Optimal spill selection and load/store placement are used to minimize the number of additional issue slots needed, given a schedule for the non-memory reference instructions and a fixed number of available physical registers. The generic multi-issue machine model closely models the operation of vector and VLIW processors, and could be extended to model super-scalar processors. The algorithm uses dynamic programming to search the state space of plausible register allocations; implicit and explicit state pruning are used to make the problem tractable. The optimal allocation produced by the algorithm for a substantial example is presented.

1 Introduction

High performance processors are increasingly memory bottlenecked due to increasing processor issue bandwidth and clock speeds relative to the limited number of ports and access latency to memory. Reducing memory traffic by improving data reuse at all levels of the memory hierarchy is therefore essential for improving system performance. Machines that execute more than one operation per clock are becoming increasingly common, but the problem of reducing their memory bottleneck through optimal register allocation has not been solved heretofore. In this work we consider a straight-line code segment before registers have been allocated and before memory references have been inserted. We present an algorithm that chooses values for spilling and places load and store instructions in the code segment to use only a specific number of registers and achieve minimum runtime on a generic multi-issue machine.

Heuristic solutions to local and global register allocation problem for single-issue machines have been discussed by Chaitin et al [6], [7], Chow et al [4], and Aho et al [1]. Horwitz et al. [8] first addressed the problem of finding an allocation of index variables to registers that minimizes the number of load and store spills that are required for a basic block. This approach is extended in Hsu et al. [9] to handle a wider class of schedules. A single-issue machine is assumed in each case, so the goal of the algorithm is to minimize the *number* of loads and stores that are needed. This paper discusses the extension of the single-issue load/store minimization algorithm for a multi-issue machine.

On a multi-issue machine, the latency of a memory operation may be hidden by executing other instructions simultaneously. Therefore the *placement*, or insertion points, of the loads and stores in the schedule is as important as the total number required. Consider a straight-line code segment whose instructions operate on and produce *values* (or virtual registers): quantities that are written (defined) once and read (used) one or more times. Input values are considered to be defined before the start of the code segment and must be loaded before being used. Output values are defined within the code segment and are considered to be used sometime after the segment ends. They must be stored after being defined. A value is *live* at a point in the schedule if it is defined there or earlier and used at that point or later. A *register allocation* is a selection, at every point in the code segment, of values to keep in registers. A *legal* allocation is one where each value is in a register when defined or used, and the number of values in registers at each point never exceeds the number of available registers. A live value that is not in a register at some point is said to have been *spilled*. A value spilled at a point in the schedule must be stored to memory sometime earlier and reloaded into the register file before its next use. Therefore a value is stored at most once, but may be loaded several times.

In our generic multi-issue model, we assume that a group of instructions can be simultaneously issued if they reserve distinct functional units for their execution. The period during which a group of simultaneously issued instructions are executing is called a *slot*. For simplicity of illustration, the model assumes that the instructions in the same slot may be dependent (i.e. read after write chaining is allowed) and they complete their execution before the next slot

begins. As registers are allocated, spill code is inserted to write values to memory and read values into registers. The cost of an allocation is the number of slots needed to issue the original code with its spill code. Given a straight-line code segment without any loads and stores, the *multi-issue local register allocation problem* is to find a minimum-cost, legal allocation of values to registers.

In section 2 we define the multi-issue scheduling problem by describing the generic multi-issue machine model in terms of its instruction issue restrictions and a model of the register file. In this section we also define the optimal register allocation subproblem. In section 3 we describe our motivation for pursuing this subject and discuss relevant previous work. In section 4 we describe our algorithm for optimal register allocation for a multi-issue machine and present the result of applying this algorithm to sample routines running on the Convex C2 vector supercomputer. In section 5 we describe our plans for extending this approach to jointly optimize scheduling and register allocation for multi-issue machines. We then discuss how the machine model and the algorithm might be generalized to handle multiple-issue scalar architectures: VLIW and superscalar machines.

2 Problem Definition

The application input to the scheduling problem is a set of instructions, without loads and stores, each of which accesses single-write, multiple-read quantities, i.e. *values*. Each instruction *defines* (writes) one value and *uses* (reads) up to two values. A data dependence exists between two instructions if one defines a value that the other uses. Output and anti dependences do not exist because register allocation has not yet been performed. Loads and stores, when they are inserted, each reference a single data value. Loads and stores that are inserted to make room for other data in the register file are called *spill code*. Values that are not defined by any instruction are called *input values* and are assumed to be defined before the start of the code segment. Input values must be loaded before being used. Values that are used after the code segment ends are identified as *output values* and must be stored after being defined. For convenience, we define the term "spill code" to include those input loads and output stores. The solution to the scheduling problem is a minimum cost ordering of the instructions, with loads and stores inserted, that satisfies the multi-issue machine model and the register file model. We refer to this ordering of instructions with loads and stores inserted as a *final schedule*. The cost of a final schedule is defined in the next subsection.

2.1 Generic Multi-Issue Machine Model

In our generic multi-issue machine model, we assume that a group of instructions can be issed simultaneously if they reserve distinct functional units for their execution. The period during which a group of simultaneously issued instructions are executing is called a *slot*. For simplicity of illustration, the model assumes that the instructions in the same slot may be dependent (i.e. read-after-write chaining is allowed) and they complete their execution before the next slot begins. As registers are allocated, spill code is inserted to write register values to memory and

read memory values into registers. The cost of a final schedule is the number of slots needed to issue the original code with its spill code.

In the generic model, all instructions execute in one of three functional units: an adder, a multiplier and a load/store unit (or port). Therefore a slot may contain 1, 2 or 3 instructions. The instructions in a slot are all assumed to be issued simultaneously and complete their execution at the end of that slot. In this simple generic model the slots roughly correspond to the chimes of a vector machine with chaining. A data dependence between instructions does not preclude their inclusion in the same slot; however a dependent instruction may not be scheduled in an earlier slot than the instruction on which it depends. Adding more types of instructions or function units, allowing several function units of each type, increasing their execution times or latencies, and increasing or restricting issue bandwidth would result in a straightforward extension of the algorithm, but would clutter this discussion with unnecessary details. This simple generic model is assumed for simplicity of illustration. It does model the performance of the Convex C2 vector supercomputer quite well.

2.2 Generic Register File Model

We assume that the final schedule will have access to r registers. Data is written to registers by loads and by the definitions in (results of) adds and multiplies. Data is read from registers by stores and by the uses in (sources of) adds and multiplies. There is no limit on the number of register reads and writes that may take place within a slot. However, because a slot contains at most 3 instructions, the register file as a whole will be read at most 5 times (2 uses each by an add and a multiply instruction, and one read access by a store), and written at most 3 times (1 definition each by an add and a multiply instruction, and one write access by a load). If a value is assigned to a register in a particular slot, that register may not be assigned another value until the next slot. That is, two values may not *share* a register within a slot.[1] As a consequence, within each slot at most r values can be assigned to registers. Every value that is used, defined, loaded, or stored in a slot must be assigned to a register during tht slot. Additional values may also be assigned to registers in that slot up to a total of r.

2.3 Multi-Issue Register Allocation

Given an *initial schedule*, i.e. an assignment of the non load/store instructions to slots so that no dependences are violated, the optimal register allocation minimizes the number of additional slots that need to be added to accommodate the spill code. A slot with no load/stores (as yet) is called an *empty slot*. A load/store placed into an empty slot is called a *free load/store* (and the slot is no longer empty). Each load/store temporarily placed in that slot thereafter has a cost of 1 and will subsequently cause an *extra slot* to be inserted. The inserted slot will be placed immediately before this slot for a load, and immediately after for a store. Note that extra slots need not be numbered. The slots of the initial schedule are numbered sequentially starting

[1] Many machines allow values to share a register within an issue slot. For example, it is often possible to begin writing to a vector register before an earlier read of that register has completed, i.e. within the same slot. In section 4.3.3 we discuss extensions to the model that do allow register sharing.

from 0. The following formalization of the allocation problem uses this convention. Although no more than one load/store can be placed in a slot, it is convenient to allow the algorithm to do so temporarily. A formal definition of the register allocation problem is as follows:

Inputs

1. An inital schedule for a code segment, i.e. a set of add and multiply instructions that define and use values, a list of output values, and an assignment of these instructions to a sequence of slots that does not violate any data dependences.
2. The number of available physical registers.

Boundary Conditions

1. All values are in a register during a slot where they are defined or used.
2. At the start of the initial schedule, before the first slot, no values are in a register.
3. At the start of the initial schedule, only the input values are in memory.
4. At the end of the final schedule, all output values must be in memory.

Constraints

1. During any slot where a value is loaded or stored, the value is said to be present in both a register and in memory.
2. If a value is present in a register in slot i and is not in a register in slot $i-1$, then that value must either be defined in slot i, or loaded from memory in slot i and present in memory in slot $i-1$.
3. If a value is present in memory in slot i and is not in memory in slot $i-1$, then that value must be in a register and stored during slot i.
4. The number of values in registers in each slot must not exceed the number of available physical registers.

Objective Function Find a minimum cost final schedule, i.e. an insertion of loads and stores into the initial schedule, that satisfies the boundary conditions and constraints above. The first memory operation inserted into a slot is free, and each subsequent operation added causes an extra slot and has a cost of one.

3 Motivation and Previous Research

Our experience with register allocation and code rescheduling began with the analysis of the performance of heavily used Fortran routines that make up a large, finite-element simulation running on a Convex C2. We found that over the entire code, load/store spills inserted to free registers made up 28% of the total memory traffic, and up to 40% in the case of particular complex routines in the compiled code. Since the processor bottleneck was clearly the single port to memory, we concluded that by somehow eliminating these spills performance might improve by up to 28% over all. We chose a representative routine that contained a large amount of spill code and used a simple manual technique to arrange the operations so that no spill code was required. However instead of the expected 40% performance gain, we observed only a 20% speedup over the compiled code. While our final schedule had optimized the data reuse in the register file, it had failed to achieve optimal utilization of the functional units. In the process of reducing memory bandwidth we created new bottlenecks in the processor.

To improve performance further we developed an automatic rescheduling tool to rearrange the no-spill code in search of schedules that fully utilize at least one functional unit. The tool did in fact find no-spill schedules that approached this criterion, giving us a total speedup of 30% over the original compiled code. The remaining gap between this speedup and the expected 40% speedup results from the occasionally idle memory port, and the limitations of our processor model. This preliminary rescheduling tool finds reorderings of code that keep the functional units busy without ever overfilling the register file, if any such reordering exists. As such, it is inadequate to handle routines where spilling is required. Our initial efforts to optimally reschedule more complex routines failed for this reason. The class of routines that can be scheduled optimally without needing to spill values temporarily to memory is quite small, so we concluded that a general treatment of the scheduling problem must address spilling.

Chaitin et al. described the application of graph coloring to global register allocation in [6]. Values that are live in the same basic block cannot be assigned to the same register and are said to *interfere* with one another. Values are associated with nodes of a graph, and edges between the nodes indicate value interference. An assignment of k-colors to the nodes such that no connected nodes are assigned the same color represents a k-coloring of the graph. Such a k-coloring can represent an assignment of values to k registers without spilling. Deciding whether a given grph can be k-colored is NP-complete. If k exceeds the number of available registers, nodes are removed from the graph until a coloring is possible. Algorithms based on Chaitin's work use heuristics to determine which nodes to delete and then generate spill code each time those values are referenced. Deleting the minimum number of nodes from a graph to make it k-colorable is also NP-complete. Heuristics for register allocation are also described in [4]. While register coloring can be adapted to perform local allocation, it has traditionally been used in global allocation across multiple basic blocks.

The spill insertion problem is to insert the minimum amount of spill code into a given initial schedule. Horwitz et al, and Hsu et al show that the spill insertion problem is NP-complete in [8] and [9] when multiple-use values (common subexpressions) are present. This is a striking result because it demonstrates that optimal code generation is difficult even when the instruction schedule is fixed. They derive pruning rules for optimal spilling and present heuristics. This

work is described further in section 4.1.

4 Preliminary Work

The Convex C2 processor was chosen to be illustrative of the class of single-port computers. Its limited bandwidth to memory, few vector registers, and simple issue rules make it an ideal target for attempts to reduce register-memory spill traffic. We used the C2 in our early development nefforts as the focus of the research on optimal scheduling algorithms. It is modeled as a multi-issue machine since multiple vector instructions may be issued in one slot. Although individual slots may differ somewhat in the number of clock periods they contain, the long vector operations (up to 128 elements) allow us to approximate run time by counting slots.

On a multi-issue machine, the latency of a memory operation may be hidden by executing other instructions simultaneously. Therefore the *placement*, or insertion points, of the loads and stores in the initial schedule is as important as the total number required. Consider a straight-line code segment whose instructions operate on and produce *values* (or virtual registers): quantities that are written (defined) once and read (used) one or more times. Input values are considered to be defined before the start of the code segment and must be loaded before being used. Output values are defined within the code segment and are considered to be used sometime after the segment ends. They must be stored after being defined. A value is *live* at a point in the schedule if it is defined there or earlier, and used at that point or later. A *register allocation* is a selection, at every point in the code segment, of values to keep in registers. A *legal allocation* is one where each value is in a register when defined or used, and the number of values in registers in each slot never exceeds the number of available registers. A live value that is not in a register at some point is said to have been *spilled*. A value spilled at a point in the schedule must be stored to memory sometime earlier and reloaded into the register file before its next use. Input values are defined as spilled as their initial state. Therefore each value is stored at most once, but may be loaded several times.

Given a straight-line code segment without any loads and stores, the *multi-issue register allocation problem* is to find a legal allocation of values to registers that produces a final schedule with the fewest slots.

The following sections describe our dynamic programming approach to optimal multi-issue register allocation. We discuss optimal allocation for single-issue machines in section 4.1, optimal allocation for multi-issue machines in section 4.2, and the results of applying the allocation algorithm to sample code in section 4.3.

4.1 Optimal Single-Issue Register Allocation

Our algorithm is an extension of the algorithms described by Horwitz [8] and Hsu [9] for the minimization of loads and stores in straight-line single-issue code. In their model of a single issue machine, only the selection of which values to spill and the resulting number of loads and stores is significant; the actual placement of the loads and stores is irrelevant because all instructions

take the same time to execute regardless of where they are placed in the schedule.

In contrast, multi-issue allocation must optimize spill selection and load/store placement jointly to minimize the final schedule's runtime. Both problems have efficient solutions when each data value is used at most once, but in the general multi-use case they are NP complete. We begin by summarizing the work done in [8] and [9] to efficiently solve the single-issue load/store minimization problem using dynamic programming and pruning rules. We then describe our extension of their algorithm.

The single-issue local register allocation problem is to insert the minimum number of loads and stores into a given schedule so that the resulting allocation is legal. Here the legality of an allocation is defined in the same way as for multi-issue allocation above; however the cost of an allocation is simply the number of loads and stores.

The algorithm in [8] reduces spill selection to a graph traversal problem, where each node of the graph represents the state of the values at the current point in the schedule. A value's state (in our terminology) is either REG (the value is in a register and not in memory), MEM (the value is in memory but not in a register), BOTH (the value is in a register and in memory), or DEAD (the value is neither in a register nor in memory). The set of value states at a particular instruction is called a *configuration*. A configuration is legal if all values used or defined at that instruction are in state REG or BOTH, and the number of values in state REG or BOTH does not exceed the number of registers. The cost of moving from one configuration to the next is the number of loads and stores that are needed to modify the value states.

Each value that goes from state REG to MEM requires a store. Note that values do not go directly from state REG to BOTH because when a value needs to be replaced in registers in instruction i, it is assumed to have been stored *between* instructions i and $i-1$. Therefore the value is in state REG at instruction $i-1$, and in state MEM at instruction i. Each value that goes from state MEM to BOTH requires a load. The only other legal transition among these three states is from BOTH to MEM which makes the register available for another value, but does not require any memory access for this value. As in our algorithm only the instructions (slots) in the initial schedule, not the load/store instructions (extra slots in our algorithm) are numbered.

The rules described in [8] and [9] are slightly different because the instructions access pseudo-registers that can be written multiple times. The algorithm constructs all legal configurations for each instruction and finds a cheapest path from the initial configuration (before the first instruction) to a final configuration for the last instruction. The required loads and stores are then implied by the sequence of configurations along this path. The number of configurations considered by this algorithm grows exponentially with the number of instructions. Horwitz [8] describes a set of pruning rules that eliminate configurations that cannot lie on a cheapest path and merges equivalent configurations. Using these rules, optimal local register allocations can be found.

4.2 Optimal Multi-Issue Register Allocation for an Initial Schedule

For a multi-issue machine, the cost of a transition from one configuration to the next does not depend on the value states alone. The cost of loading or storing a value also depends on which slots are empty (not yet used by other memory operations). An extended configuration that includes information regarding usable empty slots is called a *partial solution*. The multi-issue allocation algorithm generates legal partial solutions at the current slot in the schedule by considering all legal register replacements (value state changes), and updating the partial solution value states and empty slots information accordingly. The incremental cost of a new partial solution depends on the number of loads and stores that have just been placed that required extra slots. Implicit pruning rules are used to reduce the number of partial solutions that need to be considered at a slot. Explicit pruning rules delete partial solutions that are

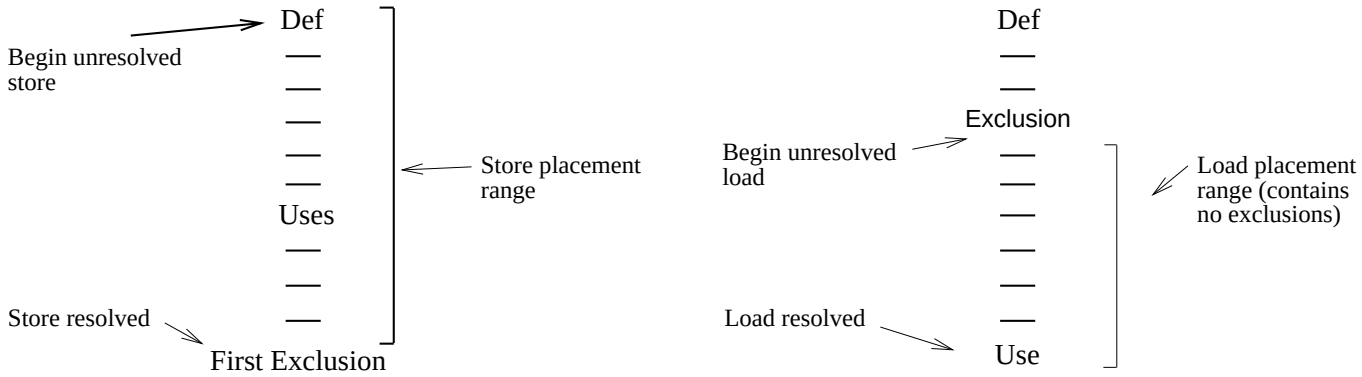

Figure 1: Load/store placement ranges

clearly no better than some other partial solution. The algorithm employs a single forward scan of the initial schedule with breadth-first expansion and pruning.

4.2.1 Unresolved Loads and Stores

If n values are live in the current slot, then $n-r$ values are *excluded* from registers in this slot. This process insures that the allocation remains legal. Later sections discuss how the algorithm chooses which values to exclude.

Definitions, uses, and exclusions of a value throughout a schedule define a *placement range* for each load and store (see Figure 1). A store of a value may be placed anywhere between the value's definition and its first exclusion. A load of a value must be placed between the last exclusion before a use of a value, and that use. In the course of making exclusion decisions for values in the current slot, we say that a load or a store is *unresolved* if the lower bound of its placement range is not yet known. Thus a live value that has been defined and never excluded has an unresolved store. Similarly, a live value that has not been used since its last exclusion

has an unresolved load.

When a value with an unresolved store is excluded, the store becomes *resolved.* Similarly, when a value with an unresolved load is used, the load becomes resolved. As soon as a load or store becomes resolved, it is placed in the highest empty slot in its placement range. Placing it in this slot minimizes the number of currently unresolved placement ranges with which the placed memory operation conflicts. If no empty slots remain in its placement range when a load/store becomes resolved, then the memory operation is placed in an extra slot, and the cost of the partial solution is incremented by 1. The extra slots may be placed anywhere in the placement range with equal (unit) cost; however the most registers are freed for uses outside of this problem context by placing extra slots for stores immediately after the value definition (top of the range) and for loads immediately before the value use (bottom of the range). If more than one range is resolved in the same slot, the load/store with the largest range is resolved first, followed by those with successively smaller ranges. It can be shown that for a given set of exclusions this method of resolving placement ranges is optimum (see Appendix).

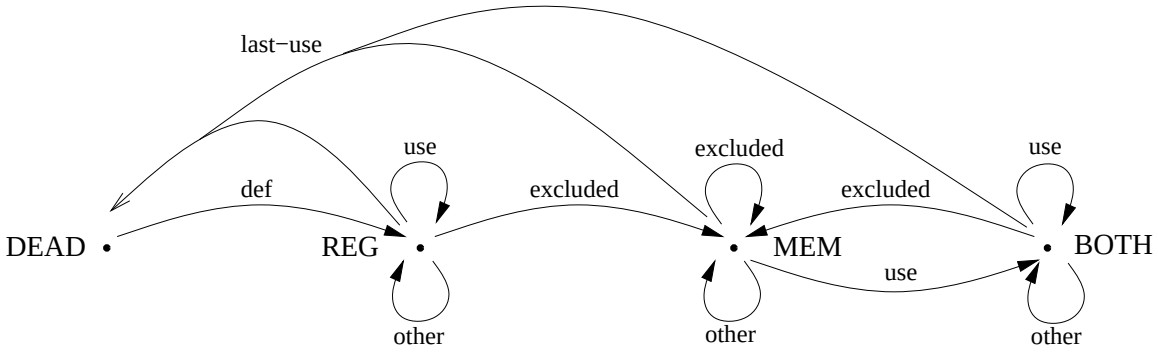

Figure 2: Transitions between last known states.

4.2.2 Value State Transitions

The value transitions from state to state are similar to those described above for single-issue spill insertion, except that a value's "state" now actually refers to its last-known state. A value moves from DEAD to REG state when defined (see Figure 2). It enters MEM state whenever it is excluded. The store of the value becomes resolved upon its first exclusion after being defined. Since the store may be placed anywhere in its range, the value may be present in memory sometime earlier than the exclusion, but at the time the intervening slots in its range were being considered by the algorithm, it was not yet known where within its range the value

would actually be stored. Thus the algorithm considers the value to be in REG state (its "last known" state) after definition until its first exclusion. Similarly, a value is in MEM state when excluded, and is considered to enter BOTH state only upon its next use. The load of the value will be placed somewhere in the load's placement range: between the last exclusion and the use. Thus the value may be present in a register sometime earlier than the use, but at the time any of those slots were being considered it was not yet known where within its range the value would actually be loaded.

A value in REG state has an unresolved store that becomes resolved upon entering MEM state. A value in MEM state has an unresolved load that becomes resolved upon entering BOTH state. Input values begin in MEM state. Output values have a use after the end of this code segment and must therefore be in MEM or BOTH state at the end.

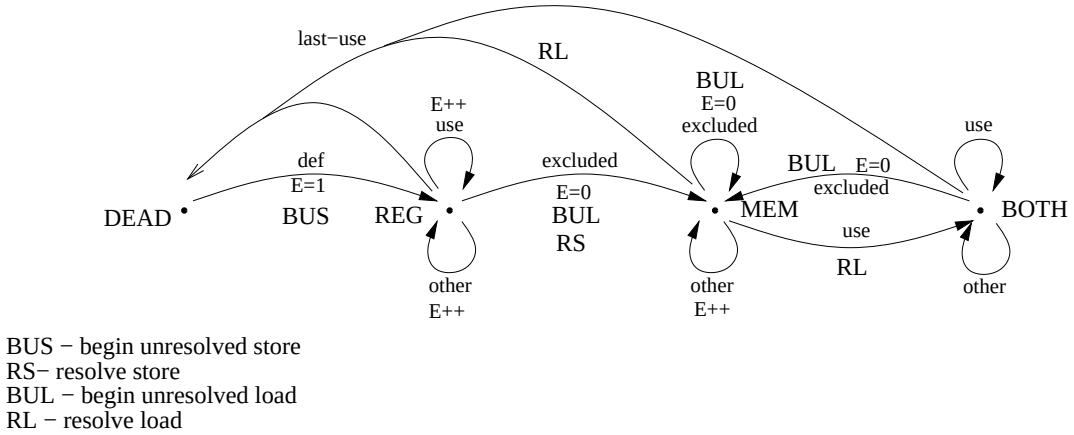

Figure 3: Empty-slots (E) calculations.

4.2.3 Empty Slots

The cost of a partial solution is the number of resolved loads and stores that need to be placed in extra slots. An extra slot is needed when placing a memory operation if all empty slots in its placement range have already been used for loads and stores of other values. Each value, v, with an unresolved load/store has an attribute, $E(v)$, which, upon entering the current slot, is 1 plus the number of empty slots above the current slot that are within its placement range. When resolved, a load/store is placed in the highest empty slot in its range. Thus $E(v)$ is decremented if $E(v) \geq E(v')$ whenever a load/store of some other value, v', is resolved. If $E(v) = 0$ when a load/store is resolved for value v, an extra slot is created and the partial solution cost is incremented by 1.

As shown in Figure 3, $E(v)$ is initialized to 1 when value v is defined, indicating that there

is one empty slot in which a store of the value can be placed (the slot in which the define took place). As each succeeding slot becomes the current slot, $E(v)$ is incremented to indicate that an additional empty slot is available. If v is excluded from the register file at the current slot, the unresolved store is resolved and $E(v)$ is reset to 0 to begin an unresolved load. $E(v)$ is again incremented upon entering each succeeding slot. At the first use following an exclusion, the load is resolved. No unresolved load/store for v begins at this slot and $E(v)$ is undefined until the next time (if ever) that the value is excluded from the register file and the next unresolved load begins.

4.2.4 Implicit Pruning Rules

Horwitz describes a set of pruning rules that effectively reduce the number of configurations that need to be considered. Using these rules, configurations that are clearly no less costly than other configurations are deleted. Our algorithm employs pruning rules that apply to the partial solutions. In both the scalar and multi-issue algorithms, given a configuration, the set of successor configurations is found by considering, at least implicitly, all *sufficient* sets of value exclusions. A sufficient set of exclusions excludes all but r live values from the register file so that even if all the remaining live values were in registers, they would fit in the r registers. For example, if there are 14 live values and r = 8, then the number of sufficient sets of value exclusions is 14 choose 6, assuming all 14 are excludable (i.e. they are neither used nor defined in the current slot). To reduce this number, we define for each slot a partial ordering $\succ$ on the excludable values at that slot. If $v_1 \succ v_2$, then at this slot value v_2 will never be excluded unless value v_1 is also excluded. The pruning rules discussed below define this partial ordering so that the optimality of the resulting register allocation will not be sacrificed.

Several pruning rules used in the multi-issue register allocation algorithm are similar to those described in [9]. In that work the spill candidates are partitioned into 4 sets, dead values, live values not written to memory with a single use remaining, live values not written to memory with more than one use remaining, and live values that have been written to memory. The function NEXTREAD(x) is defined to return the distance (in number of instructions) from the current instruction to the one that next uses the current value x. A series of pruning rules are then described that depend on the set membership and NEXTREAD(x).

We use the following implicit pruning rules to reduce the number of possible sets of exclusions for the current slot. These rules are called implicit because they eliminate exclusion sets before they are actually considered by the algorithm. The first three rules follow the 2nd, 3rd and 4th observations in [9]. The other three rules are new. Note that since we refer only to values which are written at most once, there is no need to verify that a live value will be read before being overwritten. (Note that $x \succ y$ implies that y will not be excluded unless x is also excluded, and $x > y$ implies that x is a larger numbered value than y. The second relation is used to arbitrarily pick a value to exclude when no preference exists.)

Rule 1: If values x and y are in MEM state or BOTH state, and NEXTREAD(x) > NEXTREAD(y) (see Figure 4), then $x \succ y$. Under the same conditions, if NEXTREAD(x) = NEXTREAD(y) and $x > y$, then $x \succ y$.

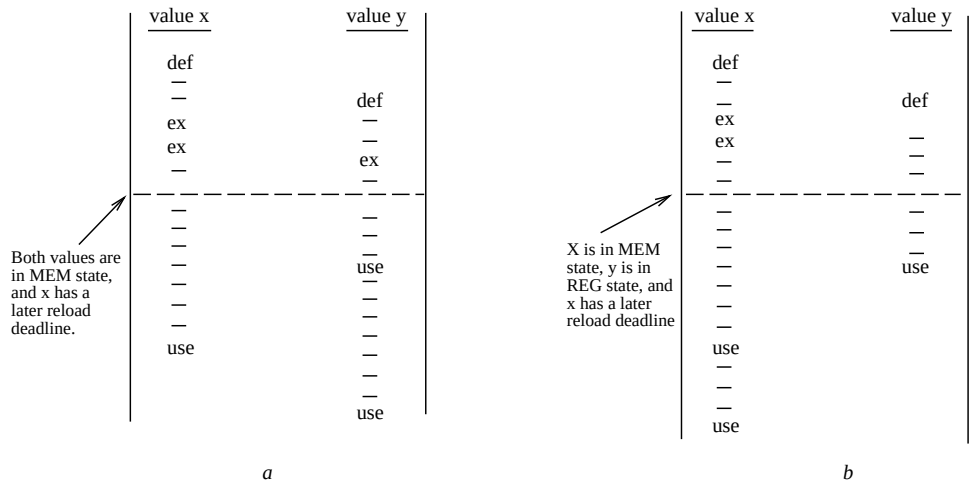

Figure 4: Examples of pruning rules 1 and 2.

This rule is an application of Belady's MIN algorithm [2].

Rule 2: If value x is in MEM state or BOTH state, value y is in REG state, value y has a single use remaining, and NEXTREAD(x) > NEXTREAD(y) (see Figure 4), then $x \succ y$. Under the same conditions, if NEXTREAD(x) = NEXTREAD(y) and $x > y$, then $x \succ y$.

Excluding x does not require a store whereas y would need to be stored, and y has an earlier reload deadline (its next use) than x. However, if y has more than one future use, a later exclusion of y might make it better to exclude y rather than x here.

Rule 3: If both values x and y are in REG state, both have a single use remaining, NEXTREAD(x) > NEXTREAD(y), and x is not defined later than y, then $x \succ y$ (see Figure 5). Under the same conditions, if x and y are defined in the same slot, NEXTREAD(x) = NEXTREAD(y), and $x > y$, then $x \succ y$.

Storing value x cannot be more expensive than storing value y, and y has an earlier reload deadline than x.

Rule 4: If value y is in REG state, E(y) = 0, and NEXTREAD(x) > NEXTREAD(y) then $x \succ y$ (see Figure 5). Under the same conditions, if NEXTREAD(x) = NEXTREAD(y) and $x > y$, then $x \succ y$.

If E(y) = 0, a store of y will increase the cost of the partial solution by 1. On the other hand, storing x, if needed, may or may not require an extra slot, and x has a later reload deadline than y. That is, the store cost of x is 0 or 1, the store cost of y is 1, and the future reload cost of x is no greater than the reload cost of y. Therefore excluding y now rather than x cannot reduce the cost of the resulting allocation.

Rule 5: If an output value is in MEM state or BOTH state, and has no future uses, exclude

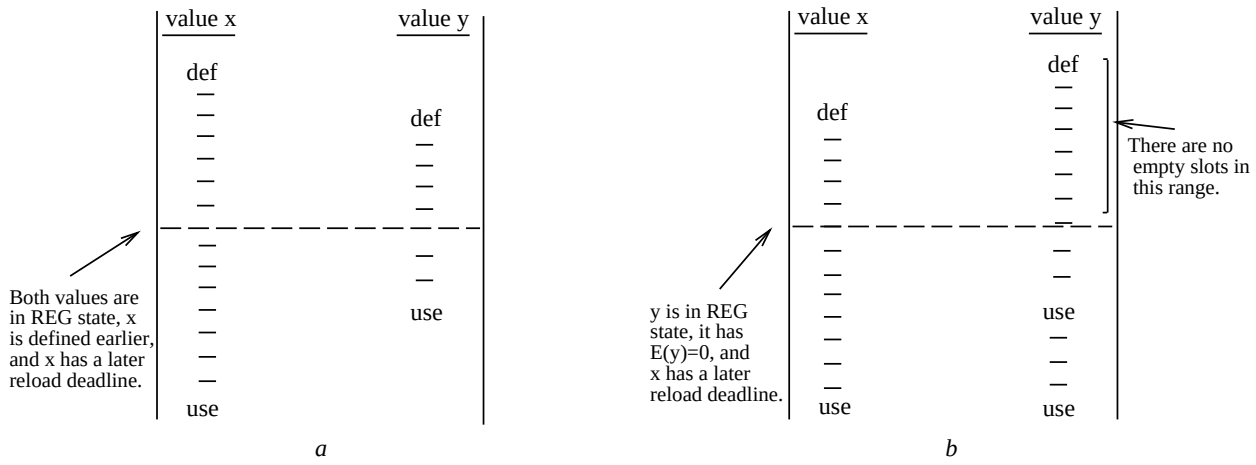

Figure 5: Examples of pruning rules 3 and 4.

it.

There is no reason for it to be in a register, and excluding it will not cost anything.

Rule 6: If x was in MEM state in the previous slot, x is used in the current slot, and x must be loaded in the current slot (see Figure 6), then exclude all values y that were in MEM state in the previous slot, were excluded more recently than x, and are not used in the current slot.

No y can be reloaded any earlier than the current slot without cost, and since another value, x, must be reloaded in the current slot, excluding y in the current slot cannot possibly increase the cost of the allocation.

4.2.5 Explicit Pruning

In addition to the implicit pruning rules, our algorithm uses one explicit pruning rule that compares two partial solutions, P_1 and P_2, produced by the algorithm at the current slot and deletes P_2 if it can determine that P_2 cannot possibly result in a better final allocation than the best final allocation obtainable from P_1.

Rule 7: If all values v are in the same state in P_1 and P_2, $E_1(v) \geq E_2(v)$ for all values v (excluding values in BOTH state for which $E(v)$ is undefined), and the cost of P_1 is no greater than the cost of P_2, then P_2 can be pruned.

Whatever exclusions might be used in later slots of P_2, the same exclusions applied to P_1 cannot have higher cost because all currently unresolved loads and stores in P_1 have larger placement ranges. Since P_2 is no less expensive to start with, P_2 cannot result in a less expensive

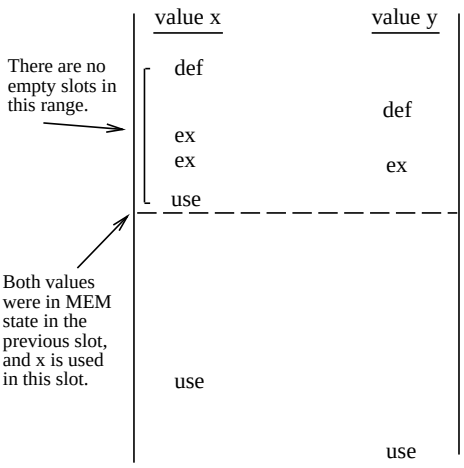

Figure 6: Example of pruning rule 6.

final allocation than the best allocation for P_1.

4.3 Optimal Allocation for the Convex C2 Vector Processor

4.3.1 Machine Model

The multi-issue allocation algorithm has been used to improve the performance of a Fortran code running on a Convex C2 vector supercomputer [5]. The generic multi-issue machine/code model described above is based on the Convex C2 machine model described in [3]. This C2 model assumes that a set of 1 to 3 vector instructions, referred to as a *chime*, can be issued to different vector functional units simultaneously. Chimes are analogous to slots in the generic machine model. The instructions in a chime are assumed to complete execution in VL plus some constant number of clocks. The vector length, VL, is assumed to be the same for all instructions, so all instructions in a slot complete execution simultaneously in the model. The partition of an ordered list of vector instructions into disjoint chimes is obtained by scanning the list sequentially from top to bottom and adding instructions to a chime until a functional unit reservation conflict exists between two instructions in the chime. Then a new chime is begun, the conflicting instruction is placed there, and the instruction scan continues until the initial schedule is produced when the last instruction is assigned to a chime. Chaining between the functional units allows vector instructions with flow dependences to occupy the same chime. Since the C2 is a register-register architecture, data must be loaded into registers before being used by an instruction. Scalar instructions that do not reference vector data are ignored by this model under the assumption that the time to process a scalar instruction is masked by the much longer execution time of a vector instruction. The C2's limited bandwidth to memory, few vector registers, and simple issue rulres make it an ideal target for studying the reduction

of register-memory spill traffic.

4.3.2 Allocation Results

Two memory bottlenecked vector routines, Comp1 and Mul5, were chosen to test the allocation algorithm. Comp1 was selected because, as compiled by the Convex Fortran compiler, it is heavily memory bottlenecked. It is fully vectorized and contains 117 load and store instructions, 70 multiply instructions and 45 add instructions. 78 of the memory operations are *required* (needed to load an input value or store an output value), so the remaining 39 represent spill code. The 117 loads and stores actually increase the number of chimes in the compiled schedule from 73 to 125. The insertion of loads and stores consumes 52 extra chimes. This routine is therefore a good candidate for performance improvement via optimum allocation.

The optimum allocation found by our multi-issue allocation algorithm consumes 43 extra chimes, or 116 chimes altogether, a 7% improvement in predicted performance. The algorithm took 4 minutes to run on an NCR 3550 system with a 50 MhZ Intel 486 processor and 256 MB of main memory. The optimal allocation for Comp1 is shown in Figure 7. The instructions are shown in the chime (slot) they occupied before placing loads and stores. The instructions reference values that will each be assigned to a physical register. The values that occupy registers in each chime are shown. Because no more than 8 values are ever in registers, the allocation is legal.

Mul5 is a completely unrolled, fully vectorized, 5x5 matrix multiplication routine. It contains 234 load and store instructions, 100 add instructions and 125 multiply instructions. 76 of the memory operations are required, and the remaining 158 in the compiled code are spills. The 234 loads and stores increase the number of chimes in the compiled schedule from 125 to 234.

The optimum allocation algorithm finds an allocation that requires only 66 extra chimes, or 191 chimes altogether, representing an 18% improvement in predicted performance. This run took 1 minute on the same NCR computer.

4.3.3 Register Sharing

The Convex C2 actually allows a value that is being used for the last time and a value that is being defined in the same chime to *share* a register. While the earlier use is reading from the register, the later define is writing the new value. Such register sharing allows the compiler to reduce the amount of spilling that is needed by reusing registers within a chime. If the value definition and the value use take place in the same instruction, *same-instruction register sharing* is possible. In Figure 8, values 3 and 6 can share a register because value 3 is not used again. The C2 register file allows value 3 to be read out of a register as value 6 is being written into the same register.

If both values used in the first instruction of a two-instruction chime are never used again, it is possible to reuse two registers, as shown in Figure 9. Same-instruction register sharing occurs between the first value used and the first value defined. In addition, the second value used can share a register with the second value defined in the chime. We refer to this sharing of registers

```
Slot #    Instruction           Values in Registers
    ------------------------
 0)     Load 0                0 1 2
        Load 1
        2 <- 1d + 0d
    ------------------------
 1)     Load 3                2 3 4 5 6
        Load 4
        5 <- 3d + 2d
        6 <- 4 *
    ------------------------
 2)     Load 7                4 5 6 7 8 9
        8 <- 5d + 7d
        9 <- 8 * 4d
    ------------------------
 3)     Load 10               6 8 9 10 11 12
        11 <- 10 *
        12 <- 11d + 6d
    ------------------------
 4)     Load 13               8 9 10 12 13 14 15
        Load 14
        15 <- 14d + 13d
    ------------------------
 5)     Load 16               8 9 10 12 15 16 17
        17 <- 16d + 15d
    ------------------------
 6)     Load 18               8 9 10 12 17 18 19 20
        19 <- 17d + 18d
        20 <- 19 * 10d
    ------------------------
 7)     Load 21               8 9 12 19 20 21 22 23
        22 <- 19 * 21
        23 <- 22d +
        Store 9
    ------------------------
 8)     24 <- 21d *           8 12 19 20 21 23 24 25
        25 <- 24d + 12d
        Store 19
    ------------------------
 9)     Load 26               8 19 20 23 25 26 27 28
        27 <- 19 * 26
        28 <- 27d +
        Store 28
    ------------------------
10)     29 <- 26d *           8 19 20 23 25 26 29 30
        30 <- 29d + 25d
        Store 8
    ------------------------
11)     Load 31               8 19 20 23 30 31 32 33
        32 <- 19d * 31
        33 <- 32d +
        Store 33
    ------------------------
12)     Load 36               8 20 23 30 31 34 35 36
        34 <- 31d *
        35 <- 34d + 30d
    ------------------------
13)     37 <- 20d +           8 20 23 35 36 37 38
        38 <- 8 * 36
    ------------------------
14)     39 <- 37d + 38d       8 23 35 36 37 38 39 40
        40 <- 36d *
    ------------------------
15)     Load 41               8 23 35 39 40 41 42 43
        42 <- 40d + 35d
        43 <- 8 * 41
    ------------------------
16)     44 <- 23d + 43d       8 23 39 41 42 43 44 45
        45 <- 41d *
        Store 39
    ------------------------
17)     Load 46               8 39 42 44 45 46 47 48
        47 <- 45d + 42d
        48 <- 8d * 46
    ------------------------
18)     Load 28               28 39 44 46 47 48 49 50
        49 <- 28d + 48d
        50 <- 46d *
        Store 49
    ------------------------
19)     Load 33               33 39 44 47 50 51 52
        51 <- 50d + 47d
        52 <- 51d *
    ------------------------
20)     Load 9                9 33 39 44 52 53 54 55
        Load 53
        54 <- 33d + 9d
        55 <- 54 * 53d
    ------------------------
21)     Load 56               39 44 52 54 55 56 57 58
        57 <- 54 * 56d
        58 <- 54 +
        Store 55
        Store 58
    ------------------------
22)     Load 59               39 44 52 54 57 59 60 61
        60 <- 39 * 59d
        61 <- 60d + 57d
    ------------------------
23)     Load 62               39 44 52 54 61 62 63 64
        63 <- 44 * 62d
        64 <- 63d + 61d
        Store 54
    ------------------------
24)     Load 49               39 44 49 52 64 65 66 67
        Load 65
        66 <- 49 * 65d
        67 <- 66d + 64d
    ------------------------
    ------------------------
25)     Load 68               39 44 49 52 67 68 69 70
        69 <- 68 * 52d
        70 <- 49 +
        Store 52
        Store 67
        Store 70
    ------------------------
26)     Load 75               39 44 68 69 71 72 75
        71 <- 68d *
        72 <- 44 +
    ------------------------
27)     73 <- 69 * 71d        39 44 69 71 72 73 74 75
        74 <- 39 +
        Store 74
    ------------------------
28)     76 <- 69d * 75        39 44 69 72 73 75 76
        Store 72
    ------------------------
29)     77 <- 73 * 75d        39 44 73 75 76 77
        Store 76
    ------------------------
30)     78 <- 77d *           39 44 73 77 78
    ------------------------
31)     Load 54               39 44 54 73 78 79 80
        Load 79
        80 <- 54 * 79d
        Store 73
    ------------------------
32)     Load 81               39 44 78 80 81 82 83
        82 <- 39 * 81d
        83 <- 82d + 80d
    ------------------------
33)     Load 84               39 44 78 83 84 85 86
        85 <- 44 * 84d
        86 <- 85d + 83d
    ------------------------
34)     Load 49               39 44 49 78 86 87 88 89
        Load 87
        88 <- 49 * 87d
        89 <- 88d + 86d
    ------------------------
35)     Load 90               39 44 49 78 89 90 91 92
        91 <- 78 * 89d
        92 <- 91d + 90d
        Store 92
    ------------------------
36)     Load 67               39 44 49 67 78 93 94 95
        Load 93
        94 <- 78d * 67d
        95 <- 94d + 93d
    ------------------------
37)     Load 54               39 44 49 54 78 95 96 97
        Load 96
        97 <- 54 * 96d
        Store 78
        Store 95
    ------------------------
38)     Load 98               39 44 49 54 97 98 99 100
        99 <- 39 * 98d
        100 <- 99d + 97d
    ------------------------
39)     Load 101              39 44 49 54 100 101 102 103
        102 <- 44 * 101d
        103 <- 102d + 100d
    ------------------------
40)     Load 104              39 44 49 54 103 104 105 106
        105 <- 49 * 104d
        106 <- 105d + 103d
    ------------------------
41)     Load 73               39 44 54 73 106 107
        107 <- 73d *
    ------------------------
42)     Load 108              39 44 54 106 107 108 109 110
        109 <- 107d * 106d
        110 <- 109d + 108d
    ------------------------
43)     Load 58               39 44 54 58 92 107 110 111
        Load 92
        111 <- 58 * 92
        Store 107
        Store 111
    ------------------------
44)     Load 70               39 44 54 58 70 92 110 112
        112 <- 70 * 92
        Store 110
        Store 112
    ------------------------
45)     Load 113              39 44 54 58 92 113 114
        114 <- 54d * 113d
    ------------------------
46)     Load 115              39 44 58 92 114 115 116 117
        116 <- 39 * 115d
        117 <- 116d + 114d
    ------------------------
47)     Load 118              44 58 92 117 118 119 120
        119 <- 44 * 118d
        120 <- 119d + 117d
    ------------------------
48)     Load 49               44 49 58 92 120 121 122 123
        Load 121
        122 <- 49 * 121d
        123 <- 122d + 120d
    ------------------------
49)     Load 76               44 58 76 92 123 124
        124 <- 76d *
    ------------------------
    ------------------------
50)     Load 125              44 58 92 123 124 125 126 127
        126 <- 124 * 123d
        127 <- 126d + 125d
    ------------------------
51)     Load 72               44 58 72 92 124 127 128
        128 <- 72 * 92
        Store 128
    ------------------------
52)     Load 74               44 58 72 74 92 124 127 129
        129 <- 74 * 92d
        Store 129
    ------------------------
53)     Load 95               44 58 72 74 95 124 127 130
        130 <- 58 * 95
        Store 124
        Store 130
    ------------------------
54)     Load 70               44 58 70 72 74 95 127 131
        131 <- 70 * 95
        Store 131
    ------------------------
55)     132 <- 72 * 95        44 58 70 72 74 95 127 132
        Store 132
    ------------------------
56)     133 <- 74 * 95d       44 58 70 72 74 95 127 133
        Store 133
    ------------------------
57)     Load 110              44 58 70 72 74 110 127 134
        134 <- 58 * 110
        Store 134
    ------------------------
58)     135 <- 70 * 110       44 58 70 72 74 110 127 135
        Store 135
    ------------------------
59)     136 <- 72 * 110       44 58 70 72 74 110 127 136
        Store 136
    ------------------------
60)     137 <- 74 * 110d      44 58 70 72 74 110 127 137
        Store 137
    ------------------------
61)     138 <- 58 * 127       44 58 70 72 74 127 138
        Store 138
    ------------------------
62)     139 <- 70 * 127       44 70 72 74 127 139
        Store 139
    ------------------------
63)     140 <- 72 * 127       44 72 74 127 140
        Store 140
    ------------------------
64)     141 <- 74 * 127d      44 74 127 141
        Store 127
        Store 141
    ------------------------
65)     Load 39               39 44 55 74 142 143 144
        Load 55
        Load 142
        143 <- 39d * 142d
        144 <- 143d + 55d
    ------------------------
66)     Load 145              44 74 144 145 146 147
        146 <- 44d * 145d
        147 <- 146d + 144d
    ------------------------
67)     Load 49               49 74 147 148 149 150
        Load 148
        149 <- 49d * 148d
        150 <- 149d + 147d
    ------------------------
68)     Load 124              74 124 150 151 152 153
        Load 151
        152 <- 124d * 150d
        153 <- 152d + 151d
        Store 153
    ------------------------
69)     Load 58               58 74 153 154
        154 <- 58d * 153
        Store 154
    ------------------------
70)     Load 70               70 74 153 155
        155 <- 70d * 153
        Store 155
    ------------------------
71)     Load 72               72 74 153 156
        156 <- 72d * 153
        Store 156
    ------------------------
72)     157 <- 74d * 153d     74 153 157
        Store 157
    ------------------------
```

Figure 7: Optimal register allocation for Comp1

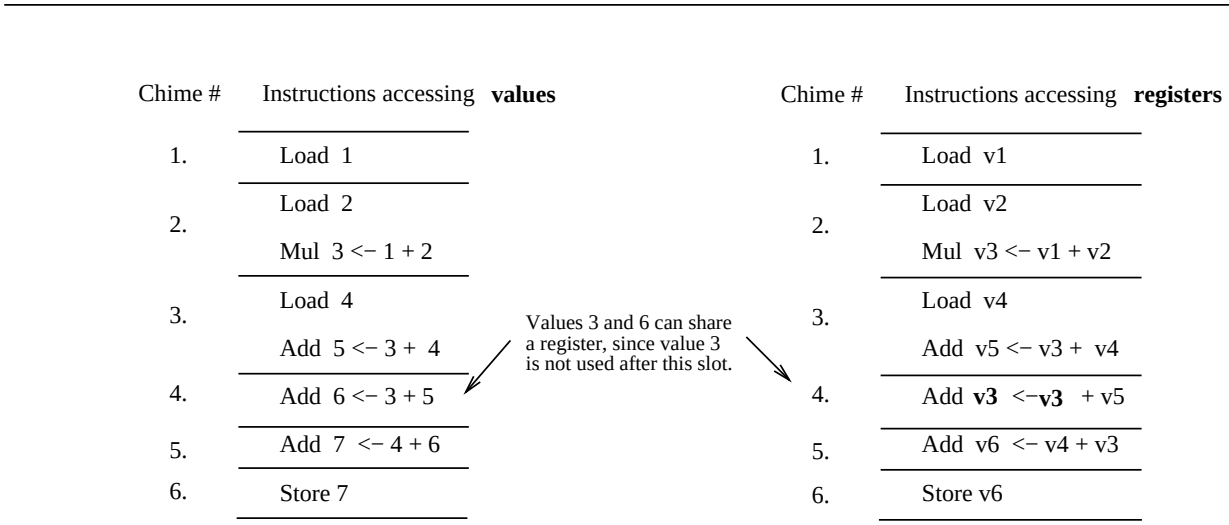

Figure 8: Same-instruction register sharing

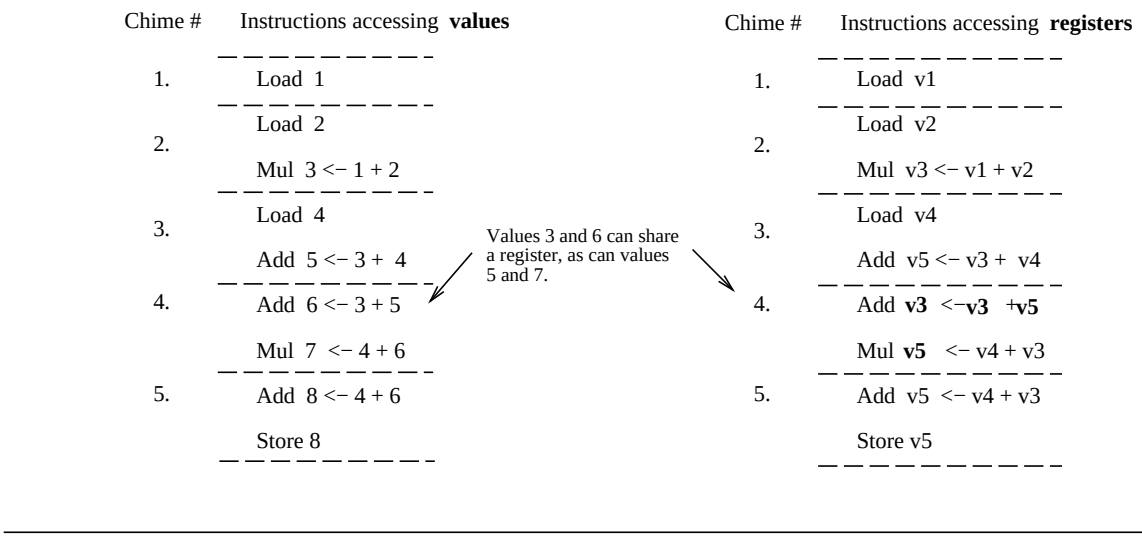

Figure 9: Inter-instruction register sharing

between values in different instructions as *inter-instruction register sharing.* In the example, values 3 and 6 share a register, and values 5 and 7 also share a register. In this way, the number of registers needed to execute a code segment is reduced.

The comparison made earlier between the multi-issue allocation algorithm's output and the Convex Fortran compiler's output is not entirely fair because our generic machine model does not allow register sharing. A value that is used in a chime occupies a register for the entire chime. Therefore, when used to optimize Convex C2 vector code, the allocation algorithm will in some instances spill values unnecessarily. A modified version of the algorithm allows register sharing, but may not exploit this capability optimally. A reduction in the number of chimes in the final schedule is achieved through the use of this heuristic. Comp1 improved to 40 extra chimes, 113 total, giving nearly a 10% reduction in chimes over the compiled code. Mul5 improved to 45 extra, 170 total, giving a 27% reduction in chimes over the compiled code.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5:78–101, 1966.

[3] E. L. Boyd. Hierarchical performance modeling with MACS: A case study of the Convex C-240. In *Proc. 20th Intl Symposium on Computer Architecture*, pages 203–212, 1993.

[4] F. C. Chow and J. L. Hennessy. The priority-based approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12:503–536, 1990.

[5] CONVEX Computer Corporation. *CONVEX Theory of Operation - C200*, volume 081-005030-000. 1990.

[6] G. J. Chaitin et al. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[7] G. J. Chaitin et al. Register allocation and spilling via graph coloring. In *Proc. ACM SIGPLAN '86 Symp. Compiler Construction*, pages 98–105, New York, 1982.

[8] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *Journal of the Association of Computing Machinery*, 13:43–61, 1966.

[9] W. Hsu, C. N. Fischer, and J. R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15:1252–1260, 1989.

A Appendix

A.1 Placement Problem Description

Given:

1. S, a set of placement slots that are numbered from 0 to $|S| - 1$.
2. R, a set of placement ranges that are numbered from 0 to $|R| - 1$. A memory operation is associated with each range. A memory operation is assigned the same number as its range.
3. $\forall r \in R, E(r)$ and $L(r)$ represent the bounds of the placement range of r. The placement range of r is then $[E(r), L(r)]$.
4. The ranges are ordered so $\forall r,r' \in R$, $r < r' \Rightarrow L(r) \leq L(r')$, i.e. the ranges are assumed to be ordered by $L(r)$. r is said to have a higher priority than r' if $r < r'$. If $L(r) = L(r')$, the order is arbitrary.

Find a function $P : R \rightarrow S \cup \{\text{NONE}\}$, called a *placement*, such that P is legal and optimal. P is *legal* if and only if:

1. $\forall r, r' \in R$, if $r \neq r', r \neq$ NONE, and $r' \neq$ NONE, then $P(r) \neq P(r')$.
   No two memory operations are placed in the same slot.
2. $\forall r \in R$, if $P(r) \neq$ NONE, $P(r) \in [E(r), L(r)]$.
   Memory operations are only placed within their placement range.

The *cost* of a legal placement P is the number of $r \in R$ such that $P(r) =$ NONE. P is *optimal* if for any legal placement P', $cost(P') \geq cost(P)$. If $P(r) =$ NONE, we say r is *unplaced*. Otherwise r is *placed*. Memory operations that are never placed will be assumed to occupy *extra* slots that are added to the schedule.

A.2 Placement Algorithm

The placement algorithm places the memory operations as follows. Each placement range is considered in order, and for each the memory operation is placed in the highest slot that is currently empty. That is, $P(r) = s$ where s is the highest empty slot in r. If there are no empty slots in r, the operation is not placed. That is, $P(r) =$ NONE.

A.3 Proof of Optimality

We prove that the placement produced by this algorithm is optimal by first defining a *standard* placement P, and then showing that any legal placement P' that is different cannot cost less than P.

A.3.1 A Standard Placement

We define a function C: $S \rightarrow R \cup \{EMPTY\}$. Given a legal placement P, $C(s)$ = EMPTY if $\nexists r \in R$ such that $P(r) = s$. Slot s is said to be *empty*. Otherwise $C(s) = r$ where $P(r) = s$. Since P is legal, this function is well defined. If more than one placement is being discussed, a subscript will be used to distinguish them (e.g. C_P and $C_{P'}$).

Given a legal placement P, the placement of $r \in R$ is *standard* if it satisfies the following conditions:

1. Condition 1. If $P(r)$ = NONE, then $\forall s \in [E(r), L(r)]$, $C(s) \neq$ EMPTY and $C(s) < r$.
   If r is not placed, then its entire placement range contains higher priority memory operations.
2. Condition 2. If $P(r) \neq$ NONE, then $\forall s \in [E(r), P(r))$, $C(s) \neq$ EMPTY and $C(s) < r$.
   If r is placed, then all earlier slots in its placement range contain higher priority memory operations.

A legal placement P is standard if and only if the placement of all $r \in R$ is standard.

A.3.2 The placement produced by the algorithm is standard

Consider a placement P produced by the placement algorithm. If P is not standard, there exists a range $r \in R$ whose placement is not standard. Range r must violate one of the four conditions:

- *r violates condition 1:* If an unplaced r violates condition 1, there must be a slot $s' \in [E(r), L(r)]$ that is empty or contains a lower priority memory operation r'. If s' is empty, then it must have been empty when r was being considered for placement. But this is a contradiction since the algorithm places r in the highest empty slot within its placement range, if such a slot exists. If s' contains a lower priority memory operation r', then r' would not yet have been placed when r was under consideration for placement. Therefore slot s' must have been empty when r was considered. But this is again a contradiction since r was assumed to have been placed in the highest empty slot within its placement range, if it exists. Therefore r cannot violate condition 1.
- *r violates condition 2:* If a placed r violates condition 2, there must be a slot $s' \in [E(r), P(r))$ that is empty or contains a lower priority memory operation r'. If s' is empty, then it must have been empty when r was being considered for placement. But this is a contradiction since r was assumed to have been placed in the highest empty slot within its placement range. If s' contains a lower priority memory operation r', then r' would not yet have been placed when r was under consideration for placement. Therefore slot s' must have been empty when r was considered. But this is again a contradiction since r was assumed to have been placed in the highest empty slot within its placement range. Therefore r cannot violate condition 2.

There cannot be such a slot $s \in S$ that violates one of the conditions, so the placement P produced by the placement algorithm is standard.

A.3.3 No placement is less expensive than a standard placement

Given a standard placement P, we show that no other placement P' can cost less than P. We show this by applying a series of operations to P' that transform it into P. At each stage, the intermediate placements are legal and cost no more than P'. Since P' is finally transformed into P, P must cost no more than P'.

If placements P and P' are not exactly the same, there must be a highest slot s such that $C_P(s) \neq C_{P'}(s)$. We consider all the ways the placements can differ:

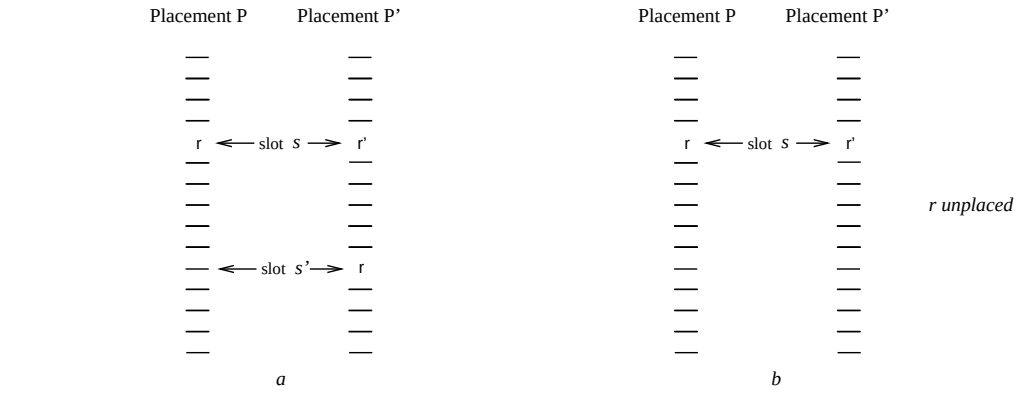

Figure 10: Slot s is not empty in either placement

- *$C_P(s) \neq EMPTY$ and $C_{P'}(s) \neq EMPTY$:* Slot s is not empty in either placement so $\exists r, r' \in R$ such that $P(r) = s$, $P'(r') = s'$ and $r \neq r'$. Since P is standard, $r < r'$. If this is not the case, P would violate one of the conditions because r would be a lower priority memory operation within the placement range of r' (whether r' is placed or unplaced in P). There are two cases to consider: r may be placed in P', or it may not.
  - If r is placed in P', it must be placed below s since the higher slots are assumed to have the same contents as P (see Figure 10a). Assume $P'(r) = s'$. Set $P'(r) = s$ and $P'(r') = s'$. The first is legal because $s \in [E(r), L(r)]$. The second is legal because $r < r' \Rightarrow L(r) \leq L(r')$. The cost of P' has not changed and the placement of memory operations above s is unchanged.
  - If r is not placed in P', set $P'(r) = s$ and $P'(r') =$ NONE (see Figure 10b). P' remains legal because $s \in [E(r), L(r)]$ and no more than one range is assigned to slot s. The cost of P' has not changed because one previously unplaced memory operation is now placed, and one previously placed memory operation is now unplaced. The placement of memory operations above s is unchanged.

Therefore, if the memory operations placed at s are different, P' is transformed so that it matches P at that slot, without increasing the cost of P'. Furthermore, the placement of memory operations above s is unchanged.

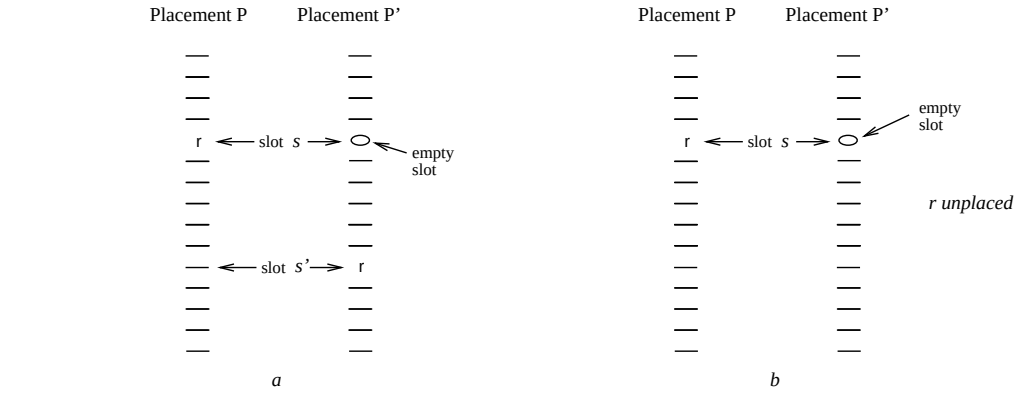

Figure 11: Slot s is not empty in placement P, but is empty in placement P'

- $C_P(s) \neq EMPTY$ *and* $C_{P'}(s) = EMPTY$*:* Slot s is not empty in placement P so $\exists r \in R$ such that $P(r) = s$. There are two cases to consider: r may be placed in P', or it may not.
  - If r is placed in P', it must be placed below s since the higher slots are assumed to have the same contents as P (see Figure 11a). Assume $P'(r) = s'$. Set $P'(r) = s$. This is legal because $s \in [E(r), L(r)]$. This operation does not change the cost of P' and the placement of memory operations above s is unchanged.
  - If r is not placed in P', $P'(r) = NONE$ (see Figure 11b). Set $P'(r) = s$. This is legal because $s \in [E(r), L(r)]$, and this operation reduces the cost of P' by 1. the placement of memory operations above s is unchanged.

  Therefore, if a memory operation is placed at s in P and P' has an empty slot at s, P' can be transformed so that it matches P at that slot, without increasing the cost of P'. Furthermore the placement of memory operations above s is unchanged.

- $C_P(s) = EMPTY$ *and* $C_{P'}(s) \neq EMPTY$*:* Slot s is not empty in placement P' so $\exists r \in R$ such that $P'(r) = s$. There are two cases to consider: r may be placed in P, or it may not.
  - If r is placed in P, it must be placed below s since the higher slots are assumed to have the same contents in P' (see Figure 12a). The empty slot at s is higher than r and within its placement range which violates condition 2. Since P is assumed to be standard, this is a contradiction.
  - If r is not placed in P, the empty slot at s is within its placement range which violates condition 1 (see Figure 12b). Since P is assumed to be standard, this is a contradiction.

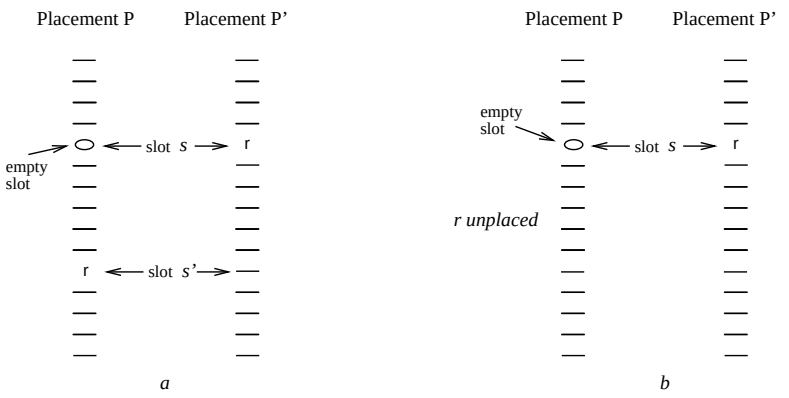

Figure 12: Slot s is empty in placement P, but is not empty in placement P'

Therefore it is not possible for P to have an empty slot at s if $P'(r) = s$.

We have shown that regardless of how placements P and P' differ at slot s, P' can be transformed to eliminate the difference, without increasing its cost. These transformations can be performed for each slot $s \in S$, so P' is eventually transformed into P. Since the cost of each intermediate placement is never more than the cost of P', the cost of P is not more than the cost of P'. P is an arbitrary standard placement and P' is an arbitrary placement, so a standard placement is never more expensive than any other placement.

Since the placement produced by the placement algorithm is standard, and there are no placements cheaper than a standard placement, the placement produced by the algorithm is optimal.