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@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 feasible register allocations; implicit and explicit state pruning are used to make the problem tractable without sacrificing optimality. 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 no more than a given number of registers and achieve minimum runtime on a generic multi-issue machine.

Heuristic solutions to local and global register allocation problems for single-issue machines have been discussed by Chow et al., [3], and Aho et al. [1]. Chaitin et al. described the application of graph coloring to global register allocation in [5] and [6]. 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 NP-complete. While register coloring can be adapted to perform local allocation, it has traditionally been used in global allocation across multiple basic blocks.

Horwitz et al. [7] 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. [8] 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 sometime their first use. *Output* values are defined within the code segment and are considered to be used sometime after the segment ends. They must be stored sometime 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 (unless it is an input value) 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 machine model, we assume that a group of instructions can be issued 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

1994 International Conference on Supercomputing, pages 107-116, Manchester, England.

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. For convenience, this spill code is considered to include loads and stores of input and output values, respectively. 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.

The following sections describe our dynamic programming approach to optimal multi-issue register allocation. In the remainder of section 1 we discuss the slot-based multi-issue machine model and the constraints that define a legal register allocation. In section 2 the details of the allocation algorithm and in section 3 we show results of running the algorithm on sample code.

1.1 A Generic Multi-Issue Machine/Code Model

The input to the allocation problem is a schedule of instructions, without loads and stores, each of which *defines* (writes) one value and *uses* (reads) up to two values, and a legal assignment of those instructions to issue slots. 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.

In the generic model, all instructions execute in one of three functional units: an adder, a multiplier, and a port (load/store unit). 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 constraints of the Convex C2 vector supercomputer quite well.

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 contain 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 that slot. Additional values may also be assigned to registers in that slot up to a total of r.

1.2 Formalization of the Multi-Issue Allocation Problem

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 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 given schedule are numbered sequentially starting from 0. The following formalization of the allocation problem uses this convention:

Inputs

1. A schedule of add and multiply instructions that define and use values, and a legal assignment of these instructions to a sequence of slots.
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 schedule, before the first slot, no values are in a register.
3. At the start of the 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. A value is either in memory or not in memory during each slot.
2. A value is either in a register or not in a register during each slot.
3. During any slot where a value is loaded or stored, the value must be present in both a register and in memory.
4. 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.

[1] Many machines do 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 3.3 we discuss extensions to the model that do allow register sharing.

5. If a value is present in memory in slot i and is not in memory in slot i-1, then that value must be stored during slot i (and therefore in a register in slot i by constraint 3).
6. The number of values in registers in each slot must not exceed the number of available physical registers.

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

2 Optimal Register Allocation using Dynamic Programming

Our algorithm is an extension of the algorithms described by Horwitz [7] and Hsu [8] 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 [7] and [8] to efficiently solve the single-issue load/store minimization problem using dynamic programming and pruning rules. We then describe our extension for the multi-issue case.

2.1 Single-Issue Load/Store Minimization

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 final schedule 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 [7] 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 never exceeds 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 original input schedule, not the load/store instructions (extra slots in our algorithm), are numbered.

The rules described in [7] and [8] 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 [7] 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.

2.2 Multi-Issue Register Allocation

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 have already been generated but are clearly no better than some other unpruned partial solution. The algorithm employs a single forward scan of the given schedule with breadth-first expansion and pruning.

2.2.1 Unresolved Loads and Stores

If n values are live in the current slot, then $n - r$ values are *excluded* from registers in this slot (for $n \geq r$, no values are excluded otherwise). This process insures that the allocation remains legal. Later sections discuss how the algorithm chooses which values to exclude. A value is said to be *excludable* in a slot if it is live and neither defined nor used in the slot.

Definitions, uses, and exclusions of a value throughout a schedule define a *placement range* for each load and store. A store of a value may be placed anywhere between the value's definition and its first exclusion (see Figure 1). A load of a value must be placed between the last exclusion before a use of the value, and that use (see Figure 2). 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

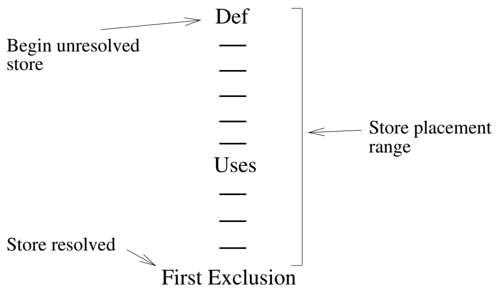

Figure 1: Store placement ranges

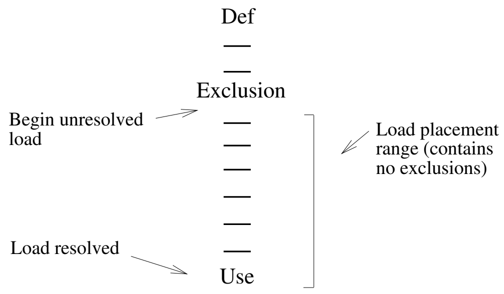

Figure 2: Load placement range

as a load or store becomes resolved, it can simply be 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 the extra slots for stores immediately after the value definition (top of the range) and for loads immediately before the value use at the (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 optimal.

2.2.2 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 in the slot where it is defined (see Figure 3). It enters MEM state in any slot where 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 whether 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.

2.2.3 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 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, becomes 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, whenever a load/store of some value, v', is resolved, $E(v)$ is decremented if $E(v) \geq E(v')$. 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.

2.2.4 Mandatory Exclusion Rules

Our algorithm uses mandatory exclusion rules to insert value exclusions that cannot increase the cost of the optimal allocation. In following two rules, the *current slot* refers to the slot currently being considered for value exclusions. Note that the value states referred to below are the states as seen by the allocation algorithm. They are unaffected by the actual load/store placement.

Rule 1: If values x and y enter the current slot in MEM state with $E(x) = E(y) = 0$, and y is used in the current slot but x is not, then x must be excluded in the current slot (see Figure 4):

Value y must be loaded in the current slot. Since $E(x) = 0$ just before entering the slot and y is loaded in the current slot, loading x in or above the current slot would add one extra slot. Since x is not used in the current slot, it can be loaded later and will at worst add one extra slot then. Therefore there is no reason to avoid excluding x in the current slot.

Rule 2: If value x is an output value that enters the current slot in MEM or BOTH state and x is not used in or below the current slot, then x must be excluded.

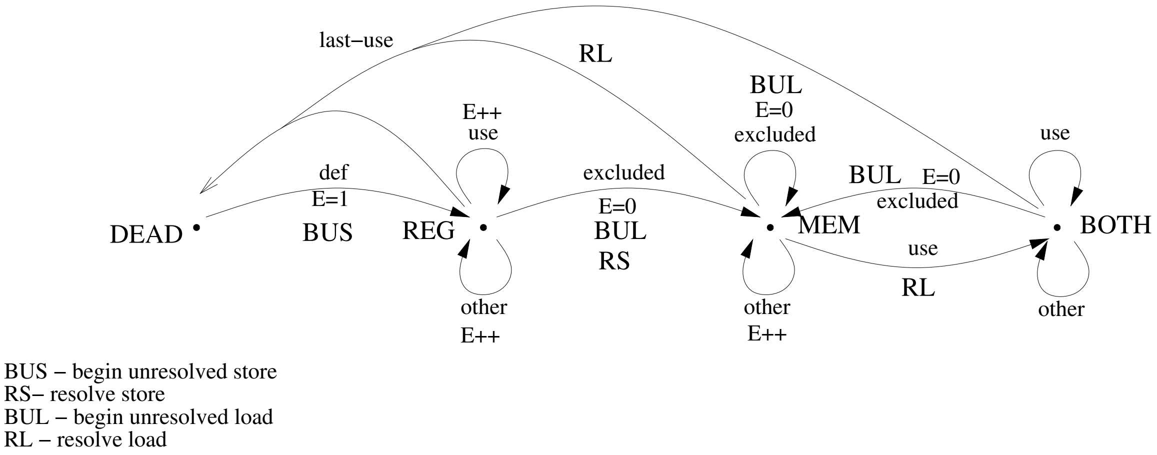

Figure 3: Transitions between last known states, and empty-slots (E) calculations.

Since x is never used again within the code segment, it need not occupy a register. Since x is already in memory, excluding x in the current slot does not cause any loads or stores to be needed within the code segment.

2.2.5 Implicit Pruning Rules

Our algorithm employs pruning rules that eliminate partial solutions that are clearly no less costly than other unpruned 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. To reduce this number, we define for each slot a partial ordering $\succ$ on the excludable values at that slot. If $x \succ y$, then at this slot value y will never be excluded unless value x is also excluded. If a rule produces $x \succ y$ and $y \succ x$ for some pair of values, *lexical tie breaking* is used to define the partial ordering in one direction. The pruning rules described 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 [8]. 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. Rules #3, #4 and #5 bear some similarity to the 2nd, 3rd and 4th observations in [8]. 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. The other three rules, #1, #2 and #6, are new.

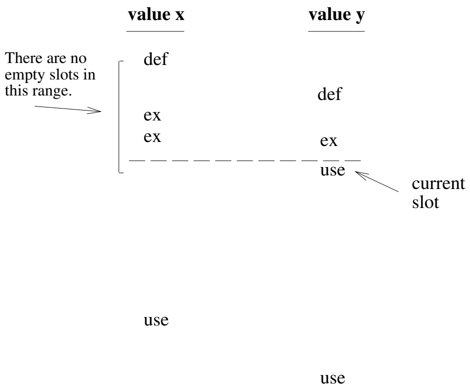

Figure 4: Rule 1.

The rule descriptions refer to the value states and the number of extra slots stored in the partial solution, and to two relations on live value pairs: x COVERS y in a particular slot if value x is excludable in every slot in which y is excludable from the current slot to the end of the schedule. x LCOVERS y, i.e. locally covers, in a particular slot if value x is excludable in every slot in which y is excludable from the current slot to the next slot in which y is not excludable. Note that x LCOVERS y if and only if (NEXTREAD(x) $\geq$ NEXTREAD(y)), and whenever x and y have a single use remaining x LCOVERS y if and only if x COVERS y.

Rule 3: If value x is in MEM state, value y is in REG, MEM or BOTH state and x LCOVERS y then $x \succ y$.

Rule 3 ensures that under the stated conditions, y will never be excluded in the current slot unless x is also excluded in the current slot. Consider a complete allocation for a schedule (which defines the register assignment for every slot and the load/store placement) and the set of exclusions, value states and extra slot attributes that produced that allocation. Suppose that rule 3 is violated at some slot, i.e.

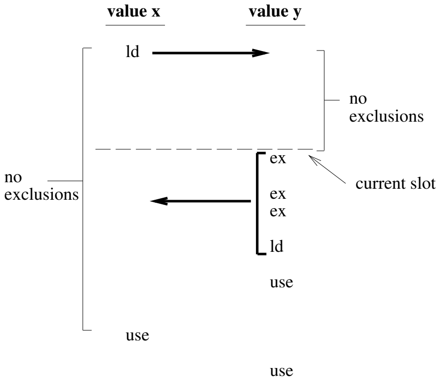

Figure 5: Rule 3, case 2 transformation.

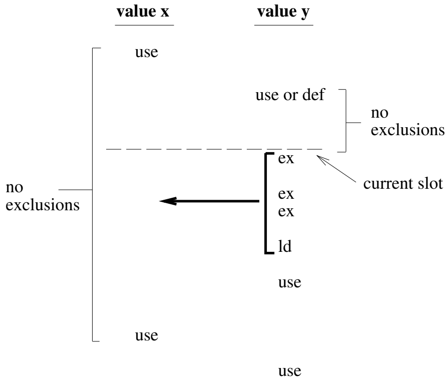

Figure 6: Rule 4, case 2 transformation.

x and y satisfy the conditions of rule 3 but y is excluded and x is not. We call the first such slot the *current slot* and construct another allocation that is no more expensive and yet does not violate rule 3 in the current slot. We will therefore conclude that the application of this rule will not cause our algorithm to produce a suboptimal allocation.

Note that since x is in MEM state there is some load of x before its next use. Consider the following two placements of this load:

1. The load of x is placed below the current slot: Since x is not in a register in the current slot, x can be excluded in the current slot at no additional cost. This new allocation is as expensive as the given allocation, and rule 3 is no longer violated in the current slot or above.

2. The load of x is placed at or above the current slot (see Figure 5): Since y is excluded in the current slot, a load of y must be placed between the current slot and the next use of y.

   Note that there are no exclusions of x between the load and its next use. Since the current slot is the first violation of rule 3, there are no exclusions of y at the slot where x is loaded or at any slot after that down to the current slot. Since x LCOVERS y, the load of x can be moved down to the slot where y is now loaded. Then all exclusions of y from the current slot (inclusive) up to the load of y can be moved to x instead, and the load of y, if still needed, can be moved up to the slot where x was loaded in the given allocation.

   As a result of this transformation, rule 3 is no longer violated in the current slot or above. This new allocation is as expensive as the given allocation if the load of y is still needed, and cheaper if not.

Rule 4: If value x is in BOTH state and y is in REG or BOTH state, and x LCOVERS y, then $x \succ y$.

Suppose that rule 4 is violated at some slot, i.e. x and y satisfy the conditions of rule 4 but y is excluded and x is not. Consider the following two cases for the first violation of rule 4:

1. Value x is excluded below the current slot before its next use: In this case, it is possible to construct an allocation in which x is excluded in the current slot at no additional cost. This new allocation is as expensive as the given allocation, and rule 4 is no longer violated in the current slot or above.

2. Value x is not excluded before its next use (see Figure 6): Since y is excluded in the current slot, a load of y must be placed between the current slot and the next use of y.

   Note that there are no exclusions of x between the current slot and its next use. Since x LCOVERS y and y is in REG or BOTH state in the current slot, the load of y and all the exclusions of y from the current slot through the next use of y can be applied to x instead. This new allocation is as expensive as the given allocation, and rule 4 is no longer violated in the current slot or above.

Rule 5: If both values x and y are in REG state, x COVERS y, and $E(x) \geq E(y)$, then $x \succ y$.

Suppose that rule 5 is violated at some slot, i.e. x and y satisfy the conditions of rule 5 but y is excluded and x is not. Consider the following two cases for the first violation of rule 5. Consider the following three cases for the first violation of rule 5:

1. Value x is never excluded: Since x COVERS y and y is in REG state in the current slot, all the exclusions, load(s) and store of y from the current slot to the end of the schedule can be applied to x instead. Since x and y are in REG state at the current slot, neither is excluded above the current slot. Therefore y will not be loaded above the current slot. If y is stored above the current slot, $E(x) \geq E(y)$ ensures that x is defined no later than y and this store can be applied to x instead. This new allocation is as expensive as the original allocation, and rule 5 is no longer violated in the current slot or above.

2. Value x is eventually excluded and the store of x is placed at or below the current slot (see Figure 7): Since y is excluded in the current slot, the store of y will be placed somewhere at or below the definition of y and above the current slot.

   All exclusions of y from the current slot up to, but not including, the first exclusion of x can be applied to x. This is legal because x COVERS y. Furthermore, all loads of y in the same range can now be applied to x since y is no longer excluded in this range. Since x COVERS y there must be a use of y in the same slot as

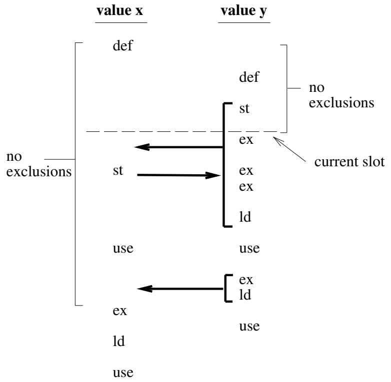

Figure 7: Rule 5, case 2 transformation.

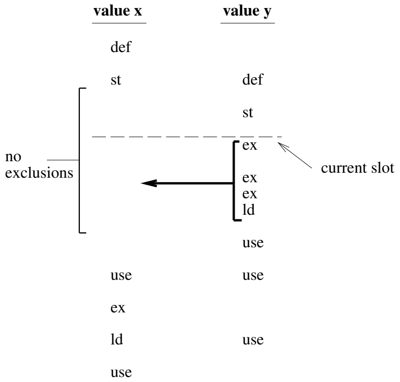

Figure 8: Rule 5, case 3 transformation.

each use of x in this range. Hence these loads will accomplish all the necessary reloading of x in this range caused by moving the exclusions. Furthermore, some of these loads may not be needed. Finally, the stores of x and y, which both occur above the first exclusion of x, can be interchanged. Note that the store of y may no longer be necessary. This new allocation is no more expensive than the original allocation, and rule 5 is no longer violated in the current slot or above.

3. Value x is eventually excluded and the store of x is placed above the current slot: In this case, the stores need not be interchanged.

   If x is excluded before its next use, then x may be excluded in the current slot at no cost and rule 5 is no longer violated in the current slot or above.

   If x is not excluded before its next use (see Figure 8), then all exclusions and the reload of y before its next use can be applied to x. Note that since x COVERS y, this next use of y will not occur below the next use of x.

   This new allocation is no more expensive than the original allocation, and rule 5 is no longer violated in the current slot or above.

It can be shown that the implicit pruning rules with lexical tie breaking impose a partial ordering on the excludable values that remain after mandatory exclusions have been performed. Note that for a given pair of values, at most one implicit pruning rule can apply in the current slot. Since none of these rules moves any excludes that appear above the current slot, and all rules are considered before moving to the next slot, the rules do not interfere with one another and use of the entire set of rules must leave at least one optimum solution in the search space.

2.2.6 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 6: If for each value v, its state in P_1 is the same as its state in P_2, and $E_1(v) \geq E_2(v)$ for all values v (excluding values in BOTH or DEAD state for which $E(v)$ is undefined), and if 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 each currently unresolved load and store in P_1 has at least as many empty slots as it does in P_2. Since P_2 is no less expensive to start with, P_2 cannot result in a less expensive final allocation than the best allocation for P_1.

3 Optimal Allocation for the Convex C2 Vector Processor

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 [4]. The generic multi-issue machine/code model described above is based on the Convex C240 machine model described in [2]. This C240 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 rules makepp it an ideal target for studying the reduction of register-memory spill traffic.

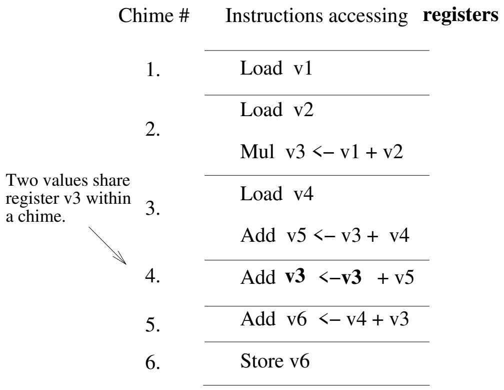

Figure 9: Same-instruction register sharing.

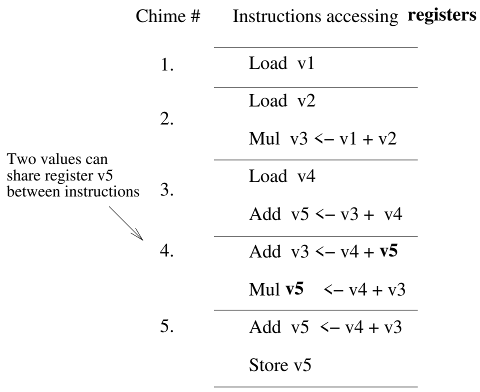

Figure 10: Inter-instruction register sharing.

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 optimal allocation.

The optimal allocation found by our multi-issue allocation algorithm consumes 27 extra chimes, or 100 chimes altogether, a 20% improvement in predicted performance. The algorithm took 62 minutes to run on an NCR 3550 system with a 50 MhZ Intel 486 processor and used approximately 20 MB of memory. The optimal allocation for Comp1 is shown in Figure 11. 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 optimal allocation algorithm finds an allocation that requires only 68 extra chimes, or 193 chimes altogether, representing a 17% improvement in predicted performance. This run took 15 hours and 43 minutes on the same NCR computer.

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 in this write after read fashion 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 9, two values can share register v3. The C2 register file allows a value to be read out of v3 as a new value is being written into it.

If a value used in the first instruction of a two-instruction chime is never used again, it is possible to reuse the register for its defined value, as shown in Figure 10. We refer to this sharing of registers between values in different instructions as *inter-instruction register sharing.* 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. The number of extra chimes in the final schedule achieved through the use of this heuristic is reduced to 26. With one more read port into the register file, the number of extra chimes is reduced to 18.

4 Conclusion

The algorithm presented finds an allocation of values to registers that maximizes the performance of a generic multi-issue machine. The generic machine can be adapted in a straightforward way to model particular vector and VLIW machines, and we believe it can be extended to model superscalar machines. We have applied the algorithm to the Convex C2 vector supercomputer and presented two examples of performance improvements that can be achieved through optimal allocation. Finally, we believe that this algorithm may form the basis for algorithms to jointly optimize code scheduling and register allocation for multi-issue machines.

Acknowledgment: We gratefully acknowledge the advice of Professor Kevin Compton who was very helpful in formulating the algorithmic approach.

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

Figure 11: Optimal register allocation for Comp1.

References

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

[2] E. L. Boyd and E. S. Davidson. 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.