# Efficient Instruction Scheduling for Delayed-Load Architectures

STEVEN M. KURLANDER
University of Wisconsin
and
TODD A. PROEBSTING
University of Arizona
and
CHARLES N. FISCHER
University of Wisconsin

A fast, optimal code-scheduling algorithm for processors with a delayed load of one instruction cycle is described. The algorithm minimizes both execution time and register use and runs in time proportional to the size of the expression-tree. An extension that spills registers when too few registers are available is also presented. The algorithm also performs very well for delayed loads of greater than one instruction cycle. A heuristic that schedules DAGs and is based on our optimal expression-tree-scheduling algorithm is presented and compared with Goodman and Hsu's algorithm Integrated Prepass Scheduling (IPS). Both schedulers perform well on benchmarks with small basic blocks, but on large basic blocks our scheduler outperforms IPS and is significantly faster.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Code scheduling, load/store architecture, register allocation

## 1. INSTRUCTION SCHEDULING

Modern RISC architectures are characterized by small, simple-instruction sets, and general-purpose registers. While simple functionally, many of the instructions are complicated by instruction-scheduling requirements. For instance, on a MIPS R3000, an integer load from memory into a register requires a single de-

This work was supported by NSF Grant CCR-9122267.
An earlier version of this article was presented at the ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation.
Authors' addresses: S. M. Kurlander, C. N. Fischer, Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706; email: {smk; fischer}@cs.wisc.edu; T. A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721; email: todd@cs.arizona.edu.

| Optimal | Nonoptimal |
|---|---|
| , r1 | , r1 |
| , r2 | , r2 |
| , r3 | **nop** |
| **add** r1, r2, r1 | **add** r1, r2, r1 |
| **add** r1, r3, r1 | , r2 |
| - | **nop** |
| - | **add** r1, r2, r1 |

Fig. 1. Two legal schedules to evaluate `(a+b)+c` on a MIPS R3000.

lay cycle before the loaded value can be accessed. It is necessary to find another instruction—that does not rely on the loaded value, or contribute to the load's address computation—to be placed immediately after the load. If no useful instruction can be found, it is necessary to put a NOP after the load to absorb the *delay* cycle.

Figure 1 gives two legal code sequences for evaluating `(a+b)+c` for the MIPS R3000. (All our examples, including this one, will use an instruction set with destinations as the rightmost operand.) The instructions selected to evaluate the expression are the same except for register assignment. The useful instructions differ only in their schedules (orders) and numbers of registers used. The right sequence requires two NOP's because the values loaded are accessed immediately by the subsequent instructions. A compiler (or assembler) must order instructions carefully to minimize the costs of scheduling constraints.

While the optimal evaluation order in Figure 1 requires two fewer instructions than the nonoptimal, it does require one more register. Avoiding scheduling conflicts requires the ability to move operations *away* from the instructions that load their operands. This lengthens the span of those register operands and, therefore, increases the number of registers in use. Because registers are scarce, and can be advantageously used to hold temporary and global values, it is important not to overuse them when scheduling instructions.

## 2. OVERVIEW

The problem of optimally scheduling instructions under arbitrary pipeline constraints is NP-complete [Garey and Johnson 1979; Hennessy and Gross 1982; Lawler et al. 1987; Palem and Simons 1990]. Many heuristics have been proposed for scheduling pipelined code; all assume, however, that pipeline constraints can occur after any instruction, and that operators may share common subexpressions. The intractability of finding an optimal schedule holds even if an unlimited number of registers is available. Optimal local register allocation in itself is also NP-complete in the presence of common subexpressions [Garey and Johnson 1979]. Such negative results have led to the belief that generating good-quality code for RISC machines with pipeline constraints is too difficult to do well except in complex optimizing compilers.

Fast, optimal algorithms, however, can be devised for simpler, yet realistic architectures. Our results show that for a restricted set of pipeline constraints and a simple RISC load/store architecture, optimal code can be generated in linear time for expressions without operand sharing. Our *delayed-load-scheduling* algorithm, DLS, efficiently combines instruction scheduling and register allocation. Initially, we restrict our discussion to handling expression-trees in which all leaf nodes are direct memory references and all operators binary. DLS is as an attractive, simple, fast, and effective alternative to more-complicated, slower heuristic solutions.

## 3. PREVIOUS WORK

An adaptation of Hu's algorithm [Hu 1961] gives an optimal solution to scheduling a tree-structured task system on multiple identical processors if each task has unit execution time [Coffman 1976], but the algorithm does not handle register allocation constraints. For an architecture with two functional units, one for loads and one for operations, with identical pipeline constraints, Bernstein et. al. [1984; 1989] have investigated code scheduling with register allocation for trees. Although applicable to a much different machine, Bernstein's results and algorithms are similar to ours[1]—both minimize pipeline interlocks and register usage, and both run in $O(n)$ time (where $n$ is the number of nodes in the expression).

Code-scheduling algorithms and heuristics for pipelined architectures have been extensively studied in recent years. Most of the attention to code scheduling has been directed at scheduling expressions represented by directed acyclic graphs (DAGs) for architectures with pipeline constraints after both loads and operations.[2] Heuristic attacks on this general problem can be found in Hennessy and Gross [1982; 1983], Gibbons and Muchnick [1986], Warren [1990], Lawler et al. [1987], and Palem and Simons [1990]. These techniques are similar in spirit; they schedule instructions from the bottom of the DAG based on differing priority heuristics. The heuristics tend to favor those instructions that (a) are ready to execute (i.e., do not face pipeline constraints); (b) will cause subsequent pipeline constraints (i.e., need to be scheduled early); (c) are "far" from the roots of the DAG (i.e., may be on a critical execution path).

Many heuristic solutions treat register allocation as a separate issue that occurs either before or after scheduling. Most heuristics work in a breadth-first manner from the bottom of the DAG up; they tend to cause many values to be live at once—filling up scarce registers. Unlike DLS, these algorithms fail to integrate code scheduling and register allocation fully, and therefore suffer from phase-ordering problems. In addition, whereas DLS runs in $O(n)$ time, these algorithms run in $O(n^2)$ time and must have an additional register allocation phase.

Attempts to integrate register allocation and scheduling have been made at the basic-block level. The techniques express the data dependences between instructions within a basic block as a DAG. Given the DAG, they attempt to schedule the instructions while both obeying pipeline constraints and minimizing registers. Since both optimal scheduling and register allocation on DAGs are NP-complete problems, their solutions to the integrated problem are heuristic.

Goodman and Hsu [1988] describe a system, Integrated Prepass Scheduling (IPS), that combines register allocation and instruction scheduling. IPS is conceptually simple. The input is an instruction DAG for which registers have not been assigned. IPS consists of two possible schedulers: CSP and CSR. CSP does heuristic

[1] Ours can issue only one instruction per cycle.

[2] We will use *operations* to denote nonload instructions.

| pattern | | | instruction |
|---|---|---|---|
| *reg* | $\leftarrow$ | *memory* | , *reg* |
| $reg_i$ | $\leftarrow$ | $reg_j$ **op** $reg_k$ | **op** $reg_j$, $reg_k$, $reg_i$ |
| *memory* | $\leftarrow$ | *reg* | **store** *reg*, *memory* |

Fig. 2. Initial DLS machine model.

scheduling at the cost of voracious register use, and CSR tends to minimize register use while possibly doing poor scheduling. Given a DAG, IPS schedules instructions using CSP and maintains a count of available registers. When the count no longer exceeds a threshold, IPS switches to CSR to reduce register usage. Once reduced appropriately, IPS reverts to CSP. This oscillation continues until the scheduling process is complete.

Bradlee et al. [1991] describe another integrated system, Register Allocation with Schedule Estimates (RASE), and compare it to IPS. RASE works in three sequential passes: PRESCHED, GRA, and FINALSCHED. For each basic block, PRESCHED estimates the cost of evaluating that basic block with $n$ registers available, for all legal register counts. Given these cost vectors, the global register allocator, GRA, computes the optimal number of registers to give to the block in face of register competition for global values. FINALSCHED simply completes the schedule required by the register level determined by GRA.

Bradlee et al. found that IPS and RASE work well in practice—reducing execution time by an average of 12%. While RASE worked better occasionally, the resulting improvement was not significant. Both systems rely on heuristic scheduling techniques that are slow ($O(n^2)$) and require an ad hoc integration of register allocation and instruction scheduling.

## 4. DELAYED-LOAD ARCHITECTURE

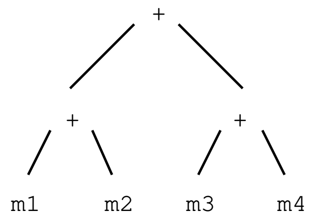
Initially we considered a simple class of architectures—RISC load/store architectures with delayed loads. Their machine instruction set is given in Figure 2. All instructions require a single instruction cycle to issue, and only loads are pipelined. This architecture is an approximation of the integer functional units of many modern RISC processors such as the SPARC and MIPS R3000 [Patterson and Hennessy 1990].

A delayed load requires that the destination of a load not be accessed by subsequent instructions for some number of instruction cycles, although other, unrelated instructions may execute. *Delay* will be used to refer to the number of cycles that must elapse before the destination register is ready to be used. An attempt to use a destination register prior to the elapsing of *Delay* cycles forces a pipeline interlock that blocks processor execution until the register has finished loading.

Figure 3 shows two possible evaluations of an example expression-tree. It is assumed that *Delay* = 1. The (naively produced) left sequence wastes cycles due to unfilled delay slots at times 3 and 7—asterisks (*) denote the registers with which the delays are associated. The right sequence incurs no delays.

## 5. REGISTER ALLOCATION TRADE-OFFS

Register allocation and instruction scheduling interact because the order of instructions determines the register needs for computing a given expression. Likewise,




| Cycle# | Unfilled Delay Slots | Filled Delay Slots |
|---|---|---|
| 1. | 1, r1 | 1, r1 |
| 2. | 2, r2 | 2, r2 |
| 3. | | 3, r3 |
| 4. | **add** r1, r2*, r2 | 4, r4 |
| 5. | 3, r1 | **add** r1, r2, r2 |
| 6. | 4, r3 | **add** r3, r4, r4 |
| 7. | | **add** r2, r4, r4 |
| 8. | **add** r1, r3*, r3 | |
| 9. | **add** r2, r3, r3 | |

Fig. 3. Sample expression-tree and two evaluation sequences.

register allocation can limit or expand the possibilities for reordering code to fill delay slots.

If register allocation precedes instruction scheduling, the ability to schedule the code can be severely limited by constraints induced not by data dependences, but by constraints introduced by potential register interference. If register allocation follows instruction scheduling, a given schedule may require unnecessarily many registers, thus limiting the effectiveness of a global optimizer and possibly requiring spill code. This well-known phase-ordering problem is accepted in practice, but can lead to suboptimal register use because the instruction schedulers minimize scheduling delays *without* taking into account the possibility that increased register demands could lead to costly register spilling.

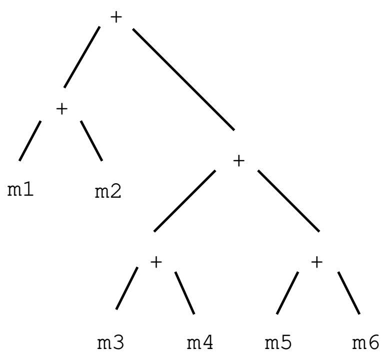
The DLS algorithm avoids this phase-ordering problem by combining instruction scheduling and local register allocation for expression-trees. DLS schedules instructions optimally to avoid all unfilled delay slots for expression-trees when $Delay = 1$. Furthermore, it finds a delay-free schedule that minimizes register usage. When $Delay > 1$, DLS serves as an excellent heuristic while retaining its conceptual simplicity, guaranteed linear performance, and integrated register allocation.

### 5.1 Canonical Form

Generating code and allocating registers is much simpler for expression-trees than for arbitrary DAGs. Once a preliminary schedule for the code has been generated for a tree, and the register needs determined, it is possible to reschedule the code and reassign the registers to obtain a code sequence in a canonical form. Ordering code in a canonical form represents the last phase of the DLS algorithm described in Section 7.

Our canonical form has three important invariants: the relative order of the operators remains unchanged; the relative order of the loads remains unchanged; and the number of registers needed remains unchanged. For a given number of registers and specific operation and load orders, the canonical order will minimize unfilled delay slots for a delayed-load machine.

The canonical schedule is produced by moving loads as early as possible in the initial instruction sequence (subject to the three invariants). Shifting the loads will move a load away from its parent in the tree and therefore increase the number of instructions between the load and its dependent operation.




| # | Sethi-Ullman(3) | Canonical(3) | Canonical(4) |
|---|---|---|---|
| 1. | 3, r1 | 3, r1 | 3, r1 |
| 2. | 4, r2 | 4, r2 | 4, r2 |
| 3. | | 5, r3 | 5, r3 |
| 4. | **add** r1, r2*, r2 | **add** r1, r2, r2 | 6, r4 |
| 5. | 5, r1 | 6, r1 | **add** r1, r2, r2 |
| 6. | 6, r3 | | 1, r1 |
| 7. | | **add** r3, r1*, r1 | **add** r3, r4, r4 |
| 8. | **add** r1, r3*, r3 | 1, r3 | 2, r3 |
| 9. | **add** r2, r3, r3 | **add** r2, r1, r1 | **add** r2, r4, r4 |
| 10. | 1, r1 | 2, r2 | **add** r1, r3, r3 |
| 11. | 2, r2 | | **add** r4, r3, r3 |
| 12. | | **add** r3, r2*, r2 | |
| 13. | **add** r1, r2*, r2 | **add** r1, r2, r2 | |
| 14. | **add** r3, r2, r2 | | |

Fig. 4. Expression-tree and canonical instruction sequences.

To produce the canonical ordering of an instruction sequence using R registers that has L loads and (L–1) operations,[3] create an ordering that consists of R loads followed by an alternating sequence of L–R (op,load) pairs, followed by the remaining R–1 operations. Loads are moved before operations that they had previously followed—this does not affect data dependences since (1) all operations depend on registers and (2) all loads depend on memory. The movement of the loads relative to the operations will cause the necessary register assignments to change; if done systematically this will not cause the register needs to increase. Since each load increases the number of registers in use by one, and each operation decreases the number of registers in use by one, the number of registers in use at any point in the evaluation is equal to the number of loads performed minus the number of operations performed. A canonical order evaluation, therefore, ensures that the number of registers in use will never exceed R.

Figure 4 gives an example expression with a standard Sethi-Ullman (SU) instruction schedule [Sethi and Ullman 1970], a canonical order with three registers, and a canonical order assuming four registers. (The Sethi-Ullman order, which is optimal with respect to register usage, orders instructions by scheduling subtrees separately

[3]There are (L–1) operations in a binary tree with L loads.

| § | pattern | | | instruction |
|---|---|---|---|---|
| §7 | *reg* | $\leftarrow$ | *memory* | , *reg* |
| | *memory* | $\leftarrow$ | *reg* | **store** *reg*, *memory* |
| | $reg_i$ | $\leftarrow$ | $reg_j$ **op** $reg_k$ | **op** $reg_j$, $reg_k$, $reg_i$ |
| §8 | *reg* | $\leftarrow$ | *const* | **loadi** *const*, *reg* |
| | $reg_i$ | $\leftarrow$ | **op** $reg_j$ | **op** $reg_j$, $reg_i$ |
| §9 | *reg_or_imm* | $\leftarrow$ | $reg_{var}$ | |
| | *reg_or_imm* | $\leftarrow$ | $const_{small}$ | |
| | *reg_or_imm* | $\leftarrow$ | *reg* | |
| | *reg* | $\leftarrow$ | $reg\_or\_imm_j$ **op** $reg\_or\_imm_k$ | **op** $reg\_or\_imm_j$, $reg\_or\_imm_k$, *reg* |
| | *reg* | $\leftarrow$ | **op** *reg_or_imm* | **op** *reg_or_imm*, *reg* |
| | *memory* | $\leftarrow$ | *reg_or_imm* | **store** *reg_or_imm*, *memory* |

Fig. 5. DLS machine model.

so that the subtree with the greater register need is scheduled first.) Simply putting the SU-generated instructions into canonical form without additional registers removes one unfilled delay slot. Adding the extra register eliminates all unfilled delay slots. Note that the relative orders of loads and the relative order of operations are the same in all three sequences.

### 5.2 Adding Registers Helps

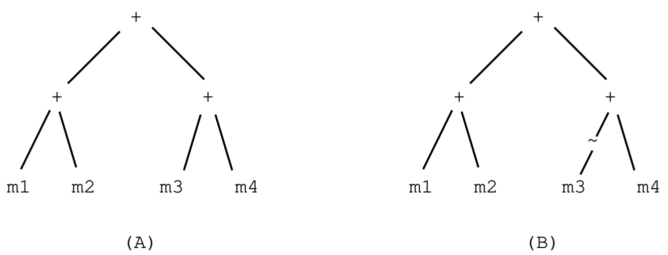
As is seen in Figure 4, adding registers helps sometimes. This follows from the observation that loads can often be *shifted* backward (i.e., earlier) in the instruction sequence without affecting the outcome of the computation. This shifting does not change the relative ordering of the loads with respect to one another, or the relative ordering of the operations with respect to one another—it simply shifts the loads farther from the operations that use them. Shifting a load farther away allows its delay slot to be filled with an intervening load or operation.

Minimizing the number of registers needed to evaluate an expression without load delays is an essential consideration. If the operations in the expression-tree in Figure 4 were evaluated from left to right, it would be necessary to use five registers rather than four to produce a canonical evaluation without unfilled delay slots. It is therefore necessary to treat the problem of optimal code generation as one of minimizing unfilled delay slots and register usage through code scheduling.

## 6. OVERVIEW OF DLS

Figure 5 outlines our discussion of DLS. In Section 7 the basic DLS algorithm for finding optimal schedules for expressions with loads, stores, and binary operators is presented. Adding unary operators and literals, both of which are delay free, allows optimal schedules to sometimes use one fewer register than is needed by the basic DLS algorithm of Section 7. An algorithm that determines the subtree ordering and the minimum number of registers needed for an optimal schedule in the presence of unary operators and literals is described in Section 8.

Figure 6 presents an example in which a tree with a unary operator has a delay-free schedule requiring one less register than a delay-free schedule for the tree without the unary operator. In Figure 6, tree B is similar to tree A, but tree B has the unary minus operator (represented as "∼") in the tree and **neg** in the instruction sequence below. Both trees have a Sethi-Ullman number of three. The




| Cycle# | Tree A: Canonical(3) | Tree A: basic DLS(4) | Tree B: DLS(3) |
|---|---|---|---|
| 1. | 1, r1 | 1, r1 | 1, r1 |
| 2. | 2, r2 | 2, r2 | 2, r2 |
| 3. | 3, r3 | 3, r3 | 3, r3 |
| 4. | **add** r1, r2, r2 | 4, r4 | **add** r1, r2, r2 |
| 5. | 4, r1* | **add** r1, r2, r2 | 4, r1 |
| 6. | | **add** r3, r4, r3 | **neg** r3, r3 |
| 7. | **add** r3, r1*, r3 | **add** r2, r3, r3 | **add** r3, r1, r3 |
| 8. | **add** r2, r3, r3 | | **add** r2, r3, r3 |

Fig. 6. Column one presents a canonical schedule of the nodes in Tree A. The canonical schedule uses three registers, and there is an empty delay slot in cycle six. Column two shows a basic DLS schedule for Tree A. This schedule uses four registers and has no empty delay slot. Column three shows a DLS schedule for Tree B, which includes a unary operator. Only three registers are needed for a delay-free schedule.

first column of the table in Figure 6 shows a three-register canonical-form schedule for tree A. This schedule has an empty delay slot in cycle six. The basic DLS algorithm generates a delay-free schedule for tree A using four registers, one more than the Sethi-Ullman count of the tree. This delay-free schedule is shown in column two. Extending DLS to schedule unary operators (and literals), a delay-free schedule for tree B requires only three registers, one fewer register than the basic DLS schedule for tree A. The DLS schedule for tree B is shown in column three. The unary operator **neg** fills the delay slot between the n cycle five and the **add** in cycle seven in the DLS schedule.

After discussing unary operators and literals, the machine model is extended in Section 9 to include register variables and immediate operands. A register variable is a register that has been allocated in a separate expression-tree. An immediate operand is a small literal value that can be referenced directly without loading its value into a register. In Section 10 the extensions of Sections 8 and 9 are incorporated into the DLS algorithm.

Register spilling within expressions is introduced in Section 11, and an investigation of the behavior of DLS with $Delay > 1$ is discussed in Section 12. In Section 13 a heuristic which is based on DLS is outlined and compared with Goodman and Hsu's scheduler IPS [Goodman and Hsu 1988].

## 7. OPTIMAL ALGORITHM FOR $Delay = 1$

Optimal instruction scheduling and register allocation for an expression-tree when $Delay = 1$ can be done in time proportional to the size of the expression-tree. Our

DLS algorithm is a variation of the Sethi-Ullman algorithm adapted to our machine model.

Both the SU algorithm and the DLS algorithm are driven by minimizing the register needs for evaluating an expression. These needs are denoted as the *minReg* of a node and refer to the minimal number of registers needed for computing the subtree rooted at that node without spilling. The *minReg* value of a node is simply the standard SU number, adapted to our load/store architecture.[4] It is calculated by the following procedure, *label*().

```
procedure label(node : ExprNode)
  if isLeaf(node) then
    node.minReg ← 1;
  else
    label(node.left);
    label(node.right);
    if node.left.minReg = node.right.minReg then
      node.minReg ← node.left.minReg+ 1;
    else
      node.minReg ← MAX(node.left.minReg, node.right.minReg);
    end if
  end if
end procedure
```

### 7.1 Exceptional Cases for $Delay = 1$

When $Delay = 1$, exactly two trees in our model have schedules that must always incur unfilled delay slots: the tree consisting of a single node and the tree consisting of a single operator and two leaf (memory) nodes. It is trivial to verify that these must incur unfilled delay slots and that the register needs for these trees are one and two, respectively.

### 7.2 Algorithm

The DLS algorithm presented in Figure 7 finds an instruction schedule and register assignment that is optimal for a given expression-tree. For all trees with the exception of the two just mentioned, the DLS schedule will have no unfilled delay slots and will use the minimal number of registers for *any* schedule without unfilled delay slots.

The number of registers needed for such a schedule is exactly one more than the minimal number of registers needed to evaluate the expression without any spills (i.e., the SU *minReg* value of the root of the expression).

The DLS algorithm is a simple three-pass algorithm for finding the optimal instruction sequence and register allocation. The procedure *label*() (given earlier) labels the nodes with their SU *minReg* values. Procedure *order*() finds the operation and load orders. *order*() is similar to the original Sethi-Ullman algorithm. *schedule*() then emits the instructions from the Sethi-Ullman ordering in canonical order.

[4]The original SU algorithm was based on a machine model in which binary operations could access their right operands directly from memory. Our model requires all operands to be in registers.

```
// Sethi-Ullman Ordering
procedure order(root : ExprNode; var opSched, loadSched : nodeList)
  if not isLeaf(root) then
    if root.left.minReg < root.right.minReg then
      order(root.right, opSched, loadSched);
      order(root.left, opSched, loadSched);
    else
      order(root.left, opSched, loadSched);
      order(root.right, opSched, loadSched);
    end if
    opSched ← root || opSched;                      // append opSched to root
  else
    loadSched ← root || loadSched;                  // append loadSched to root
  end if
end procedure

// Canonical Ordering
procedure schedule(opSched, loadSched : nodeList; Regs : integer)
  initialLoads : integer ← MIN(Regs, length(loadSched));
  forall i ← 1 to initialLoads do                   // Loads First
    ld ← popHead(loadSched);
    ld.reg ← getReg();
    gen(Load, ld.name, ld.reg);
  end forall
  while not Empty(loadSched) do                     // (Operation,Load) Pairs
    op ← popHead(opSched);
    op.reg ← op.right.reg;
    gen(op.op, op.left.reg, op.right.reg, op.reg);
    ld ← popHead(loadSched);
    ld.reg ← op.left.reg;
    gen(Load, ld.name, ld.reg);
  end while
  while not Empty(opSched) do                       // Remaining Operations
    op ← popHead(opSched);
    op.reg ← op.right.reg;
    gen(op.op, op.left.reg, op.right.reg, op.reg);
    freeReg(op.left.reg);
  end while
end procedure

// Schedule Instructions
procedure generate(root : ExprNode; Delay : integer)
  label(root)                                       // Compute minReg
  opSched ← loadSched ← emptyList();                // Initialize
  order(root, opSched, loadSched);                  // Find Load/Op Order
  schedule(opSched, loadSched, root.minReg+ Delay); // Emit Canonical Order
end procedure
```

Fig. 7. Optimal delayed-load-scheduling (DLS) algorithm.

### 7.3 Optimality Proof

The argument that the DLS algorithm creates an optimal instruction schedule and register allocation follows from two observations: the number of registers needed to avoid unfilled delay slots must be at least *minReg* + 1, and the canonical order generated by the algorithm using *minReg* + 1 registers does not incur unfilled delay slots.

The following theorem was proved in Proebsting and Fischer [1991].

THEOREM 7.3.1. *Let $T$ be an expression-tree with Sethi-Ullman number, minReg, and let Delay* $= 1$. *If only minReg registers are available to schedule $T$, then $T$ will have an unfilled delay slot.*

PROOF. The evaluation cannot take fewer than *minReg* registers. If only *minReg* registers are available, there must be a point at which a just-loaded register must be used in the next instruction, which would result in a load delay. This follows from the fact that only loads can increase the number of registers in use, and thus at some point a load must put *minReg* registers in use. Because only *minReg* registers are available, this load must be followed by an operation on the just-loaded register (if another operation could have been scheduled, it would have been to keep the number of registers in use at a minimum). Therefore, more than *minReg* registers are needed to avoid unfilled delay slots. □

We will refer to the following corollary of Theorem 7.3.1. Let *t.minReg* be the *minReg* value for a subtree $t$ of $T$.

COROLLARY 7.3.2. *Let $T$ be an expression-tree, Delay* $= 1$, *and $t$ be a subtree of $T$. Assume in a Sethi-Ullman ordering of $T$ there are only t.minReg registers available to schedule $t$. Then a load in $t$ has an empty delay slot.*

PROOF. This follows immediately from the proof of Theorem 7.3.1. □

Assuming *minReg* registers are available, there are only two trees in which a load does not follow an operator in a Sethi-Ullman ordering—a single identifier and a binary operator whose two operands are identifiers. Let set $E$ contain all expression-trees except these two. The following theorem identifies loads that will have filled delay slots in a canonical-form schedule of a Sethi-Ullman ordering of a tree in $E$.

THEOREM 7.3.3. *Let $T \in E$, and let Delay* $= 1$. *Assume at least minReg registers are available to schedule $T$. Let $O_1$ be a Sethi-Ullman ordering of $T$, and $O_2$ be the canonical-form schedule based on $O_1$. If there are at least two available registers for a load $L$ in $O_1$, one of which will be assigned to $L$, then $L$'s delay slot is filled in $O_2$.*

PROOF. To produce $O_2$, we slide each load forward in $O_1$ in succession beginning with the first, such that we never overallocate registers. Immediately prior to sliding a load $L$ forward, the same number of registers available to $L$ in $O_1$ are now available to $L$, since only loads ahead of $L$ in $O_1$ have been moved forward.

Let load $L$ have at least two available registers in $O_1$. If a load follows $L$ in $O_1$, then after $L$ is scheduled in $O_2$, the nearest load following $L$ will slide ahead of the operator referencing $L$. This nearest load will be assigned a register that was available to $L$ before $L$ was scheduled. $L$'s delay slot is filled. Next, assume no loads follow $L$ in $O_1$. Since $T \in E$, an operator must precede $L$ in $O_1$. Since at least two registers are available to $L$, $L$ slides ahead of the nearest operator. Again, $L$'s delay slot is filled. □

COROLLARY 7.3.4. *Let $T \in E$. Given minReg + 1 registers, DLS produces a delay-free schedule.*

PROOF. DLS produces a canonical-form schedule from a Sethi-Ullman ordering of $T$. Given *minReg* + 1 registers, each identifier has at least two available registers in a Sethi-Ullman ordering of $T$. By Theorem 7.3.3, DLS generates a delay-free schedule. □

Assume that there are *only minReg* registers available to schedule an expression-tree $T$ and that a load $L$ in $T$ has two available registers in a Sethi-Ullman ordering of $T$. If $T$ is a single identifier or binary operator whose two operands are identifiers, then $L$ will still have a filled delay slot in a canonical-form schedule based on the ordering.

THEOREM 7.3.5. *Let $T$ be an expression-tree, and let Delay = 1. Assume only minReg registers are available to schedule $T$. Let $O_1$ be a Sethi-Ullman ordering of $T$, and $O_2$ be the canonical-form schedule based on $O_1$. If there are at least two available registers for a load $L$ in $O_1$, one of which will be assigned to $L$, then $L$'s delay slot is filled in $O_2$.*

PROOF. If $T \in E$, then by Theorem 7.3.3, a load in $O_1$ that has at least two available registers has its delay slots filled in $O_2$. Assume $T \notin E$. If $T$ is a single load, then *minReg* = 1, and there is only one register available to $T$, which has an unfilled delay slot. If $T$ is a binary operator whose two operands are loads, then *minReg* = 2. The first load scheduled in $O_1$ has two available registers and has its delay slot filled in $O_2$ by the second load. There is one register available for the second load in $O_1$, which has an unfilled delay slot in $O_2$. □

## 8. SCHEDULING LITERALS AND UNARY OPERATORS

Above we presented an algorithm to schedule expression-trees optimally in which *Delay* = 1 and *minReg* + 1 registers are used. Some trees can be scheduled using only *minReg* registers if the nodes of those trees include unary operators or literals.

In this section we extend the machine model to include unary operators and literals, both of which are delay free. We assume a **loadi** instruction is used to move the literal's value to a register. In Section 9 we extend our algorithm to allow operators to reference immediate operands.

When scheduling unary operators and literals, choosing which subtree to schedule first can be based on the number of empty delay slots in each subtree. We need to identify which loads have unfilled delay slots in the final schedule.

Assume a tree is given exactly *minReg* registers. By Theorem 7.3.5, if we have a canonical-form schedule *sched* based on a Sethi-Ullman ordering of the tree, those loads that do not increase the register count to *minReg* in a Sethi-Ullman ordering will have filled delay slots in *sched*. Loads that increase the register count to *minReg* have only one register available. By Corollary 7.3.2, these loads will have empty

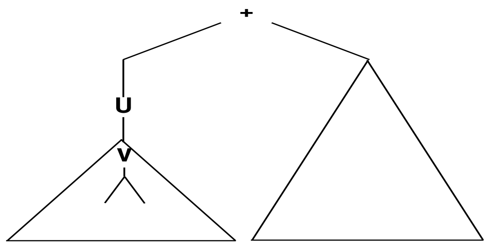



Fig. 8. Unary operators $U$ are the only operators from the left subtree of $+$ that can fill delay slots in the right subtree.

delay slots in *sched*. By introducing delay-free loads and delay-free unary operators, loads that increase the register count to *minReg* may be delay free, or have their delay slots filled by unary operators. We now sometimes find a delay-free schedule that utilizes only *minReg* registers.

### 8.1 Filling Delay Slots with Unary Operators

Assume first that $L.minReg = R.minReg$ or $L.minReg = R.minReg + 1$, and let scheduling begin with $L$. In both cases $T.minReg = R.minReg + 1$. Since more than $L.minReg$ registers are available to $L$, each load in $L$ has at least two registers available. Delay slots in $L$ will be filled. Since there are only $R.minReg$ registers available to schedule $R$, unary operators from $L$ will be used to fill delay slots in $R$.

The left subtree in general may have a chain $U$ of $n$ consecutive unary operators, $n \geq 0$, descending from its root and extending down to a binary operator or a leaf, $v$ (see Figure 8). The only unary operators in $L$ available to fill delay slots in $R$ are those in $U$. After $v$ is scheduled there will be $R.minReg$ registers available to schedule $R$. By Corollary 7.3.2 and Theorem 7.3.3, slots must be filled in $R$ at points where $R.minReg$ registers are in use. These delay slots can be filled by unary operators $U$ in the left subtree, as $v$ will have been scheduled (with its value already computed into some register, which can be reused by all the unary operators in $U$).

If $L.minReg > R.minReg + 1$, then once $L$ is scheduled, there will be at least $R.minReg + 1$ registers available to schedule $R$. Scheduling $R$ after $L$ is therefore sufficient to fill all the delay slots of $R$ without use of $L$'s unary operators.

### 8.2 Finding a Schedule with Unary Operators and Literals

The algorithm *UnaryCalc* in Figure 9 calculates the number of unary operators needed to fill the remaining load delay slots for each subtree of $T$. $T$ may contain literals, which are subtrees of $T$ having no empty load delay slots. To allocate only *minReg* registers, *UnaryCalc* applies the Sethi-Ullman algorithm to select the subtree to schedule first. If the Sethi-Ullman algorithm does not have a preference, *UnaryCalc* selects the ordering that minimizes the number of unfilled delay slots.

$T.need$ represents the number of extra unary operators needed to fill $T$'s delay slots. If $T.need$ is 0, then $T.minReg$ registers are sufficient for a delay-free schedule. $T.have$ is the number of unary operators chained at the root of $T$ that may fill $T$'s sibling's delay slots.

```
procedure UnaryCalc(Tree T)
  if isLeaf(T) then
    if isDelayedLoad(T) then // Delayed-Loads have delay slot
      T.need← 1; T.have ← 0;
    else // Delay-free Load
      T.need← 0; T.have ← 0;
    end if
  elsif isUnary(T) then // Increment number of unary operators available
    Tree C← T.sched_unary; // Child subtree
    UnaryCalc(C);
    T.have ← C.have + 1; T.need ← C.need;
  else                         // T's root node is a binary operator
    Tree L ← T.sched_left , R ← T.sched_right; // Left and right subtrees
    UnaryCalc(L);
    UnaryCalc(R);
    T.have← 0;  // Binary operators cannot contribute unary operators
    if L.minReg = R.minReg then    // T.minReg ← L.minReg + 1
      integer lcount, rcount;   // Unary operators needed by T if left or right
                                // subtrees are scheduled first, respectively
      lcount ← MAX(0,R.need − L.have); // Determine which ordering has
      rcount ← MAX(0,L.need − R.have); // fewer unfilled delay slots
      if lcount ≤ rcount then
          // Scheduling left subtree first allows for fewer unfilled delay slots
        T.need ← lcount; T.sched_first ← left;
      else
        T.need ← rcount; T.sched_first ← right;
      end if
    elsif L.minReg > R.minReg then
        // Left subtree scheduled before right; T.minReg ← L.minReg
      if L.minReg = R.minReg + 1 then
        T.need ← L.need + MAX(0, R.need − L.have);
      else // right subtree has no unfilled delay slots
        T.need ← L.need;
      end if
      T.sched_first ← left;
    else L.minReg < R.minReg
      // symmetric to R.minReg > L.minReg case.
    end if
  end if
end procedure
```

Fig. 9. Determine the number of unary operators needed for each subtree for a delay-free schedule with *minReg* registers.

THEOREM 8.2.1. *For a tree $T$, $T.need$ represents the number of unary operators $T$ needs to be delay free using exactly $T.minReg$ registers.*

PROOF.

*Base Step* ($height = 1$). Let $T$ be a tree of height one (a singleton node). Then $T$ is either a delayed load or a delay-free load. In the former case, $T$ needs one unary operator to fill the load's delay slot, so $T.need = 1$. In the latter case, there is no delay slot to fill, so $T.need = 0$. In both cases no unary operators are made available, so $T.have = 0$.

*Inductive Step* ($height > 1$).

*Case* 1. The operator at the root node of $T$ is unary. This unary operator cannot be used as one of the unary operators its subtree might need. However, this unary operator may be used to fill a delay slot in another subtree. So, $T.have = C.have + 1$, where $C$ is the subtree of $T$. Since the unary operator is delay free, $T.need = C.need$.

*Case* 2. The root of $T$ is a binary operator. Let $L$ be $T$'s left operand and $R$ be $T$'s right operand.

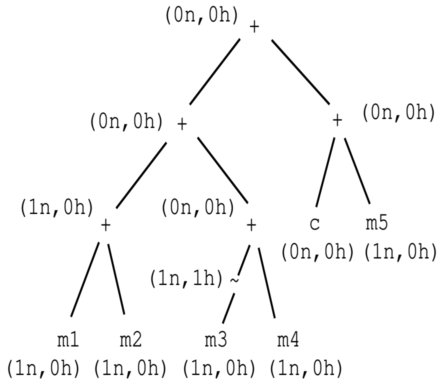
Unary operators below the root of $T$ cannot fill the load delay slots of subtrees above the root of $T$, so $T.have = 0$.

*Case* 2a. $L.minReg = R.minReg$ ($T.minReg = L.minReg + 1$). $L.minReg + 1$ registers are available for the first subtree scheduled. When the first subtree is scheduled, its result before (and after) scheduling the unary operators at the root of the subtree will be in a single register, leaving $L.minReg$ registers to schedule the second subtree. Since $L.minReg + 1$ registers are available for the first subtree, all of its delay slots will be filled. If $L$ is scheduled first, any unary operators descending from the root of $L$ may fill delay slots of $R$. In this case $T.need = MAX(0, R.need - \mathrm{L}.have)$. If $R$ is scheduled first, $T.need = MAX(0, L.need - R.have)$. If more unary operators are provided than are needed, then $T.need = 0$. We choose the ordering that minimizes $T.need$.

*Case* 2b. $L.minReg > R.minReg + 1$ ($T.minReg = L.minReg$). Since the *minReg* count of $L$ is greater than $R$'s, $L$ is scheduled first. After $L$ is scheduled, there will be at least $R.minReg + 1$ registers to schedule $R$. $R$ will have no empty delay slots. Therefore, the number of delay slots needed to be filled by $T$ is the number of delay slots needed by $L$. So, $T.need = L.need$.

*Case* 2c. $L.minReg = R.minReg + 1$ ($T.minReg = L.minReg$). Again $L$ has a larger *minReg* count and is scheduled before $R$. Before scheduling any unary operators at the root of $L$, $L$ will compute its result in a register. So $R.minReg$ registers, the minimum number of registers necessary for $R$, will be available to schedule $R$. The number of $T$'s delay slots to be filled will be those of $L$ plus the number of delay slots unfilled in $R$ after using $L$'s available unary operators to fill delay slots. $T.need = L.need + MAX(0, R.need - L.have)$.

*Case* 2d. $R.minReg > L.minReg + 1$ and $R.minReg = L.minReg + 1$. Symmetric to cases 2b and 2c. □




| *Cycle#* | DLS Schedule |
|---|---|
| 1. | 1, r1 |
| 2. | 2, r2 |
| 3. | 3, r3 |
| 4. | **add** r1, r2, r1 |
| 5. | 4, r2 |
| 6. | **neg** r3, r3 |
| 7. | **add** r3, r2, r3 |
| 8. | 5, r2 |
| 9. | **add** r1, r3, r1 |
| 10. | **loadi** c, r3 |
| 11. | **add** r2, r3, r2 |
| 12. | **add** r1, r2, r1 |

Fig. 10. Minimizing register usage in the presence of unary operators and delay-free loads.

### 8.3 Example

Figure 10 is an example of the *UnaryCalc* algorithm. The symbol $\sim$ is unary minus; m1 through m5 are addresses; and c is a constant. Each node is labeled with a pair indicating the number of unary operators needed and available for a delay-free schedule using $T.minReg$ registers. Leaves m1 through m5 will require delayed loads. The constant leaf, c, requires a nondelayed load. At the $\sim$ node above m3, one unary operator becomes available.

The variables *lcount* and *rcount* in routine *UnaryCalc* (Figure 9) represent the number of delay slots needed to be filled if the left or right subtree, respectively, is scheduled first. At the the parent of $\sim$, the left and right subtrees have equal *minReg* counts. Here we have $lcount = 0$ and $rcount = 1$; the left subtree is scheduled before the right. Moving up another level in the tree, we find the left subtree needs one unary operator. Again, $lcount = 0$ and $rcount = 1$; the left subtree is scheduled first since an additional register is available to eliminate its unfilled delay slot. Moving to the root of the tree, neither operand needs unary operators to fill delay slots, so the entire tree may be scheduled with *minReg* number of registers ($= 3$).

## 9. MAPPING EXPRESSIONS TO SINGLE NODES

In this section, we expand the machine model to include register variables and immediate operands. A register variable is a register that has been defined in a prior expression. We assume that a register variable does not free a register after it is referenced. An immediate operand is a small literal value that can be referenced directly without moving its value to a register; a **loadi** instruction can be avoided.

Register variables and immediate operands are treated identically. Operators with register variables or immediate operands are scheduled either as nondelayed loads or unary operators. A binary operator with one register variable operand may be treated as a unary operator. The binary operator can reuse the register belonging to its nonregister variable operand and may potentially fill a load's delay slot. A unary or binary operator with only register variables is equivalent to a nondelayed load—the operator may fill a delay slot and requires one register.

```
procedure MapNodes(Tree T)

  if isLeaf(T) then
    if isDelayedLoad(T) then T.sched_op ← DelayedLoad;
    elsif isNonDelayedLoad(T) then T.sched_op ← NonDelayedLoad;
    else T.sched_op ← RegisterVariable;    // node is a register variable
    end if                                  // or small literal value

  elsif isUnary(T) then
    Tree C ← T.sched_unary;

    MapNodes(C);
    if isRegisterVariable(C) then T.sched_op ← NonDelayedLoad;
    else T.sched_op ← Unary;
    end if
  else
    Tree L ← T.sched_left; Tree R ← T.sched_right;

    MapNodes(L); MapNodes(R);
    if isRegisterVariable(L) ⋀ isRegisterVariable(R) then
      T.sched_op ← NonDelayedLoad;
    elsif isRegisterVariable(R) then
      T.sched_op ← Unary; T.sched_unary ← L;
    elsif isRegisterVariable(L) then
      T.sched_op ← Unary; T.sched_unary ← R;
    else T.sched_op ← Binary;
    end if
  end if
end procedure
```

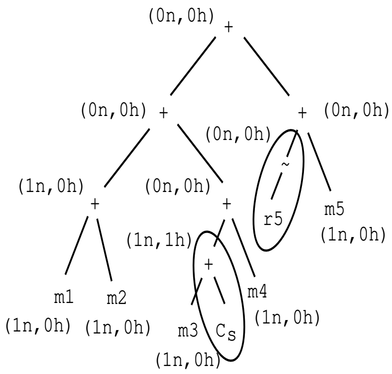
Fig. 11. *MapNodes* schedules nodes that reference register variables or immediate operands as nondelayed loads or unary operators.

Algorithm *MapNodes* in Figure 11 marks which operators should be treated as nondelayed loads or unary operators by setting the node's *sched_op* field. On some architectures immediate operands are only allowed as the rightmost operand of an instruction. *MapNodes* can readily be modified to treat only a small literal value occurring as the rightmost operand of a node as an immediate operand.

In Figure 12 the unary operator and constant of Figure 10 are replaced, respectively, by a binary operator with an identifier and small constant ($c_s$) as operands and a unary operator with a register variable operand. Each subtree is scheduled in the same manner as the subtree it replaces. The new schedule substitutes instructions 6 and 10 of Figure 10 with these corresponding instructions. This schedule is in extended canonical form, which is described in Section 10.3. Single lines separate the subsequences in extended canonical form.

## 10. THE FINAL SCHEDULE

After running *MapNodes* and *UnaryCalc*, *DLS* produces the final schedule for $T$. This version of *DLS* returns a delay-free schedule, if one exists, using the minimum




| *Cycle#* | DLS Schedule |
|---|---|
| 1. | 1, r1 |
| 2. | 2, r2 |
| 3. | 3, r3 |
| 4. | **add** r1, r2, r1 |
| 5. | 4, r2 |
| 6. | **add** r3, $c_s$, r3 |
| 7. | **add** r3, r2, r3 |
| 8. | 5, r2 |
| 9. | **add** r1, r3, r1 |
| 10. | **neg** r5, r3 |
| 11. | **add** r2, r3, r2 |
| 12. | **add** r1, r2, r1 |

Fig. 12. Minimizing register usage in the presence of register variables and immediate operands.

```
function DLS(Tree T): nodeList

   nodeList loads ← NULL;        // list of loads of tree T
   nodeList ops ← NULL;          // list of operators of tree T
   nodeList sched ← NULL;        // schedule in extended canonical form
   nodeList final_sched ← NULL;  // schedule with unary operators
                                 // filling delay slots

   MapNodes(T);
   label(T);
   UnaryCalc(T);
   OrderNodes(T, loads, ops, NULL);
   if T.need = 0 then
      // A delay-free schedule exists with minReg registers
      sched ← ExtendedCanonicalForm(loads, ops, T.minReg);
      final_sched ← ScheduleUnarys(sched);
   else
      // If a delay-free schedule exists, minReg + 1 registers are needed
      final_sched ← ExtendedCanonicalForm(loads, ops, T.minReg + 1);
   end if
   return final_sched;
end function
```

Fig. 13. Algorithm to schedule trees.

number of registers and, therefore, subsumes the version of *DLS* presented in Section 7.2. In the following sections we classify instructions in the manner they are scheduled. For example, we will refer to a binary operator whose two operands are register variables as a delay-free load.

### 10.1 *DLS* Algorithm

Algorithm DLS is shown in Figure 13. Initially, DLS calls *MapNodes*, *label*, and *UnaryCalc*. *label* computes *minReg* for each node of a tree. *UnaryCalc* determines

```
procedure OrderNodes(Tree T, nodeList loads, nodeList ops, node parent)

  if isLeaf(T) then
      // An operator with only register variables is considered a leaf
    loads ← loads || T;
  elsif isUnary(T) then
    OrderNodes(T.sched_unary, loads, ops, parent);
    T.parent ← parent; ops ← ops || T;
  else
    Tree L ← T.sched_left; Tree R ← T.sched_right; // left and right subtrees
    if T.sched_first = left then
      OrderNodes(L, loads, ops, T); OrderNodes(R, loads, ops, T);
    else
      OrderNodes(R, loads, ops, T); OrderNodes(L, loads, ops, T);
    end if
    ops← ops || T;
  end if
end function
```

Fig. 14. Create lists of loads and operators.

(1) the number of unary operators the tree needs for a delay-free schedule with *minReg* registers and (2) an ordering to schedule the subtrees. DLS calls *OrderNodes* to divide $T$'s nodes into two lists: a list of the tree's loads (*loads*) and a list of operators (*ops*). Instructions equivalent to loads because of register variables or immediate operands are included in list *loads*. To assist a later phase in filling delay slots with unary operators, DLS passes a third argument, initially **NULL**, that represents the parent of a chain of unary operators.

## 10.2 *OrderNodes*

Function *OrderNodes* is shown in Figure 14. *OrderNodes* walks the tree in the order determined by *UnaryCalc*, adding loads to list *loads*, and operators to list *ops*. The root of a chain of unary operators is contained in *parent*. *OrderNodes* assigns *parent* to each node in the sequence to assist a later phase in filling empty delay slots with unary operators.

Figure 15 illustrates *OrderNodes*'s operation. Node $-$ acts as a unary operator since one operand is a register variable. The other nodes and the labels beside each node are described in the example of Section 8.3. A delay-free schedule exists for this tree with $minReg = 3$ registers. The lists of operators and loads computed by *OrderNodes* is shown below the tree in Figure 15.

After calling *OrderNodes*, DLS examines the value of $T.need$. If $T.need = 0$, then a delay-free schedule exists with *minReg* registers. Otherwise, if a delay-free schedule exists, $minReg + 1$ registers are needed. In both cases, DLS produces a delay-free schedule, if one exists.

Routine *ExtendedCanonicalForm* puts the lists of loads and operators into an extended canonical form schedule, as described in the following section. The com-

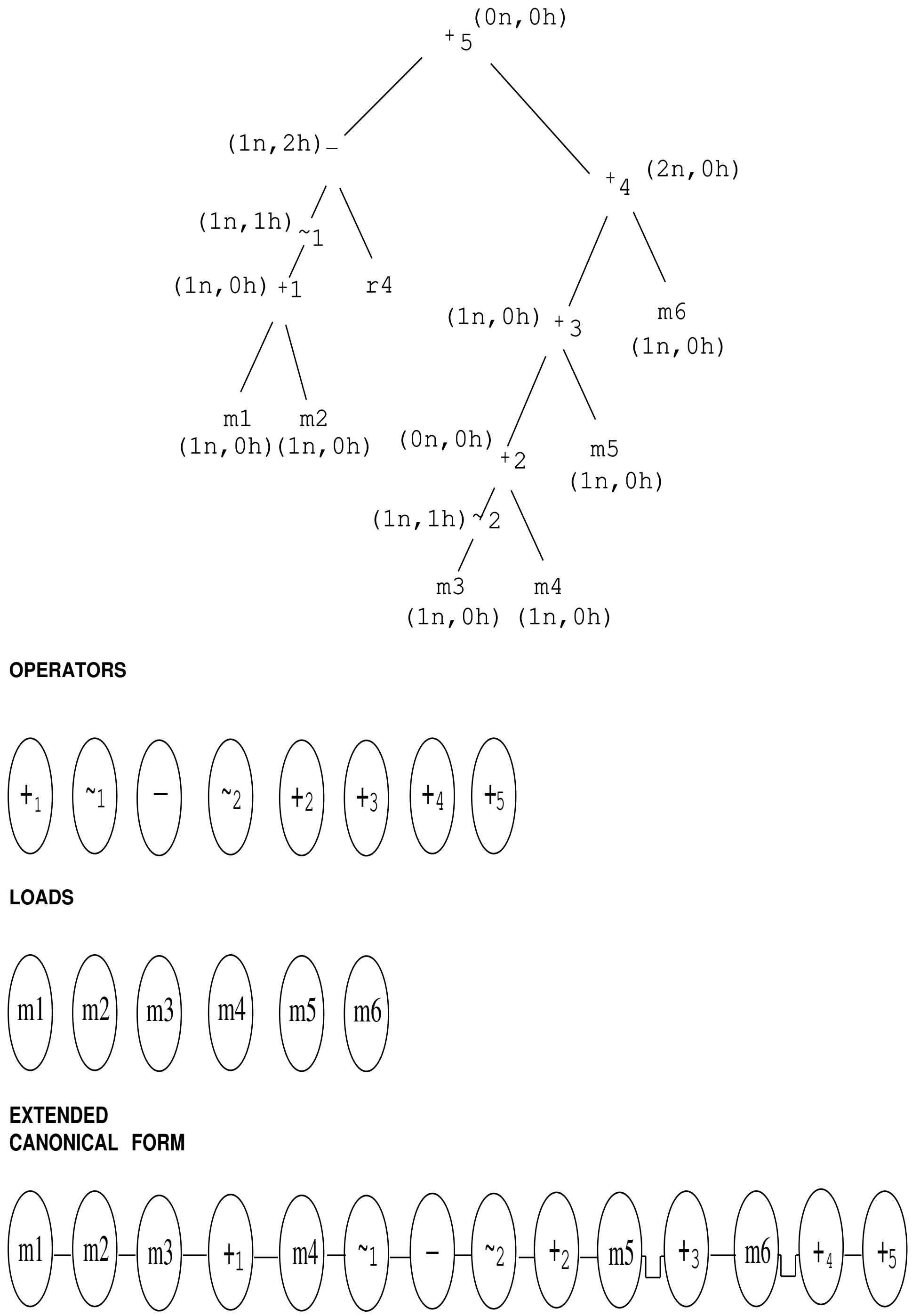



Fig. 15. Example tree, list of tree's loads and operators returned by routine *OrderNodes*, and an extended canonical-form schedule of the tree.

puted schedule is delay free when $minReg + 1$ registers are required. If only $minReg$ registers are needed, *ScheduleUnarys* generates the final schedule.

## 10.3 Extended Canonical Form

DLS produces schedules in extended canonical form. Whereas canonical form (as defined in Section 5.1) includes only loads and binary operators, extended canonical form includes unary operators too. Extended canonical form is comprised of four subsequences. The first subsequence consists of $r$ loads, where $r$ is the number of available registers. As in canonical form, loads are moved as early as possible in the schedule, such that the number of available registers is not exceeded. The second subsequence consists of triples of the form

$$< unary\ operator^n, binary\ operator, load >,$$

where $n$ is a number of adjacent unary operators ($n \geq 0$). Letting *binops* be the number of binary operators in the tree, there are $binops + 1 - r$ triples. Extended canonical form's third subsequence includes $r - 1$ pairs of the form

$$< unary\ operator^n, binary\ operator >,$$

where again $n$ is a number of unary operators ($n \geq 0$). The final subsequence consists of zero or more unary operators. This final subsequence contains those unary operators appearing at the root of the tree (and hence evaluated last) that extends down to a binary operator or leaf.

The schedule in Figure 12 is in extended canonical form. Recall that we treat the binary operator in cycle 6 as a unary operator and the unary operator in cycle 10 as a nondelayed load instruction because both operators reference register variables or immediate operands. Each subsequence is divided by a single line. Since there are three registers available, the first subsequence, cycles 1–3, is filled by load instructions. The second subsequence consists of three triples, cycles 4–5, cycles 6–8, and cycles 9–10. Only the second triple, cycles 6–8, contains a unary operator. The two binary operators at cycles 11 and 12 form the third subsequence. Since the root of the tree is a binary operator, the fourth subsequence of unary operators is empty.

The *ExtendedCanonicalForm* algorithm (Figure 16) is passed ***loads***, a list of the tree's loads, ***ops***, a list of the tree's operators, and $r$, the number of available registers. The algorithm returns a schedule in extended canonical form. First the initial subsequence of loads is scheduled. Following these loads is the second subsequence—triples of unary operators, one binary operator, and one load. When all loads have been scheduled, the remaining operators are scheduled.

The unary operators have not yet filled all delay slots; in Figure 15, the delay slots between m5 and $+_3$ and m6 and $+_4$ are still unfilled as indicated by the notches between these nodes. Function *ScheduleUnarys* fills them by sliding unary operators to the right.

## 10.4 *ScheduleUnarys*

When scheduling unarys, we view a list of instructions as beads on a string. As chains of unary operators are encountered they may be pushed along the string. We allow unary operator chains to slide past loads and binary operators (moving from

```
function ExtendedCanonicalForm(nodeList loads, nodeList ops, integer r):nodeList

  nodeList sched← NULL;        // final schedule
  integer initialLoads ← MIN(r,length(loads));
          // number of loads in first subsequence

  // schedule the first subsequence
  for i← 1 to initialLoads do
    sched ← sched || popHead(loads);
  end for

  // schedule the second subsequence
  while not Empty(loads) do
    while isUnary(Head(ops)) do
      sched ← sched || popHead(ops);
    end while
    sched ← sched || popHead(ops);
    sched ← sched || popHead(loads);
  end while

  // append third and fourth subsequences and return final schedule
  return sched || ops;
end function
```

Fig. 16. Produce a schedule in extended canonical form.

one subtree to another). Notches in the string model unfilled delay slots between instructions. Notches are points where a unary operator can be usefully scheduled.

A unary operator must always be scheduled within the range of nodes in a tree beginning at its child and ending at its parent. Since routine *ExtendedCanonicalForm* does not change the relative ordering among the operators, we check for an unfilled delay slot in this range by sliding each chain of unary operators to the right in the schedule.

We must decide which available chain will supply a unary operator to fill a given delay slot. The parent of the most-recent chain encountered is included within the ranges of the other chains (except the previous one if they have the same parent). Therefore, the range of the most-recent chain cannot extend beyond the ranges of the others, and so we schedule unary operators from the most-recent chain first. Among this chain's unary operators, we choose the earliest unscheduled one to avoid violating dependencies between instructions.

Figure 17 shows the extended canonical-form schedule from Figure 15. This example also illustrates sliding unary operators through the schedule. The tree has two chains of unary operators: $\sim_1$ and $-$ form the first chain, and $\sim_2$ forms the second. The first unary chain is shown below schedule (1) and the second chain above schedule (1). The range of the second chain ends before the first. Notches represent empty delay slots. Schedule (1) includes delay slots that must be filled by unary operators. Since unary operators were already filling the delay slots between m4 and $+_2$, adding these unary operators to the two chains results in an additional

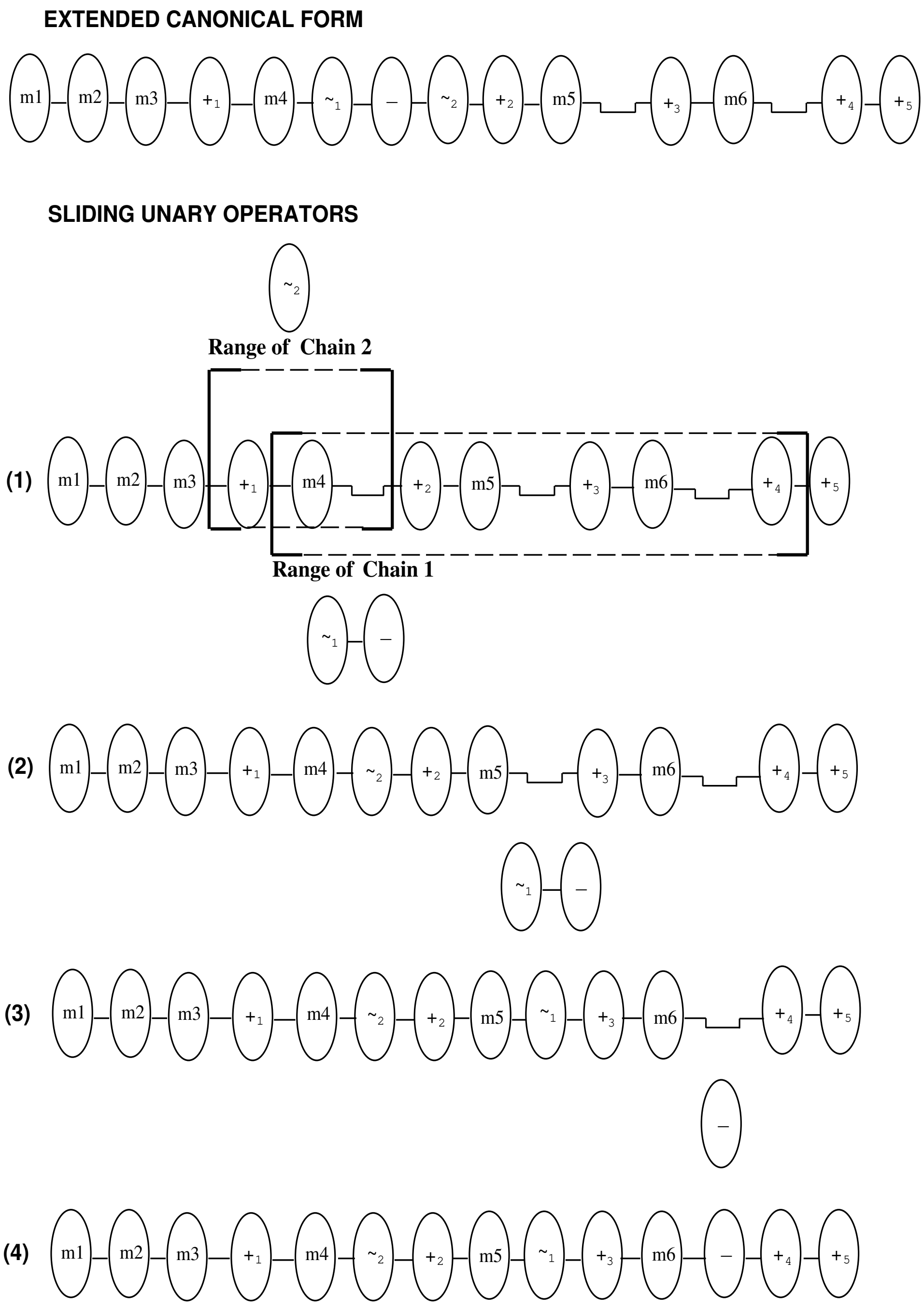



Fig. 17. Filling delay slots with unary operators.

```
1  function ScheduleUnarys(nodeList sched): nodeList
2
3     nodeList final_sched ← NULL;     // schedule with all delay slots filled
4     nodeList unylist ← NULL;         // list of unary operators to fill delay slots
5     node curr_entry ← Head(sched);   // current entry examined in sched
6     node next_entry;                 // next entry to be examined
7
8     while curr_entry ≠ NULL do
9        next_entry ← curr_entry.next;
10       // Check for unfilled delay slot
11       if (final_sched≠ NULL) ⋀ EmptyDelaySlot(last(final_sched),curr_entry) then
12          // Schedule unary from unylist to fill delay slot
13          final_sched ← final_sched || popHead(unylist);
14       end if
15
16       if isUnary(curr_entry) then
17          // Remove all unary operators in a single chain
18          unylist ← RemoveUnarys(curr_entry) || unylist;
19       else   // Schedule unary operators before their parent in the tree
20          while (unylist≠ NULL) ⋀isParent(curr_entry,Head(unylist)) do
21             final_sched ← final_sched || popHead(unylist);
22          end while
23          final_sched ← final_sched || curr_entry;
24          curr_entry ← next_entry;
25       end if
26    end while
27    final_sched← final_sched|| unylist;
28    return final_sched;
29 end function
```

Fig. 18. Fill empty delay slots with unary operators.

empty delay slot in schedule (1). We fill each delay slot with the first unary operator in the chain whose range ends the earliest. Thus, $\sim_2$ fills the first slot as shown in schedule (2). Schedule (4) is the final delay-free schedule.

Figure 18 presents *ScheduleUnarys*, which uses unary operators to fill empty delay slots of *sched*. *ScheduleUnarys* computes *final_sched*: the delay-free schedule. Variable *curr_entry* points to the current entry in *sched*. Empty delay slots are filled by unary operators in list *unylist*.

*ScheduleUnarys* checks for an empty delay slot at line 11. If routine *EmptyDelaySlot* determines that the last entry in *final_sched* is a delayed load, and *curr_entry* references that load, then the first unary operator from *unylist* is removed to fill the empty delay slot. Placing chains at the beginning of *unylist* allows the first unary operator in the last chain to be easily removed.

If *curr_entry* is a unary operator (line 16), *ScheduleUnarys* calls *RemoveUnarys* to remove a chain of unary operators. Any unary operator after the first in a chain must be the parent of the previous one in the chain. *RemoveUnarys* checks for this condition to isolate the unary operator chain.

| *Cycle#* | DLS Schedule |
|---|---|
| 1. | 1, r1 |
| 2. | 2, r2 |
| 3. | 3, r3 |
| 4. | **add** r1, r2, r1 |
| 5. | 4, r2 |
| 6. | **neg** r3, r3 |
| 7. | **add** r3, r2, r3 |
| 8. | 5, r2 |
| 9. | **neg** r1, r1 |
| 10. | **add** r3, r2, r3 |
| 11. | 6, r2 |
| 12. | **sub** r1, r4, r1 |
| 13. | **add** r3, r2, r3 |
| 14. | **add** r1, r3, r1 |

Fig. 19. Delay-free schedule of our example tree.

If *curr_entry* is not a unary operator, and *curr_entry* is the parent of unary operator chains in *unylist* (lines 20 to 22), then all the unary operators from these chains must be scheduled immediately. Routine *isParent* indicates if *curr_entry* is the parent of the unary chain that includes the first unary operator on *unylist*. If the first unary operator's *parent* field, assigned in routine *OrderNodes*, has the value *curr_entry*, then *isParent* returns true.

*ScheduleUnarys* adds *curr_entry* to the final schedule (line 23). After registers are assigned, the delay-free schedule for our example tree is shown in Figure 19.

### 10.5 Proof of Correctness

We must show that all delay slots are filled if a delay-free schedule exists given either *minReg* or *minReg* + 1 registers. Theorem 7.3.3 holds for a schedule in extended canonical form, since both canonical form and extended canonical form move loads ahead of operators, as loads are assigned the earliest available registers. Assume a delay-free schedule exists for a tree given *minReg* + 1 registers. With *minReg* + 1 registers, each load has at least two registers available in a Sethi-Ullman evaluation. This schedule is delay free, applying the argument in the proof of Theorem 7.3.3 to this schedule's loads.

Assume a delay-free schedule exists given *minReg* registers. Since the proof of Theorem 7.3.3 holds for a schedule in extended canonical form, loads with two available registers have their delay slots filled. Since *UnaryCalc* and *OrderNodes* determine the relative ordering of the loads in a delay-free schedule, the delay-free loads are scheduled correctly. As we know by Theorem 8.2.1 that a positioning of the unary operators yields a delay-free schedule, all that remains is scheduling the unary operators to fill all remaining empty delay slots. Each unary operator must be scheduled after its child and before its parent. Since routine *ExtendedCanonicalForm* maintains the relative ordering among operators, we slide each chain of unary operators to the right, checking for unfilled delay slots.

Among the unary operators available to fill a delay slot, *ScheduleUnarys* chooses one from the last chain added to *unylist*. These unary operators have the shortest range in which they can be scheduled. Among the unary operators in the last chain, *ScheduleUnarys* chooses the first unscheduled one to avoid violating dependencies between them. Following this procedure, *ScheduleUnarys* returns a delay-free schedule.

### 10.6 Algorithm Complexity

The complexity of DLS is $O(n)$, where $n$ is the number of nodes in the tree. Routines *UnaryCalc*, *MapNodes*, and *OrderNodes* perform a constant number of operations at each node visited. *ExtendedCanonicalForm* is also linear since removing a node from a list and appending it to a schedule takes constant time. *ScheduleUnarys* is linear since we add and remove an instruction from the unary list in constant time for each scheduled node. Adding each node to the final schedule is also in constant time. DLS, which is composed of these routines, is thus linear in the number of nodes scheduled.

## 11. SPILLING

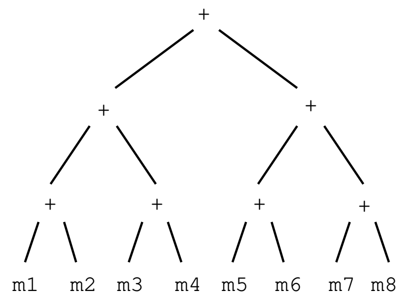
For the DLS algorithm to be practical, it must also be able to produce good schedules when too few registers are available for allocation. Suppose, for example, that exactly *minReg* registers are available, but they are insufficient for a delay-free schedule. Should the algorithm introduce spill code so that subtrees may be computed without unfilled delay slots? If so, where should the spills be introduced? If not, will the computation incur excessively many unfilled delay slots? The best solution depends on the form of the expression-tree.

The tree in Figure 20 can be best handled by allowing the canonical execution order (with $minReg = 4$ registers) and incurring a single load delay of one cycle. Having no unfilled delay slots would have required a spill and hence cost two extra instructions: a store and load. The tree in Figure 21 will incur three delay cycles, but would incur only the cost of a single load and store if a spill were introduced.

Ordering code so that spill/delay costs will be minimized requires extending the DLS algorithm. Given an expression with two subtrees that have identical *minReg* values, the algorithm orders the subtrees such that the subtree having fewer empty delay slots is scheduled last. The number of delay slots corresponding to a subtree $T$ is represented by $T.need$ in routine *UnaryCalc* (Section 8.2).

The decision to spill a node is made when calculating the *minReg* and *need* values. A node is spilled if its *minReg* value is equal to the number of available registers and if its *need* value is greater than two (the cost of a store and load). If a node has a *minReg* value greater than the number of available registers, then its operand with the greater number of unfilled delay slots should be spilled. Spilling information is calculated bottom-up in the tree while calculating *minReg* and *need*. The algorithm avoids spilling until absolutely necessary *or* until it is advantageous.

```
if (node.minReg = Registers ∧ node.need > 2) ∨ node.minReg > Registers then
  if node.left.need > node.right.need then
    Spill node.left; Make node.left a Leaf temporary;
    Set node.left.need ← 1; Set node.left.minReg ← 1;
    Set node.need ← node.right.need; Set node.minReg ← node.right.minReg;
  else // Spill node.right, etc...
  end if
end if
```



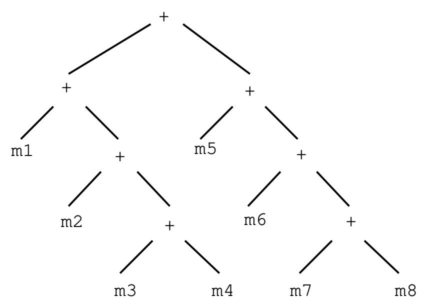

| # | No Spill—Unfilled Delay Slots | Spill—Filled Delay Slots |
|---|---|---|
| 1. | 1, r1 | 1, r1 |
| 2. | 2, r2 | 2, r2 |
| 3. | 3, r3 | 3, r3 |
| 4. | 4, r4 | 4, r4 |
| 5. | **add** r1, r2, r2 | **add** r1, r2, r2 |
| 6. | 5, r1 | 5, r1 |
| 7. | **add** r3, r4, r4 | **add** r3, r4, r4 |
| 8. | 6, r3 | 6, r3 |
| 9. | **add** r2, r4, r4 | **add** r2, r4, r4 |
| 10. | 7, r2 | 7, r2 |
| 11. | **add** r1, r3, r3 | **store** r4, TEMP |
| 12. | 8, r1 | 8, r4 |
| 13. | | **add** r1, r3, r3 |
| 14. | **add** r2, r1*, r1 | EMP, r1 |
| 15. | **add** r3, r1, r1 | **add** r2, r4, r4 |
| 16. | **add** r4, r1, r1 | **add** r3, r4, r4 |
| 17. | | **add** r1, r4, r4 |

Fig. 20. Spilling may be more expensive than delay slots—example with four registers.

Loads introduced by spills will not have unfilled delay slots because they will occur only at a node whose sibling has a *minReg* value of at least $Registers-1$. This ensures that the load will be part of a tree whose root has *minReg* of at least $Registers-1$. Since this new leaf (spill) node has a *minReg* of 1, it cannot increase the *minReg* or number of unfilled delay slots of the entire tree. The cost of the spill is restricted to the cost of the store/load and with the unfilled delay slots associated with evaluating the subtree below the spilled node (which cannot be greater than two).

## 12. BEHAVIOR FOR $Delay > 1$

In this section we study the behavior of DLS for machines where the load delay exceeds one cycle. For example, the HP PA-8000 processor has a load delay of two cycles [Gwennap 1994]. For this study we restrict our model to our original machine model, which included only binary operations and delayed loads.




| # | No Spill—Unfilled Delay Slots | Spill—Filled Delay Slots |
|---|---|---|
| 1. | 3, r1 | 3, r1 |
| 2. | 4, r2 | 4, r2 |
| 3. | 2, r3 | 2, r3 |
| 4. | **add** r1, r2, r2 | **add** r1, r2, r2 |
| 5. | 1, r1 | 1, r1 |
| 6. | **add** r3, r2, r2 | **add** r3, r2, r2 |
| 7. | 7, r3 | 7, r3 |
| 8. | **add** r1, r2, r2 | **add** r1, r2, r2 |
| 9. | 8, r1 | 8, r1 |
| 10. | | **store** r2, TEMP |
| 11. | **add** r3, r1*, r1 | 6, r2 |
| 12. | 6, r3 | **add** r3, r1, r1 |
| 13. | | 5, r3 |
| 14. | **add** r1, r3*, r3 | **add** r2, r1, r1 |
| 15. | 5, r1 | EMP, r2 |
| 16. | | **add** r3, r1, r1 |
| 17. | **add** r3, r1*, r1 | **add** r2, r1, r1 |
| 18. | **add** r2, r1, r1 | |

Fig. 21. Spilling may save cycles—example with three registers.

The optimality results for *Delay* $= 1$ do not directly extend to greater *Delay* values. DLS is, however, an excellent heuristic for larger *Delay*'s, retaining its simplicity and linear running time. As a heuristic for instruction scheduling with *Delay* $> 1$, DLS may require more than *minReg* $+ 1$ registers to achieve a delay-free schedule in canonical order. If a delay-free schedule exists for a given expression, it can be found, however, by using *minReg* + *Delay* registers with the DLS canonical form. The same argument made for the optimality of the case *Delay* $= 1$ shows that the number of registers needed for a delay-free schedule will never be greater than *minReg* + *Delay*. Not all expressions require *minReg* + *Delay* registers for a delay-free schedule—they may have delay-free schedules requiring fewer registers. For this reason, a heuristic approximation to the DLS algorithm for *Delay* $> 1$ is to use a DLS-generated canonical order with *minReg* + *Delay* registers. This heuristic retains the optimal scheduling results of the *minReg* $+ 1$ case, but may, in a few cases, overallocate registers. In Section 12.2, we give the lower bounds on the fewest possible registers needed for a delay-free schedule when *Delay* $> 1$.

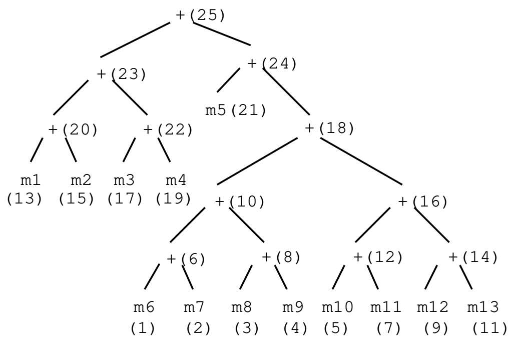



Fig. 22. Noncontiguous optimal evaluation for *Delay* = 2.

## 12.1 Noncontiguous Operand Ordering

The Sethi-Ullman algorithm generates code that is *contiguous*. That is, instructions generated for one subtree do not mix with the instructions for a sibling subtree. The DLS algorithm does not possess this property because the loads from one subtree may be mixed with the operations from another. The algorithm does, however, produce schedules that exhibit some contiguity: the loads taken alone and the binary operations taken alone do have contiguous orders. It is precisely this property that allows a "divide-and-conquer" approach that treats each subtree separately.

It is not always possible to generate code with this contiguous operation/load property and still have the code be optimal with respect to unfilled delay slots and register usage if *Delay* is greater than 1. Figure 22 is the *smallest* example of a tree that does not have an optimal schedule in which the operations/loads are ordered contiguously for *Delay* = 2—the tree is labeled with the optimal evaluation order.

Lacking the contiguous property for optimal results, it is unlikely that a linear-time algorithm exists for optimally scheduling trees with $Delay > 1$. Whether the optimal algorithm is polynomial time or exponential time, it will be much more expensive to run than DLS (and in practice not all that more effective).

## 12.2 Register Bounds

The fact that $minReg + Delay$ registers and a canonical ordering will always produce a delay-free schedule (when one exists) gives an upper bound on the number of registers necessary to find such an evaluation. The optimality proof in Section 7.3 gives a lower (and upper) bound of $minReg + Delay$ for the case where $Delay = 1$. The previous section showed that $minReg + Delay$ is *not* a lower bound when $Delay > 1$.

Given any Sethi-Ullman order evaluation of an expression, it is not difficult to show, however, that $minReg + \lceil Delay\ /\ 2 \rceil$ is a lower bound on the registers needed for delay-free evaluation when put into canonical form (when $Delay > 1$). Given the canonical evaluation order with exactly *minReg* registers, it must be the case that some operation immediately follows the load of one of its operands. Adding a single register shifts the loads earlier and operations later in the schedule. For each additional register, each load moves at most one instruction earlier in the schedule,

| *Delay* | DLS Optimal | DLS Suboptimal | Total | % DLS Optimal |
|---|---|---|---|---|
| 2 | 1,015,481 | 17,930 | 1,033,411 | 98.3 |
| 3 | 1,015,481 | 17,930 | 1,033,411 | 98.3 |
| 4 | 1,007,509 | 25,902 | 1,033,411 | 97.5 |
| 5 | 1,007,509 | 25,902 | 1,033,411 | 97.5 |
| 6 | 1,007,511 | 25,900 | 1,033,411 | 97.5 |
| 7 | 1,007,535 | 25,876 | 1,033,411 | 97.5 |
| 8 | 1,007,703 | 25,708 | 1,033,411 | 97.5 |
| 9 | 1,008,631 | 24,780 | 1,033,411 | 97.6 |

Fig. 23. Heuristic results for all binary trees of 25 or fewer nodes.

and each operation moves at most one instruction later. (Loads in the initial (nonalternating) sequence of loads do not move at all—likewise for the operations in the latter part of the schedule.) Thus, the addition of each register can move a load/operation pair at most two instructions further apart.

Therefore, if *Delay* instructions are necessary to fill a delay slot, and each register will move at most two instructions between any pair, it will be necessary to use a minimum of $\lceil Delay \ / \ 2 \rceil$ registers in addition to the original *minReg*.

It is possible to find the minimal number of registers needed for a particular combination of operation and load orders in linear time. This can be done by computing the schedule (in canonical order) for each possible number of registers. Starting with $minReg + Delay - 1$ registers and working down, the canonical orders are created and tested to make certain that the loads are separated from their successors by at least *Delay* instructions. Because the lower bound on number of registers for a legal canonical order is $minReg + \lceil Delay \ / \ 2 \rceil$, this process will require at most $\lceil Delay \ / \ 2 \rceil$ iterations. With this inexpensive extra step, it is possible to fine-tune DLS further to use even fewer registers in many cases when $Delay > 1$.

### 12.3 Empirical Results

The DLS algorithm works extremely well as a heuristic for *Delay* values greater than 1. By enumerating all possible expression-trees of 25 or fewer nodes, and testing the algorithm against an exhaustive search algorithm, the effectiveness of the algorithm as a heuristic can be easily verified. We ran trials for *Delay*'s of 2 through 9 to obtain the results given in Figure 23. A schedule produced by DLS is considered suboptimal if it uses $r$ registers but contains no unfilled delay slots when there exists a delay-free schedule needing fewer than $r$ registers, or if it uses $r$ registers and contains $i$ unfilled delay slots when there exists a schedule using $r$ registers that contains fewer than $i$ unfilled delay slots. DLS never uses more than $minReg + Delay$ registers and produces a delay-free schedule for all but a finite number of trees for any given *Delay*.

In Section 13.6, we compare schedules produced by DAG_DLS, a DAG scheduler based on DLS, with optimal schedules generated using branch and bound.

### 12.4 Anomaly

Using DLS for $Delay = 2$, an interesting (and surprising) counterintuitive result has been found. It is possible for an expression-tree to have a subtree whose optimal (delay-free) evaluation requires *more* registers than the entire tree's optimal

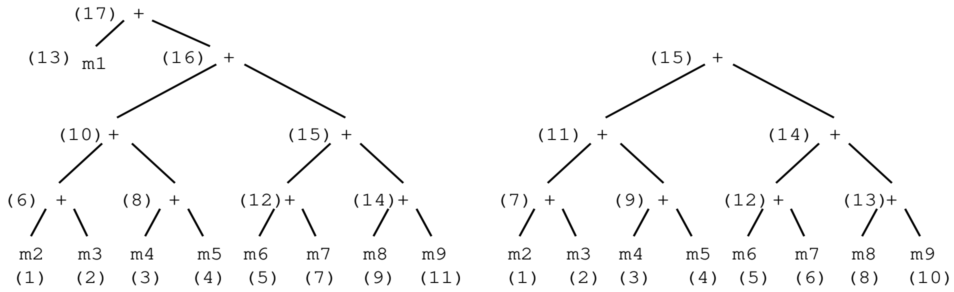



Fig. 24. Anomaly for *Delay* = 2. Entire tree needs five registers—Right subtree alone needs six. (The respective optimal evaluation orders are given.)

evaluation. The left tree of Figure 24 can be evaluated optimally with five registers; however its right subtree taken alone requires six registers for a delay-free evaluation. (No delay-free evaluation exists using fewer than six registers, and DLS will find this optimal evaluation.) Note also that the full tree has $minReg = 4$, and $Delay = 2$, yet it needs only five registers for a spill-free, delay-free evaluation.

## 13. DAG_DLS

This section outlines scheduler DAG_DLS (Figure 25), which schedules a DAG that represents a basic block (a more-detailed discussion of DAG_DLS will be provided in a subsequent paper). DLS assumes that each expression is a tree in which (1) loads appear only at the leaves and (2) all operators are delay free. DAG_DLS lifts these restrictions, allowing for common subexpressions, internal loads, and operators with delay slots. DAG_DLS is a heuristic, as finding an optimal schedule for a DAG whose nodes can have an arbitrary number of delay slots is NP-Complete [Hennessy and Gross 1982; Palem and Simons 1990].

DAG_DLS is based on DLS. DLS uses the fewest number of registers to generate an optimal schedule for an expression-tree. Similarly, DAG_DLS aims to use as few registers as possible to generate a delay-free schedule. To fill empty delay slots, DAG_DLS slides nodes earlier in the schedule just as DLS does in producing a canonical-form schedule.

### 13.1 Forming Expressions whose Edges are True Dependences

DLS routines *MapNodes*, *label*, and *UnaryCalc* assume that each edge in an expression is a true dependence. False dependences, however, can affect the ordering of operands in an expression. For example, if the left operand of a node reads a register variable that is to be written to by its right operand, then the left operand must be scheduled first.

Routine *TrueDependences* of DAG_DLS (line 7 of Figure 25) divides a DAG into a list of expressions *eList*. The only edges between nodes within each expression in *eList* are true dependences. If there is a false dependence between two nodes, then these nodes are part of separate expressions in *eList*. Given a false dependence from node $n_1$ in expression $e_1$ to node $n_2$ in expression $e_2$, $e_1$ occurs before $e_2$ in the list. Scheduling expressions in *eList* in the order they appear generates a schedule that does not violate false dependences.

```
1  function DAG_DLS(DAG n): nodeList
2    DAG e;                                  // current expression
3    DAGList eList;                          // list of expressions
4    nodeList sched ← NULL;                  // linearized schedule
5
6    // divide a DAG into a list of expressions whose edges are true dependences
7    eList ← TrueDependences(n);
8
9    // for each expression in eList
10   for e ← First(eList) to Last(eList) do
11     // schedule nodes within each expression to limit register pressure
12     // and fill delay slots
13     D_MapNodes(e);          // map nodes whose operands are register variables
14                             // or small literal values to other nodes
15     D_Label(e);             // estimate register pressure
16     D_UnaryCalc(e);         // find an ordering to fill delay slots and
17                             // minimize register pressure
18
19     // form a list of nodes from the current expression and append to list sched
20     sched ← sched || Linearize(e);
21   end for;
22
23   // slide nodes in sched earlier in the schedule to fill empty delay slots
24   return SlideNodes(sched);
25 end function
```

Fig. 25. Routine to schedule a DAG that represents a basic block.

### 13.2 Finding a Node Ordering that Limits Register Pressure and Fills Delay Slots

The next phase of DAG_DLS orders nodes within each expression in *eList* to limit register pressure and enable unary operators and literals to fill delay slots. Since finding a schedule that minimizes register pressure for an expression is NP-Complete [Garey and Johnson 1979], our approach is heuristic.

The expressions in *eList* are scheduled in the order they occur (lines 10 to 21 of Figure 25). In DLS, *MapNodes* maps nodes whose operands include register variables or small literal values to nondelayed loads or unary operators; *label* computes the minimum number of registers needed to evaluate an expression-tree; and *UnaryCalc* determines a preliminary node ordering for a tree to minimize register pressure and allow unary operators and literals to fill delay slots. DAG_DLS routines *D_MapNodes*, *D_Label*, and *D_UnaryCalc* are similar to those in DLS, but the DAG_DLS routines traverse expressions that may contain common subexpressions. The results of *D_Label* and *D_UnaryCalc* are only approximations of optimal solutions if the expression is not a tree.

*D_MapNodes*, *D_Label*, and *D_UnaryCalc* visit each common subexpression once. After *D_MapNodes*, *D_Label*, and *D_UnaryCalc* are called, routine *Linearize* generates a list of nodes based on the ordering determined by *D_UnaryCalc*.

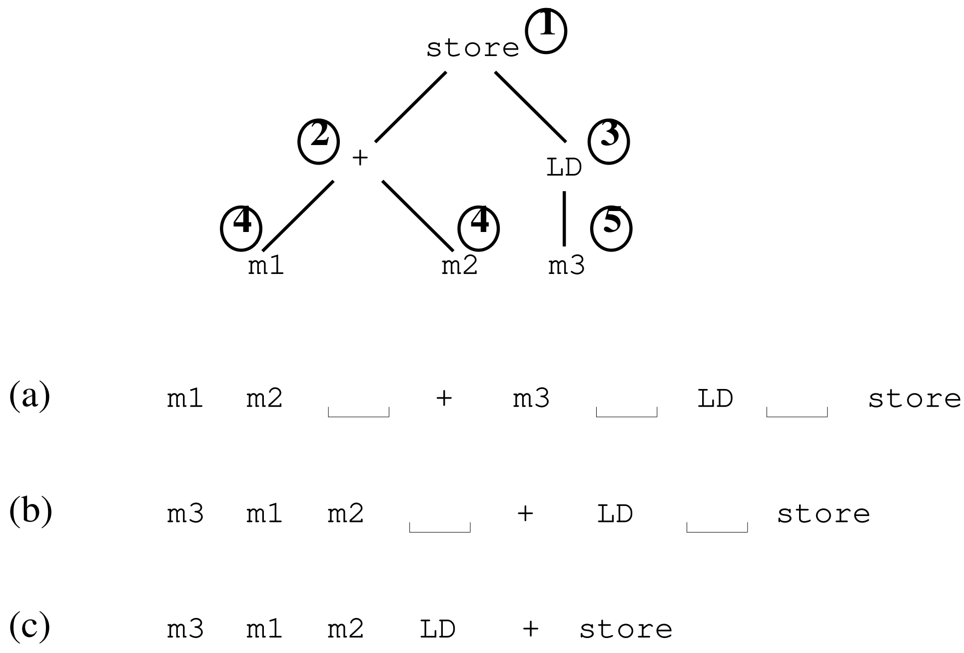



Fig. 26. Deciding how far to slide a node.

### 13.3 Sliding Nodes to Fill Delay Slots

To fill empty delay slots, both DLS and DAG_DLS slide nodes toward the front of the schedule. DLS assumes that only loads can have delay slots and that loads occur only at the leaves. Sliding only loads; DLS optimally fills all delay slots. In DAG_DLS expressions can have internal loads, and operators can have delay slots—simply moving loads at the leaves may leave delay slots unfilled in the schedule.

The distance to move a node depends not only on the empty delay slots occurring earlier than the node in the schedule, but also on the nodes dependent on it. Figure 26 shows an expression. Node LD represents an internal load in the expression. An ordering of the nodes of this tree is shown in row (a). Underscores represent empty delay slots. Since m2 does not depend on the value of m1, m2 fills the delay slot following m1.

The distance to move m3 in the schedule depends on LD, whose operand is m3, and the empty delay slot after m2 in the schedule. If node m3 in row (a) slides to the delay slot following m2, then the node LD, which is dependent on m3, may be unable to slide far enough in the schedule to have its delay slot filled. However, if m3 slides past the delay slot following m2, then no node may fill the delay slot following m2.

To gauge how far to move a node, a cumulative cost for each node is computed. The concept of cumulative cost is borrowed from Goodman and Hsu's widely used scheduler Integrated Prepass Scheduling (IPS) [Goodman and Hsu 1988]. The cumulative cost for a node $i$ is computed by taking the maximum cumulative cost of its successors in the DAG and adding the cost of executing node $i$. Scheduling a node early with a large cumulative cost will often improve the chance of filling the delay slots of those nodes dependent on it.

In Figure 26, each node in the subtree is marked by its cumulative cost—loads have a cumulative cost of two, and all other nodes have a cumulative cost of one. When sliding a node earlier in the schedule, the node may fill an empty delay slot only when the next node to be passed has a greater or equal cumulative cost than the current node, or a node with greater or equal cumulative cost has already been passed. The nodes to be moved after the current node can potentially fill the delay slots that the current node passed over. Also, moving the current node far enough may allow the nodes moved subsequently to have their own empty delay slots filled. For example, in row (b) m3 has a larger cumulative cost than m1, m2, and +, so it passes the empty delay slot after m2 and moves to the beginning of the schedule. In row (c), node LD, which has a smaller cumulative cost than m2, but a larger cumulative cost than +, slides ahead of + and into the empty delay slot following m2. All delay slots are filled.

DAG_DLS calls *SlideNodes* (line 24 of Figure 25) to move nodes in the schedule. While sliding a node, the node may pass at most a constant number of other nodes. This restriction allows *SlideNodes* to have an $O(n)$ time complexity, where $n$ is the number of nodes in the DAG. In addition, since sliding nodes earlier in the schedule can increase register pressure, *SlideNodes* updates the register pressure at the positions a node slides past. To avoid overallocating registers, the routine stops moving a node if there are an insufficient number of registers available for allocation to nodes.

### 13.4 Complexity

The time complexity of DAG_DLS is $O(n)$, where $n$ is the number of nodes in the DAG. DAG_DLS calls *TrueDependences*, *D_MapNodes*, *D_Label*, *D_UnaryCalc*, and *Linearize* which are all linear, since (1) each node is visited once by each routine and (2) the number of dependences between nodes is linear in the number of nodes in the DAG. DAG_DLS calls *SlideNodes*, which is also linear, as nodes slide past at most a constant number of other nodes, and computing the cumulative cost for each node requires a single pass over the nodes.

### 13.5 Performance of DAG_DLS and IPS

We implemented DAG_DLS and IPS in the back end of lcc [Fraser and Hanson 1991], which generates code for a DECstation 5000 with R3000/R3010 processors. A node is allowed to slide past at most 10 nodes in DAG_DLS, as this number works well in practice.

Figure 27 presents the execution-time improvement on selected SPEC benchmarks when scheduling with DAG_DLS as a replacement for IPS. Benchmarks *eqntott*, *espresso*, and *xlisp* are C benchmarks and the others are Fortran benchmarks coverted to C using the utility *f2c*. DAG_DLS performs significantly better than IPS on the Fortran benchmarks, whereas IPS performs slightly better on average for the C benchmarks.

Fortran benchmarks often compute multiple expressions in a single basic block. DAG_DLS performs better on Fortran benchmarks because it effectively schedules each expression and only schedules nodes from the next expression when there are

| *Execution-time improvement* | |
|---|---|
| *benchmark* | *improvement* |
| dnasa | 1.2% |
| doduc | 2.8% |
| eqntott | 0.0% |
| espresso | -0.3% |
| fpppp | 5.3% |
| spice | 0.4% |
| tomcatv | 3.1% |
| xlisp | -0.2% |

Fig. 27. Execution-time improvement on SPEC benchmarks when scheduling with DAG_DLS as a replacement for IPS.

| *Comparing Execution Times of DAG_DLS and IPS* | | |
|---|---|---|
| *benchmark* | *execution time of DAG_DLS* | *execution time of IPS as a percentage of the execution time of DAG_DLS* |
| dnasa | 4.5s | 84.7% |
| doduc | 16.7s | 113.9% |
| eqntott | 3.7s | 97.4% |
| espresso | 24.8s | 99.1% |
| fpppp | 10.2s | 172.8% |
| spice | 40.1s | 118.9% |
| tomcatv | 0.5s | 106.6% |
| xlisp | 12.1s | 97.1% |

Fig. 28. Time spent scheduling by DAG_DLS and time spent by IPS as a percentage of the execution time of DAG_DLS.

available registers. IPS may try to intermix evaluation of expressions. This can increase the register pressure beyond the threshold allowed by IPS. To decrease register pressure, IPS may search for nodes that can free registers at the expense of not filling delay slots, and if nodes that free registers cannot be found, register spilling can occur.

When there is less of a demand for registers, as in the case for the C benchmarks, IPS performs well. IPS can select nodes individually from a larger section of the DAG. DAG_DLS only slides nodes earlier in the schedule. Sliding a node earlier in the schedule cannot fill an empty delay slot occurring at a later position in the schedule.

Figure 28 presents the execution time of DAG_DLS and the percentage improvement this represents over IPS, an $O(n^2)$ scheduler. DAG_DLS is significantly faster than IPS on benchmarks *doduc*, *fpppp*, and *spice*, all of which have large basic blocks. On benchmark *fpppp*, DAG_DLS spends about 10 seconds; IPS spends approximately 70% more time than is required by DAG_DLS. For benchmarks with smaller basic blocks, the scheduling times for both schedulers are competitive.

### 13.6 Comparing DAG_DLS and IPS to an Optimal Scheduler

We implemented a branch-and-bound algorithm that finds a schedule requiring the fewest number of cycles for all basic blocks with 20 or fewer instructions for the

| Comparing performance of DAG_DLS and IPS with an Optimal Scheduler | | | | | |
|---|---|---|---|---|---|
| *benchmark* | *dynamic cycle count of optimal schedules* | DAG_DLS | | IPS | |
| | | *additional cycles* | *% above optimal* | *additional cycles* | *% above optimal* |
| dnasa | 8,890,557,534 | 119,482,481 | 1.34% | 107,091,098 | 1.20% |
| eqntott | 1,178,718,422 | 13,357 | < 0.01% | 333 | < 0.01% |
| espresso | 2,152,731,998 | 154,417 | 0.01% | 246,478 | 0.01% |
| tomcatv | 773,006,807 | 13,231,403 | 1.71% | 13,231,660 | 1.71% |
| xlisp | 3,226,952,871 | 9,328,767 | 0.29% | 9,328,767 | 0.29% |

Fig. 29. The number of cycles above optimal of DAG_DLS and IPS schedules and the percentage above optimal these additional cycles represent. Basic blocks with 20 or fewer instructions are compared.

SPEC benchmarks listed in Figure 29. These basic blocks were instrumented to compute their dynamic cycle counts.

The second column of Figure 29 shows the dynamic cycle count for the optimal schedules. The remaining columns present the number of cycles above optimal for the DAG_DLS and IPS schedules and the percentage above optimal these additional cycles represent.

As mentioned in Section 13.5, DAG_DLS outperforms IPS on benchmarks with large basic blocks. However, the performance of both DAG_DLS and IPS are excellent for smaller blocks. For these blocks their performance is either identical or nearly identical. On the C benchmarks (*eqntott*, *espresso*, and *xlisp*), both schedulers are within 0.3% of optimal. Since DAG_DLS often generates betters schedules for large basic blocks than IPS, and both schedulers perform well on small basic blocks, DAG_DLS seems a reasonable alternative to IPS in production compilers.

## 14. CONCLUSION

The DLS algorithm performs optimal code scheduling and optimal register allocation in linear time for binary expression-trees with load delays of one cycle. The algorithm can be modified to predict optimal locations for register spilling. Unlike other code-scheduling algorithms, it does not suffer from phase-ordering problems with register allocation.

Extensions to the binary tree algorithm allow DLS to schedule trees optimally with nondelayed load instructions and unary operations. DLS can also handle register variables and constants optimally.

DAG_DLS, which schedules DAGs and is based on DLS, often generates better schedules than Goodman and Hsu's scheduler IPS on large basic blocks and is significantly faster. Both schedulers generate a near-perfect schedule for small basic blocks. As a heuristic, DAG_DLS coordinates register allocation and code scheduling without sacrificing the algorithmic linearity of DLS.

REFERENCES

Bernstein, D., Jaffe, J. M., and Rodeh, M. 1989. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM J. Comput. 18,* 6 (Dec.), 1098–1127.

Bernstein, D., Pinter, R. Y., and Rodeh, M. 1984. Optimal scheduling of arithmetic operations in parallel with memory access. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 325–333.

Coffman, E. G., Jr., Ed. 1976. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York.

Fraser, C. W. and Hanson, D. R. 1991. A retargetable compiler for ANSI C. *ACM SIGPLAN Not. 26,* 10 (Oct.), 29–43.

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

Gibbons, P. B. and Muchnick, S. S. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. ACM, New York, 11–16.

Goodman, J. R. and Hsu, W.-C. 1988. Code scheduling and register allocation in large basic blocks. In *Proceedings of the International Conference on Supercomputing*. ACM, New York, 442–452.

Gwennap, L. 1994. PA-8000 combines complexity and speed. *Microprocess. Rep. 8,* 15, 1, 6–9.

Hennessy, J. L. and Gross, T. R. 1982. Code generation and reorganization in the presence of pipeline constraints. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 120–127.

Hennessy, J. L. and Gross, T. R. 1983. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst. 5,* 3 (July), 422–448.

Hu, T. C. 1961. Parallel sequencing and assembly line problems. *Oper. Res. 9,* 6, 841–848.

Lawler, E., Lenstra, J. K., Martel, C., Simons, B., and Stockmeyer, L. 1987. Pipeline scheduling: A survey. Computer Science Research Rep., IBM Research Division, San Jose, Calif.

Palem, K. and Simons, B. 1990. Scheduling time-critical instructions on RISC machines. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 270–280.

Patterson, D. A. and Hennessy, J. L. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Palo Alto, Calif.

Proebsting, T. A. and Fischer, C. N. 1991. Linear-time optimal code scheduling for delayed-load architectures. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, New York, 256–267.

Sethi, R. and Ullman, J. D. 1970. The generation of optimal code for arithmetic expressions. *J. ACM 17,* 4 (Oct.), 715–728.

Warren, H. S., Jr. 1990. Instruction scheduling for the IBM RISC system/6000 processor. *IBM J. Res. Dev. 34,* 1, 85–92.

Received August 1993; Revised June 1995; Accepted July 1995