Optimal Dual-Issue Instruction Scheduling With Spills For Binary Expression Trees

Waleed M. Meleis Edward S. Davidson

Advanced Computer Architecture Lab
University of Michigan, Ann Arbor MI
waleed@eecs.umich.edu

Abstract

We describe an algorithm that produces code, with spills, for a register-constrained machine that can issue up to one arithmetic operation and one memory access operation per time slot, under the restrictions that the code's dependence graph is represented as a binary tree with no unary operations, and the latency of the operations is 1. *We prove that the algorithm produces a minimum cost schedule, and show that its complexity is* $O(nk)$ *where* n *is the number of operations to be scheduled and* k *is the number of spills in the schedule. The number of issue slots in the minimum length schedule is* $\rho + 2k + |A|$ *where* ρ *is the number of registers used,* k *is the number of spills and* $|A|$ *is the number of arithmetic operations in the tree. We show this is the cost of any schedule that is in "contiguous form".*

1 Introduction

High performance processors are increasingly memory bottlenecked due to increasing processor issue bandwidth and clock speeds relative to the limited number of ports and access latency to memory. Reducing memory traffic by improving data reuse at all levels of the memory hierarchy is therefore essential for improving system performance. Processors that can issue more than one instruction per cycle can hide the latency of spill code by overlapping it with instructions that do useful work. In this paper, we introduce an algorithm that produces a minimal length code schedule for a processor that can issue up to one 2-operand arithmetic operation and one memory access operation per time slot, provided that the code's dependence graph is represented as a binary tree and the latency of each operation is 1. The complexity of the algorithm is shown to be $O(nk)$ where n is the number of operations to be scheduled and k is the number of spills in the schedule.

The problem of minimizing the number of registers needed to evaluate an expression tree without spills on a single-issue processor was first solved by Nakata [10] and by Redziejowski [11]. The algorithm uses a postorder evaluation of the expression tree and executes in time proportional to the number of operations to be scheduled. Sethi and Ullman extend this result in [13] to minimize the amount of spill code needed to evaluate an expression tree, given a fixed number of registers. This algorithm also uses a postorder traversal of the expression tree and executes in linear time. Aho and Johnson develop a dynamic programming algorithm that solves the same problem in [1]. Bernstein et al. [2] show how to find the cheapest dual-issue schedule without spills for a binary expression tree, given a fixed number of registers. They also give an efficient heuristic that applies if the expression tree contains unary operations.

The problem of minimizing the number of registers needed in an evaluation of an expression DAG without spills was shown to be NP complete by Sethi in [12]. An expression DAG differs from an expression tree in that a computed quantity may be used more than once in the course of the computation. Bruno and Sethi show in [6] that evaluating an expression DAG using minimal spill code for a 1-register machine is NP-complete.

The spill insertion problem is to insert the minimum amount of spill code into a given initial schedule. The most efficient algorithms known for spill insertion are shown by Horwitz et al. [8], and Hsu et al. [9] to be exponential in the number of registers in the worst case. This is a striking result because it demonstrates that optimal code generation is difficult even when the instruction schedule is fixed. They derive pruning rules for optimal spilling and present heuristics.

The basic multi-processor scheduling problem is to optimally schedule a set of unit time tasks, given a partial ordering on the tasks and a set of processors that can each execute disjoint subsets of the tasks. This is equivalent to the problem of optimally scheduling an expression on a machine with distinct functional units and infinite registers. The multi-processor scheduling problem for DAGs is shown to be NP-complete by Ullman in [14]. The multi-processor scheduling problem for trees is shown to be NP-complete by Bernstein et al. in [3]. These results demonstrate that optimal code generation for a multiple-issue machine is difficult even in the absence of register restrictions.

2 The Machine/Computation Model

The notation used in [2] formalizes the sequential and parallel scheduling problems without spills. We begin by extending this notation to allow the use of spill code. We consider a machine model that has access to R registers: r_1 r_2,...,r_R and an arbitrarily large memory space. The machine has three types of operations: load operations which copy a data value from memory into a register, store operations which copy a data value from a register into memory, and binary arithmetic operations which read two data values from registers and write a new data value to a register.

A computation tree, F, contains leaf nodes that represent load operations and internal nodes that represent arithmetic operations. In this work, every internal node is assumed to have exactly two children. An edge between two nodes represents a data dependence between the two operations. The set of *arithmetic operations* in F is denoted $A(F) = A_1, A_2, ..., A_{|A(F)|}$, where A_i may denote either the operation or its *arithmetic result* (or simply, *value*). The set of *load operations* is $L(F) = L_1, L_2, ..., L_{|L(F)|}$ where L_i represents a load of A_i if $1 \leq i \leq |A(F)|$, and L_i represents a leaf node of F if $|A(F)| < i \leq |L(F)|$. The set of *store operations* is denoted $S(F) = S_1, S_2, ..., S_{|A(F)|}$ where S_i represents a store of A_i. A load operation L_i or a store operation S_i is referred to as *spill code* if $1 \leq i \leq |A(F)|$. Note that if value A_i is spilled in an evaluation of F, S_i and later L_i will appear in the schedule somewhere between A_i and the use of A_i; if A_i is not spilled, S_i and L_i will not appear at all.

For a given node v of F, F_v denotes the subtree of F with v as its root. If u is a child of v in F, we say u is *used* in the computation of v.

A *sequential evaluation* P of a computation F is an ordered list of operations (including spill code): $P = P_1, P_2, ..., P_n$ where P_i is executed in slot i, and the following three conditions are satisfied:

1. Every arithmetic operation and load leaf is executed, i.e. for every node v, there exists a unique j such that $P_j = v$.

2. No dependences of F are violated, i.e. if P_i, P_j are nodes in F, then $P_j \in F_{Pi}$ implies $j \leq i$. This implies that the latency of each arithmetic and load operation is 1 slot, i.e. if value P_j is used by P_i then $i \geq j + 1$.

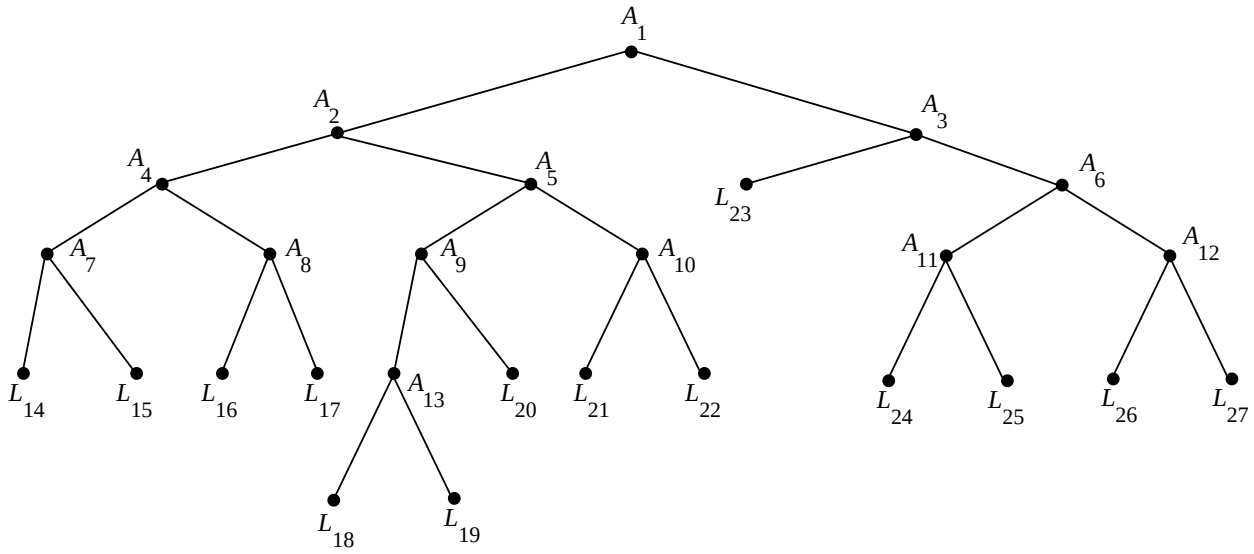
3. If the value A_v corresponding to a non-leaf node P_i is used by P_j $(j > i)$, then A_v may be stored in slot k_1, and loaded in slot k_2, provided that $i < k_1 < k_2 < j$. In this case, $P_{k_1} = S_v$ and $P_{k_2} = L_v$ are spill code.

Conditions 2 and 3 are collectively referred to as the *precedence constraints* of a sequential evaluation. The *completion time* of P, $c(P)$, equals $|P|$, the number of operations in P.

A *parallel evaluation* PQ of a computation F is an ordering of operation pairs: $PQ = PQ_1, PQ_2, ...PQ_n$ where $PQ_i = (p_i, q_i)$ such that $p_i \in L(F) \cup S(F) \cup \{NOP\}$ and $q_i \in A(F) \cup \{NOP\}$. NOPs are used to indicate that no operation is scheduled in either or both operation pairs at a slot. A parallel evaluation PQ must satisfy the following four constraints:

1. For every nonleaf node v, there exists a unique PQ_j such that $q_j = v$.

2. For every leaf node v, there exists a unique PQ_j such that $p_j = v$.



(a) A computation F

slot # i =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P_i =	L_{18}	L_{19}	A_{13}	L_{20}	A_9	L_{21}	L_{22}	A_{10}	A_5	S_5	L_{14}	L_{15}	A_7	L_{16}	L_{17}	A_8	A_4	L_5	A_2
$\|inreg\|_i$ =	1	2	1	2	1	2	3	2	1	0	1	2	1	2	3	2	1	2	1

slot # i =	20	21	22	23	24	25	26	27	28	29	30	31
P_i =	S_2	L_{24}	L_{25}	A_{11}	L_{26}	L_{27}	A_{12}	A_6	L_{23}	A_3	L_2	A_1
$\|inreg\|_i$ =	0	1	2	1	2	3	2	1	2	1	2	1

(b) A sequential evaluation of F

slot # i =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
p_i =	L_{14}	L_{15}	–	L_{16}	L_{17}	–	L_{18}	L_{19}	–	L_{20}	L_{21}	S_4	L_{22}	–	–	L_4	–	L_{24}	L_{25}
q_i =	–	–	A_7	–	–	A_8	A_4	–	A_{13}	–	A_9	–	–	A_{10}	A_5	–	A_2	–	–
$\|inreg\|_i$ =	1	2	1	2	3	2	2	3	2	3	3	2	3	2	1	2	1	2	3

slot # i =	20	21	22	23	24	25	26	27	28
p_i =	L_{26}	S_{11}	L_{27}	–	L_{11}	–	L_{23}	–	–
q_i =	A_{11}	–	–	A_{12}	–	A_6	–	A_3	A_1
$\|inreg\|_i$ =	3	2	3	2	3	2	3	2	1

(c) A parallel evaluation of F

Figure 1: Examples of notation.

3. For every PQ_i, PQ_j, if $q_i \neq$ NOP, then $p_j \in F_{q_i}$ implies $j < i$, and $q_j \in F_{q_i}$ where $q_j \neq q_i$ implies $j < i$. (once again, latency is 1).

4. For every PQ_i, PQ_j, if $q_i = A_v$ and q_j uses A_v $(j > i)$, then A_v may be stored in slot k_1, and loaded in slot k_2, provided that $i < k_1 < k_2 < j$. In this case, $PQ_{k_1} = (S_v, q_{k_1})$ and $PQ_{k_2} = (L_v, q_{k_2})$ are spill code.

Conditions 3 and 4 are collectively referred to as the precedence constraints of a parallel evaluation. A pair (p_i, NOP) is called an *empty arithmetic slot*, (NOP, q_i) is an *empty load/store slot*, and (NOP, NOP) is an *empty slot*. The completion time of PQ, $c(PQ)$, equals $|PQ|$.

The concatenation, $P_1|P_2$ (or simply P_1P_2), of two sequential evaluations represents the sequential evaluation formed by the operations in P_1 followed by the operations in P_2.

For a sequential or parallel evaluation E, $inreg_i$ is the set of all values that are in a register in slot i, namely the set of all values v such that v is defined or loaded in slot j, where $j \leq i$, and is neither used nor stored in any slot k, where $j < k \leq i$. A nonleaf value v is said to be *spilled* in slot i if it is stored in slot j, where $j \leq i$, and is loaded in slot k, where $k > i$. Thus if v is spilled in slot i, then v is not in a register in slot i.

Each load operation adds one element to $inreg$, each store removes one, and each arithmetic operation adds one and removes two. Therefore, $|inreg_i|$ = the total number of loads, minus stores, minus arithmetic operations in slot i or earlier. The number of values in registers at slot i of an evaluation E, $|\text{inreg}_i|$, is denoted $\rho_i(E)$, and $\rho(E)$ denotes the maximum over all i of $\rho_i(E)$. An example is shown in Figure 1.

Given a computation, F, and the number of available registers, R, the sequential scheduling problem is to construct a sequential evaluation P such that $c(P)$ is minimized subject to $\rho(P) \leq R$. The parallel scheduling problem is to construct a parallel evaluation PQ such that $c(PQ)$ is minimized subject to $\rho(PQ) \leq R$. In each case, the evaluation constructed is said to *use* R *registers*.

3 Minimizing spills in a parallel evaluation

Given a tree F and a number of available registers R, the solution to the parallel scheduling problem is found by using the sequential scheduling algorithm to construct a sequential evaluation with the minimum number of spills, transforming this evaluation into a parallel evaluation, and then transforming this evaluation into a "contiguous" form. At each stage of the transformation the amount of spill code remains unchanged and the cost of the evaluation does not increase. In this section we show that no parallel evaluation of F can have fewer spills than the optimal sequential evaluation of F, and that a minimum cost sequential evaluation can therefore be transformed into a minumum spill parallel evaluation. In the following sections, we show that every parallel evaluation, PQ, of F in contiguous form that contains k spilled values has a cost of $\rho(PQ) + 2k + |A(F)|$ and that no parallel evaluation can cost less.

3.1 Minimizing spills in a sequential evaluation

In [1] and [13], efficient algorithms to solve a variant of the sequential scheduling problem are described. In addition to the three types of operations described above, these algorithms include another latency 1 binary arithmetic operator that reads one operand from a register and one operand from memory and writes its result to a register, causing no net change in $inreg_i$. In [4],

Bose describes a simple adaptation of the algorithm in [1] that solves the sequential scheduling problem without the fourth instruction class. The algorithm first computes a label, M, for each vertex of F, and then uses these labels to construct the schedule.

Sequential scheduling algorithm

Visit the vertices of the tree in postorder. For every vertex v compute its label, $M(v)$, as follows:

```
Label(v)
     1.If v ∈ L(F):   M(v) = 1.
     2.If v ∈ A(F) with children v1, v2:
          a) if M(v1) ≠ M(v2), M(v) = max(M(v1), M(v2))
          b) if M(v1) = M(v2), M(v) = M(v1) + 1
End
```

Apply $Schedule$ to the vertex of F to produce the sequential evaluation.

```
Schedule(v)
     if M(v) = 1 {i.e. v is a leaf} Compute v;
          {otherwise, vertex v has children v1, v2}
     else if M(v1), M(v2) ≥ R:
          Schedule(v1); Store v1; Schedule(v2); Load v1; Compute v;
     else if M(v2) < M(v1):
          Schedule(v1); Schedule(v2); Compute v;
     else
          Schedule(v2); Schedule(v1); Compute v;
End
```

As argued in [4], the sequential scheduling algorithm computes a cheapest evaluation of F that uses no more than R registers. Furthermore, it can be shown that if the cheapest evaluation contains no spills, the algorithm minimizes the maximum number of values in registers. The proof of this, which we omit, is similar to the one given in [13]

Clearly $c(P)$ equals the number of nodes in F plus the number of spill operations. Therefore an optimal sequential evaluation must contain the fewest spill operations among all sequential evaluations. Furthermore, if k values are spilled, there are $2k$ spill operations, i.e. k store/load pairs. Figure 1(b) shows the result of the sequential scheduling algorithm for $R = 3$.

3.2 A lower bound on spills in a parallel evaluation

A simple transformation allows us to transform any k-spill parallel evaluation that uses R registers into a k-spill sequential evaluation that uses R registers. Given such a parallel evaluation, PQ, recall that each slot i has $PQ_i = (p_i, q_i)$. Let $J_i = q_i p_i$, and let $P' = J_1 J_2 ... J_{|PQ|}$, then delete the NOPs to create a sequential evaluation, P. In [2] the same transformation is applied to parallel evaluations that do not contain spills. We first show that when P is constructed in this manner it represents a sequential evaluation. Then we show that $\rho(P) \leq \rho(PQ)$.

If operation v is in an earlier slot than operation w in parallel evaluation PQ, v will be in an earlier slot than w after PQ is transformed into P. Since each non NOP operation in PQ is located uniquely in P (and visa versa), and since the precedence conditions among operations

slot # i=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
p_i=	L_{18}	L_{19}	–	L_{20}	–	L_{21}	L_{22}	–	–	S_5	L_{14}	L_{15}	–	L_{16}	L_{17}	–	–	L_5	–
q_i=	–	–	A_{13}	–	A_9	–		A_{10}	A_5	–	–	–	A_7	–	–	A_8	A_4	–	A_2
$\|inreg\|_i$ =	1	2	1	2	1	2	3	2	1	0	1	2	1	2	3	2	1	2	1

slot # i=	20	21	22	23	24	25	26	27	28	29	30	31
p_i=	S_2	L_{24}	L_{25}	–	L_{26}	L_{27}	–	–	L_{23}	–	L_2	–
q_i=	–	–	–	A_{11}	–	–	A_{12}	A_6	–	A_3	–	A_1
$\|inreg\|_i$ =	0	1	2	1	2	3	2	1	2	1	2	1

Figure 2: The result of transforming a sequential evaluation in Figure 1b into a parallel evaluation.

in PQ refer only to operations in distinct slots and the relative order of these operations is unchanged in P, it follows that P represents a sequential evaluation.

Consider some PQ_i such that $p_i, q_i \neq$NOP, and assume that $P_j = q_i$ and $P_{j+1} = p_i$. That is, assume that q_i and p_i become the jth and $(j+1)$th operations in P. Clearly the number of values in registers in slot i of PQ is the same as the number of values in registers in slot $j+1$ of P because in both evaluations the same operations have executed, i.e. $\rho_i(PQ) = \rho_{j+1}(P)$. Similarly, $\rho_{i-1}(PQ) = \rho_{j-1}(P)$. Since $q_i = P_j$ is an arithmetic operation, the number of values in registers decreases by one between slots $j-1$ and j of P, i.e. $\rho_j(P) = \rho_{j-1}(P) - 1$. Therefore, $\rho_j(P) < \rho_{i-1}(PQ)$. We have shown that the number of values in registers in slots j and $j+1$ of P is no larger than the number of values in registers in slots $i-1$ and i, respectively, of PQ. A similar analysis can be applied to all slots of P and PQ. For those slots of PQ for which p_i or q_i =NOP, the analysis is degenerate since NOPs do not alter ρ. Therefore $\rho(P) = \rho(PQ)$.

Since P is a sequential evaluation of F that contains the same operations as PQ, evaluations P and PQ contain the same number of spills. Therefore we may now prove the following theorem.

Theorem 1: *Given a computation F and a number of available registers R, if an optimal sequential evaluation, P, of F contains k spills, then no parallel evaluation of F contains fewer than k spills.*

To prove this theorem, assume that PQ is a parallel evaluation of F that uses no more than R registers and contains k' spills with $k' < k$. Then the transformation described above can be used to create a sequential evaluation of F that uses no more than R registers and contains k' spills. This contradicts the assertion that P is optimal. Hence, no parallel evaluation of F can contain fewer than k spills.

3.3 Creating a minimum-spill parallel evaluation

The optimal serial evaluation of F thus directly yields a lower bound, k, on the number of spills in any parallel evaluation of F. We can easily achieve this lower bound through a straightforward transformation of the optimal sequential evaluation. Given a k-spill sequential evaluation $P = P_1, P_2, ..., P_n$, then for each slot i:

- if $P_i \in A(F)$, let PQ_i = (NOP,P_i), and

slot # i=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_i=	L_{18}	L_{19}	L_{20}	L_{21}	L_{22}	L_{14}	L_{15}	S_5	L_{16}	L_{17}	L_5	L_{24}	L_{25}	S_2	L_{26}	L_{27}	L_{23}	L_2	–	–
q_i=	–	–	A_{13}	A_9	–	A_{10}	A_5	–	A_7	–	A_8	A_4	A_2	–	A_{11}	–	A_{12}	A_6	A_3	A_1
$\|inreg\|_i$ =	1	2	2	2	3	3	3	2	2	3	3	3	3	2	2	3	3	3	2	1

Figure 3: The result of applying the load-contiguous transformation to Figure 2 with $R = 3$.

- if $P_i \in L(F) \cup S(F)$, let $PQ_i = (P_i, \text{NOP})$.

No precedence is violated by this transformation since the operations are executed in the same order. Therefore PQ represents a k-spill parallel evaluation of F that uses R registers. Note, however, that PQ is degenerate since there is a NOP in every slot; thus PQ is unlikely to have the minimum number of slots. The result of applying this transformation to the sequential evaluation shown in Figure 1b can be seen in Figure 2.

4 Load-contiguous parallel evaluations

We have shown that a parallel evaluation with the minimum number of spills can be constructed by applying a transformation to the optimal sequential evaluation. We now describe the process of transforming this parallel evaluation into a canonical form. We define a load-contiguous parallel evaluation and show that every parallel evaluation can be made load-contiguous without increasing its cost, the maximum number of values in a register, or the number of spills.

A parallel evaluation of a computation F is said to be *load-contiguous* if and only if no load operation is scheduled in a later slot than an empty load/store slot. Let PQ be a k-spill parallel evaluation of a computation F that uses R registers. Consider the first slot i containing an empty load/store slot such that there is a later slot containing a load operation. Let j be the first slot after i that contains a load operation. We create an evaluation PQ' that is the same as PQ except either

1. the empty slot i has been deleted,
2. the load operation in slot j has been moved to i, or
3. the load operation in slot j has been deleted along with the store of the same value.

Clearly, in all three cases, $c(PQ') \leq c(PQ)$ and PQ' has no more spills than PQ. It remains to show that $\rho(PQ') \leq \rho(PQ)$ and that precedence is preserved.

First consider the degenerate case where slot i is an empty slot containing neither an arithmetic operation nor a load/store. Slot i can clearly be deleted without increasing ρ or affecting precedence.

Next consider the case where slot i contains an arithmetic operation. Since slot i contains only an arithmetic operation, $\rho_i(PQ) = \rho_{i-1}(PQ) - 1$. From slot i to slot j by assumption there are no load operations, so the number of registers used here can only stay the same or decrease. That is, for all slots s such that $i \leq s < j$, $\rho_s(PQ) \leq \rho_{i-1}(PQ) - 1$.

We tentatively create a new evaluation PQ' that is identical to PQ except the load operation in slot j is moved to slot i. That is, $p'_i = p_j$, and p'_j = NOP. For all $k < i$, $\rho_k(PQ') = \rho_k(PQ)$. Similarly, for all $k \geq j$, $\rho_k(PQ') = \rho_k(PQ)$. However, $\rho_i(PQ') = \rho_i(PQ) + 1$ due to the new load operation in slot i. Similarly, for all slots s such that $i \leq s < j$, $\rho_s(PQ') = \rho_s(PQ) + 1$. But since $\rho_s(PQ) \leq \rho_{i-1}(PQ) - 1$ from above, we can conclude that for $i \leq s < j$, $\rho_s(PQ') = \rho_s(PQ) + 1 \leq \rho_{i-1}(PQ)$. Therefore $\rho(PQ') \leq \rho(PQ) \leq R$. Moving a load operation earlier in the evaluation cannot violate its precedence with respect to an arithmetic operation that uses the loaded value. However, if the load operation is spill code which has been moved earlier than its associated store, precedence has been violated - but, in this case both operations can be deleted, thereby decreasing the number of spills without increasing either the cost of the evaluation or the maximum number of values in registers.

This transformation can be applied iteratively until the evaluation is load-contiguous. The transformation does not affect any operation earlier than i, and each time the transformation is applied slot i is either no longer empty or eliminated entirely, or a spill is eliminated. Therefore the iterations will eventually terminate, giving an evaluation in which the initial string of slots will contain a load or store operation until the load operations are exhausted. We have thus proven the following theorem.

Theorem 2: *Given any parallel evaluation PQ of a computation F, there exists a load-contiguous parallel evaluation PQ' of F such that:*

1. $c(PQ') \leq c(PQ)$,
2. $\rho(PQ') \leq \rho(PQ)$, *and*
3. PQ' *has no more spills than* PQ.

The result of applying this transformation to the parallel evaluation in Figure 2 is seen in Figure 3. We now extend the notion of load-contiguity to stores.

5 Store-contiguous parallel evaluations

We now define a store-contiguous parallel evaluation and show that every load-contiguous parallel evaluation can be made store-contiguous as well without increasing the evaluation's cost or the number of spills, and without causing the maximum number of values in registers to exceed R.

Given a R available registers, a parallel evaluation of a computation F is *store-contiguous* if and only if every store operation is in a slot i where $\rho_{i+1} = R$.

Consider the largest i such that slot i contains a store operation and $\rho_{i+1}(PQ) < R$. We show by contradiction that slot $i+1$ cannot contain a store. Suppose slot $i+1$ does contain a store operation. Then $\rho_{i+1}(PQ) < \rho_i(PQ)$. Furthermore, since slot i contains a store operation, $\rho_i(PQ) < \rho_{i-1}(PQ)$. Thus $\rho_{i+1}(PQ) \leq \rho_{i-1}(PQ) - 2$. The number of values in registers at a slot can increase by 1 in the next slot, which occurs when the next slot contains *only* a load operation. Therefore, $\rho_{i+2}(PQ) \leq \rho_{i-1}(PQ) - 1$. Since $\rho_{i-1}(PQ) \leq R$, we have $\rho_{i+2} \leq R - 1$. In this case, the store in slot $i+1$ contradicts the assumption that the store in slot i is the last store that precedes a slot with fewer than R values in registers. Therefore slot $i+1$ does not contain a store.

We now show that since $\rho_{i+1}(PQ) < R$, we can construct a new, evaluation PQ' in which the store in slot i is moved one slot later without increasing the evaluation's cost, the number

slot # i=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_i=	L_{18}	L_{19}	L_{20}	L_{21}	L_{22}	L_{14}	L_{15}	L_{16}	S_5	L_{17}	L_5	L_{24}	L_{25}	L_{26}	S_2	L_{27}	L_{23}	L_2	–	–
q_i=	–	–	A_{13}	A_9	–	A_{10}	A_5	A_7	–	–	A_8	A_4	A_2	A_{11}	–	–	A_{12}	A_6	A_3	A_1
$\|inreg\|_i$ =	1	2	2	2	3	3	3	3	2	3	3	3	3	3	2	3	3	3	2	1

Figure 4: The result of applying the store-contiguous transformation to Figure 3.

of spills, or the maximum number of values in registers. We separately consider the following six cases depending on the operations in slots i and $i+1$.

Case 1: $PQ_i = (S, A)$ and $PQ_{i+1} = (L, NOP)$: Swap the S and L, i.e. let $PQ'_i = (L, A)$ and $PQ'_{i+1} = (S, NOP)$. Moving a load operation earlier in an evaluation cannot violate a dependence. Clearly for all $j \leq i-1$, $\rho_j(PQ) = \rho_j(PQ')$. Since $PQ'_i = (L, A)$, then $\rho_i(PQ') = \rho_{i-1}(PQ')$. Finally, for all $k \geq i+1$, $\rho_k(PQ) = \rho_k(PQ')$ since for each of these k, the same set of operations have been executed in PQ and PQ'. Therefore $\rho(PQ') \leq \rho(PQ) \leq R$. If the store and load reference the same value, they can both be deleted from PQ' without increasing the maximum number of values in registers.

Case 2: $PQ_i = (S, A)$ and $PQ_{i+1} = (L, A')$: Let $PQ'_i = (L, A)$ and $PQ'_{i+1} = (S, A')$. The justification for this transformation is identical to that given for case 1.

Case 3: $PQ_i = (S, A)$ and $PQ_{i+1} = (NOP, A')$: Let $PQ'_i = (NOP, A)$ and $PQ'_{i+1} = (S, A')$. For all $j \neq i$, $\rho_j(PQ) = \rho_j(PQ') \leq R$. Since $PQ'_i = (NOP, A)$, then $\rho_i(PQ') = \rho_{i-1}(PQ') - 1 < R$. Therefore $\rho(PQ') \leq \rho(PQ) \leq R$.

Case 4: $PQ_i = (S, NOP)$ and $PQ_{i+1} = (NOP, A)$: Let $PQ'_i = (NOP, NOP)$ and $PQ'_{i+1} = (S, A)$. For all $j \neq i$, $\rho_j(PQ) = \rho_j(PQ')$. Since $PQ'_i = (NOP, NOP)$, then $\rho_i(PQ') = \rho_{i-1}(PQ')$. Therefore $\rho(PQ') \leq \rho(PQ) \leq R$. The empty slot PQ'_i can be deleted without increasing the maximum number of values in registers.

Case 5: $PQ_i = (S, NOP)$ and $PQ_{i+1} = (L, A)$, Swap PQ_i and PQ_{i+1}, i.e. let $PQ'_i = (L, A)$ and $PQ'_{i+1} = (S, NOP)$. Moving the arithmetic operation earlier than a store cannot violate precedence. Moving a load operation earlier than a store violates precedence only if L and S reference the same value, in which case they can be deleted as in case 1. For all $j \neq i$, $\rho_j(PQ) = \rho_j(PQ')$. Since $PQ'_i = (L, A)$, then $\rho_i(PQ') = \rho_{i-1}(PQ')$. Therefore $\rho(PQ') \leq \rho(PQ) \leq R$.

Case 6: $PQ_i = (S, NOP)$ and $PQ_{i+1} = (L, NOP)$: Let $PQ'_i = (L, NOP)$ and $PQ'_{i+1} = (S, NOP)$. No precedence is violated unless the load and store reference the same value, in which case they can be deleted as in case 1. By assumption, $\rho_{i+1}(PQ) < R$. The load operation in PQ_{i+1} increases the number of values in registers by 1, so $\rho_i(PQ) < R - 1$. The store operation in PQ_i decreases the number of values in registers by 1, so $\rho_{i-1}(PQ) < R$. Clearly for all $j \neq i$, $\rho_j(PQ') = \rho_j(PQ) \leq R$.. The load operation in PQ'_i increases the number of values in registers by 1, so $\rho_i(PQ') \leq R$. Therefore $\rho(PQ') \leq R$.

In each case we have shown that if slot i contains a store operation and $\rho_{i+1}(PQ) < R$, it is always possible to move the store one slot later without increasing the evaluation's cost or the number of spills, and without causing the maximum number of values in registers to exceed R.

This transformation can be applied repeatedly to the same store, moving it 1 slot later in each case, until either it does precede a slot in which R values are in registers, or the store can be deleted (if it is moved later than the load of the same value). Then the transformation can be applied to the preceding store. Since the evaluation contains a finite number of slots, each store will eventually either precede a slot containing R values in a register or be deleted.

The movement of one store cannot affect the prior placement of a later store since, as we have shown, stores cannot be in adjacent slots if they precede slots with R values in registers. Since there are finitely many stores, the transformations will eventually terminate with each remaining store preceding a slot with R values in registers.

It remains to show that it is possible to satisfy both load-contiguity and store-contiguity in an optimal schedule. Notice that up to the point of deleting load/store pairs, the store contiguity transforms that move a load interchange the load with a store. Thus they preserve load-contiguity. However, empty load/store slots are introduced wherever a load/store pair is deleted and these may destroy load-contiguity. In this case, however, the load-contiguity transformation can be reapplied. In the worst case, the load-contiguity and store-contiguity transformations are reapplied up to k times, i.e. each time a spill is eliminated. We have thus proven the following theorem.

Theorem 3: *Given a k-spill, load-contiguous, parallel evaluation PQ of a computation F such that $\rho(PQ) \leq R$, there exists a parallel evaluation PQ' of F such that:*

1. $c(PQ') \leq c(PQ)$,
2. $\rho(PQ') \leq R$,
3. *PQ' uses k or fewer spills,*
4. *PQ' is load-contiguous, and*
5. *PQ' is store-contiguous.*

An evaluation PQ' is said to be *contiguous* if it is both load-contiguous and store-contiguous. The result of applying this transformation to the parallel evaluation shown in Figure 3 is seen in Figure 4.

6 Cost of a k-spill parallel evaluation

We have shown that given a k-spill parallel evaluation of a computation F that uses R registers, there exists another parallel evaluation of F that is no more expensive, does not use more than R registers or more spills, and is contiguous. We now examine the structure of such a k-spill contiguous evaluation and show that its cost is $\rho(PQ) + 2k + |A(F)|$.

Let PQ be a k-spill, contiguous parallel evaluation of a computation F such that $\rho(PQ) \leq R$, and consider an arbitrary store operation S in PQ_i. We show by contradiction that that $PQ_i = (S, NOP)$ and $PQ_{i+1} = (L, NOP)$ for some load operation, L.

Assume that $PQ_i = (S, A)$. Then $\rho_i(PQ) = \rho_{i-1}(PQ) - 2$. The number of values in registers at a slot can increase by at most 1 in the next slot, so $\rho_{i+1}(PQ) \leq \rho_{i-1}(PQ) - 1 \leq R - 1$. Thus PQ is not store-contiguous, which is a contradiction. We conclude that $PQ_i = (S, NOP)$.

Similarly, assume that $PQ_{i+1} = (L, A)$. Then $\rho_{i+1}(PQ) = \rho_i(PQ)$. Then since $PQ_i = (S, NOP)$, $\rho_i(PQ) = \rho_{i-1}(PQ) - 1 \leq R - 1$. Thus PQ is not store-contiguous, which is a contradiction. Slot $i + 1$ cannot contain an empty load/store slot because this would violate

load-contiguity. We conclude that $PQ_{i+1} = (L, NOP)$. We have thus proven the following lemma.

Lemma 1: *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers, if PQ_i contains a store operation S, then $PQ_i = (S, NOP)$, and $PQ_{i+1} = (L, NOP)$ for some load operation L.*

Consider a store operation in slot i of a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$, and the following store operation, if it exists, in slot j. We show by contradiction that for all m such that $i < m < j$, $\rho_m(PQ) = R$. Assume that for some slot m, $i < m < j$, $\rho_m(PQ) < R$. Since PQ is store-contiguous, $\rho_{i+1}(PQ) = R$. Therefore the number of values in registers decreases by 1 between slots $i+1$ and m. Since there are, by assumption, no stores among these slots, there must be a slot m', $i + 1 < m' < j$ such that $PQ_{m'} = (NOP, A)$. We can assume that there is a load operation after slot j that loads the value stored in slot j. Thus the empty load/store slot m' causes PQ not to be load-contiguous, which is a contradiction. Therefore, for all $i < m < j$, $\rho_m(PQ) = R$. This immediately implies that all such slots m contain a load operation and an arithmetic operation. We have thus proven the following lemma.

Lemma 2: *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers, if PQ_i and PQ_j contain store operations, then for all m such that $i < m < j$, $\rho_m(PQ) = R$ and $PQ_m = (L, A)$ for some load operation L and arithmetic operation A.*

Consider the slot i containing the first store operation S, if one exists, in a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$. We show by contradiction that there are R empty arithmetic slots before slot i. Note that load-contiguity implies that there can be no empty load/store slots before slot i, and since there are no stores there, each must contain a load.

Suppose there are more than R empty arithmetic slots before slot i. Then $\rho_{i-1}(PQ)$ would exceed R, a contradiction. Thus, there cannot be more than R empty arithmetic slots before slot i.

Similarly, if there are fewer than R empty arithmetic slots before slot i, $\rho_{i-1} \leq R - 1$. Then, since $PQ_i = (S, NOP)$, $\rho_i(PQ) \leq R - 2$, and since $PQ_{i+1} = (L, NOP)$, $\rho_{i+1}(PQ) \leq R - 1$, which violates store-contiguity. Thus, there cannot be fewer than R empty arithmetic slots before slot i. The number of empty arithmetic slots before the first store slot must thus exactly equal R. We have therefore proven the following lemma.

Lemma 3: *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers, if the first store operation of PQ is in slot m, then there are exactly R empty arithmetic slots preceding slot m.*

Finally, consider the slot m containing the last store operation S, if one exists, of a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$. We claim that if PQ contains no empty slots, there are no empty arithmetic slots after slot $m + 1$, and show this by contradiction. Note first that there can be no empty load/store slots between slot m and the last load operation in slot t. As argued above, if there is such an empty load/store slot, PQ would not be load-contiguous. Assume that PQ does contain an empty arithmetic slot, m', such that $m + 1 < m'$. If there is more than one such slot, we assume that m' is the first one. Therefore, there is an arithmetic operation in every slot between slot $m + 1$ and m'. Since PQ contains no empty slots, $m' \leq t$. Since

PQ is contiguous, every slot between slots m and m' must contain a load operation. Therefore, $\rho_{m'-1}(PQ) = R$. Since $PQ_{m'} = (L, NOP)$ for some load operation L, $\rho_{m'}(PQ) = R + 1$, which is a contradiction. Therefore, the slot m' cannot exist. We have proven the following theorem.

Lemma 4: *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers and contains no empty slots, if the last store operation of PQ is in slot m, then there are no empty arithmetic slots after slot $m + 1$.*

We have shown that in a contiguous parallel evaluation there are two empty arithmetic slots for each store operation (spill), and, if any stores exist, there are R empty arithmetic slots before the first store operation and no empty arithmetic slots after the last store. If there are no spills in a contiguous parallel evaluation PQ, the number of empty arithmetic slots can be seen to be $\rho(PQ)$. If there is at least one spill, $\rho(PQ) = R$ since PQ is contiguous. Therefore, the cost of a k-spill contiguous evaluation is $\rho(PQ) + 2k + |A(F)|$, as stated in the following theorem.

Theorem 4: *The cost of a k-spill contiguous parallel evaluation PQ of a computation F containing no empty slots, with $\rho(PQ) \leq R$, is $c(PQ) = \rho(PQ) + 2k + |A(F)|$.*

It can be seen that the contiguous evaluation in Figure 4 satisfies Theorem 4 in addition to the preceding lemmas. In the following we only consider parallel evaluations that contain no empty slots.

7 The optimal parallel evaluation

We now describe an algorithm that solves the parallel scheduling problem, given a computation, F, and R available registers. Using the sequential scheduling algorithm, we construct a k-spill sequential evaluation P. Using the transformation of Section 3.3, we construct a k_1-spill parallel evaluation PQ_1. Using the transformation of Section 4, we construct a k_2-spill parallel evaluation PQ_2. Using the transformation of Section 5, we construct a k_3-spill parallel evaluation PQ_3. We show by contradiction that PQ_3 is the solution to the parallel scheduling problem.

Assume that there is some k'-spill parallel evaluation PQ' of F, with $\rho(PQ') \leq R$ and $c(PQ') < c(PQ_3)$. We apply the transformation of Section 4 to PQ', giving a k'_2-spill parallel evaluation PQ'_2. Using the transformation of Section 5, we construct a k'_3-spill parallel evaluation PQ'_3. By Theorems 2 and 3, $k_3 \leq k_2 \leq k_1 = k$, and $k'_3 \leq k'_2 \leq k'$. By Theorem 1, $k \leq k_3$, so $k = k_3$. Similarly, $k \leq k'_3$. We consider the following two cases depending on whether or not $k = 0$.

Case 1: $k > 0$: By Theorems 2 and 3, $c(PQ'_3) \leq c(PQ'_2) \leq c(PQ')$. But by Theorem 4, $c(PQ'_3) = \rho(PQ'_3) + 2k'_3 + |A(F)|$, and $c(PQ_3) = \rho(PQ_3) + 2k + |A(F)|$. Since $k, k' \neq 0$ and PQ_3 and PQ'_3 are store-contiguous, $\rho(PQ_3) = \rho(PQ'_3) = R$. Thus, $c(PQ_3) \leq c(PQ'_3)$, and $c(PQ_3) \leq c(PQ')$, which is a contradiction. So if $k \neq 0$, PQ' cannot exist.

Case 2: $k = 0$: The sequential scheduling algorithm produces the evaluation P such that $\rho(P)$ is minimized, if such an evaluation exists without any spills. In Section 3.2 we show that a parallel evaluation can be transformed into a sequential evaluation without increasing the maximum number of values in registers. Therefore, $\rho(P) \leq \rho(PQ_3)$. If this were not

the case, a sequential evaluation using fewer than $\rho(P)$ registers could be constructed. By Theorem 2, $\rho(PQ_2) \leq \rho(PQ_1) = \rho(P)$. Since PQ_2 contains no spills, it is already store-contiguous. Therefore $\rho(PQ_3) = \rho(PQ_2) \leq \rho(P)$ and $\rho(P) = \rho(PQ_3)$. By Theorem 4, $c(PQ_3) = \rho(PQ_3) + |A(F)|$, and $c(PQ'_3) = \rho(PQ'_3) + |A(F)|$. By assumption, $c(PQ') < c(PQ_3)$, and therefore $c(PQ'_3) < c(PQ_3)$. Then $\rho(PQ'_3) < \rho(PQ_3) = \rho(P)$, which is a contradiction. Therefore, if $k = 0$, PQ' cannot exist.

Thus, we have proven the following theorem.

Theorem 5: *Given a computation F and R available registers, the parallel evaluation PQ_3 that is produced using the algorithm described above is a solution to the parallel scheduling problem.*

It can be shown that the worst-case computational complexity of the parallel scheduling algorithm is $O(nk)$, where n is the number of nodes in F, and k is the number of spills in the optimal sequential evaluation of F. This result is described in the appendix.

8 Conclusions

We have described an $O(nk)$ algorithm that solves the parallel scheduling problem by applying a series of transformations to an optimal sequential evaluation. The cost of the k-spill parallel evaluation PQ found by the algorithm is shown to be $\rho(PQ) + 2k + |A(F)|$. Possible extensions of this work would include dependence graphs whose arithmetic operations can use only one value or more than two, arithmetic and load/store operations with varying latencies, a machine model that allows more than one arithmetic or load/store operation to issue in a slot, and dependence graphs that are DAGs.

References

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23:488–501, 1976.

[2] D. Bernstein, J. Jaffe, and M. Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM Journal of Computing*, 18:1098–1127, 1987.

[3] D. Bernstein, M. Rodeh, and I. Gertner. On the complexity of scheduling problems for parallel/pipeplined machines. *IEEE Transactions on Computers*, 38:1308–1313, 1989.

[4] P. Bose. Optimal code generation for expressions in super scalar machines. In *Proc. Fall Joint Computer Conference*, pages 372–379, 1986.

[5] D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction scheduling for RISCS. In *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[6] J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23:502–510, 1976.

[7] J. R. Goodman and W. C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. 1988 Intl. Conf. on Supercomputing*, pages 442–452, 1988.

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

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

[10] I. Nakata. On compiling algorithms for arithmetic expressions. *Communications of the ACM*, 10:492–494, 1967.

[11] R. R. Redziejowski. On arithmetic expressions and trees. *Communications of the ACM*, 12:81–84, 1969.

[12] R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4:226–248, 1975.

[13] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17:715–728, 1970.

[14] J. D. Ullman. Np-complete scheduling problems. *Journal of Computing Systems and Sciences*, 10:384–393, 1975.

Appendix: Computational complexity

The sequential scheduling algorithm visits every node of the dependence graph once. Therefore if F contains n nodes, the algorithm's worst-case time complexity is $O(n)$.

The transformation of Section 3.3 visits every slot in the sequential evaluation once. A sequential evaluation containing k spills uses $n+2k$ slots. Thus, the transformation's complexity is $O(n+k)$.

The load-contiguous transformation in Section 4 can be implemented with two pointers. The first pointer points to each load operation in turn, starting from the beginning of the evaluation, and the second pointer always points to the latest empty load/store slot that is earlier than the first pointer. In the worst case, each pointer scans the $n+2k$ slots in the parallel evaluation once. Thus, the transformation's complexity is $O(n+k)$.

The store-contiguous transformation in Section 5 can be implemented by considering each store operation in turn, starting from the end of the evaluation. The transformation is repeatedly applied to the same store, moving it one slot later each time, until the transformation can no longer be applied. Applying the transformation once takes constant time. In the worst case a store is moved $n+2k$ slots, from the first slot in the evaluation to the last slot. Since there are k stores, the transformation will take time $nk+2k^2$ in the worst case. Each value can be spilled at most once, so $k \leq n$. Thus, the complexity of the store-contiguous transformation is $O(nk)$. Since k is the minimum number of spills in any parallel evaluation of F, no stores will be deleted by the transformation and the load-contiguous transformation will not need to be reapplied. Empty slots can be deleted from the evaluation in $O(n+k)$ time. Thus, the complexity of the parallel scheduling algorithm is $O(nk)$.