Dual-Issue Scheduling With Spills For Binary Trees

Waleed M. Meleis* Edward S. Davidson†

Abstract

We describe an algorithm that finds a minimum cost schedule, including spill code, for a register-constrained machine that can issue up to one arithmetic operation and one memory access operation at a time, under the restrictions that the dependence graph is a full binary tree, and all operations have unit processing times. This problem is a generalization of two problems whose efficient solutions are well understood: optimal dual-issue scheduling without spills, solved by Bernstein, Jaffe and Rodeh (1987), and optimal single-issue scheduling with spill code, solved by Sethi and Ullman (1970), both assuming a binary dependence graph and a fixed number of registers. We show that the algorithm's 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 cost of an optimal schedule (i.e. its length) 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. We show that this is the cost of all schedules that are in "contiguous form" and therefore that all minimum-spill contiguous form schedules have minimum cost.

1 Introduction

We consider the problem of finding a minimum cost schedule, with spill code, on a machine that can issue one 2-operand arithmetic operation and one memory access operation at a time, provided that the dependence graph is a binary tree and all operations have unit processing times. This problem is a generalization of two problems whose efficient solutions are well understood: optimal dual-issue scheduling without spills, solved by Bernstein, Jaffe and Rodeh [2], and optimal single-issue scheduling with spill code, solved by Sethi and Ullman [14], both assuming a binary dependence graph and a fixed number of registers. In this paper we describe an efficient algorithm to solve the general dual-issue scheduling problem with spill code and completely describe the structure of the optimal schedules.

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 an important way to improve 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. The dual-issue machine assumed in this paper can be used to model the behavior of a wide range of register-limited RISC microprocessors and supercomputers [2].

Many related scheduling problems can be solved efficiently. Minimizing the number of registers needed to schedule operations without spill code on a single-issue machine when the dependence graph is a tree was first solved by Nakata [11] and Redziejowski [12]. These algorithms use a postorder evaluation of the expression tree and execute in time proportional to the number of operations to be scheduled. Sethi and Ullman [14] extended this result to minimize the amount of spill code needed to evaluate an expression tree. This algorithm also uses a postorder traversal of the expression tree and executes in linear time. Aho and Johnson [1] describe a dynamic programming solution to this problem that applies to a wider range of architectures.

When the dependence graph is a directed, acyclic graph (DAG) the related scheduling problems are usually NP-complete. In this case, more than one operation can depend on a particular operation. Minimizing the number of registers needed to schedule operations without spill code on a single-issue machine when the dependence graph is a DAG was shown to be NP-complete by Sethi [13]. Bruno and Sethi [5] show that when the dependence graph is a DAG, minimizing spill code is NP-complete even when only one register is available. Furthermore, the easier problem of inserting the minimum amount of spill code into a fixed schedule is also NP-complete when the underlying dependence graph is a DAG [6, 8]. Heuristics for the spill insertion problem based on dynamic programming and pruning rules are described by Horwitz, Karp, Miller, and Winograd [9] and Hsu, Fischer, and Goodman [10]. Finally, scheduling operations on multiple identical machines was shown to be NP-complete by Ullman [15] when the dependence graph is a DAG, and by Bernstein, Rodeh, and Gertner [3] when the dependence graph is a tree.

2 The Machine/Computation Model

The notation used by Bernstein, Jaffe, and Rodeh in [2] formalizes the sequential and parallel scheduling

*Department of Electrical and Computer Engineering, Northeastern University, Boston MA, 02115 (meleis@ece.neu.edu).

†Department of Electrical Engineering and Computer Science, Ann Arbor MI, 48105 (davidson@eecs.umich.edu).

problems without spills. We extend 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.

The dependence graph F is a rooted binary tree that contains leaf nodes that represent load operations and internal nodes that represent arithmetic operations. We assume that every internal node has exactly two children. An edge between two nodes represents a data dependence between the two operations, and an operation must be scheduled before its children. 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*). An arithmetic operation is said to *use* a value that it directly depends on. 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* by 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 at time 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 processing time of each arithmetic and load operation is one. 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 nonleaf node P_i is used by P_j $(j > i)$, then A_v may be stored at time k_1, and loaded at time k_2, provided that $i < k_1 < k_2 < j$. In this case, $P_{k_1} = S_v$ and $P_{k_2} = L_v$, and S_v and L_v are spill code.

Conditions 2 and 3 are collectively referred to as the *precedence constraints* of a sequential evaluation. The cost of P, $c(P)$, equals 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 some time. A parallel evaluation PQ must satisfy the following four conditions:

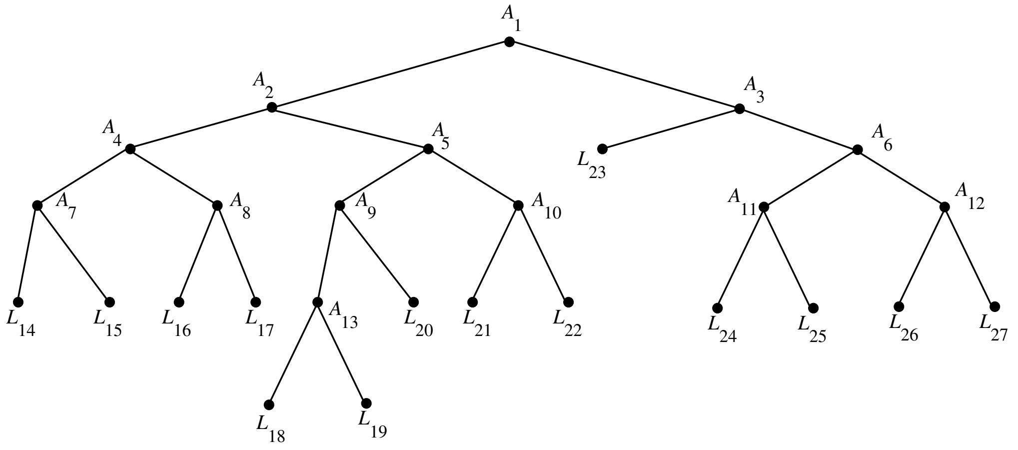
1. For every nonleaf node v, there exists a unique PQ_j such that $q_j = v$.
2. For every leaf node v, there exists a unique PQ_j such that $p_j = v$.
3. For every PQ_i, PQ_j, if $q_i \neq$ NOP, then $p_j \in F_{q_i}$ implies $j < i$, and $q_j \in F_{q_i}$ implies $j \leq i$. (once again, processing times are assumed to be one)
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 at time k_1, and loaded at time 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 cost of PQ, $c(PQ)$, equals the number of operations pairs in PQ.

Sequential and parallel evaluations defined in this way ensure that every operation is executed, the operations' unit processing time is not violated, and spills (if any) are properly placed in the evaluation. The limitations imposed by a register file of finite size are described below. The concatenation $P|P'$ (or simply PP') of two sequential evaluations represents the sequential evaluation formed by the operations in P followed by the operations in P'.

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



(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
$\vert inreg\vert_i$ =	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
$\vert inreg\vert_i$ =	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	–	–
$\vert inreg\vert_i$ =	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
$\vert inreg\vert_i$ =	3	2	3	2	3	2	3	2	1

(c) A parallel evaluation of F

Figure 1: Examples of sequential and parallel evaluations.

operations at time i or earlier. The number of values in registers at time i of an evaluation E, $|\text{inreg}_i|$, is denoted $\rho_i(E)$, and $\rho(E)$ denotes the maximum over all i of $\rho_i(E)$. That is, $\rho_i(E) = |\mathit{inreg}_i|$ and $\rho(E) = \max_i \rho_i(E)$. 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*.

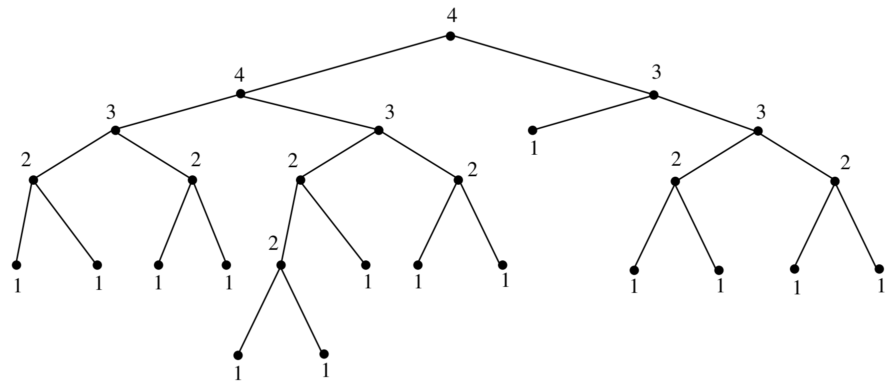


Figure 2: Value of M labels for Figure 1(a).

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 be transformed into a minimum 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 k-spill parallel evaluation can cost less.

3.1 Minimizing spills in a sequential evaluation

Efficient algorithms that solve a variant of the sequential scheduling problem are described by Aho, Sethi, and Ullman [1], and Sethi and Ullman [14]. In addition to the three types of operations described above, these algorithms include another 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 $inreg_i$. Bose [4] describes an adaptation of the algorithm in [1] that solves the sequential scheduling problem without this fourth instruction class. The algorithm, described below, 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 and compute the label $M(v)$ for each vertex v using the following labeling algorithm:

Step 1. `If` $v \in L(F)$: $M(v) = 1$.
Step 2. `If` $v \in A(F)$ `with children` v_1, v_2:
  `a) if` $M(v_1) \neq M(v_2)$,
    $M(v) = max(M(v_1), M(v_2))$
  `b) if` $M(v_1) = M(v_2)$, $M(v) = M(v_1) + 1$

Apply *Schedule*, described below, to the root vertex of F to produce the sequential evaluation. `Compute` v schedules operation v at the next available time. Similarly, `Store` v and "Load v" schedule a store or a load, respectively, of value v at the next available time.

```
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) ≥ R and 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
```

For a given vertex v, the labeling routine computes $M(v)$, the minimum number of registers needed to evaluate the subtree rooted at v without spill code. The scheduling routine uses the labels to optimally schedule the entire tree. The values of the labels for Figure 1(a) are shown in Figure 2. Spill code (i.e. the loads and stores of v_1 in *Schedule*) is inserted when neither subtree can be scheduled using R or fewer registers. The operations that compute a particular subtree are executed consecutively. That is, the computation of a subtree is never interrupted by the computation of another subtree. The subtree with the higher register requirement at a particular vertex is always executed first. As argued in [4], the sequential scheduling algorithm finds a minimum cost 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 [14].

The cost of a sequential evaluation 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 this 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. Recall that for a parallel evaluation PQ and a time i, $PQ_i = (p_i, q_i)$.

Serial Transformation. Given a k-spill parallel evaluation PQ that uses R registers, we construct a serial evaluation P as follows. For each time i, let $J_i = q_i p_i$. 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 $\rho(P) \le \rho(PQ)$ (Lemma 3.1).

If operation v is scheduled at an earlier time than operation w in parallel evaluation PQ, v will be scheduled at an earlier time than w after PQ is transformed into P. Therefore these operations cannot violate the precedence constraints in the serial evaluation P. If operations v and w are scheduled at the same time in PQ then by definition of a parallel evaluation they are not dependent on one another. Therefore these operations cannot violate the precedence constraints in P, and it follows that P represents a sequential evaluation. We can therefore state the following lemma without proof.

LEMMA 3.1. *Given a k-spill parallel evaluation PQ that uses R registers, and the serial evaluation P formed by applying the serial transformation to PQ, $\rho(P) \le \rho(PQ)$.*

Since P is a sequential evaluation of F that contains the same non-NOP operations as PQ, evaluations P and PQ contain the same number of spills. We can now prove the following theorem.

THEOREM 3.1.. *Given a computation F and a number of available registers R, if an optimal sequential evaluation P of F that uses R registers contains k spills, then no parallel evaluation of F that uses R registers contains fewer than k spills.*

Proof. 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 serial transformation 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. Therefore, 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.

Parallel Transformation. Given a k-spill sequential evaluation $P = P_1, ..., P_n$, then for each time i:

1. if $P_i \in A(F)$, let $PQ_i =$ (NOP,P_i), and
2. if $P_i \in L(F) \cup S(F)$, let $PQ_i = (P_i$,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 scheduled in PQ for every time i; thus PQ is unlikely to have the minimum cost. The result of applying the parallel transformation to the sequential evaluation in Figure 1b is shown in Figure 3.

4 Load-contiguous parallel evaluations

We have shown that a parallel evaluation with the minimum number of spills can be constructed by applying the parallel transformation to the optimal sequential evaluation. We now describe the process of transforming this parallel evaluation into a contiguous 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 registers, or the number of spills. We will then show in §5 that a load-contiguous evaluation can be made contiguous, i.e. both load-contiguous and store-contiguous, without increasing its cost, the maximum number of values in registers, or the number of spills. Finally, in §§6-7 we show that a minimum-spill contiguous evaluation is necessarily optimal. That is, applying the contiguity transformations to a minimum-

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
$\lvert inreg\rvert_i=$	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
$\lvert inreg\rvert_i=$	0	1	2	1	2	3	2	1	2	1	2	1

Figure 3: The result of transforming the sequential evaluation in Figure 1(b) into a parallel evaluation.

spill parallel evaluation produces a minimum cost evaluation.

A parallel evaluation of a computation F is said to be *load-contiguous* if and only if no load operation is scheduled at a later time than an empty load/store slot. Using this definition we can summarize the proof of the following theorem.

THEOREM 4.1. **(Load contiguous transformation).** *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.*

Proof. Let PQ be a k-spill parallel evaluation of a computation F that uses R registers, but is not load-contiguous. Consider the first empty load/store slot i. Let j be the first time after i where a load operation is scheduled. We create an evaluation PQ' that is the same as PQ except either

1. the empty slot at time i has been deleted,
2. the load operation at time j has been moved to i, or
3. the load operation at time 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. We can also show that $\rho(PQ') \leq \rho(PQ)$ and that precedence is preserved.

The result of applying this transformation to the parallel evaluation in Figure 3 is shown in Figure 4. We now extend the notion of load-contiguity to stores.

5 Store-contiguous parallel evaluations

We define a store-contiguous parallel evaluation and show that every load-contiguous parallel evaluation can be made store-contiguous as well. Given R available registers, a parallel evaluation of a computation F is *store-contiguous* if for any i such that a store operation is scheduled at time i, $\rho_{i+1} = R$. Using this definition we can summarize the proof of the following theorem.

THEOREM 5.1. **(Store contiguous transformation).** *Given a k-spill, load-contiguous, parallel evaluation PQ of a computation F with $\rho(PQ) \leq R$, 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.*

Proof. Let PQ be a load-contiguous, k-spill parallel evaluation of a computation F that uses R registers, but is not store-contiguous. We show that PQ can be transformed into an evaluation that is both load-contiguous and store-contiguous without increasing the evaluation's cost or the number of spills, and without causing the maximum number of values in registers to exceed R. We begin by showing that the last store operation in the schedule that violates the store-contiguous property cannot immediately precede another store operation. We then show that the parallel evaluation can be transformed so that the store operation no longer violates the store-contiguous property. Finally we show that both load-contiguity and store-contiguity can be achieved for the entire schedule.

Consider the largest time i such that a store operation is scheduled at time i and $\rho_{i+1}(PQ) < R$. We

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
$\lvert inreg \rvert_i$ =	1	2	2	2	3	3	3	2	2	3	3	3	3	2	2	3	3	3	2	1

Figure 4: The result of applying the load-contiguous transformation to the parallel evaluation in Figure 3 with $R = 3$.

show by contradiction that there cannot be a store operation scheduled at time $i + 1$. Suppose there is a store operation scheduled at time $i + 1$. Then $\rho_{i+1}(PQ) < \rho_i(PQ)$. Furthermore, since a store operation is scheduled at time i, $\rho_i(PQ) < \rho_{i-1}(PQ)$. Thus $\rho_{i+1}(PQ) \leq \rho_{i-1}(PQ) - 2$. The number of values in registers at any time can increase by at most one in the next time period, which occurs when the next time period 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 scheduled at time $i + 1$ contradicts the assumption that the store scheduled at time i is the last store that precedes a time period with fewer than R values in registers. Therefore there is not a store operation scheduled at time $i + 1$.

We now show that since $\rho_{i+1}(PQ) < R$, we can construct a new evaluation PQ' in which the store scheduled at time i is moved one time period later without increasing the evaluation's cost, the number of spills, or the maximum number of values in registers. There are six cases depending on the operations scheduled at times i and $i+1$. In each case we can show that the new evaluation does not violate the precedence constraints. That is, we show that no values are used before they are defined, and that values are spilled only after being defined and are reloaded before being used. Furthermore the cost of the new evaluation does not exceed that of the original evaluation, and the maximum number of values in registers is not increased.

This transformation can be applied repeatedly to the same store, moving it one time period later in each case, until either it does precede a time period 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 is of finite length, each store will eventually either precede a time period 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, the last store to violate the store-contiguous property cannot immediately precede another store operation. Since there are finitely many stores, the sequence of transformations will eventually terminate with each remaining store preceding a time period 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 contiguous transformations that move a load interchange the load with a store. Thus they preserve load-contiguity. But 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-contiguous transformation can be reapplied. At worst, the load-contiguous and store-contiguous transformations are reapplied k times, i.e. each time a spill is eliminated.

An evaluation PQ' is said to be *contiguous* if it is both load-contiguous and store-contiguous, i.e. if no load operation is scheduled in a later time period than an empty load/store slot, and for any i such that a store operation is scheduled at time i, $\rho_{i+1} = R$. The result of applying this transformation to the load-contiguous parallel evaluation in Figure 4 is shown in Figure 5.

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 costs no more, uses no more than R registers and no more than k spills, and is contiguous. We now examine the structure of such a k-spill contiguous evaluation with a sequence of lemmas, given without proof, and show in Theorem 6.1 that its cost is $\rho(PQ) + 2k + |A(F)|$.

Lemma 6.1. **(Each store operation and the following operation).** *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.*

Lemma 6.2. **(Operations between two successive store operations).** *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers, if $PQ_i = (S, NOP)$ for some store operation S, and the following store operation exists and is scheduled at time*

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 5: The result of applying the store-contiguous transformation to the load-contiguous evaluation in Figure 4.

$j, j > i$, then for all m, $i+1 < m < j$, PQ_m is of the form (L, A) and $\rho_m(PQ) = R$.

LEMMA 6.3. **(Operations before the first store operation).** *Given a k-spill, contiguous parallel evaluation PQ of a computation F that uses R registers, if the first store operation of PQ is scheduled at time m, then a load operation is scheduled at each time preceding time m, and exactly R empty arithmetic slots precede time m.*

LEMMA 6.4. **(Operations after the last store operation).** *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 scheduled at time m, then there are no empty arithmetic slots after time $m+1$.*

THEOREM 6.1. **(Cost of a contiguous parallel evaluation).** *A k-spill contiguous parallel evaluation PQ of a computation F with no empty slots and with $\rho(PQ) \leq R$, has cost $c(PQ) = \rho(PQ) + 2k + |A(F)|$.*

Proof. In a contiguous parallel evaluation there are two empty arithmetic slots for each store operation (spill), namely the slot with the store operation and the one immediately after. Furthermore there are no other empty arithmetic slots between a pair of store operations. If any stores exist, there are R empty arithmetic slots before the first store operation and, assuming that there are no empty slots, there are no empty arithmetic slots after the last store, except for the slot immediately following the last store which is of the form (L, NOP). If there is at least one spill, $\rho(PQ) = R$ since PQ is contiguous. Therefore, the cost of a k-spill contiguous evaluation, where $k \geq 1$, is the sum of the arithmetic slots plus the empty arithmetic slots, i.e. $\rho(PQ) + 2k + |A(F)|$.

As a consequence of Theorem 6.1 and the preceding lemmas, a k-spill contiguous evaluation that uses R registers has the following general form. The first store operation, if it exists, is preceded by a series of operation pairs of the form (L, A) interspersed with R operation pairs of the form (L, NOP). Each of the k store operations is scheduled in an empty arithmetic slot, (S, NOP), and is followed by an operation pair of the form (L, NOP). Such pairs of load/store operations are followed by a series of operation pairs of the form (L, A). The evaluation ends with a series of operation pairs of the form (NOP, A). It can be seen that the contiguous evaluation in Figure 5 satisfies Lemmas 6.1, 6.2, 6.3, and 6.4 and Theorem 6.1.

In the following discussion, since empty slots serve no useful purpose and increase the evaluation cost, 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 parallel transformation of §3.3, we construct the parallel evaluation PQ_1. Using the load contiguous transformation of §4, we construct the parallel evaluation PQ_2. Using the store contiguous transformation of §5, we construct the parallel evaluation PQ_3 which is optimal by the following theorem, which we state without proof.

THEOREM 7.1.. *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.*

8 Computational complexity

We show in this section 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 schedule.

The labeling routine of the sequential scheduling algorithm performs a depth-first traversal of the dependence graph and does a constant amount of work at each vertex. The worst-case complexity of a depth-first traversal of a binary tree is $O(n)$ [7]. Similarly, the worst-case complexity of the scheduling routine is $O(n)$.

The parallel transformation of §3.3 visits every time period in the sequential evaluation once and does a constant amount of work per visit. A sequential evaluation containing k spills is of length $n + 2k$, so the transformation's worst-case complexity is $O(n+k)$.

The load-contiguous transformation in §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 $n+2k$ time periods in the parallel evaluation once and is bounded by some constant time per time period. Thus, the transformation's complexity is $O(n+k)$.

The store-contiguous transformation in §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 time period later each time, until the transformation can no longer be applied. Applying the transformation once takes constant time. No store is moved more than $n+2k$ time periods, i.e. from the beginning of the schedule to the end. 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 worst-case complexity of the parallel scheduling algorithm is $O(nk)$.

9 Summary

We have presented an $O(nk)$ algorithm that solves the parallel scheduling problem by applying a series of transformations to an optimal sequential evaluation. We have shown that only contiguous evaluations need to be considered in searching for optimal parallel evaluations, and that optimal serial and optimal parallel evaluations have the same number of spills. We have fully described the structure of contiguous parallel evaluations and shown that the cost of the k-spill parallel evaluation PQ found by the algorithm is $\rho(PQ)+2k+|A(F)|$.

References

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

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

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

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

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

[6] M. Carlisle, *On local register allocation*, Thesis, Computer Science Department, University of Delaware, 1991.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.

[8] M. Farach and V. Liberatore, *On local register allocation*, in ACM-SIAM Symposium on Discrete Algorithms, 1998.

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

[10] 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 (1989), pp. 1252–1260.

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

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

[13] R. Sethi, *Complete register allocation problems*, SIAM Journal on Computing, 4 (1975), pp. 226–248.

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

[15] J. D. Ullman, *NP-complete scheduling problems*, Journal of Computing Systems and Sciences, 10 (1975), pp. 384–393.