SIAM J. COMPUT.
Vol. 30, No. 6, pp. 1921–1941
© 2001 Society for Industrial and Applied Mathematics

DUAL-ISSUE SCHEDULING FOR BINARY TREES WITH SPILLS AND PIPELINED LOADS*

WALEED M. MELEIS†

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, all arithmetic and store operations have unit latency, and all load operations have a latency of 1 or all load operations have a latency of 2. This problem is a generalization of two problems whose efficient solutions are well understood: optimal dual-issue scheduling without spills for binary expression trees, solved by Bernstein, Jaffe, and Rodeh [*SIAM J. Comput.*, 18 (1989), pp. 1098–1127], and optimal single-issue scheduling with spill code and delayed loads, solved by Kurlander, Proebsting, and Fischer [*ACM Transactions on Programming Languages and Systems*, 17 (1995), pp. 740–776], both assuming 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 a "contiguous" schedule (i.e., its length) is shown to be $\rho + 2k + g + |A|$, where ρ is the number of registers used, $|A|$ is the number of arithmetic operations, k is the number of spills, and g is the number of empty slots in the associated single processor schedule. Therefore all contiguous schedules formed from optimal single processor schedules have minimum cost.

Key words. scheduling algorithms, code generations, parallel functional units, registers

AMS subject classifications. 68Q20, 68Q22, 68N20, 68R05, 90B35

PII. S009753979834610X

1. Introduction.

In modern processors, memory bandwidth is a major performance bottleneck due to increased processor issue widths 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 critical to improve system performance. In addition, the latency of pipelined memory operations is often larger than that of most arithmetic operations. One source of memory traffic is spilling, which moves data values from registers to memory and back. This data movement is necessary because the number of registers that instructions can access is limited. Reducing this traffic can be achieved through better register allocation and register sensitive scheduling and through the judicious use of spill code to minimize performance degradation. Processors that can issue more than one instruction per cycle can hide the latency of spill operations and other memory operations by overlapping them with instructions that do useful work.

Many scheduling problems for related processor architectures 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 [30] and Redziejowski [34]. 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 [36] 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 John-

*Received by the editors October 26, 1998; accepted for publication (in revised form) September 25, 2000; published electronically March 20, 2001. A preliminary version of this paper appeared in the Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms.
http://www.siam.org/journals/sicomp/30-6/34610.html
†Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 (meleis@ece.neu.edu).

son [1] describe a dynamic programming solution to this problem that applies to a wider range of architectures. While the preceding results only apply to operations with unit latencies, Kurlander, Proebsting, and Fischer [28] describe an efficient algorithm for optimally scheduling expressions trees, with spills, for a delayed load architecture. This architecture assumes that all load operations have a latency of two cycles.

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 [35]. Bruno and Sethi [11] show that when the dependence graph is a DAG, minimizing spill code is NP-complete even when only one register is available. This result assumes an accumulator-based architecture which is different than the load/store architecture we consider. 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 [13, 19]. Scheduling operations with arbitrary latencies on multiple identical machines was shown by Ullman to be NP-complete when the dependence graph is a DAG [37], by Bernstein, Rodeh, and Gertner when the dependence graph is a tree [5], and by Palem and Simons when the dependence graph is a collection of chains [31].

However, a number of related problems with unit-time operations can be solved efficiently in the absence of register constraints. If all latencies equal one, polynomial algorithms are described by Coffman and Graham that schedule operations on two processors when the dependence graph is a DAG [15] and by Hu for arbitrarily many processors when the dependence graph is a tree [27]. If all latencies are equal and the dependence graph is a tree, Bruno, Jones, and So give an efficient algorithm to schedule operations on multiple processors [10]. Finally, if all latencies equal 1 or 2, Bernstein and Gertner describe an efficient algorithm to schedule a single processor with arbitrary precedence [3].

Heuristics for the spill insertion problem based on dynamic programming and pruning rules are described by Horwitz et al. [25] and Hsu, Fischer, and Goodman [26]. More practical heuristics for register allocation and spill insertion are described and evaluated by Chaitin et al. [17, 18] and Chow and Hennessy [14]. More recently, improved allocation algorithms were evaluated by Callahan and Koblenz [12], Pinter [32], Briggs et al. [8, 9], Proebsting and Fischer [33], George and Appel [20], Lueh and Gross [29], and Bergner et al. [2]. These algorithms attempt to minimize the number of spills within a basic block or over an entire program, but without explicitly handling pipelines or instruction-level parallelism. The interdependency of code scheduling and register allocation in practical applications was described in [7] and [22]. These papers address the problem of efficient code generation for basic blocks on single-issue RISC processors and document the tradeoffs that exist between good data reuse in the register file and the minimization of pipeline interlocks. Other scheduling algorithms for pipelined processors were described by Hennessy and Gross [23, 24] and by Gibbons and Muchnick [21].

The algorithms described in this paper are the first to simultaneously optimize a schedule in the presence of multiple processors, and either spill code or instruction pipelines. (In fact, we solve the scheduling problem in the presence of both spills and instruction pipelines.) These results therefore increase the space of scheduling problems that can be solved efficiently along these important dimensions. Bernstein, Jaffe, and Rodeh [4] first described an efficient algorithm to schedule a dual-issue

processor with a limit on the number of available registers. They show that an optimal dual-issue schedule can be derived from an optimal single-issue schedule. However, this result does not apply when spill code must be inserted and the authors state that "... more work concerning store operations is needed."

In this paper we describe an efficient optimal scheduling algorithm that allocates registers and inserts spill code for a dual-issue processor with pipelined memory access operations. We show that an optimal single-issue schedule can be transformed into an optimal dual-issue schedule without increasing the number of spill operations. We then show that this transformation can be further generalized to apply to processors with pipelined memory operations, and we completely describe the structure of an optimal solution to this general problem. The dual-issue machine assumed in this paper is a simplified representation of a range of pipelined RISC microprocessors and supercomputers that have a limited number of registers. By solving the problem of scheduling with spills for a dual-issue processor with pipelined memory operations, we have solved the following two previously open subproblems: optimal scheduling for a dual-issue processor with spills and optimal scheduling for a dual-issue processor without spills with pipelined memory operations. Our main result represents a generalization of optimal algorithms described by Sethi and Ullman [36] and Bernstein, Jaffe, and Rodeh [4]. This result also represents an extension of the algorithm described by Kurlander, Proebsting, and Fischer [28]; however, their machine model also includes unary operations and literals which our algorithm does not handle. We summarize the main results of this paper below.

1. A sequential schedule with arbitrary load latencies and spills can be transformed into a dual-issue schedule with the same number of spills and idle cycles, and visa versa.

2. A dual-issue schedule with arbitrary load latencies and spills can be transformed into a contiguous dual-issue schedule by moving load operations earlier in the schedule and store operations later, without increasing the schedule's length or the number of spill operations.

3. We give an efficient algorithm that solves the dual-issue scheduling problem with spills for load latencies = 1 or load latencies = 2.

Our paper is organized as follows. In section 2 we describe our machine and computation model and define the sequential and parallel scheduling problems. In section 3 we describe some basic properties of parallel schedules. In sections 4 and 5 we show that we only need to consider contiguous schedules. In section 6 we describe the structure and cost of a contiguous schedule, and in section 7 we describe an algorithm that solves the parallel scheduling problem.

2. The machine/computation model. The notation used by Bernstein, Jaffe, and Rodeh in [4] formalizes the sequential and parallel scheduling problems without spills. We extend this notation to allow the use of spill code and nonunit latencies for loads. We consider a machine model that has access to R registers: $r_1\ r_2, \ldots, r_R$ 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. Arithmetic and store operations have unit latencies, and each load operation has a latency that is some positive integer. In the definitions below, we assume that an operation's latency gives the minimum number of cycles that must elapse before a dependent operation is scheduled. We also assume that all functional units are fully

pipelined, which allows an operation to be issued to each functional unit in each cycle. We let NOP indicate that no operation is scheduled at some time.

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 predecessors. An edge between two nodes represents a data dependence between the two operations, and an operation must be scheduled after its predecessors. The set of *arithmetic operations* in F is denoted $A(F) = \{A_1, A_2, \ldots, A_{|A(F)|}\}$, 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 $L(F) = \{L_1, L_2, \ldots, L_{|L(F)|}\}$, 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 latency of load operation L_i is denoted $latency(L_i)$. The set of *store operations* is denoted $S(F) = \{S_1, S_2, \ldots, S_{|A(F)|}\}$, 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 evaluation 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 predecessor 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 and NOPs): $P = P_1, P_2, \ldots, P_n$, 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, and P_j is an arithmetic operation, then $P_j \in F_{P_i}$ implies $j \leq i - 1$. If P_j is a load operation, then $P_j \in F_{Pi}$ implies $j \leq i - latency(P_j)$. This implies that the latency of each arithmetic operation is one, and the latency of each load operation P_j is $latency(P_j)$.

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 $i < k_1 < k_2 \leq j - latency(P_{k_2})$. 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. An operation $P_i = NOP$ is called an *empty slot* of a sequential evaluation. The cost of P, $c(P)$, equals the number of operations in P, i.e., we define an evaluation's cost to be equal to its length.

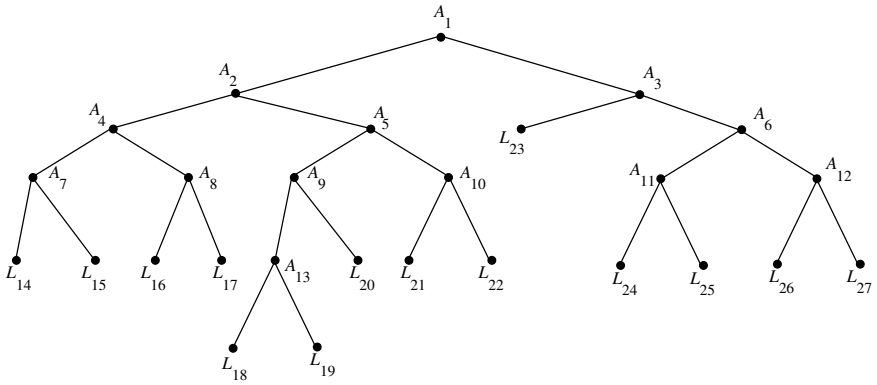
We now extend the definitions in [4] to handle dual-issue processors. A *parallel evaluation* PQ of a computation F is an ordering of operation pairs: $PQ = PQ_1, PQ_2, \ldots, PQ_n$ where $PQ_i = (p_i, q_i)$ such that $p_i \in L(F) \cup S(F) \cup \{NOP\}$ and $q_i \in A(F) \cup \{NOP\}$. NOPs are used 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 $j \leq i - latency(q_i)$. Furthermore, $q_j \in F_{q_i}$ implies $j \leq i - 1$. Once again, the latency of each arithmetic operation is one, and the latency of load operation q_i is $latency(q_i)$.

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 $i < k_1 < k_2 \leq j - latency(k_2)$. In this case, $PQ_{k_1} = (S_v, q_{k_1})$ and $PQ_{k_2} = (L_v, q_{k_2})$ are spill code.



(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	20
$P_i =$	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}	–
$\rho_i =$	1	2	2	1	2	2	1	2	3	3	2	1	0	1	2	2	1	2	3	3

Slot $i =$	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
$P_i =$	A_8	A_4	L_5	–	A_2	S_2	L_{24}	L_{25}	–	A_{11}	L_{26}	L_{27}	–	A_{12}	A_6	L_{23}	–	A_3	L_2	–	A_1
$\rho_i =$	2	1	2	2	1	0	1	2	2	1	2	3	3	2	1	2	2	1	2	2	1

(b) *A serial evaluation of F.*

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_{14}	L_{15}	–	L_{16}	L_{17}	–	–	L_{18}	L_{19}	–	L_{20}	–	L_{21}	S_4	L_{22}	–	–	–	L_4	–
$q_i =$	–	–	–	A_7	–	–	A_8	A_4	–	–	A_{13}	–	A_9	–	–	–	A_{10}	A_5	–	–
$\rho_i =$	1	2	2	2	3	3	2	2	3	3	3	3	3	2	3	3	2	1	2	2

Slot $i =$	21	22	23	24	25	26	27	28	29	30	31	32	33
$p_i =$	L_{24}	L_{25}	–	L_{26}	S_{11}	L_{27}	–	L_{11}	–	L_{23}	–	–	–
$q_i =$	A_2	–	–	A_{11}	–	–	–	A_{12}	–	A_6	–	A_3	A_1
$\rho_i =$	2	3	3	3	2	3	3	3	3	3	3	2	1

(c) *A parallel evaluation of F.*

FIG. 1. *Examples of sequential and parallel evaluations for latency $= 2$ and $R = 3$.*

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* if p_i is not a NOP, and (NOP, q_i) is called an *empty load/store slot* if q_i is not a NOP. A pair (NOP, NOP) is called an *empty slot* of a parallel evaluation. Note that by definition the sets of empty arithmetic slots, empty load/store slots, and empty slots in an evaluation are disjoint. The cost of PQ, $c(PQ)$, equals the number of operation pairs in PQ, and note that we again define an evaluation's cost to be equal to its length.

Sequential and parallel evaluations defined in this way ensure that every operation is executed, the operations' latencies are 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 operations at time i or earlier. The number of values in registers at time i of an evaluation E, $|inreg_i|$, is denoted $\rho_i(E)$, and $\rho(E)$ denotes the maximum over all i of $\rho_i(E)$. That is, $\rho_i(E) = |inreg_i|$ and $\rho(E) = \max_i \rho_i(E)$. When all load latencies are equal, we let $latency = latency(L_i)$. An example is shown in Figure 1 in which $latency = 2$ and $R = 3$.

Given a computation F, the number of available registers R, and the latencies of the load operations, 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*. In this paper we describe algorithms that solve the parallel scheduling problem for $latency = 1$ and $latency = 2$.

3. Minimizing spill operations plus empty slots in a parallel evaluation. Given a tree F, the number of available registers R, and the latencies of the load operations, a solution to the parallel scheduling problem is found by using the sequential scheduling algorithm to construct a sequential evaluation that minimizes the number of spill operations plus empty slots, transforming this evaluation into a parallel evaluation, and then transforming this evaluation into a "contiguous" form. At each stage of the transformation the number of spill operations plus empty slots remains unchanged and the cost of the evaluation does not increase. In this section we show that for arbitrary load latencies, no parallel evaluation of F can have fewer spill operations plus empty slots than an optimal sequential evaluation of F, and that a minimum cost sequential evaluation can be transformed into a parallel evaluation that minimizes the number of spill operations plus empty slots. In the following sections, we show that every parallel evaluation PQ of F in contiguous form that contains k spilled values and g empty slots has a cost of $\rho(PQ) + 2k + g + |A(F)|$, and, for $latency = 1$ or $latency = 2$, that no parallel evaluation can cost less.

3.1. Minimizing spills in a sequential evaluation. An efficient algorithm that solves the sequential scheduling problem with $latency = 1$ was described by Aho, Sethi, and Ullman [1] and Sethi and Ullman [36]. 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 [6] describes an adaptation of the algorithm in [1] that solves the sequential scheduling problem without this fourth instruction class.

For a given vertex v, the algorithm first computes the minimum number of registers needed to evaluate the subtree rooted at v without spill code. The scheduling routine uses these quantities to optimally schedule the entire tree. Spill code 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 [6], this sequential scheduling algorithm finds a minimum cost evaluation of F that uses no more than R registers assuming $latency = 1$. 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 [36].

An efficient algorithm to solve the sequential scheduling problem with $latency = 2$ was described by Kurlander, Proebsting, and Fischer [28]. They show that all nontrivial trees can be scheduled without empty slots or spill code if the number of available registers is at least one more than the minimum number of registers needed to evaluate the tree without spills. In this case the sequential scheduling algorithm for $latency = 1$ is first used to create an evaluation, and the ordering of the load operations and the ordering of the arithmetic operations becomes an invariant. The new evaluation then consists of a series of load operations, followed by an alternating series of arithmetic and load operations, and ending with a series of arithmetic operations. This algorithm therefore solves the sequential scheduling problem with $latency = 2$ for sufficiently large values of R.

For smaller values of R, spill code may need to be inserted in the evaluation. For each vertex v, the algorithm computes the *register pressure* caused by v by applying the unit load latency sequential scheduling algorithm to the subtree rooted at v. The register pressure caused by v is equal to the number of times during this evaluation that the number of live values equals the maximum number of live values. A vertex v is then spilled if the minimum number of registers needed to evaluate the subtree rooted at v equals R and v's register pressure is greater than two. If the minimum number of registers needed to evaluate v is greater than R, then a predecessor of v with the largest register pressure is spilled.

The cost of a sequential evaluation equals the number of nodes in F plus the number of spill operations and empty slots. Therefore an optimal sequential evaluation must contain the fewest spill operations plus empty slots among all sequential evaluations. Furthermore, if k values are spilled, there are $2k$ spill operations, i.e., k store/load pairs.

3.2. A lower bound on spill operations plus empty slots in a parallel evaluation. A simple transformation allows us to transform a k-spill, g-empty slot parallel evaluation with arbitrary load latencies and that uses R registers into a k-spill, g-empty slot 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 with arbitrary load latencies that uses R registers, we construct a serial evaluation P as follows. For each time i, if $PQ_i = (L, A)$ for some load L and arithmetic operation A, let $J_i = A, L$. Similarly, if $PQ_i = (L, NOP)$, (S, NOP), or (NOP, A), let $J_i = L$, S, or A, respectively. If $PQ_i = (S, A)$, let $J_i = A, S$. Finally if $PQ_i = (NOP, NOP)$, let $J_i = NOP$. Then let $P' = J_1 J_2 \dots J_{|PQ|}$. In [4] a similar transformation is applied to parallel evaluations that do not contain spills and with $latency = 1$. We first show that when P is constructed in this manner it represents a sequential evaluation. Then we show that $\rho(P) \leq \rho(PQ)$ (Lemma 3.1).

If two operations v and w are scheduled $d \geq 1$ cycles apart in parallel evaluation PQ, these operations are scheduled d or more cycles apart 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.

LEMMA 3.1. *Given a k-spill parallel evaluation PQ of a computation F with arbitrary load latencies and R available registers, and given the serial evaluation P formed by applying the serial transformation to PQ, then $\rho(P) \leq \rho(PQ)$.*

Proof. Consider some arbitrary time j of the evaluation P. Suppose first that P_j contains an arithmetic operation, i.e., $P_j = q_i$ for some time i of PQ. By definition of the serial transformation, all operations scheduled earlier than q_i in P must have been scheduled at time $i-1$ or earlier in PQ. Therefore, the number of values in registers at time $j-1$ of P and at time $i-1$ of PQ is the same, i.e., $\rho_{j-1}(P) = \rho_{i-1}(PQ)$. Since q_i is an arithmetic operation, $\rho_j(P) = \rho_{j-1}(P) - 1 \leq \rho_{j-1}(P)$, which implies that $\rho_j(P) \leq \rho_{i-1}(PQ)$.

Now suppose that P_j contains a load/store operation, i.e., $P_j = p_i$ for some time i of PQ. By definition of the serial transformation, all operations scheduled at time j or earlier in P must have been scheduled at time i or earlier in PQ. Therefore, $\rho_j(P) \leq \rho_i(PQ)$. Finally if P_j contains a NOP, then $\rho_j(P) \leq \rho_{j-1}(P)$. □

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. Furthermore, by construction both evaluations contain the same number of empty slots. We can now state the following lemmas whose proofs follow from the discussion above.

LEMMA 3.2. *Consider a computation F with arbitrary load latencies and R available registers. If an optimal sequential evaluation P of F that uses R registers contains $2k$ spill operations and g empty slots, then the number of spill operations plus empty slots in any parallel evaluation of F that uses R registers is not less than $2k+g$.*

LEMMA 3.3. *Consider a computation F with arbitrary load latencies and R available registers, and let minreg be the minimum number of registers needed to evaluate F without spills in a serial evaluation. Then all parallel evaluations of F that use fewer than minreg registers have at least one spill.*

3.3. Creating a minimum-spill operation plus empty slot parallel evaluation. An optimal serial evaluation of F thus directly yields a lower bound $2k+g$ on the number of spill operations plus empty slots in any parallel evaluation of F. We can achieve this lower bound through a straightforward transformation of an optimal sequential evaluation.

Parallel transformation. Given a k-spill, g-empty slot sequential evaluation $P = P_1, \ldots, P_n$ with arbitrary load latencies, then for each time i,

(i) if $P_i \in A(F)$, let $PQ_i = (NOP, P_i)$, and
(ii) if $P_i \in L(F) \cup S(F)$, let $PQ_i = (P_i, NOP)$,
(iii) if $P_i = NOP$, let $PQ_i = (NOP, NOP)$.

No precedence is violated by this transformation since the operations are executed in the same order, and all pairs of operations are the same number of cycles apart in both evaluations. Therefore PQ represents a k-spill, g-empty slot 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 1(b) is shown in Figure 2.

4. Load-contiguous parallel evaluations. We have shown that a parallel evaluation with the minimum number of spill operations plus empty slots can be constructed by applying the parallel transformation to an 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

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}	–	–	–	S_5	L_{14}	L_{15}	–	–	L_{16}	L_{17}	–
$q_i =$	–	–	–	A_{13}	–	–	A_9	–	–	–	A_{10}	A_5	–	–	–	–	A_7	–	–	–
$\rho_i =$	1	2	2	1	2	2	1	2	3	3	2	1	0	1	2	2	1	2	3	3

Slot $i =$	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
$p_i =$	–	–	L_5	–	–	S_2	L_{24}	L_{25}	–	–	L_{26}	L_{27}	–	–	–	L_{23}	–	–	L_2	–	–
$q_i =$	A_8	A_4	–	–	A_2	–	–	–	–	A_{11}	–	–	–	A_{12}	A_6	–	–	A_3	–	–	A_1
$\rho_i =$	2	1	2	2	1	0	1	2	2	1	2	3	3	2	1	2	2	1	2	2	1

FIG. 2. *The result of transforming the sequential evaluation in Figure* 1(b) *into a parallel evaluation for latency* $= 2$ *and* $R = 3$.

every parallel evaluation can be made load-contiguous without increasing its cost or the maximum number of values in registers. Furthermore, for load latencies ≤ 2 we show that the number of spill operations plus empty slots also does not increase. We then show in section 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 spill operations plus empty slots. Finally in sections 6 and 7 we show that for *latency* $= 1$ or *latency* $= 2$, a contiguous evaluation that minimizes the number of spill operations plus empty slots is necessarily optimal. That is, applying the contiguity transformations to a parallel evaluation with the minimum number of spill operations plus empty slots 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, i.e., a pair (NOP, q) where q is not a NOP. Using this definition we can prove the following theorem and lemma.

THEOREM 4.1. *Given any* k*-spill parallel evaluation* PQ *of a computation* F *with arbitrary load latencies, there exists a* k'*-spill load-contiguous parallel evaluation* PQ' *of* F *such that*

(i) $c(PQ') \leq c(PQ)$,
(ii) $\rho(PQ') \leq \rho(PQ)$, *and*
(iii) $k' \leq k$.

Proof. Let PQ be a k-spill, g-empty slot 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 one of of the following two transformations has been applied.

1. If the load scheduled at time j loads a value that is stored at time m such that $i < m < j$, both the load and store are deleted.

2. Otherwise, the load operation at time j is moved to i.

In both cases, $c(PQ') \leq c(PQ)$ and PQ' has no more spill operations than PQ. It remains to show that $\rho(PQ') \leq \rho(PQ)$, and that precedence is preserved.

Since no load/store operation is scheduled at time i, $\rho_i(PQ) = \rho_{i-1}(PQ) - 1$. From time i through $j-1$ by assumption there are no load operations, so the number of registers used as we progress forward can only stay the same or decrease. In particular, for all time 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 at time j is moved to time i. That is, $p'_i = p_j$, and $p'_j = NOP$. For all $m < i$, $\rho_m(PQ') = \rho_m(PQ)$. Similarly, for all $m \geq j$, $\rho_m(PQ') = \rho_m(PQ)$. However, $\rho_i(PQ') = \rho_i(PQ) + 1$ due to the new load operation at time i. Similarly, for all 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}	S_5	–	L_{16}	L_{17}	–	L_5	L_{24}	–	L_{25}	S_2
$q_i =$	–	–	–	A_{13}	–	A_9	–	–	A_{10}	A_5	–	–	A_7	–	–	A_8	A_4	–	A_2	–
$\rho_i =$	1	2	2	2	2	2	3	3	3	3	2	2	2	3	3	3	3	3	3	2

Slot $i =$	21	22	23	24	25	26	27	28	29	30
$p_i =$	–	L_{26}	L_{27}	–	L_{23}	L_2	–	–	–	–
$q_i =$	–	A_{11}	–	–	A_{12}	A_6	–	A_3	–	A_1
$\rho_i =$	2	2	3	3	3	3	3	2	2	1

FIG. 3. *The result of applying the load-contiguous transformation to the parallel evaluation in Figure* 2 *for latency* $= 2$ *and* $R = 3$.

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 if the load operation at time j is moved to i, $\rho(PQ') \leq \rho(PQ) \leq R$. By a similar argument we can show that if the load at time j and a store at time m are deleted, $\rho(PQ') \leq \rho(PQ) \leq R$.

Moving a load operation earlier in the evaluation cannot violate its precedence with respect to an arithmetic operation that uses the loaded value. If the load operation scheduled at time j loads a value that is stored at time m such that $i < m < j$, both the load and store operations are deleted. Note that the number of empty slots in the evaluation will increase if the load operation at time j or the store operation at time m is in an empty arithmetic slot.

The transformation described above 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 either a load operation is scheduled at time i or a spill is eliminated. Therefore the iterations will eventually terminate, giving an evaluation in which the initial portion of the evaluation contains load operations, store operations and empty slots, until the load operations are exhausted. □

We now extend this result by showing that for *latency* ≤ 2, the number of spill operations plus empty slots also does not increase.

LEMMA 4.2. *Given any* k*-spill,* g*-empty slot, parallel evaluation* PQ *of a computation* F *with latency* ≤ 2*, there exists a* k'*-spill,* g'*-empty slot, load-contiguous parallel evaluation* PQ' *of* F *such that*

(i) $c(PQ') \leq c(PQ)$,
(ii) $\rho(PQ') \leq \rho(PQ)$,
(iii) $k' \leq k$, *and*
(iv) $g' = g$.

Proof. Let PQ be a k-spill, g-empty slot 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 apply the transformation described in Theorem 4.1 to either move the load operation at time j to i or delete the load operation at time j along with the store of the same value. In each case we show that an evaluation can be created that contains g empty slots.

If PQ_j originally contained an empty arithmetic slot, moving the load operation scheduled there to time i will create a new empty slot at time j. Deleting this empty slot does not violate precedence. Deleting an empty slot cannot violate precedence between two arithmetic operations, between an arithmetic operation and a store operation, or between a store operation and a load operation, because arithmetic and store operations have a latency of 1. Since the load latency is not larger than 2,

deleting an empty slot at time j could only violate precedence if PQ_{j-1} contains a load operation and PQ_{j+1} contains an arithmetic operation dependent on that load. However, by assumption there are no load operations scheduled between time i and time j in PQ. Therefore PQ_{j-1} does not contain a load operation, and precedence is not violated if an empty slot at time j is deleted.

A similar argument applies if the load operation scheduled at time j loads a value that is stored at time m such that $i < m < j$. Deleting both the load and store operations will create new empty slots if either operation was originally scheduled in an empty arithmetic slot. Precedence would be violated by deleting these empty slots if either operation was preceded by a load operation and followed by an arithmetic operation dependent on that load. However, by assumption there cannot be a load operation scheduled between the first empty load/store slot at time i and time j. Therefore any new empty slots can be deleted without violating precedence.

Furthermore by arguments similar to those used in the proof of Theorem 4.1 we can show that $c(PQ') \leq c(PQ)$, $\rho(PQ') \leq \rho(PQ)$, $k' \leq k$, and that this transformation can be applied iteratively until the evaluation is load contiguous. □

Load-contiguous transformation for load latencies ≤ 2. Given any parallel evaluation PQ of a computation F, we create a load-contiguous parallel evaluation PQ' using the transformation described in Lemma 4.2: we apply the transformation from Theorem 4.1 and delete any slots that are empty as a result of each phase of the transformation.

The result of applying the load-contiguous transformation to the parallel evaluation in Figure 2 is shown in Figure 3. 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 and only if for any i such that a store operation is scheduled at time i, $\rho_{i+1} = R$. This implies that store operations are scheduled as late as possible. Using this definition we can prove the following theorem.

THEOREM 5.1. *Given a k-spill, g-empty slot, load-contiguous, parallel evaluation PQ of a computation F with arbitrary load latencies and with $\rho(PQ) \leq R$, there exists a k'-spill, g'-empty slot, parallel evaluation PQ' of F such that*

(i) $c(PQ') \leq c(PQ)$,
(ii) $\rho(PQ') \leq R$,
(iii) $k' \leq k$,
(iv) $g' \geq g$,
(v) $2k' + g' \leq 2k + g$,
(vi) *PQ' is load-contiguous, and*
(vii) *PQ' is store-contiguous.*

Proof. Let PQ be a load-contiguous, k-spill, g-empty slot 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, the number of spill operations, or the number of spill operations plus empty slots, and without causing the number of empty slots to decrease or the maximum number of values in registers to exceed R. We begin by showing that the last store operation in the evaluation 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 evaluation.

Consider the largest time i such that a store operation is scheduled at time i and $\rho_{i+1}(PQ) < R$. We 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 $\rho_{i+1}(PQ) < \rho_i(PQ)$ because the number of values in registers must decrease by at least 1 between time i and time $i+1$. 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. We consider seven cases depending on the operations scheduled at times i and $i+1$. The transformation from PQ to PQ' is shown in Figure 4. For each case, the upper pairs of operations represent PQ_i and PQ_{i+1}, while the lower pairs represent PQ'_i and PQ'_{i+1}. For conciseness we let X and Y each represent either an arithmetic operation or an empty slot, and we let Z represent either a load operation or an empty slot. In each case it can be seen that $c(PQ') \leq c(PQ)$, $k' \leq k$, and $g' \geq g$.

The new evaluation does not violate the precedence constraints because in each case one or more of the following transformations is performed: a load operation is moved earlier in the evaluation, a store operation is moved later in the evaluation, an arithmetic operation is moved one slot later into a previously empty slot, or an arithmetic operation is moved before a store operation. None of these transformations can violate the precedence constraints, unless a load operation is moved earlier than a store operation of the same value. In this case deleting both operations does not violate precedence. We now show that in each case $\rho(PQ') \leq R$ and $2k' + g' \leq 2k + g$.

The transformations affect only operations scheduled at time i or $i+1$, so for all $j \leq i-1$ and $j \geq i+1$, $\rho_j(PQ) = \rho_j(PQ')$. As a result, showing that $\rho_i(PQ') \leq \rho_{i-1}(PQ')$ allows us to conclude that $\rho(PQ') \leq R$.

(i) Case 1 in Figure 4 such that S and L reference the same value: Delete the load and store operations, i.e., let $PQ'_i = (NOP, X)$ and $PQ'_{i+1} = (NOP, Y)$. Since $PQ'_i = (NOP, A)$ or (NOP, NOP), $\rho_i(PQ') \leq \rho_{i-1}(PQ')$. The number of empty slots does not decrease. The quantity $2k+g$ does not increase because the number of spills decreases by one and the number of empty slots increases by no more than two.

(ii) Cases 2–6 in Figure 4 such that S and L, if present, do not reference the same value: Since $\rho_{i-1}(PQ) \leq R$ and $\rho_{i-1}(PQ) = \rho_{i-1}(PQ')$, it can be seen in each case that $\rho_i(PQ') \leq R$. Also, the number of spill operations and empty slots is unchanged.

(iii) Case 7 in Figure 4 such that S and L do not reference the same value: In this case, $\rho_{i-1}(PQ) \leq R-1$. If this were not true, then $\rho_{i+1}(PQ) = R$, which contradicts the assumption that $\rho_{i+1}(PQ) < R$. Therefore $\rho_i(PQ') \leq R$. Again, the number of spill operations and empty slots is unchanged.

In each case we have shown that if a store operation is scheduled at time i and $\rho_{i+1}(PQ) < R$, it is always possible to move the store one time period later without violating precedence, without increasing the evaluation's cost, the number of spill

Case =	1	2	3	4	5	6	7
$p =$	$S\ L$	$S\ -$	$S\ L$	$S\ Z$	$S\ -$	$S\ -$	$S\ L$
$q =$	$X\ Y$	$-\ -$	$A\ X$	$-\ A$	$A\ A'$	$A\ -$	$-\ -$
$p' =$	$-\ -$	$-\ S$	$L\ S$	$Z\ S$	$-\ S$	$-\ S$	$L\ S$
$q' =$	$X\ Y$	$-\ -$	$A\ X$	$A\ -$	$A\ A'$	$-\ A$	$-\ -$

FIG. 4. *The contents of PQ_i, PQ_{i+1}, PQ'_i, PQ'_{i+1} for each case of the store-contiguous transformation. X and Y each represent either an arithmetic operation or an empty slot, and Z represents either a load operation or an empty slot. In Case* 1 *the store and load operations reference the same value. In the remaining cases the store and load operations, if present, reference different values.*

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}	$-$
$q_i =$	$-$	$-$	$-$	A_{13}	$-$	A_9	$-$	$-$	A_{10}	A_5	$-$	A_7	$-$	$-$	$-$	A_8	A_4	$-$	A_2	$-$
$\rho_i =$	1	2	2	2	2	2	3	3	3	3	3	3	2	3	3	3	3	3	3	3

Slot $i =$	21	22	23	24	25	26	27	28	29	30
$p_i =$	L_{26}	S_2	L_{27}	$-$	L_{23}	L_2	$-$	$-$	$-$	$-$
$q_i =$	A_{11}	$-$	$-$	$-$	A_{12}	A_6	$-$	A_3	$-$	A_1
$\rho_i =$	3	2	3	3	3	3	3	2	2	1

FIG. 5. *The result of applying the store-contiguous transformation to the load-contiguous evaluation in Figure* 3 *with $R = 3$.*

operations, or the number of spill operations plus empty slots, and without causing the number of empty slots to decrease or the maximum number of values in registers to exceed R. The transformation described above 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 this 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 evaluation. Notice that up to the point of deleting load/store pairs, any transformation that moves a load interchanges the load with a store. Thus load-contiguity is preserved. But empty load/store slots can be 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 transformation described above and the load-contiguous transformation are reapplied k times, i.e., each time a spill is eliminated. □

Store-contiguous transformation. Given a load-contiguous parallel evaluation PQ of a computation F, we create a parallel evaluation PQ' that is both load-contiguous and store-contiguous using the transformation described in Theorem 5.1.

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 the store-contiguous transformation to the load-contiguous parallel evaluation in Figure 3 is shown in Figure 5.

6. Cost of a *k*-spill parallel evaluation. We have shown that given a k-spill, g-empty slot 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 is contiguous. Furthermore, for load latencies ≤ 2 the contiguous evaluation has no more than $2k + g$ spill operations plus empty slots. We now examine the structure of such a k-spill, g-empty slot contiguous evaluation with a sequence of lemmas and show in Theorem 6.5 that its cost is $\rho(PQ)+2k+\mathrm{g}+|A(F)|$. The results in this section apply with arbitrary load latencies.

LEMMA 6.1 (each store operation and the following operation). *Consider a k-spill, g-empty slot contiguous parallel evaluation PQ of a computation F with arbitrary load latencies and R available 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.*

Proof. Let PQ be a k-spill, g-empty slot 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 $PQ_i = (S, NOP)$, and show that $PQ_{i+1} = (L, NOP)$ for some load operation L. Assume that $PQ_i \neq (S, NOP)$, i.e., $PQ_i = (S, A)$. Then $\rho_i(PQ) = \rho_{i-1}(PQ) - 2$. The number of values in registers at some time can increase by at most one in the next time period, so $\rho_{i+1}(PQ) \leq \rho_i(PQ) + 1 = \rho_{i-1}(PQ) - 1 \leq R - 1$. Thus PQ is not store-contiguous, which is a contradiction. We conclude that $PQ_i = (S, NOP)$.

Now from the store contiguity of PQ we know that $\rho_{i+1}(PQ) = R$. We also know that $\rho_{i-1}(PQ) \leq R$ and therefore, since $PQ_i = (S, NOP)$, $\rho_i(PQ) \leq R - 1$. Thus $\rho_{i+1}(PQ) - \rho_i(PQ) \geq 1$ which implies that $PQ_{i+1} = (L, NOP)$ for some load operation L. □

LEMMA 6.2 (operations between two successive store operations). *Consider a k-spill, g-empty slot contiguous parallel evaluation PQ of a computation F with arbitrary load latencies and R available registers. If $PQ_i = (S, NOP)$ for some store operation S and the following store operation exists and is scheduled at time j, $j > i$, then for all m, $i + 1 < m < j$, PQ_m is of the form (L, A) or (NOP, NOP), and $\rho_m(PQ) = R$.*

Proof. Consider a store operation scheduled at time i of a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$, and the following store operation scheduled at time j. We know from Lemma 6.1 that $PQ_{i+1} = (L, NOP)$ and from store-contiguity that $\rho_{i+1}(PQ) = R$. We now show by contradiction that for all m such that $i+1 < m < j$, $\rho_m(PQ) = R$. Assume that for some time m, $i+1 < m < j$, $\rho_m(PQ) < R$. Therefore the number of values in registers decreases by at least one between time $i + 1$ and m. Since there are, by assumption, no stores scheduled at these times, there must be at least one time m', $i + 1 < m' \leq m$ such that $PQ_{m'} = (NOP, A)$. However, there must be a load operation after time 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 $i + 1 < m < j$, $\rho_m(PQ) = R$. This implies that for all m, $i + 1 < m < j$, PQ_m is of the form (L, A) or (NOP, NOP). □

LEMMA 6.3 (operations before the first store operation). *Consider a k-spill, g-empty slot contiguous parallel evaluation PQ of a computation F with arbitrary load latencies and R available registers. If the first store operation of PQ is scheduled at time i, then a load operation or an empty slot is scheduled at each time preceding time i, and exactly R empty arithmetic slots precede time i.*

Proof. Consider the first time i when a store operation S, if one exists, is scheduled in a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$. Load-contiguity and the fact that there is a load operation scheduled at time $i+1$ imply that there can be no empty load/store slots before slot i, and since there are no stores there, there must be a load operation or an empty slot scheduled at each time before time i.

Store-contiguity implies that $\rho_{i+1}(PQ) = R$, so $\rho_{i-1} = R$ because $PQ_i = (S, NOP)$ and $PQ_{i+1} = (L, NOP)$ for some store operation S and load operation L. Therefore there are R more load operations than arithmetic operations scheduled before time i, which implies that there are exactly R empty arithmetic slots before time i. ∎

LEMMA 6.4 (operations after the last store operation). *Consider a k-spill, g-empty slot contiguous parallel evaluation PQ of a computation F with arbitrary load latencies and R available registers. If the last store operation of PQ is scheduled at time m, then there are no empty arithmetic slots after time $m+1$.*

Proof. Consider the last time m when a store operation S, if one exists, is scheduled in a contiguous parallel evaluation PQ with $\rho(PQ) \leq R$. We claim that there are no empty arithmetic slots after time $m+1$, and show this by contradiction. Note first that there can be no empty load/store slots between time m and the last load operation at time 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 let m' be the first one. Therefore, there is an arithmetic operation or an empty slot scheduled at every time j, where $m+2 \leq j \leq m'-1$. (Recall that the operations scheduled at time $m+1$ have the form (L, NOP).) Since PQ is contiguous, there must be a load operation or an empty slot scheduled at every time j, where $m+1 \leq j \leq m'$. 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 empty arithmetic slot m' cannot exist. ∎

THEOREM 6.5 (cost of a contiguous parallel evaluation). *A k-spill, g-empty slot contiguous parallel evaluation PQ of a computation F with arbitrary load latencies and with $\rho(PQ) \leq R$ has cost $c(PQ) = \rho(PQ) + 2k + g + |A(F)|$.*

Proof. We have shown that 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 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 number of slots containing arithmetic operations, plus the number empty arithmetic slots, plus the number of empty slots, i.e., $\rho(PQ) + 2k + g + |A(F)|$.

If there are no spills in a contiguous parallel evaluation PQ, the number of empty arithmetic slots can be seen to be $\rho(PQ)$. The number of values in registers only increases when operation pairs scheduled at some time are of the form (L, NOP). Since PQ contains no spills, the number of values in registers only decreases when operation pairs scheduled have the form (NOP, A). The number of values in registers is unchanged when operation pairs scheduled have the form (L, A) or (NOP, NOP). Since PQ is load-contiguous, each operation pair of the form (NOP, A) must follow all operation pairs of the form (L, NOP). Therefore, $\rho(PQ) =$ the number of

operation pairs of the form (L, NOP). Since arithmetic operations or empty slots are scheduled at all other times, the cost of a 0-spill contiguous evaluation is $\rho(PQ) + g + |A(F)|$. □

As a consequence of Theorem 6.5 and the preceding lemmas, a k-spill, g-empty slot 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 empty slots and 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) interspersed with empty slots. The evaluation ends with a series of operation pairs of the form (NOP, A), also interspersed with empty slots. It can be seen that the contiguous evaluation in Figure 5 satisfies Lemmas 6.1, 6.2, 6.3, and 6.4 and Theorem 6.5.

Note also that there are $R - 1$ final operations ($\rho(PQ) - 1$ operations if there are no stores) of the form (NOP, A) interspersed with empty slots. This will always be the case since there are R values in registers in the time period after the last store, schedules of dependence trees leave one final value in a register, operation pairs of the form (L, A) or (NOP, NOP) do not change the number of values in registers, and each operation pair of the form (NOP, A) reduces the number of values in registers by one. Any empty slots that follow the first empty arithmetic slot can be deleted without violating precedence. Furthermore the evaluation can be made to begin with R operation pairs of the form (L, NOP), followed by only (L, A) and (NOP, NOP) operation pairs up to the first store operation by simply pushing the leading A operations to the right until they are scheduled consecutively up to the first store operation (or consecutive slots up to the last (L, A) operation pair if there are no stores), and deleting any empty slots that precede the first arithmetic operation by more than one cycle.

7. Optimal parallel evaluations for latency = 1 or latency = 2. We describe an algorithm that solves the parallel scheduling problem given a computation F and R available registers. We first construct a k-spill, g-empty slot sequential evaluation P using an optimal sequential scheduling algorithm, such as the two described in section 3.1. Using the parallel transformation of section 3.3, we construct the parallel evaluation PQ_1. Using the load-contiguous transformation of section 4, we construct the evaluation PQ_2. Using the store-contiguous transformation of section 5, we construct the evaluation PQ_3. If $latency = 1$, we show that using the $latency = 1$ sequential scheduling algorithm to produce P in the first step gives an evaluation PQ_3 that is a solution to the parallel scheduling problem. Similarly, if $latency = 2$ we show that using the $latency = 2$ sequential scheduling algorithm to produce P in the first step gives an evaluation PQ_3 that is a solution to the parallel scheduling problem.

Note that the sequential evaluations produced by the two sequential scheduling algorithms can be quite different since they make different assumptions about the underlying processor. We begin by describing two properties that are independent of the optimal sequential scheduling algorithm used.

LEMMA 7.1. *Given a computation F with arbitrary load latencies and R available registers, there exists a solution to the parallel scheduling problem that is contiguous.*

Proof. By Theorem 4.1 and Theorem 5.1, given any parallel evaluation PQ of F that uses R registers, it is possible to construct another contiguous evaluation that uses R registers, contains no additional spills, and costs no more than PQ. □

As a result of Lemma 7.1, we consider only contiguous evaluations of F.

LEMMA 7.2. *Given a computation F with latency ≤ 2 and R available registers, evaluations P and PQ_3 produced by the parallel scheduling algorithm have the same number of spill operations plus empty slots, and PQ_3 has no fewer empty slots than P.*

Proof. By Lemma 3.2, no parallel evaluation of F that uses R registers can contain fewer spill operations plus empty slots than an optimal sequential evaluation. Since P is optimal, the number of spill operations plus empty slots in the parallel evaluations PQ_3 is not less than $2k + g$. But the transformations used to produce PQ_3 never increase the number of spill operations plus empty slots in the evaluation, as shown in Lemma 4.2 and Theorem 5.1. Therefore the number of spill operations plus empty slots in the parallel evaluation PQ_3 is exactly $2k+g$. Similarly, the number of empty slots in evaluation PQ_3 is not less than g. □

We now show that the parallel scheduling algorithm for *latency* $= 1$ gives an optimal evaluation.

THEOREM 7.3. *Given a computation F with latency $= 1$ and R available registers, the parallel scheduling algorithm for latency $= 1$ gives an evaluation, PQ_3, that is a solution to the parallel scheduling problem.*

Proof. By Lemma 7.2, the number of spill operations plus empty slots in parallel evaluation PQ_3 is $2k + g$. Since load latencies equal one we can assume that all evaluations contain no empty slots. Therefore, PQ_3 contains exactly k spills. We first show that only k-spill contiguous evaluations need to be considered. We then show that PQ_3 is an optimal k-spill contiguous evaluation of F.

Consider two contiguous evaluations of F, PQ and PQ', that contain k and k' spills, respectively, such that both use R registers and $k \neq k'$. We show that $c(PQ) \leq c(PQ')$. By Lemma 3.2 and since $g = 0$, no parallel evaluation of F that uses R registers can contain fewer than k spills, so $k < k'$. We first consider the case where $k > 0$. Since PQ is contiguous, and since it contains at least one spill, $\rho(PQ) = R$. Since $k < k'$, PQ' also contains at least one spill. Therefore $\rho(PQ) = \rho(PQ')$. Since both PQ and PQ' are contiguous, $c(PQ) = \rho(PQ) + 2k + |A(F)|$, and $c(PQ') = \rho(PQ') + 2k' + |A(F)|$. Since $k < k'$, $c(PQ) < c(PQ')$. Therefore if $k \neq 0$, no contiguous evaluation with more than k spills is optimal.

We now consider the case where $k = 0$. Since PQ uses R registers, $\rho(PQ) \leq R$ and $c(PQ) = \rho(PQ) + |A(F)|$. Since $k < k'$, PQ' contains at least one spill, so $\rho(PQ') = R$ and $c(PQ') = R + 2k' + |A(F)|$. Therefore, $c(PQ) < c(PQ')$. We have shown that given an arbitrary contiguous evaluation PQ' that uses R registers, if PQ' does not contain k spills, then PQ' cannot be an optimal evaluation of F. We will therefore only consider k-spill contiguous evaluations of F.

By Theorem 6.5, every k-spill contiguous evaluation PQ of F that uses R registers has cost $c(PQ) = \rho(PQ) + 2k + |A(F)|$. If $k > 0$, $c(PQ) = R + 2k + |A(F)|$, which is a constant. That is, all k-spill contiguous evaluations of F have the same cost and every one of them is optimal. In particular, PQ_3 is optimal.

If $k = 0$, $\rho(PQ)$ must be minimized in order to minimize $c(PQ) = \rho(PQ)+|A(F)|$. In [36] it is shown that the sequential scheduling algorithm minimizes the number of registers used. By Lemma 3.1, any parallel evaluation of a computation F can be transformed into a serial evaluation of F without increasing the maximum number of values in registers. Therefore, no parallel evaluation of F uses fewer than $\rho(P)$ registers. The evaluation P is transformed by our algorithm into PQ_2 by applying the parallel transformation followed by the load-contiguous transformation. Since $k = 0$ by assumption, the load-contiguous evaluation is also store-contiguous, and therefore contiguous. As described in section 3.3, the parallel transformation does

not change the maximum number of values in registers. By Theorem 4.1, the load-contiguous transformation also does not increase the maximum number of values in registers. Therefore, $\rho(PQ) = \rho(P)$ and $\rho(PQ)$ is minimized. We conclude therefore that PQ_3 is an optimal no-spill contiguous evaluation of F. □

The following theorem shows that the parallel scheduling algorithm for *latency* = 2 gives an optimal evaluation. The theorem relies on results regarding the optimal sequential scheduling algorithm for *latency* = 2 [28]. In particular we use the fact that if the number of available registers is large enough, the algorithm produces an evaluation containing no empty slots, and otherwise the evaluation produced must contain at least one empty slot. While the proof of Theorem 7.3 considered cases based on the number of spills, the proof of Theorem 7.4 considers cases based on the relative sizes of R and the minimum number of registers needed to avoid the use of spill code. With *latency* = 2, it is the number of available registers that affects the existence of empty slots in an optimal parallel evaluation.

THEOREM 7.4. *Given a computation F with latency = 2 and R available registers, the parallel scheduling algorithm for latency = 2 gives an evaluation, PQ_3, that is a solution to the parallel scheduling problem.*

Proof. We prove that no contiguous evaluation of F costs less than PQ_3. Let PQ' be an arbitrary k'-spill, g'-empty slot contiguous evaluation of F that uses R registers, and let $\rho' = \rho(PQ')$. We further let *minreg* equal the minimum number of registers needed to evaluate F without spill code on one processor with load latencies = 1. We first consider the case where $R \geq minreg + 1$. Since R is larger than *minreg*, evaluation P contains no spills and therefore PQ_3 contains no spills. Furthermore, when R is larger than *minreg* the sequential scheduling algorithm produces an evaluation containing no empty slots [28]. Since by Lemma 3.2 the transformations do not change $2k + g$, $c(PQ_3) = minreg + 1 + |A(F)|$. We now compare this cost to the possible costs of PQ'. If $\rho' \geq minreg + 1$, then by Theorem 6.5, $c(PQ_3) \leq c(PQ')$. If $\rho' = minreg$, then the sequential scheduling algorithm produces an evaluation containing at least one empty slot [28]. Since the transformations do not decrease the number of empty slots, $g \geq 1$, and again $c(PQ_3) \leq c(PQ')$. Finally if $\rho' < minreg$, then by assumption $\rho' < R$ and therefore k' must equal zero since PQ' is contiguous. This is a contradiction by Lemma 3.3 because all parallel evaluations that use fewer than *minreg* registers must contain at least one spill.

We next consider the case where $R = minreg$. The cost of evaluation PQ_3 is $R+2k+g+|A(F)|$. If $k' > 0$, then $\rho' = R$ since PQ' is contiguous. Therefore the cost of PQ' is $R+2k'+g'+|A(F)|$ which is not less than $c(PQ_3)$ because $2k+g \leq 2k'+g'$ by Lemma 3.2. If $k' = 0$, then it must be the case that $\rho' = R$, because if $\rho' < minreg$ then PQ' must contain at least one spill by Lemma 3.3. Therefore $c(PQ_3) \leq c(PQ')$ because $2k + g \leq 2k' + g'$.

Finally, we consider the case where $R < minreg$. The cost of evaluation PQ_3 is again $R + 2k + g + |A(F)|$. If $k' > 0$, then $\rho' = R$ and $c(PQ_3) \leq c(PQ')$ as before. If $k = 0$, then this contradicts Lemma 3.3 because $R < minreg$. Therefore in each case $c(PQ_3) \leq c(PQ')$, and we conclude that PQ_3 is an optimal evaluation of F. □

8. Computational complexity. We show in this section that the worst-case computational complexity of the parallel scheduling algorithms is $O(nk)$, where $n = |A| + |L|$ is the number of nodes in F and k is the number of spills in the evaluation.

The labeling routine of the sequential scheduling algorithms 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 $O(n)$

[16]. Similarly, the worst-case complexity of the scheduling routine with load latencies $= 2$ is $O(n)$.

The parallel transformation of section 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 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 $n + 2k$ 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 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 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 evaluation 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. Unnecessary Empty slots can be deleted from the evaluation in $O(n + k)$ time. Thus, the worst-case complexity of the parallel scheduling algorithms is $O(nk)$.

9. Summary. We have presented an $O(nk)$ algorithm that solves the parallel scheduling problem for $latency = 1$ or $latency = 2$ 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 spill operations plus empty slots. 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 $\rho(PQ) + 2k + g + |A(F)|$.

These results may encourage further research on other, related problems. For example, extensions of our results may apply to trees with arbitrary arity, and to processors that can issue more operations simultaneously. The new transformations we describe may also form the basis for worst-case results for more general problems that cannot be efficiently solved.

Acknowledgments. We thank the anonymous referees for their detailed suggestions which have greatly improved this paper.

REFERENCES

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

[2] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, *Spill code minimization via interference region spilling*, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, NV, ACM Press, 1997.

[3] D. Bernstein and I. Gertner, *Scheduling expressions on a pipelined processor with a maximal delay of one cycle*, ACM Trans. Prog. Lang. Syst., 11 (1989), pp. 57–66.

[4] D. Bernstein, J. Jaffe, and M. Rodeh, *Scheduling arithmetic and load operations in parallel with no spilling*, SIAM J. Comput., 18 (1989), pp. 1098–1127.
[5] D. Bernstein, M. Rodeh, and I. Gertner, *On the complexity of scheduling problems for parallel/pipeplined machines*, IEEE Trans. Comput., 38 (1989), pp. 1308–1313.
[6] P. Bose, *Optimal code generation for expressions in super scalar machines*, in Proceedings of the Fall Joint Computer Conference, IEEE Computer Society Press, Los Alamitos, CA, 1986, pp. 372–379.
[7] D. Bradlee, S. Eggers, and R. Henry, *Integrating register allocation and instruction scheduling for RISCS*, in Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, 1991.
[8] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, *Coloring heuristics for register allocation*, SIGPLAN Notices, 24 (1989), pp. 275–284.
[9] P. Briggs, K. D. Cooper, and L. Torczon, *Improvements to graph coloring register allocation*, ACM Trans. Prog. Lang. Syst., 16 (1994), pp. 428–455.
[10] J. Bruno, J. Jones, and K. So, *Deterministic scheduling with pipelined processors*, IEEE Trans. Comput., C-29 (1980), pp. 308–316.
[11] J. Bruno and R. Sethi, *Code generation for a one-register machine*, J. ACM, 23 (1976), pp. 502–510.
[12] D. Callahan and B. Koblenz, *Register allocation via hierarchical graph coloring*, in ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada, ACM Press, 1991.
[13] M. Carlisle, *On Local Register Allocation*, Thesis, Computer Science Department, University of Delaware, Newark, DE, 1991.
[14] F. C. Chow and J. L. Hennessy, *The priority-based approach to register allocation*, ACM Trans. Prog. Lang. Syst., 12 (1990), pp. 503–536.
[15] E. Coffman and R. Graham, *Optimal scheduling for two-processor systems*, Acta Inform., 1 (1972), pp. 200–213.
[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1991.
[17] G. J. Chaitin, et al., *Register allocation via coloring*, Computer Languages, 6 (1981), pp. 47–57.
[18] G. J. Chaitin, et al., *Register allocation and spilling via graph coloring*, in Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, New York, 1982, pp. 98–105.
[19] M. Farach and V. Liberatore, *On local register allocation*, in ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 1998.
[20] L. George and A. Appel, *Iterated register coalescing*, ACM Trans. Prog. Lang. Syst., 18 (1996), pp. 300–324.
[21] P. B. Gibbons and S. S. Muchnick, *Efficient instruction scheduling for a pipelined architecture*, in Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, New York, 1986, pp. 11–16.
[22] J. R. Goodman and W. C. Hsu, *Code scheduling and register allocation in large basic blocks*, in Proceedings of the 1988 International Conference on Supercomputing, Orlando, FL, ACM Press, 1988, pp. 442–452.
[23] J. Hennessy and T. Gross, *Code generation and reorganization in the presence of pipeline constraints*, in 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, ACM Press, 1982, pp. 120–127.
[24] J. Hennessy and T. Gross, *Postpass code optimization of pipeline constraints*, ACM Trans. Prog. Lang. Syst., 5 (1983), pp. 422–448.
[25] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, *Index register allocation*, J. ACM, 13 (1966), pp. 43–61.
[26] 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.
[27] T. Hu, *Parallel sequencing and assembly line problems*, Oper. Res., 9 (1961), pp. 841–848.
[28] S. Kurlander, T. Proebsting, and C. Fischer, *Efficient instruction scheduling for delayed-load architectures*, ACM Trans. Prog. Lang. Syst., 17 (1995), pp. 740–776.
[29] G. Lueh and T. Gross, *Call-cost directed register allocation*, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, NV, ACM Press, 1997.
[30] I. Nakata, *On compiling algorithms for arithmetic expressions*, Communications of the ACM, 10 (1967), pp. 492–494.
[31] K. Palem and B. Simons, *Scheduling time-critical instructions on RISC machines*, ACM Trans. Prog. Lang. Syst., 15 (1993), pp. 632–658.
[32] S. S. Pinter, *Register allocation with instruction scheduling: A new approach*, SIGPLAN

Notices, 28 (1993), pp. 248–257.
[33] T. Proebsting and C. Fischer, *Demand-driven register allocation*, ACM Trans. Prog. Lang. Syst., 18 (1996), pp. 683–710.
[34] R. R. Redziejowski, *On arithmetic expressions and trees*, Communications of the ACM, 12 (1969), pp. 81–84.
[35] R. Sethi, *Complete register allocation problems*, SIAM J. Comput., 4 (1975), pp. 226–248.
[36] R. Sethi and J. D. Ullman, *The generation of optimal code for arithmetic expressions*, J. ACM, 17 (1970), pp. 715–728.
[37] J. D. Ullman, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384–393.