EFFECTS OF LOOP FUSION AND STATEMENT MIGRATION ON THE SPEEDUP OF VECTOR MULTIPROCESSORS

Mayez Al-Mouhamed (1) and Lubomir Bic (2)

(1) Department of Computer Engineering
King Fahd University of Petroleum and Minerals (KFUPM)
31261 Dhahran, Saudi Arabia

(2) Department of Information and Computer Science
University of California Irvine,
California 92717, USA.

Summary: Vector multiprocessors rely on both spatial and temporal parallelism for achieving significant speedup. For singly nested loops, we study the effect on the speedup of: 1) loop fusion and, 2) increasing the granule-size of parallel-vector loops using extracted statements from scalar loops. The proposed optimizations migrate vector statements from one loop to another, create new loops, and reduce others. Loops and statements that belong to strongly connected data paths are vertically fused, whenever possible, in order to promote chaining and cache/register reuse. To reduce loop synchronization, horizontal fusion is also used for independent loops having compatible dependence types. Finally, vector operations are scheduled based on knowledge of the timing of arithmetic pipelines, load/store operations, and management of the available resource. Testing is carried out using synthetic Fortran programs on the Convex C240 vector multiprocessor. The proposed loop fusion improves the speedup by 18% to 43% over the C240 commercial optimizing compiler. Chaining-oriented scheduling and allocation yield 9% to 15% improvement over the highest optimization option of the C240 compiler.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **31**, 56–64 (1995)

Effects of Loop Fusion and Statement Migration on the Speedup of Vector Multiprocessors

MAYEZ AL-MOUHAMED* AND LUBOMIR BIC†

*Computer Engineering Department, College of Computer Science and Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia; and †Department of Information and Computer Science, University of California–Irvine, Irvine, California 92717

Vector multiprocessors rely on both spatial and temporal parallelism for achieving significant speedup. For singly nested loops, we study the effect on the speedup of (1) loop fusion and (2) increasing the granule size of parallel-vector loops using extracted statements from scalar loops. The proposed optimization migrate vector statements from one loop to another, create new loops, and reduce others. Loops and statements that belong to strongly connected data paths are vertically fused, whenever possible, in order to promote chaining, register, and cache reuse. To reduce loop synchronization, horizontal fusion is also used for independent loops having compatible dependence types. Finally, vector operations are scheduled based on knowledge of the timing of arithmetic pipelines, load and store operations, and management of the available resource. Testing is carried out using synthetic Fortran programs on the Convex C240 vector multiprocessor. The proposed loop fusion improves the speedup by 18 to 43% over the C240 commercial optimizing compiler. Chaining-oriented scheduling and allocation yields 9 to 15% improvement over the highest optimization option of the C240 compiler. © 1995 Academic Press, Inc.

1. INTRODUCTION

Research of the past decade has generated impressive improvements [16] in the design of parallel vector processors (VPs), due primarily to decreasing cycle time, the use of faster pipelined memories, and the increasing number of VPs. Vector multiprocessors allow exploiting spatial and temporal parallelism [20, 4] bsed on recurrence analysis that enables finding the maximum inherent parallelism in loops.

Restructuring compilers such as Parafrase [15, 20], PFC [3], UFTN [8], or V-Pascal [17] perform data dependence analysis of loops in order to classify them depending on their inherent parallelism. Scalar loops (SL) are the least reduceable because of their tight recurrences. Techniques for extracting parallelism out of SL loops have been developed based on recurrence analysis [10]. For example, *Cycle Shrinking* [16] and *Graph Traverse Scheduling* [4] split the loop iteration range into independent or synchronized partitions which can run in parallel.

Loop fusion has been considered [7] as an optimization method for compiling programs targeted to distributed-memory systems. Loop fusion simplifies [13, 14] data partitioning and allows increasing memory reuse. Loop fusion, however, has not been utilized heavily with vector multiprocessors.

Vectorizing compilers support operators to extract parallelism from loops that are primarily classified as scalar or having limited parallelism. Unfortunately, the extracted parallelism is distributed only out of the original loop. The work presented in this paper is based on the use of known techniques for extracting parallelism but with the objective of reducing the granule size of some scalar loops, fusing loops, forming new loops, or migrating statements from one loop to another whenever possible. With this approach, vector scheduling is proposed in order to exploit the locality of data producers and consumers that results from fusion. Scheduling of vector operations is based on the use of an accurate model and timing of the underlying vector processor. The benefits are an increase of the granule size of parallel vector loops at the expense of scalar loops, increasing reuse of cache memory and vector registers, improved chaining, and reduced loop synchronization. The approach has been tested on the Convex C240 mainframe.

This paper is organized as follows. In Section 2, we present a review of recurrence analysis. Section 3 presents our approach to loop distribution and fusion. Loop scheduling for the Convex C240 is developed in Section 4. Resolving conflicts due to register and data-path allocations is presented in Section 5. Section 6 presents the evaluation of this work and Section 7 concludes about this work.

2. BACKGROUND

Recurrences [16, 18] are the most important inhibitors of loop vectorization. A recurrence consists of a value that is computed in one iteration and used in another iteration of the same loop. To preserve the original chronology of the references, it is important to find out whether the loop can be vectorized while still generating the results wanted by the user. The basis for the detection of recurrences is the analysis of the dependence distance and recurrences.

Data dependence analysis [10] is concerned with the

0743-7315/95 $12.00
Copyright © 1995 by Academic Press, Inc.
All rights of reproduction in any form reserved.

memory access order as required by a program. A data dependence between references indicates that they access the same memory location with a given order. The types of accesses can be: (1) read-after-read (RAR) or input dependence, (2) read-after-write (RAW) or true dependence, (3) write-after-read (WAR) or anti dependence, and (3) write-after-write or output dependence. We note that RAR does not constrain the access order.

The number of iterations d between the source and the sink of the dependence is called the dependence distance. Consider a loop indexed by $(i_1, ..., i_k, ..., i_n)$ from outermost to innermost and let $a(i_1, ..., i_k, ..., i_n) = F(a(i_1, ..., i_k - d_k, ..., i_n))$ be an expression of the innermost nest. Evaluation of $a(i_1, ..., i_k, ..., i_n)$ is dependent on reference $a(i_1, ..., i_k - d_k, ..., i_n)$ that has been evaluated d_k iterations earlier. For a given array reference, the dependence distances can be represented by a vector $D = (d_1, ..., d_n)$ whose elements are displayed from left to right to represent the dependence from the outermost to the innermost loop of the nest. The element d_k is the dependence distance of the referenced array of the kth nest of the loop.

To illustrate, assume the loop body is $a(i - 2) = F(a(i - 4), ...)$, where a is an array and F is some function. The dependence distance is $d_i = i - 4 - (i - 2) = -2$. At iteration $i = 8$, $a(4)$ is read but two iterations earlier $a(4)$ was written, which indicates the presence of a RAW dependence.

Recurrence analysis [19] is useful to determine the chronology and the relationship of operations. A recurrence order cannot be changed without affecting the results.

A loop-carried-dependency (LCD) is the minimum condition for a recurrence to exist. An LCD test uses function $lcd(r_1, r_2)$ that can be defined by using an n-nested loop for which the jth iterator is $i_j \cdot lb_j, ub_j, s_j$, where lb_j, ub_j, s_j are the lower bound, upper bound, and stride, respectively. The range of the jth normalized iterator is $b_j = [1, (ub_j - lb_j + s_j)/s_j]$. A loop carries an LCD dependency when there exists at least two references r_2 and r_2, to the same array, such that $lcd(r_1, r_2)$ gives true value. In the case of a singly nested loop, r_1 is generally defined by $a(f_1(i_1, i_2, ...), f_2(i_1, i_2, ...), ...)$. The value of r_1 is computed an iteration $(i_1, i_2, ...)$ and used by reference $r_2 = a(g_1(i_1, i_2, ...), g_2(i_1, i_2, ...), ...)$ at some iteration $(i_1 + k_1, i_2 + k_2, ...)$. Therefore, $ldc(r_1, r_2) = 1$ if there exists integers $k_1 \in b_1, k_2 \in b_2$, etc such that $f_1(i_1, i_2, ...) = g_1(i_1 + k_1s_1, i_2 + k_2s_2, ...)$, $f_2(i_1, i_2, ...) = g_2(i_1 + k_1s_1, i_2 + k_2s_2, ...)$, etc.

Recurrence analysis establishes the validity of loop transformations [15] that can be applied to each type of loops in order to exploit its inherent parallelism. The most used loop transformations [16] are *strip-mining*, *loop distribution*, and *loop interchanging*.

Strip-mining splits the original loop, that is generally the innermost loop, into two loops: (1) a sequential or vector iterator whose bounds match the vector register size and (2) a parallel iterator that controls the number of times the vector iterator should execute.

Loop distribution creates many loops from a composite loop in order to overcome the limitation of only being able to vectorize innermost loops. A simple loop nest is a multinested loop where all calculations are done in the innermost loop. This enables distributing the outermost loop and vectorizing each of the resulting simple loop nests.

Loop interchanging consists of permuting the order of loop nests to achieve any of the following objectives: (1) interchanging the most vectorizable loop (avoiding a recurrence in the original inner loop) to innermost and the most parallelisable loop to outermost, (2) increase spatial reuse of data in cache memory, (3) order the loop nests in increasing order of loop counts from outermost to innermost, which is more suitable for vectorization. The way arrays are stored in memory influences spatial reuse of the cached data. For example, column-major organization of the array $a(i, j)$ leads to positioning of the i loop as innermost.

3. LOOP DISTRIBUTION AND FUSION

Our objective is to investigate the benefit of fusing loops and statements following the extraction of parallelizable computations out of loops that are primarily classified as scalar loops. We do not search to extract all the parallelism out of such loops. There are many approaches to exploit most of the inherent parallelism in loops [15, 16, 18]. The effectiveness of the gained speedup is necessarily dependent on the reduction factor, the granule size of the loop, and the amount of needed synchronization.

In the following, we present an algorithm called *Distribute* for reducing the granule size of loops that are primarily classified as scalar loops. Algorithm *Distribute* classifies loops into a number of categories depending on the amount of available parallelism and synchronization needed to generate correct results. The algorithm for distributing loops can be described by the following steps:

1. A loop L is classified as *loop-independent-dependency* (LID) if L does not carry dependency from one iteration to another but dependency may occur across the references of the same iteration. This is true when every pair of references r_1 and r_2 of L satisfy $lcd(r_2, r_2) = 0$, where $lcd(r_1, r_2)$ is the *loop-carried-dependence test*. The test $lcd(r_1, r_2)$ yields true value only when r_1 is referenced in some iteration and assigned in another iteration. A loop-independent-dependency (L_{lid}) can be strip-mined to generate a *parallel vector loop* (PVL). If this step succeeds in classifying L as an L_{lid}, then processing of L terminates and processing of the next loop L' is started.

2. Loop L could be partitioned into a number of L_{lid} loops and one reduced L_{lcd} loop if some expressions of L are only involved in LID dependence which enables distributing each of these expressions out of the original LCD loop. In other terms, there is no reference $r_1 \in L_{lid}$ for which there exists another reference $r_2 \in L$ such that $lcd(r_1, r_2) = 1$, where both r_1 and r_2 reference the same array. Each synthesized L_{lid} loop will be formed by one expression that is free of LCD dependencies. Based on

data producer–consumer relationships, a resulting L_{lid} loop could either be predecessor or successor with respect to the remaining L_{lcd} loop in the corresponding dataflow graph.

3. *Partial vectorization* (PV) is attempted in order to reduce the granule size of the current L_{lcd} loop. PV consists of distributing parallelizable and vectorizable statements out of LCD vector expressions. The distributed statements should not be involved in LCD dependence with the remaining expressions of the same loop. Temporary arrays are used to store the results of the distributed statements that become predecessors to their original LCD expressions in the dataflow graph. PV may generate an arbitrary number of L_{lid} loops that will be classified as PVL.

4. A reduction is detected whenever two references r_1 and r_2 of one expression reference a scalar or array with $lcd(r_1, r_2) = 0$ and r_1 is used prior to storing a computed value into r_2. A *reduction* operation can be vectorized by using dedicated vector accumulator, vector compress and merge, thus leading to a *parallel vector loop with dedicated code* (PVLD).

5. The remaining L_{lcd} could be partitioned into a backward-LCD (B-LCD) and a forward-LCD (F-LCD). An F-LCD exists when one iteration references value that will be assigned in a latter iteration. This can be determined by using a write-after-read ($war(r_1, r_2) = 1$) test which can be combined with $lcd(r_1, r_2) = 1$ to indicate the presence of B-LCD dependency between r_1 and r_2. An F-LCD loop does not prevent vectorization because a write-after-read dependence proceeds in the correct order when executed on a vector pipeline. Because some iteration boundaries can concurrently proceeds across the processors, F-LCD loops cannot be safely parallelized without adding synchronization to guarantee correct results. A distributed F-LCD loop is labeled as *parallel vector loops with synchronization* (PVLS). The remainder of the original loop is L_{b-lcd} with B-LCD dependence type.

6. Two expressions for which there is a B-LCD dependence can be interchanged if the result generated in one expression is not used in evaluating the second expression and no other dependence is violated. Although, s_1 and s_2 can be involved in B-LCD dependence they can be interchanged without altering the results if s_2 does not reference the computed value of s_1. If s_1 and s_2 are not involved in other dependence, expressions s_1 and s_2 can then be distributed out of the current L_{b-lcd} loop and two new loops can be created.

7. The remaining loop is considered as unreduceable because of its limited parallelism (if any) and marked as *scalar loop* (SL) that should be executed in sequence.

The formal steps of algorithm *Distribute* can be summarized as follows:

Variables and Functions:

s	is an arithmetic expression of loop L
r_1, r_2	are two references to the same scalar or array
$lcd(r_1, r_2)$	gives true only if there is loop-carried-dependency between r_1 and r_2
$war(r_1, r_2)$	write-after-read returns true only if r_1 is read and r_2 is write
$dep(s_1, s_2)$	gives true only if computation of $s2$ depends on the result of $s1$
PV	partial vectorization routine

```
Input: A candidate loop $L$ from a program $\{L_1, ..., L_n\}$
Output: Distributing $L$ into loops of type SL, PVL, PVLD, and PVLS
Algorithm Distribute:
Step 1: If $\forall r_1, r_2 \in L$: $lcd(r_1, r_2) = 0$, then mark $L$ as PVL and exit
Step 2: Attempt partitioning $L$ into $L_{lid}$ and $L_{lcd}$:
    Step 2.1: Find expressions of $L$ that have LID dependence only: $s_1$ is selected if for $\forall r_1 \in s_1$ that is referenced in $r_2 \in s_2$ we necessarily have $lcd(r_1, r_2) = 0$
              Create a loop $L_{lid}$ for each expression $s_1$ and mark it PVL
    Step 2.2: The remaining expressions form an $L_{lcd}$: $\forall s_1 \subset L_{lcd}$ and $r_1 \in s_1$: $lcd(r_1, r_2) = 1$, where $r_2 \in s_2$, and $s_2 \subset L_{lcd}$
Step 3: Attempt applying PV to $L_{lcd}$ and mark output fragments as required
Step 4: Search for a reduction:
              If ($\exists s_1 \subset L_{lcd}$ and $r_1 \in s_1$ such that $lcd(r_1, r_2) = 1$ only for $r_2 \in s_1$ and $war(r_1, r_2) = 1$), then create $L_{redn} = \{s_1\}$ and mark it PVLD
Step 5: Attempt partitioning $L_{lcd}$ into $L_{f-lcd}$ and $L_{b-lcd}$:
    Step 5.1: Find a set of expressions $S = \{s_1, s_2, ...\}$ such that $\forall r_1 \in s_1$, then $lcd(r_1, r_2) = 1$ only for $r_2 \in s_2$ and $war(r_1, r_2) = 1$
              Create $L_{f-lcd} = S$ and mark it PVLS
    Step 5.2: The remaining expressions of $L_{lcd}$ form one single $L_{b-lcd}$
Step 6: Attempt finding pairs $s_1$ and $s_2$ such that the only $r_1 \in s_1$ and $r_2 \in s_2$ which cause $lcd(r_1, r_2) = 1$ and $war(r_1, r_2) = 0$ satisfy: $dep(s_1, s_2) = 0$
        Interchange and distribute $s_1$ and $s_2$ into two loops and mark them as required
Step 7: Mark the nonreduceable $L_{b-lcd}$ as $SL$
```

Following the above analysis the program is represented by a *data dependence graph* (DDG) in which the nodes represent loops of type SL, PVLD, or PVLS or simply PVL. The method used for loop fusion is based on *Vertical* and *Horizontal* fusion:

1. Vertical fusion: statements or loops of the same type and with the same headers which belong to a dependence path are fused whenever possible. Using the previously generated loop labeling, fusing condition for PVL or PVLS loops is examined in order to ensure preserving the loop type of the fused loop. Fusion is not performed if the fused loop could contain: (1) F-LCD (prevent parallelization), or (2) B-LCD (fusion preventing).

2. Horizontal fusion: data independent loops or statements having the same types and loop counts are also fused in order to reduce synchronization and loop overhead.

In case of different fusion alternatives, priority is given to fusing loops and statements that maximizes the number of shared vector data. For example, if ($s1$, $s2$) and ($s1$, $s3$) are valid fusions and ($s2$, $s3$) is fusion-preventing, then ($s1$, $s2$) is selected if they share the use of larger number of vector data than ($s1$, $s3$). Analysis of conditions that prevent loop fusion can be found in [18]. Loop boundaries in the output program are mainly those corresponding to loops with different types, different loop counts, or same type but with fusion preventing condition.

We examine the program shown in Fig. 1 as an example of distributing and fusing loops. The loop has an LCD with respect to references $a(i)$ and $a(i-1)$ which indicates that Step 1 of *Distribute* fails. In Step 2, $s2$ is distributed out of the original loop. In Step 3, PV distributes statements $s1' : t1(i) = b(i) * c(i)$, $s3'' : t2(i) = sqrt(e(i))$, and $s3' : t3(i) = a(i) * t2(i)$ out of the original LCD loop, where $s1'$, $s3'$, and $s3''$ are identifiers for the three new statements. Statements $s1'$, $s3'$, and $s3''$ are labeled as PVL loops because they are free of LCD dependencies. The previous statements $s1$ and $s3$ are then reduced to $s1 : a(i) = a(i-1) + t1(i)$ and $s3 : sum = sum + t3(i)$ as a result of the previous partial vectorization. Step 4 finds the reduction that is present in $s3$ and creates a new loop for $s3$ that is labeled PVLD. The remaining LCD loop is reduced to $s1$. Step 5 indicates that $s1$ is a B-LCD and there is no F-LCD. Step 6 fails because the B-LCD loop is formed by one single expression $s1$ that is labeled as SL loop.

The resulting data dependence graph has four levels: $l1 = \{s1', s2\}$, $l2 = \{s1, s3''\}$, $l3 = \{s3'\}$, and $l4 = \{s3\}$. The dependence edges are: ($s1' \rightarrow s1$), ($s2 \rightarrow s3''$), ($s1, s3'' \rightarrow s3'$), and ($s3' \rightarrow s3$).

Vertical fusion leads to fuse expressions $s2$ and $s3''$. As $s1$ is SL and $s2$ is LID, horizontal fusion inserts then term $s1'$ within loop ($s2$, $s3''$) that becomes ($s1'$, $s2$, $s3''$). The type of $s3'$ is different from that of $s1$ or $s3$. The above three loops remain separate as shown in Fig. 2.

4. LOOP SCHEDULING

The traditional approaches [9, 11] are based on the use of *reservation tables* that result from gross estimating the functional unit (*Fus*) times as one or two units of times and scheduling the vector operations. Resolving problems with respect to memory load and store operations, resource availability, and resource allocation is done in a separate

```
do  i=1,n
       s1: a(i)=a(i-1)+b(i)*c(i)
       s2: e(i)=c(i)-b(i)
       s3: sum=sum+a(i)*sqrt(e(i))
enddo
```

FIG. 1. Example of a loop carrying a B-LCD.

```
do  i=1,n                          (Parallel-vector loop)
     s2: e(i)=c(i)-b(i)
     s3'': t2(i)=sqrt(e(i))
     s1': t1(i)= b(i)*c(i)
enddo
do  i=1,n                          (Scalar loop)
     s1: a(i)=a(i-1)+t1(i)
enddo
do  i=1,n                          (Parallel-vector loop)
     s3': t3(i)= a(i) * t2(i)
enddo
do  i=1,n                          (Add-reduce loop PVLD)
     s3: sum=sum+t3(i)
enddo
```

FIG. 2. Output program following distribution/fusion.

step. The result is that isolating the above constraints from the first step leads the schedule makespan to be excessively increased by adding delays for resolving the various register and data-path constraints in the final step.

Our approach consists of *early incorporation of all the available constraints* by scheduling the load/store and arithmetic operations over the *Fus* based on accurate timing of the VP and the management of the available resources during scheduling. This results in a conflict-free schedule with respect to *Fus*, memory utilization, and resource availability. The schedule generated is used to resolve conflicting assignment of register and data-path allocations. This consists of resolving conflicting allocation of registers and data paths by minimizing the additional delays to be incorporated in the original schedule. We expect this approach to better preserve the benefits of our chaining-oriented scheduling because most of the problem constraints are early taken into account.

4.1. Earliest-Starting-Time of Vector Operations

Loop scheduling is needed in order to exploit increasing vector locality within each PVL and PVLS loop and to promote vector chaining. The reader can refer to [12] for a discussion on the synchronization needed for PVLS loops. The key point to this approach is an algorithm that selects the vector operation which can start at the earliest time as an approach to find real chaining conditions. Below we present the model of the C240 vector processor that is used as the basis for evaluating the earliest-starting-times of vector operations as well as for loop scheduling.

The vector processor unit of C240 [9, 6] (Fig. 3) has the following vector pipelines (*Fus*): (1) concurrent vector load (M_{out}) and store (M_{in}), (2) add and logical (Fu_1), (3) multiply and divide (Fu_1), and (4) merge/compress and conditional pipes (not shown). The timing of some typical operations of the C240 is shown on Fig. 4. Any of the four register bank outputs (bo_i) can route vector data to any of three functional pipelines inputs (fi). The functional unit output (fo) and M_{out} can be routed to any of the

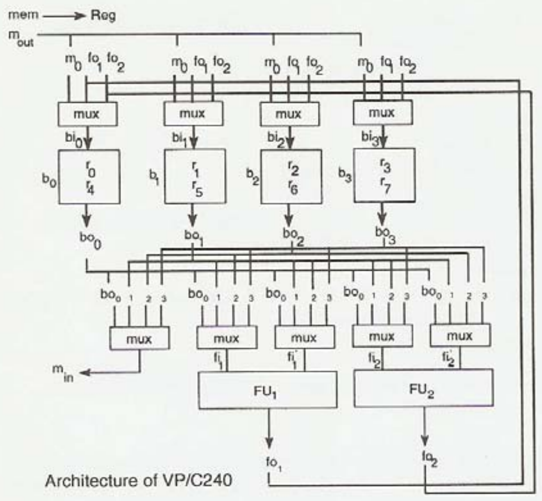

FIG. 3. Architecture of the C240 vector processor.

register bank inputs (*bi*). Each bank has two vector registers but only one bank input.

In the following, we investigate evaluation of the *earliest-starting-time* ($est(n)$) of vector operation n by using the resources of the C240.

We start by defining the notation used prior to finding the *est* time for each possible case. Variable vs denotes a vector or a scalar, $vs.n.$ denotes a vs that is input operand for vector operation n. $n.vs$ denotes a vector/scalar that is produced by n. $Fu(n)$ denotes the functional unit on which n is to run. Denote by $s(n)$, $t(n)$, $t_l(vs)$ the starting time of n, finishing time of n, and the time to load vs from memory into some register, respectively. $t(M)$ and $t(Fu)$ denote the earliest time the memory (M_{in} or M_{out}) and functional unit Fu are free, where M_{out} (loading) and M_{in} (storing) are memories used to establish two independent data paths (C240) with the vector processor.

Evaluation of the earliest-starting time ($est(n)$) is based on the previous status of the *Fus*, M_{in} and M_{out}, and the number of registers and data-paths used. Evaluation of the $est(n)$ for the C240 requires analysis of the following three cases:

• For Case 1, n requires loading of two vectors $vs_1.n$ and $vs_1.n$. Denote by $t_L(vs_1, vs_2) = t(M_{out}) + t_l(vs_1) + t_l(vs_2)$ the time at which both operands $vs_1.n$ and $vs_2.n$ will be loaded from the memory into some registers. As there is no possible chaining, $est(n)$ is defined by $est(n) = \max\{t_L(vs_1, vs_2), t(Fu(n))\}$.

• For Case 2, n requires loading of one vector $vs.n$ while the other operand is generated by a predecessor n'. If no chaining is possible ($Fu(n) = Fu(n')$), then $est(n)$ depends on the later of the loading time $t_L(vs)$ and the time $t(Fu(n'))$ the functional unit $Fu(n')$ is free, i.e., $est(n) = \max\{t_L(vs), t(Fu(n'))\}$.
In the other case ($Fu(n) \neq Fu(n')$), chaining can be done if at the chaining time ($t_c(n) = s(n') + \delta(n')$) the loading of $vs.n$ is complete and $Fu(n)$ is free, where $\delta(n')$ is the delay on of n due to its chaining with operation n'. If the previous condition is not satisfied, then running n may take place following the completion of n'. To summarize, the $est(n)$ will be defined by

$$est(n) = \begin{cases} t_c(n), & \text{if } t_c(n) \geq \max\{t_L(vs), t(Fu(n))\}, \\ \max\{t_L(vs), t(n'), t(Fu(n))\}, & \text{otherwise (no chaining).} \end{cases}$$

• For Case 3, both operands of n are generated by predecessors n' and n'', respectively. If there is no chaining ($Fu(n) = Fu(n') = Fu(n'')$), then $est(n) = t(Fu(n))$ because no loading is needed.
There is potential chaining on one unit when ($Fu(n) \neq Fu(n')$) but ($Fu(n) = Fu(n'')$). Chaining is possible if at the chaining time $t_c(n) = s(n') + \delta(n')$ the computation n'' is complete and $Fu(n)$ is free, where $\delta(n')$ is the time to store the first result of n'. In other words, $est(n)$ is defined by

$$est(n) = \begin{cases} t_c(n), & \text{if } t_c(n) \geq \max\{t(n''), t(Fu(n))\}, \\ \max\{t(Fu(n)), t(Fu(n''))\}, & \text{otherwise (no chaining).} \end{cases}$$

There is potential chaining on two units when $Fu(n) \neq Fu(n')$ and $Fu(n) \neq Fu(n'')$. We restrict ourselves to the case of the C240 that has two arithmetic units. Therefore, operations n' and n'' are sequentially executed and the chaining time of n is $t_c(n) = \max\{t'_c(n), t''_c(n)\}$, where $t'_c(n) = s(n') + \delta(n')$ and $t''_c(n) = s(n'') + \delta(n'')$. The chaining is possible if $Fu(n)$ is free at the chaining time $t_c(n)$, otherwise n is to start following the completion of both predecessors n' and n''. To summarize this case, we have

$$est(n) = \begin{cases} t_c(n), & \text{if } t_c(n) \geq t(Fu(n)), \\ \max\{t(Fu(n)), t(Fu(n')), t(Fu(n''))\}, & \\ \qquad \text{otherwise (no chaining).} & \end{cases}$$

Typical pipeline execution times for some operations

	Load Store	Integer Add/Sub	Integer Mul/Div	Floating Add/Sub	Floating Mul/Div
Clocks	3+vc/2	8+vc/2	12+11(vc-1)/8	12+vc	36+35(vc-1)/8

FIG. 4. Execution times (clocks) on the C240 vector processor.

Finally, we evaluate the earliest time ($t_{rc}(n)$) at which the needed resource (Fu, M_{out}, M_{in}, and bank inputs and outputs) for n becomes available. The register and data-path availability are examined at time point $est(n)$ that has been evaluated previously so that $t_{rc}(n) \geq est(n)$. If the needed resource is available, then $t_{rc}(n) = est(n)$. Otherwise, $t_{rc}(n)$ will be the earliest time the needed resource will be released. The final earliest-starting-time of n will be used by the scheduling algorithm.

4.2. Chaining-Oriented Scheduling

Using the above evaluation of the earliest-starting-time, the proposed vector-scheduling is based on minimizing the *Fu* idle times in an attempt to maximize the efficiency and to promote potential vector-chaining. Each loop is separately scheduled. Vector load is not considered as a separate operation but included within each of its arithmetic vector operation. At any given time, the operands of ready-to-run operations are available in memory or in some vector registers.

The notation used is as follows:

• The set of vectors or scalars that are consumed by operation n is denoted by $prod(n) = \{vs.n\}$. Set $cons(vs) = \{n\}$ denotes the set of operations that consumes vs. $Pred(n)$ and $Succ(n)$ denote the sets of predecessors and successors of n, respectively. $npred(n)$ is a variable that is initialized to the number of predecessor operations of n.

• $Fu(n)$ is the functional unit on which n should execute, i.e., the load/store, mul/div, or add/sub for the Convex C240. $Load(vs)$ is a flag that is used to state whether vs is loaded on some register. Times $t_L(vs)$, $T_s(vs)$, $t_{op}(n)$ are the times at which load, store, and operation n, will be complete respectively.

• Function $res(vs)$ is a dual operation on the number of available registers and the number of available data paths. Its operation consists of decrementing these numbers depending on the current operation. Note that the available resource is guarantee at the time operator $res(vs)$ is activated because of the method used to evaluate $est(n)$. Operation *Deallocate* frees a resource.

• Function *allocate n to Fu(n)* leads to generation of a vector machine instruction, i.e., adding vector $v1$ to $v2$ and storing the resulting data into vector $v3$ (Add $v1$, $v2$, $v3$).

The proposed vector scheduling is shown below. In Step 1, the set of ready-to-run operations are stored into A. Step 2 is the main loop. Step 2.1 selects operation n that can start at the earliest time as a heuristic to maximize the efficiency of the *Fus*. Step 2.2.1 updates the time $Fu(n)$ becomes free and allocates one register to the output of n. Step 2.2.2 performs loading input operands (if necessary), allocates one register, and updates the time M_{out} will be free. Step 2.2.3 checks whether a successor operation of n becomes ready to run ($npred(n) = 0$) and eventually inserts it in A. The case of a vector-store is treated in Step 2.3 that updates the time M_{in} becomes free following the store and the register holding the vector becomes free if the vector has no consumer operations.

The *Vector Scheduling* algorithm is the following:

```
Input: A set of parallel loops
Output: Program to execute the vector operations
(1) Initialize: $A \leftarrow \{n : Pred(n) = \emptyset\}$
(2) While $A \neq \emptyset$ Do
  Begin
  (2.1) Select $n \in A : est(n) = \min_{n' \in A}\{est(n')\}$
  (2.2) If $n \neq store$ then
    (2.2.1) Allocate $n$ to $Fu(n)$: $s(n) = est(n)$, $t(n) = s(n) +$
            $t_{op}(n)$, $t(Fu(n)) = t(n)$, $A \leftarrow A - \{n\}$, Allocate
            $reg(n.vs)$ at time $s(n)$
    (2.2.2) For each $vs.n$ do
              If $load(vs) = 0$ then Load $vs$ on $reg(vs)$,
                Allocate $reg(vs)$ at time $t(M_{out})$,
                $t(M_{out}) = t(M_{out}) + t_L(vs)$, and set
                $Load(vs) = 1$
              Endif
    (2.2.3) For each $n' \in Succ(n)$ Do:
              $npred(n') = npred(n') - 1$
                If $npred(n') = 0$ Then $A \leftarrow A + \{n'\}$
  (2.3) Else store $vs.n$: $s(n) = est(n)$, $t(n) = s(n) + t_s(vs)$,
        $t(M_{in}) = t(n)$
        If $cons(vs) = \emptyset$ then Deallocate reg($vs$) at time $t(n)$
  (2.2) Endif
end
```

The main loop of the allocation algorithm is statement 2 that executes m times because one vector operation is allocated for each run of the body, where m is the number of vector operations. Operation $npred(n') = npred(n') - 1$ executes $O(m^2)$ times but the condition $npred(n') = 0$ occurs only for each operation. The time complexity of the scheduling algorithm is $O(m^2)$.

5. MINIMIZING REGISTER AND DATA-PATH CONFLICTS

The previous scheduling guarantees that (1) different loads (M_{out}) or stores (M_{in}) are properly serialized, (2) allocation of vector operations to *Fus* do not conflict and, (3) resource availability is guaranteed. In the following we analyse the three types of conflicts that might result from the use of the previous schedule.

• *Overlapped lifetime of vectors* may lead to conflicting utilization if at least two vectors are allocated to the same register or to different registers but within the same register bank. Denote by $\omega_{sp}(v, v')$ the time cost, or delay over the finish time of the previous schedule, of spilling either v or v' whenever their lifetimes overlap. The cost $\omega_{sp}(v, v')$ is the time to store and load one vector's data.

• *Overlapped bank output* occurs when two vectors v and v' are allocated to registers of the same bank and their loading into the corresponding *Fus* overlap with respect to time. Denote by $\omega_{bout}(v, v')$ the cost associated with the least delay incurred to the schedule as a result of serializing the use of the bank output. Therefore, $\omega_{out}(v, v') = \min\{t(n') - s(n), t(n) - s(m')\}$, where n and n' are the operations that consume v and v', respectively, and $s(n)$ and $t(n)$ are the starting and ending times of n.

• *Overlapped bank input* occurs when two vectors v and v' are allocated to registers of the same bank and their storing into their registers overlap with respect to time. The associated cost ω_{bin} is the least delay incurred by the schedule when serializing the use of the bank input.

Based on the above time delays, we heuristically define the global weight $\omega(v, v')$ of edge (v, v') as the maximum delay caused by the interfering operations of vectors v and v'. Each vector v is further assigned a weight $\omega(v)$ that is the sum of all the delays caused to the schedule.

$$\omega(v) = \begin{cases} o, & \text{if no lifetime overlap,} \\ \sum_{v'} \max\{\omega_{sp}(v, v') + \omega_{bin}(v, v') + \omega_{bout}(v, v')\}, & \text{otherwise.} \end{cases}$$

In the following, we use *Weighted Graph Coloring* for allocating banks and registers to the vectors of each loop. The vectors used in each loop are associated an undirected graph in which a node v has the weight $\omega(v)$ and an edge (v, v') has the weight $\omega(v, v')$. The graph is formed by a collection of nonconnected subgraphs. The used coloring algorithm is a variant of [5]. The algorithm uses two heaps A and B:

```
1. Initially, $A$ contains all nodes except the one with the
   highest weight which is placed in $B$,
2. Repeat the following steps until $A$ is empty
   (a) Repeat the following step until $B$ is empty
       i. Remove a node $v$ from $B$ (highest weight), color
          $v$, and add to $B$ all immediate neighbors of $v$
          after removing them from $A$
   (b) Remove from $A$ the node (first node of a new com-
       ponent) with the highest weight and insert it into $B$
```

The time complexity of this algorithm is $O(cm^2)$, where c is the number of components and m is the number of vectors within a component.

6. PERFORMANCE EVALUATION

Our objective is to develop a compiler optimization to assists nonexpert programmers. We consider synthetic Fortran programs. Each is formed by a total of 30 arithmetic vector expressions. Each expression may include up to four vector operations. To generate these programs, the dependence edges between the expressions were randomly generated in order to specify the vector-operands for each expression including possible vector-loads. The maximum number of expressions that belong to the same level in the data dependence graph (DDG) were at most four and the number of levels was at least 5. All LCD was present in 20% of the expressions and all loop counts were set to 1000. We use $L(k)$ to denote that each loop is allowed to incorporate at most k expressions.

Let's $S_{C240/nopt}$ be the speedup of the highest optimization level of the Convex C240 over the No-Optimization or scalar execution. We also use the speedup $S_{opt/nopt}$, where *opt* refers to the proposed approach. Finally, we use the Speedup $S_{opt/C240}$ to compare our approach to that of the C240 highest optimization option. All the studied cases use one processor only. Table I shows the results of averaging the running of 10 programs for each value of $L(k)$.

The first entry in Table I shows the measured speedup of the C240 optimizer over the scalar processor ($-no$) when only one VP is used. The highest speedup is 11 which is measured when the program is formed by loops that contain only one vector expression. This speedup decreases with increasing the granule size of the loops ($L(1)$, $L(2)$, etc.). The reason is that the vector processor of the C240 has hardware support to implement the loop overhead that is much faster than the software approach used by the scalar processor.

With loops having few statements ($L(1)$, $L(2)$, and $L(3)$), the newly loaded vector data in the beginning of each loop could have been made available in vector registers during the running of previous loops. The overhead of multiple vector load and store is avoided in our approach through loop fusion. The effect of loop fusion allows improving the speedup as shown in $S_{opt/nopt}$ and $S_{opt/C240}$ of Table I. All assignment statements require storing data regardless of loop boundary. The difference is that with loop fusion some of these vector store can occur while the arithmetic pipelines can be active.

The effect of statement migration appears more sharply when the number of expressions within each loop is maximum ($L(max)$). In this case, the opportunity of loop fusion is reduced but there is still some benefit from transferring expressions from one loop to another in order to align vector data producers with vector data consumers as an attempt to promote pipeline chaining. The benefit of statement migration ($L(max)$) allowed to improve the speedup obtained by the C240 by 18% as shown by $S_{opt/C240}$.

Generally, loop fusion was studied for optimizing data locality through increasing reuse of data caches on scalar processors [14, 2]. In this work, loop fusion is investigated only for singly nested loops. However, it can be extended to multiple-nested loops by using the control structure that has been proposed for V-Pascal [17].

TABLE I
Speedup of C240 and This Approach (*opt*)—Different Programs and Different Allocations

Speedup	$L(1)$	$L(2)$	$L(3)$	$L(\max)$
$S_{C240/nopt}$	11	9.75	8.72	8.6
$S_{opt/nopt}$	15.73	12.87	10.55	10.15
$S_{opt/C240}$	1.43	1.32	1.21	1.18

The convex compiler has a number of traditional vector and parallel loop optimization such as strip-mining, loop-distribution, loop-interchanging, hoisting and sinking, pattern matching, etc. The newer version of the C240 optimizer now includes automatic parallelization of loops with routine calls. Currently, some loop fusion is being implemented [1] as part of the loop transformation phase of the Convex Compiler that uses the technique of loop unrolling and jamming.

6.1. *Effect of Scheduling and Allocation*

The objective of the second experiment is to study the effects of the proposed vector scheduling and resource allocation. For this, the C240 optimizer and our scheduler were run using the same restructured program in an attempt to isolate and compare the scheduling and allocation effects which have been done separately.

Table II shows the measured speedup for this experiment. The achievement of our scheduling and allocation are still those obtained in the first experiment. However, the C240 optimizer improves its speedup ($S_{C240/nopt}$), when using restructured programs, by a 10 to 24% factor compared to that achieved without restructuring the program.

The C240 can find more opportunities for chaining in the restructured programs as most producers/consumers are now located within the boundary of each loop. Our approach improves its speedup by a 9 to 15% factor over that of the C240 due to our scheduling and allocation.

The C240 optimizer uses the technique of reservation table in scheduling vector operations over the available pipelines [9]. The timing of all the operations is rounded so that it can either be one or two macro-cycles. Vector scheduling that is a variant of *list-scheduling* [11] is performed in the first pass. There are many reasons for inserting delays into the reservation table in the allocation pass. For example, delays are inserted [11] in order to satisfy the load/store timing and to avoid conflicting use of registers (spilling) and data-paths between registers and *Fus*. In general, the inserted delays negate some of the chaining benefits of the first step. When load/store, registers, and data paths requires major revision of the schedule (delays), some of the benefit of scheduling the with chaining in mind is lost.

In this approach, we use a more accurate model for the machine in the first step because the delays caused by the load/store and resource scarcity are considered in the scheduling decisions. Therefore, the generated schedule guarantees resource availability but cannot ensure conflict-free data paths. In the second step, less delays are inserted in the schedule to avoid conflicting use of registers and data paths. A conservative conclusion is that early incorporation of timing constraints in allocating the vector registers and data-paths yields better results than the approach used in commercial vectorizers such as that of the C240.

TABLE II
Effect of the Allocation and Scheduling of the VP—Same Programs but Different Allocations

Speedup	$L(1)$	$L(2)$	$L(3)$	$L(\max)$
$S_{C240/nopt}$	13.69	11.50	9.68	9.31
$S_{opt/nopt}$	15.73	12.87	10.55	10.15
$S_{opt/C240}$	1.15	1.12	1.09	1.09

7. CONCLUSION

The objective of this work was to investigate some aspects of global restructuring of synthetic programs based on restructuring of the source program and the use of a chaining-oriented machine dependent allocation. For this, we proposed reducing the granule size of scalar loops by extracting vectorizable and parallelizable statements. Furthermore, these statements can be fused within other loops of the same type and having similar loops count in an attempt to increase the granule size of parallel loops. The other benefit of this operation is that placing data producers closer to their consumers increases the opportunity for chaining their execution, and increasing reuse of data in vector-registers and cache-memory. At a lower level, we proposed a horizontal scheduling and allocation method based on locally maximizing the efficiency and incorporation of the vector load/store delays.

Our approach allows improving the highest optimization option of the convex C240 by a speedup factor fo 18 to 43%. Chaining-oriented scheduling and allocation gave an improvement of 9 to 15%.

Further extension to this work would be the integration of expert techniques in parallelism extraction for each type of loops. This would greatly help scientists in optimizing large-scale programs within the framework of an interactive optimizer.

ACKNOWLEDGMENTS

This work was done during the sabbatical year of the first author. Thanks to King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia, for its financial support of the sabbatical leave of the first co-author. Thanks to the Office of Academic Computing (OAC), the University of California Irvine, and especially to Mrs. Donald Frederick, Joseph Farran, and Paul Higgins for their assistance with the Convex C240 Mainframe. We especially thank Donald Frederick for useful discussions on vector processing.

REFERENCES

1. Based on personal communication with Steve Wallach, cofounder of Convex Computer, PO Box 833851, 300 Waterview Pkwy., Richardson, TX 75083-385. Internet: wallach@concave.convex.com.
2. Aba-Sufah, D. J., Kuck, W., and Lawrie, D. H. On the performance enhancement of paging systems through program analysis and transformation. *IEEE Trans. Comput.* **30**(5), 341–356 (1981).
3. Allen K. and Kennedy, J. R. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Programming Languages Systems* **9**(4), 491–542 (Oct. 1987).

4. Ayguade E. *et al.* GTS: Parallelization and vectorization of tight recurrences. *Proc. of the ACM Conf. on Supercomputing,* 1989, pp. 531–539.
5. Chaitan, G. J. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices* **17**(2), 201–207 (1982).
6. Chastain, G., Gostin, M., and Mankocich, J. The Convex C240 architecture. *IEEE Proc. Supercomput.* 321–329 (1988).
7. Choudhary A., and Hiranandani, S. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. *Proc. FMPC-92,* pp 4–11.
8. Coleman, H. B. The vectorizing compiler for the UNISYS ISP. *Proc. of the 1987 ICPP,* pp. 567–576.
9. Convex Computer Corporation, *The Convex Hardware Manual.* Convex Press, Richardson, TX, Feb. 1987.
10. Cytron, R. G. Doacross: Beyond vectorization for multiprocessors. *Proc. of the 1986 ICPP,* pp. 836–844.
11. Eisenbeis, W., Jalby, C., and Lichnewasky, A. Squeezing more CPU performance out of a CRAY-2 by vector block scheduling. *IEEE Proc. Supercomput.* 237–246 (1988).
12. Herbert, U. C. Some timings for synchronization on the multiprocessor system Cray X-MP. *Proc. of the Parallel Computing '85,* pp. 457–462. Elsevier, Amsterdam, 1986.
13. Kennedy K., and McKinley, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. *ACM Conference on Programming Language Design and Implementation,* 1989.
14. Kennedy K., and McKinely, K. S. Optimizing for parallelism and data locality. *Inter. Conf. On Supercomputing,* Washington, DC, 1992.

Received May 20, 1994; accepted November 22, 1994

15. Kuck D. J. *et al.* The structure of an advanced vectorizer for pipelines processors. In K. Hwang (Ed.). *Tutorial on Supercomputers: Designs and Applications,* pp. 163–178, 1984.
16. Polychronopoulos, C. P. *Parallel Programming and Compilers.* Kluwer, London, 1988.
17. Tsuda T., and Kunieda, Y. V-Pascal: An automatic vectorizing compiler for Pascal with no language extensions. *IEEE Proc. Supercomput.,* 182–198 (1988).
18. Warren, J. A hierarchical basis for reordering transformation. *Conference Record of the 11th Annual ACM Symp. on Principles of Programming Languages,* Jan. 1984.
19. Wolfe, M., and Banerjee, U. Data dependence and its application to parallel processing. *J. Parallel Programming* **16**(2), (Apr. 1987).
20. Wolfe, M. J. Optimizing supercompilers for supercomputers. Dept. of Comp. Sci. Univ. of Illinois at Urbana–Champain, R-82-1105, Oct. 1982.

MAYEZ AL-MOUHAMED received his B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Paris, in 1975, 1977, and 1982, respectively. Since 1983, he has been a member of the Computer Engineering Faculty at the King Fahd University of Petroleum and Minerals in Dhahran, Saudi Arabia. Dr. Al-Mouhamed's primary research interests lie in the areas of parallel computers and parallel programming languages with applications to real-time processing. The research problems he is currently working on include static and dynamic scheduling, parallelizing of real-time applications, shared-memory multiprocessors, robotics, and robotic vision.