Loop Fusion for Clustered VLIW Architectures

Yi Qian
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
yqian@mtu.edu

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
carr@mtu.edu

Philip Sweany
Texas Instruments
P.O. Box 660199, MS/8649
Dallas, TX 75266-0199
sweany@ti.com

ABSTRACT

Embedded systems require maximum performance from a processor within significant constraints in power consumption and chip cost. Using software pipelining, high-performance digital signal processors can often exploit considerable instruction-level parallelism (ILP), and thus significantly improve performance. However, software pipelining, in some instances, hinders the goals of low power consumption and low chip cost. Specifically, the registers required by a software pipelined loop may exceed the size of the physical register set.

The register pressure problem incurred by software pipelining makes it difficult to build a high-performance embedded processor with a single, multi-ported register bank with enough registers to support high levels of ILP while maintaining clock speed and limiting power consumption. The large number of ports required to support a single register bank severely hampers access time. The port requirement for a register bank can be reduced via hardware by partitioning the register bank into multiple banks connected to disjoint subsets of functional units, called *clusters*. Since a functional unit is not directly connected to all register banks, wasted energy and resources can result due to delays incurred when accessing "non-local" registers.

The overhead due to partitioning of the register set can be ameliorated by using high-level compiler loop optimization techniques such as unrolling, unroll-and-jam and fusion. High-level loop optimizations spread data-independent parallelism across clusters that may not require "non-local" register accesses and can provide work to hide the latency of any such register accesses that are needed.

In this paper, we examine the effects of loop fusion on DSP loops run on four simulated, clustered VLIW architectures and the Texas Instruments TMS320C64x. Our experiments show a 1.3 – 2 harmonic mean speedup.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

General Terms

Languages

Keywords

loop fusion, clustered VLIW architectures

1. INTRODUCTION

Embedded systems have attracted research interest recently due to their increasing applications in many telecommunication, image processing, and industry control products. To meet performance demands, high-performance embedded systems have adopted processors that issue and execute multiple instruction-level operations in parallel. A high degree of instruction-level parallelism (ILP) exploited via compiler techniques, such as software pipelining, allows embedded system applications to make maximum use of multiple functional units. However, high ILP also puts a large demand on machine resources. Supporting a high degree of parallelism via a single large register bank is not feasible since this kind of architecture actually impedes the overall performance. Allowing a large number of simultaneous reads/writes for registers can dramatically increase the cycle time of a processor. Therefore, some state-of-the-art embedded system architectures, in particular DSP chips (e.g., TI TMS320C6x), use clustering to simplify hardware design and achieve a high clock rate while maintaining modest power consumption and chip size.

Clustered VLIW machines use several small register banks with a low number of ports instead of one large, highly ported register bank. Each register bank is grouped with one or more functional units that can access data directly only from the local register bank. These functional unit/register bank groups are called *clusters*. The overhead for clustered VLIW machines occurs with the intercluster communication required when a functional unit needs a value that resides in another cluster. The efficiency of clustered architectures mainly depends on both wisely partitioning instructions among clusters to minimize intercluster data movement and on scheduling instructions to maximize utilization of functional units.

Previous work in code generation for clustered VLIW architectures has mostly concentrated on methods to partition virtual registers amongst the target architecture's clusters. Since embedded DSP systems usually have a large computation demand where a majority of execution time is typically spent in loops, high-level loop transformations present an excellent opportunity for improving the quality of partitioned code. Loop unrolling and unroll-and-jam have already been employed for exposing more parallelism on clustered VLIW machines [11, 19, 20]. In this paper we will in-

vestigate the effect of another important loop transformation, loop fusion, on clustered VLIW machines.

This paper is organized as follows. Section 2 summarizes the related work in generating code for clustered VLIW architectures. Section 3 gives the background related to partitioning and loop transformations. Section 4 presents a high-level optimization strategy for clustered VLIW architectures and Section 5 details our experimental results. Finally, Section 6 presents our conclusions.

2. RELATED WORK

Considerable recent research has addressed the problems of clustered architectures. Ellis [8] describes an early solution to the problem of generating code for partitioned register banks in his dissertation. His method, called BUG (bottom-up greedy), is applied to a scheduling context at a time (*e.g.*, a trace) and is intimately intertwined with instruction scheduling, utilizing machine-dependent details within the partitioning algorithm.

Özer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [16]. They state that UAS is an improvement over BUG since UAS can perform schedule-time resource checking while partitioning, allowing UAS to manage the partitioning with knowledge of the bus utilization for copies between clusters. Özer's study of entire programs shows, for their best heuristic, an average degradation of roughly 19% on an 8-wide, two-cluster architecture when compared to an unrealizable monolithic-register-bank architecture with the same level of ILP.

Nystrom and Eichenberger [15] present an algorithm that first performs partitioning with heuristics that consider modulo scheduling. Specifically, they prevent inserting copies that will lengthen the recurrence constraint of modulo scheduling. If copies are inserted off critical recurrences in recurrence-constrained loops, the initiation interval for these loops may not be increased if enough copy resources are available. Nystrom and Eichenberger report excellent results for their technique.

Hiser, et al. [9, 10], describe our experiments with register partitioning in the context of whole programs and software pipelining. Our basic approach, described in [9] and summarized in Section 3, abstracts away machine-dependent details from partitioning with edge and node weights, a feature extremely important in the context of a retargetable compiler. Using architectures similar to those used in [16], we found a degradation in execution performance of 10% on average compared to an unrealistic single-register-bank architecture. Experiments with software pipelining show that we can expect a 10–25% degradation for software pipelined loops over a 16-wide architecture with one register bank. While this is more degradation than Nystrom and Eichenberger report, an architecture with significantly more ILP was used.

Sánchez and González [19, 20] have studied the effects of inner-loop unrolling on modulo scheduled loops on architectures with partitioned register banks. In addition to inner-loop unrolling, our method will use loop fusion [12], unroll-and-jam [7] and loop alignment [5] to improve modulo scheduled loops [11].

Huang, et al., describe a loop transformation scheme to enhance ILP for software pipelined loops for partitioned register bank architectures [11]. This method uses unroll-and-jam to increase parallelism within a single cluster, then unrolls the loop level with the lowest communication overhead to improve parallelism across clusters. Experiments show that the transformed loops obtain 27% and 20% average improvement in unit II over untransformed loops on a 8-wide machine with 2 clusters and 4 clusters, respectively. In this paper, we will examine another important loop transformation, loop fusion, for clustered VLIW architectures.

3. BACKGROUND

3.1 Software Pipelining

While local and global instruction scheduling can together exploit a large amount of parallelism for non-loop code, to best exploit instruction-level parallelism within loops requires software pipelining. Software pipelining can generate efficient schedules for loops by overlapping the execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelining where speed-up is achieved by overlapping execution of different operations.

Allan et al. [3] provide a good summary of current software pipelining methods, dividing software pipelining techniques into two general categories called *kernel recognition* methods [2, 4] and *modulo scheduling* methods [14, 18]. The software pipelining used in this work is based upon modulo scheduling.

Modulo scheduling selects a schedule for one iteration of the loop such that, when that schedule is repeated, no resource or dependence constraints are violated. This requires analysis of the data dependence graph (DDG) for a loop to determine the minimum number of instructions required between initiating execution of successive loop iterations. Once this minimum initiation interval (MinII) is determined, instruction scheduling attempts to minimize the loop schedule length. If a schedule of MinII instructions can be found that does not violate any resource or dependence constraints, modulo scheduling has achieved a minimum schedule. If not, scheduling is attempted with MinII $+ 1$ instructions, and then MinII $+ 2$, ..., continuing up to the worst case which is the number of instructions required for local scheduling. The first value of initiation interval (II) to produce a "legal" schedule of the DDG becomes the actual initiation interval. After a schedule for the loop itself has been found, code to set up the software pipeline (prelude) and drain the pipeline (postlude) are added. Rau [18] provides a detailed discussion of an implementation of modulo scheduling, and in fact our implementation is based upon Rau's method.

3.2 Partitioning

In clustered VLIW architectures a distinct set of registers is associated with each cluster of functional units. An example of such an architecture is the Texas Instruments C6x [22]. Operations performed in any functional unit require registers with the proper associated register bank, and copying a value from one register bank to another is expensive. The problem for the compiler, then, is to allocate registers to banks to reduce the number of copies, while retaining a high degree of parallelism.

Our approach to this problem is as follows [10, 9]:

1. build intermediate code with symbolic registers, assuming a single infinite register bank,
2. build data dependence graphs (DDGs) and perform software pipelining still assuming an infinite register bank,
3. partition the registers to register banks (and thus preferred functional unit(s)) as described below.
4. apply value cloning for induction variables and loop invariants [13],
5. re-build DDGs and perform instruction scheduling attempting to assign operations to the "proper" (cheapest) functional unit based upon the location of the registers, and
6. with functional units specified and registers allocated, perform "standard" Chaitin/Briggs graph coloring register assignment for each register bank [6],

3.2.1 Partitioning Registers by Components

Our method builds a graph, called the *register component graph* (RCG), whose nodes represent register operands (symbolic registers) and whose arcs indicate that two registers "appear" in the same (atomic) operation. Arcs are added from the destination register to each source register. We build the register component graph with a single pass over an "ideal" instruction schedule which by our definition, uses all the characteristics of the actual architecture, except that it assumes that all registers are in a single multi-ported register bank. In this work, we will use the greedy partitioning method described in [10] to partition the RCG built from the "ideal" schedule. This method assigns edge weights to the RCG based upon communication costs with positive weights for edges that connect nodes that should go in the same register bank and negative weights for edges whose adjacent nodes should go in separate clusters. The edge and node weights are based upon scheduling freedom, parallelism and copy costs.

3.2.2 A Partitioning Example

To demonstrate the register component graph method of partitioning, consider the following matrix multiply code.

```
for (i = 0; i < n;i++)
 for (j = 0; j < n; j++)
 {
   c0 = 0.0;
   for (k = 0; k < n; k+=2)
     c0 += a[i][k] * b[k][j];
     c0 += a[i][k+1] * b[k+1][j];
 }
```

We software pipeline the inner loop, assuming a single cycle latency for add, and a two-cycle pipeline for multiply. In addition we assume that all loads are cache hits and also use a two-cycle pipeline. An ideal schedule for this inner loop is shown in Figure 1. The corresponding register component graph appears in Figure 2. The loop kernel requires 4 cycles to complete.[1]

load r1, a[i][k]	add r4,r4,r5
load r2, b[k++][j]	mult r5,r7,r6
load r7, a[i][k]	add r4,r4,r3
load r6, b[k++][j]	mult r3,r1,r2

Figure 1: Ideal Schedule

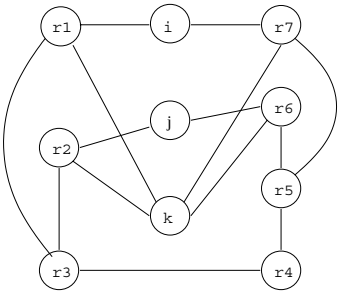

Figure 2: Register Component Graph

[1]We assume the availability of a load command that allows for either pre-increment or post-increment of the address operand.

load r1, a[i1][++k1]	copy r55, r5
load r2, b[k1++][j1]	mult r5,r7,r6
add r4,r4,r55	nop
add r4,r4,r3	load r7, a[i2][++k2]
mult r3,r1,r2	load r6, b[k2++][j2]

Figure 3: Partitioned Schedule

One potential partitioning of the graph in Figure 2 (given the appropriate edge and node weights) is the following:

$$P_1 = \{r1, r2, r3, r4, i1, k1, j1\}, P_2 = \{r5, r6, r7, i2, k2, j2\}$$

In the above partition we assume that i and j (read-only variables) and k (an induction variable) are cloned to have one copy in each of the clusters. This assumption allows us to produce the partitioned code of Figure 3, which requires 5 cycles for the loop kernel, a degradation of 20% over the ideal schedule of Figure 1. The additional instruction is necessary to copy the value of r5 from cluster 2 to cluster 1, where it is known as r55.

3.3 Loop Transformations

In this section, we review our previous work [11] showing how three loop transformations – unroll-and-jam [7], loop unrolling and loop alignment [5] – can be used to enhance software pipelining on clustered VLIW architectures. The goal with these transformations is to increase the amount of data-independent parallelism in innermost loops, leading to a better partition and software pipeline. We begin by showing how unroll-and-jam can balance the computation and memory-access requirements of a loop with the corresponding resources provided by a single cluster. Then, we show how unroll-and-jam and/or loop unrolling can spread computation across all nodes in the cluster.

Consider the following loop.

```
for(i = 0;  i < n; i++)
  for(j = 0; j < 2*n; j++)
  {
    c0 = 0;
    for(k = 0; k < n; k++)
      c0 += a[i][k] * b[k][j];
    c[i][j] = c0;
   }
```

The innermost loop has 2 load instructions and one fixed-point multiply-accumulate instruction. On an architecture that can perform 2 memory operations and 2 fixed-point operations per cycle, the loop has idle computation cycles. If we unroll-and-jam the j-loop by a factor of 2, we get

```
for(i = 0;  i < n; i++)
  for(j = 0; j < 2*n; j+=2)
  {
    c0 = 0;
    c1 = 0;
    for(k = 0; k < n; k++)
    {
      a0 = a[i][k];
      c0 += a0 * b[k,j];
      c1 += a0 * b[k,j+1];
    }
    c[i][j] = c0;
    c[i][j+1] = c1;
  }
```

This loop has 3 memory operations and 2 multiply-accumulate operations per iteration of the innermost loop. The resource requirements of this loop more closely match what the target architecture provides, resulting in better performance. We use the method described in [7] to balance loops for a single cluster.

After improving the parallelism within a cluster, unroll-and-jam (or unrolling) is performed to spread the parallelism in an innermost loop across clusters. This transformation is analogous to a parallel loop where different iterations run on different processors. On a clustered architecture, each unrolled iteration can be executed on separate clusters.

Consider the following unrolled loop.

```
for (j = 0; j < 2*n; j+=2)
{
  a[j] = a[j] + 1; /* iteration j */
  a[j+1] = a[j+1] + 1; /* iteration j+1 */
}
```

Since there are no dependences between the iterations, no intercluster communication is required if iterations `j` and `j+1` are executed on different processors and value cloning is applied [13]. However, when a loop carries a dependence, communication between register banks may be needed.

Consider the following unrolled loop.

```
for (j = 0; j < 2*n; j+=2)
{
  a[j] = a[j-1] + 1; /* iteration j */
  a[j+1] = a[j] + 1; /* iteration j+1 */
}
```

Before unrolling, there is a true dependence from the first statement to itself carried by the `j`-loop. After unrolling, if the code for iteration `j` is executed on one cluster and the code for iteration `j+1` on another, there will be communication between clusters.

Not all loop-carried dependences require communication. As in shared-memory parallel code generation, loop alignment can be used to change a loop-carried-dependence into a loop-independent dependence. In the following loop, there is a loop-carried dependence from `a[j]` to `a[j-1]`.

```
for (j = 1; j < 2*n+1; j++)
{
  a[j] = b[j] + 1;
  c[j] = a[j-1] + 1;
}
```

This loop can be aligned and unrolled as follows.

```
c[1] = a[0] + 1
for (j = 1; j < 2*n; j+=2)
{
  a[j] = b[j] + 1;     /* iteration j */
  c[j+1] = a[j] + 1;   /* iteration j */
  a[j+1] = b[j+1] + 1; /* iteration j+1 */
  c[j+2] = a[j+1] + 1; /* iteration j+1 */
}
a[2*n+1] = b[2*n+1] + 1;
```

Now, we can schedule iteration `j` on one cluster and iteration `j+1` on another cluster with good intercluster parallelism.

Unfortunately, alignment is not always possible, as is the case in the earlier example containing a recurrence. Alignment is limited by two types of dependences: recurrences and multiple dependences between two statements with differing dependence distances. Each of these restrictions on alignment is called an *alignment conflict* because alignment cannot change all loop-carried dependences into loop-independent dependences. An alignment conflict represents register-bank communication if the source and the sink of the dependence are put in separate clusters.

Although actually aligning a loop is unnecessary to expose the intercluster parallelism as shown in [11], alignment conflicts are used as a guide for choosing a loop to unroll. Our method unrolls the loop containing the fewest alignment conflicts, since this loop will introduce the fewest copies between clusters.

4. LOOP FUSION

In this section, we show how loop fusion, used as a precursor to unrolling, can be used to improve loop performance on clustered VLIW architectures. Our complete loop optimization strategy is as follows.

1. Fuse loops to enhance intracluster parallelism and enhance the intercluster parallelism later exploited by unrolling.
2. Unroll-and-jam loops to increase intracluster parallelism.
3. Unroll-and-jam or unroll loops to enhance data-independent parallelism across multiple clusters.

The profitability of fusion is computed by examining its effect on unrolling. By combining two (or more) loop nests into a single nest, operations from the second loop can be scheduled in any empty slots that exist in the original loop. The resulting loop may provide more parallelism than the two loops in isolation, may lessen the overhead due to loop code, and often takes less code space. However, loop fusion may degrade performance if the fused loop increases the amount of intercluster communication. Our method fuses as many loops as possible while retaining low communication overhead. In the rest of this section, we introduce our cost model for determining the profitability of fusion, and then present a slightly modified greedy fusion algorithm.

4.1 Communication cost

Although loop fusion provides opportunities to make efficient use of clustered VLIW machines, careless use of loop fusion may hamper performance due to high intercluster communication. For example, given the following loops:

```
for (i = 0; i < n; i++)
   a[i] = x[i] + q;
for (i = 0; i < n; i++)
   b[i] = a[i] * c[i];
for (i = 0; i < n; i++)
   d[i] = a[i-1] + b[i];
```

fusing them together gives the loop

```
for (i = 0; i < n; i++)
{
   a[i] = x[i] + q;
   b[i] = a[i] * c[i];
   d[i] = a[i-1] + b[i];
}
```

This loop provides opportunities for data reuse involving references to `a[i]` and `b[i]`. However, extra alignment conflicts are introduced into the fused loop. One dependence path from the first statement to the third statement, due to references to `a[i]` and `b[i]`, has

a total distance of 0, while a second dependence path between these two statements due to references to `a[i]` and `a[i-1]` has a distance of 1. This is an alignment conflict that may restrict intercluster parallelism when the loop is unrolled for multiple clusters.

To avoid introducing additional alignment conflicts, we can fuse the first two loops, or the second and third loop, instead of fusing all three together. The resulting loops can be unrolled to generate independent copies of statements that can be executed in parallel across multiple clusters.

The following example illustrates another case when loop fusion can adversely affect unroll-and-jam.

```
for (i = 0; i < m; i++)
{
  s = 0;
  for (j = 0; j < n; j++)
    s = s + a[j] * b[j];
}
for (i = 0; i < m; i++)
{
  c[i] = c[i-1] * q
  for (j = 0; j < n; j++)
    d[j] = c[i] + a[j] * b[j]
}
```

The `i`-loop in the first loop nest contains no alignment conflicts, indicating unroll-and-jam will be beneficial. This is not the case for the second loop nest where the `i`-loop carries a recurrence and makes unroll-and-jam unprofitable at this level. Instead, unrolling the `j`-loop in the second nest will produce independent copies of the loop body. In this example, fusing two loops together is not a good choice since either loop level of the fused loop will carry a recurrence that inhibits the ability of unroll-and-jam and unrolling to create data-independent intercluster parallelism.

Based upon the above discussion, we apply loop fusion when the fused loop does not increase the number of alignment conflicts. To compute this, we first introduce a notion of communication cost that will be used in our fusion algorithm. Communication cost at level l of a loop L, $C_L(l)$, is defined as the number of alignment conflicts at that level. We also define the set of minimal loop levels for a loop nest L, Min_L, as a set of loop levels that have the minimal communication cost among communication costs at all levels. The algorithm for determining whether loop fusion is profitable is shown in Figure 4. The algorithm first computes the communication cost for the original loops and the fused loop, and finds common loop levels at which both loops have minimal communication cost. These loop levels are good candidates for applying unroll-and-jam or unrolling during future transformations. The algorithm returns true if the communication cost of the fused loop at one of these levels is equal to the sum of the communication costs of the original loops at the same level, i.e., no extra alignment conflicts are introduced into the fused loop at this level.

It is worth noting that legal fusion does not introduce new recurrences into the fused loop, making this transformation more attractive.

4.2 Implementing Fusion

It has been shown that finding the optimal solution for a global fusion problem is NP-hard, and in practice a greedy heuristic for applying fusion is simple and sufficient for finding good solutions [12]. In our work, we perform loop fusion based upon the greedy algorithm proposed in [12] using the algorithm in Figure 4 to evaluate profitability.

```
procedure FusionIsProfitable(L1, L2)
Input: L1, L2: original loops
  compute C_L1, C_L2
  compute Min_L1, Min_L2
  compute C_Lf for the fused loop Lf
  commonLevels = Min_L1 ∩ Min_L2
  if commonLevels ≠ φ
    if ∃ l ∈ commonLevels and
      C_Lf(l) = C_L1(l) + C_L2(l)
      return true;
  return false;
```

Figure 4: Algorithm to Determine Fusion Profitability

Kennedy's approach [12] begins by constructing a fusion graph, where each node n represents a loop, and an edge (n_1, n_2) is added between two nodes if there exists a dependence between two corresponding loops. Then the legality of loop fusion is examined. An edge (n_1, n_2) is marked fusion-preventing if

- loop headers are not compatible, or
- a forward loop-independent dependence is a backward loop-carried dependence after fusion, or
- two loops are in a bad path. A bad path is a path from n_1 to n_2 which includes a node that can not be fused with either n_1 or n_2.

Finally the approach iteratively selects two fusible loops and fuses them together, if fusion is beneficial, until no such loop pair can be found.

Originally the communication cost is computed for each node and each edge as if the sink and source node of the edge were in the same loop. After fusion, two fusible nodes are collapsed into a single node, and the edges that connect the two old nodes are also collapsed. The communication cost of the new node is equal to the sum of communication costs of the two old nodes, and the communication costs for the edges into and out of the fused node are re-computed.

5. EXPERIMENTAL RESULTS

We have implemented the fusion algorithm described in Section 4 in Memoria, a source-to-source Fortran transformer based upon the DSystem [1]. Memoria already contains the unrolling algorithm described in Section 3.3. We have evaluated the effectiveness of our transformations on a simulated architecture, called the URM [17], and the Texas Instruments TMS320C64x.

The benchmark suite for this experiment consists of 119 loops that have been extracted from a DSP benchmark suite provided by Texas Instruments. The suite includes two full applications and 48 DSP kernels. We converted the DSP benchmarks from C into Fortran by hand so that Memoria could process them. After conversion, each C data type or operation is replaced with a similar Fortran data type or operation. For example, unsigned integers are converted into integers in the Fortran code and bitwise operations in C are converted into the corresponding bitwise intrinsics in Fortran. By defining the operation cycle counts properly, this conversion allows us to achieve accurate results from our experiments.

Fusion is applicable to 12 sets of loops consisting of 25 total loops. Table 1 describes the main functions of those 25 loops. The

speedups reported in the rest of this section are reported over only these 25 loops.

Loop Set #	Description
1,2	solves the error locater polynomial equation
3	scales up or down a line of data
4	performs 2D wavelet transform on input data
5,6,7,8,9	performs shift and dot product on vectors
10	performs multiplication, accumulation, and subtraction on an array and finds the largest value
11	performs dot product on vectors
12	Transposes the data to get pixel planes

Table 1: Benchmark Descriptions

5.1 URM Results

For the URM, we have compiled the code generated by Memoria with Rocket [21], a retargetable compiler for ILP architectures. Rocket performs cluster partitioning [10], software pipelining [18] and Chaitin/Briggs style register assignment [6]. In this experiment, Rocket targets four different clustered VLIW architectures with the following configurations:

1. 8 functional units with 2 clusters of size 4
2. 8 functional units with 4 clusters of size 2
3. 16 functional units with 2 clusters of size 8
4. 16 functional units with 4 clusters of size 4

Each cluster has 48 integer registers. All functional units are homogeneous and have instruction timings as shown in Table 2. Each machine with 8 functional units can perform one copy between register banks in a single cycle, while the 16 functional unit machines can perform two per cycle.

Operations	Cycles
integer copies	1
float copies	1
loads	5
stores	1
integer mult.	2
integer divide	12
other integer	1
other float	2

Table 2: Operation Cycle Counts

Tables 3, 4 and 5, show the effect of fusion, unroll-and-jam and unrolling on the 25 loops in our benchmark suite to which fusion is applicable. The rows labeled "Average" show the average speedup in unit II (II per unrolled iteration) obtained by the transformed loops when compared with the unit II of loops without the transformations. For the unfused loops we use the sum of the original unit IIs and compare that to the unit II of the fused loop. The rows labeled "Harmonic", "Median" and "Std Dev" give the harmonic mean speedup in unit II, the median speedup in unit II and the standard deviation, respectively.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.59	1.59	1.66	1.59
Harmonic	1.48	1.48	1.56	1.48
Median	1.38	1.38	1.9	1.38
Std Dev	0.45	0.45	0.4	0.45

Table 3: URM Speedups: Fused vs. Original

In Table 3, we show the effects of performing only loop fusion on our set of loops. We have observed a 1.48 – 1.56 harmonic mean speedup in unit II using fusion over the four target architectures. For most of these tests, the architecture configuration made little difference. This is because the original loops do not have enough parallelism to occupy all of the issue slots. As is often the case, the improvements on architectures with fewer partitions are greater due to a reduced communication overhead. Only the 16-wide two-cluster machine produces different results.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.61	1.62	1.83	1.62
Harmonic	1.50	1.49	1.69	1.49
Median	1.57	1.55	1.94	1.57
Std Dev	0.45	0.51	0.52	0.48

Table 4: URM Speedups: Fused & Unrolled vs. Unrolled only

Table 4 shows the improvements obtained by loop fusion over unrolling (either inner- or outer-loop unrolling). Again, we see significant gains with a 1.49 – 1.69 harmonic mean speedup in unit II. The speedups on individual loops ranged from 1.06 to 2.67, with the median speedup ranging from 1.55 to 1.94.

Finally, Table 5 shows the total improvement of fusion and unrolling over the original untransformed loop. We have observed a 1.74 – 2.03 harmonic mean speedup in unit II and a median speedup of 2. The speedups for individual loops range from 1.14 to 3.33.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	2.15	1.98	2.15	2.01
Harmonic	1.9	1.74	2.03	1.82
Median	2	2	2	2
Std Dev	0.79	0.76	0.52	0.62

Table 5: URM Speedups: Fused & Unrolled vs. Original

The largest speedups are consistently on the architecture with 16 functional units arranged in 2 clusters. This is because this architecture offers a large amount of ILP with a relatively low communication overhead since there are only two clusters.

Our results for the URM show that loop fusion offers tremendous potential for improving code on clustered VLIW architectures. Fusion is applicable to 21% of the loops found in our benchmark suite and when it is applied, significant improvements are obtained.

5.2 TMS320C64x Results

In addition to the results for the URM, we have evaluated loop fusion on the Texas Instruments TMS320C64x (or C64x). The C64x

CPU is a two-cluster VLIW fixed-point processor with eight functional units that are divided equally between the clusters. Each cluster is directly connected to one register bank having 32 general purpose registers. All eight of the functional units can access the register bank in their own cluster directly, and the register bank in the other cluster through a cross path. Since only two cross paths exist, a total of up to two cross path source reads can be executed per cycle. Other intercluster value transfers have to be done via explicit copy operations. On the C64x, multiply instructions have one delay slot, load instructions have four delay slots, and branch instructions have five delay slots. Most other instructions have zero delay slots, while some can have up to three delay slots [22].

To make use of the C64x C compiler, we converted the fused code generated by the Memoria into C by hand. Then, both the original and transformed versions of the code were compiled and run on the C64x. When using the C64x compiler, we chose the highest optimization level (-o3) [23]. The TI compiler already performs unrolling, so our measurements only show the effects of loop fusion. Turning off unrolling in the TI compiler is not acceptable since it can affect the quality of code generated even if Memoria performs the unrolling. We summarize the results obtained on the C64x in Table 6.

	Speedup
Average	1.46
Harmonic	1.33
Median	1.33
Std Dev	0.46

Table 6: TMS320C64x Speedups: Fused vs. Original

On the C64x, fusion achieves an average speedup in unit II of 1.46 and a harmonic mean speedup of 1.33. The speedups ranged from 1 to 2 with a median speedup of 1.33. Four of the loop sets have a speedup of 2, and an additional four of loop sets do not improve with fusion. The mean speedup improvement on all loop sets is less than that seen on the URM. The difference in speedup and the lack of any speedup in four of the loops can be attributed to the availability of cross paths on C64x. The cross paths allow communication between clusters without an explicit copy operation. This reduces the overhead of the communication compared to the URM and offers fewer opportunities for improvements.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the effect of loop fusion on clustered VLIW architectures. Loop fusion enhances instruction-level parallelism, generates compact code and improves the utilization of functional units. Our method uses alignment conflicts as a metric for guiding loop fusion and preventing aggressive fusion that would restrict intercluster parallelism created by unroll-and-jam.

We have implemented a greedy loop fusion algorithm and have run an experiment on DSP benchmarks for four different simulated, clustered VLIW architectures, and the TI TMS320C64x. Our results show that by performing loop fusion, we can achieve a 1.3 – 2 harmonic mean speedup in the unit II of a loop.

In the future, we plan to integrate the profitability estimate of loop fusion with a new metric based upon software pipelining. This new metric will allow us to determine if an increase in copy operations can be hidden within the code generated by software pipelining and to determine the change in efficiency in the final code. In addition, we will use this metric to guide unroll-and-jam and loop unrolling.

Since clustered VLIW embedded processors are increasing in use, it is important for compilers to generate excellent code within tight constraints. The work presented in this paper is a positive step in generating high performance code on clustered architectures. We believe that high-level transformations, such as loop fusion, should be an integral part of compilation for clustered VLIW architectures.

7. ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation under grant CCR-9870871. The authors would like to thank Texas Instruments for providing the benchmark suite used in this experiment and Dr. John Linn for his help in obtaining the benchmarks.

8. REFERENCES

[1] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Conference on Programming Language Design and Implementation*, pages 308–317, Atlanta Georgia, June 1988. SIGPLAN '88.

[3] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3), September 1995.

[4] V. Allan, M. Rajagopalan, and R. Lee. Software pipelining: Petri net pacemaker. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, January 20-22 1993.

[5] R. Allen and K. Kennedy. Advanced compilation for vector and parallel computers. Morgan Kaufmann Publishers, San Mateo CA.

[6] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, OR, July 1989.

[7] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Prog. Lang. Syst.*, 16(6):1768–1810, 1994.

[8] J. R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1984.

[9] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compiler Techniques*, pages 13–23, Philadelphia, PA, October 2000.

[10] J. Hiser, S. Carr, P. Sweany, and S. Beaty. Register partitioning for software pipelining with partitioned register banks. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 211–218, Cancun, Mexico, May 2000.

[11] X. Huang, S. Carr, and P. Sweany. Loop transformations for architectures with partitioned register banks. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 48–55, Snowbird, UT, June 2001.

[12] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[13] D. Kuras, S. Carr, and P. Sweany. Value cloning for architectures with partitioned register banks. In *The 1998 Worshop on Compiler and Architecture Support for Embedded Systems*, Washington D.C., December 1998.

[14] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, July 1988.

[15] E. Nystrom and A. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31 International Symposium on Microarchitecture (MICRO-31)*, pages 103–114, Dallas, TX, December 1998.

[16] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 308–316, Dallas, TX, December 1998.

[17] D. Poplawski. The unlimited resource machine (URM). Technical Report 95-01, Michigan Technological University, Jan. 1995.

[18] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelined loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, November 29-Dec 2 1994.

[19] J. Sànchez and A. Gonzàlez. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Proceedings of the 2000 International Conference on Parallel Processing*, Toronto, Canada, August 2000.

[20] J. Sànchez and A. Gonzàlez. Instruction scheduling for clustered VLIW architectures. In *Proceedings of 13th International Symposium on System Systhesis (ISSS-13)*, Madrid, Spain, September 2000.

[21] P. H. Sweany and S. J. Beaty. Overview of the Rocket retargetable C compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.

[22] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000. literature number SPRU189.

[23] Texas Instruments. *TMS320C6000 Optimizing Compiler User's Guide*, 2000. literature number SPRU187.