Optimizing Loop Performance for Clustered VLIW Architectures

Yi Qian
Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
{yqian, carr}@mtu.edu

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

Abstract

Modern embedded systems often require high degrees of instruction-level parallelism (ILP) within strict constraints on power consumption and chip cost. Unfortunately, a high-performance embedded processor with high ILP generally puts large demands on register resources, making it difficult to maintain a single, multi-ported register bank. To address this problem, some architectures, e.g. *the Texas Instruments TMS320C6x, partition the register bank into multiple banks that are each directly connected only to a subset of functional units. These functional unit/register bank groups are called* clusters.

Clustered architectures require that either copy operations or delay slots be inserted when an operation accesses data stored on a different cluster. In order to generate excellent code for such architectures, the compiler must not only spread the computation across clusters to achieve maximum parallelism, but also must limit the effects of intercluster data transfers.

Loop unrolling and unroll-and-jam enhance the parallelism in loops to help limit the effects of intercluster data transfers. In this paper, we describe an accurate metric for predicting the intercluster communication cost of a loop and present an integer-optimization problem that can be used to guide the application of unroll-and-jam and loop unrolling considering the effects of both ILP and intercluster data transfers. Our method achieves a harmonic mean speedup of 1.4 – 1.7 on software pipelined loops for both a simulated architecture and the TI TMS320C64x.

1. Introduction

With increasing demands for performance by DSP applications, embedded processors must increase available instruction-level parallelism (ILP) within significant constraints on power consumption and chip cost. Unfortunately, increasing the amount of ILP on a processor while maintaining a single register bank increases the cost of the chip and potentially decreases overall performance [6]. The number of read/write ports required to support high levels of ILP hampers cycle time.

To improve ILP and keep the port requirements of the register banks low, some modern embedded processors, *e.g.* the Texas Instruments TMS320C6x, employ *clustered* VLIW architectures. 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 which can access data directly from the local register bank. These functional unit/register bank groups are called *clusters*. If a functional unit needs a value stored in a remote cluster, extra copy operations or delays are needed to retrieve the value. This leads to intercluster communication overhead. Clustered VLIW architectures depend on the efficient use of partitioned register banks. Thus, the compiler must both expose more parallelism for maximal functional-unit utilization, and schedule instructions among clusters such that intercluster communication overhead is minimized.

Much recent work in compilation for clustered VLIW architectures has concentrated on methods to partition virtual registers amongst the target architecture's clusters effectively for software pipelined loops [2, 11, 15, 24]. Since DSP applications spend a large fraction of time in loops, high-level loop transformations offer an excellent opportunity to enhance these partitioning schemes and greatly improve performance. Previous methods for applying loop transformations for clustered VLIW architectures have presented ad-hoc techniques to apply loop unrolling and/or unroll-and-jam to loops to improve parallelism on clustered architectures [12, 19, 20]. In this paper, we present a new metric for predicting the intercluster communication costs of a loop and an integer-optimization problem to determine unroll amounts for loop unrolling and unroll-and-jam based upon the new metric. Our method automatically tailors unrolling and unroll-and-jam for a specific loop on a specific architecture based upon the predicted initiation interval of a software pipelined loop in the presence of intercluster communication.

2. Related Work

Considerable recent research has addressed the problem of generating code for clustered VLIW architectures. In this section, we give an overview of methods related to software pipelining and loop transformations.

Nystrom and Eichenberger [15] present an algorithm that first performs partitioning with heuristics that consider modulo scheduling for clustered VLIW architectures. Specifically, they try to 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. [11], describe our experiments with register partitioning in the context of software pipelining. Our basic approach abstracts away machine-dependent details from partitioning with edge and node weights, a feature extremely important in the context of a retargetable compiler. Experiments have shown that we can expect a 10–25% degradation due to partitioning for software pipelined loops over an unrealizable 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. They unroll inner loops by the number of clusters on the target architecture to improve register partitioning.

Codina, et al. [10], propose a framework that integrates cluster assignment with software pipelining and register allocation on clustered processors. They have developed a heuritic approach to evaluate the quality of scheduled code using intercluster communication, register pressure and memory accesses.

Aleta, et al. [2], describe a partitioning method that is based on widely accepted multi-level graph-partitioning methods. The authors have modified multi-level graph partitioning to emphasize final schedule length rather than the number of inter-partition arcs remaining in the partition graph.

Zalamea, et al. [24], present a partitioning method that considers register allocation, register spilling and intercluster communication when making a choice of cluster for an operation. Their use of “limited” backtracking provides much of the power of their method.

Huang, et al. [12], present a loop transformation scheme to enhance ILP for software pipelined loops for partitioned register bank architectures. Their method uses unroll-and-jam to increase parallelism within a single cluster, then unrolls the loop level with the estimated lowest communication overhead by the number of clusters on the target architecture to improve parallelism across clusters. In this paper, we present a new metric and integer-optimization problem that take into account the effect of unrolling and unroll-and-jam on software pipelining. Unlike Huang’s method, which unrolls loops by a fixed factor to generate intercluster parallelism, our method tailors unroll-and-jam for a specific loop based upon the target architecture’s configuration and the data dependences in the loop.

3. Background

3.1. Software Pipelining

Since the integer-optimization method used in this work is based upon software pipelining, we introduce several concepts related to modulo scheduling [14, 18]. Modulo scheduling selects a schedule for one iteration of a 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 cycles required between initiating execution of successive loop iterations, called the *minimum initiation interval* or *MinII*. Computing *MinII* involves two factors: the resource initiation interval (*ResII*) and the recurrence initiation interval (*RecII*). *ResII* is the maximum number of instructions in a loop requiring a specific functional-unit resource. *RecII* is the length of the longest recurrence in the data dependence graph (DDG) of a loop. The maximum of *RecII* and *ResII* imposes a lower bound on *MinII*. Once *MinII* is determined, instruction scheduling attempts to find the shortest legal schedule. The smallest schedule length to produce a legal schedule of the DDG becomes the actual initiation interval (*II*). 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.

Because our optimization method uses unroll-and-jam and unrolling, using the *MinII* of a loop to measure performance improvements is inadequate. Unrolling will increase the *MinII* rather than lower it even though loop performance may be improved. Instead of using *MinII*, we measure performance improvement in terms of unit *MinII*, or *uMinII* to get an actual measure of speedup. The *uMinII* of a loop is the *MinII* divided by the product of the unroll factors of the loops [7].

3.2. Unroll-and-jam and Loop Unrolling

Unroll-and-jam [3] is a transformation that can improve inner-loop parallelism and enhance software pipelining. The transformation unrolls an outer loop and then fuses the copies of the inner loops back together. Consider the following loop.

```
for (i = 0; i < 2*n; i++)
  for (j = 0; j < n; j++)
     y[i] += x[j]*m[i][j];
```

After unroll-and-jam of the `i`-loop by a factor of 2 the loop becomes

```
for (i = 0; i < 2*n; i+=2)
  for (j = 0; j < n; j++) {
     y[i] += x[j]*m[i][j];
     y[i+1] += x[j]*m[i+1][j];
  }
```

The original loop contains two memory operations (the store and load of `y[i]` are removed by scalar replacement [9]) and one multiply-accumulate operation. On a machine with one memory unit, one computational unit and unit-cycle operations, the *uMinII* of the loop would be 2. The unroll-and-jammed loop contains three memory operations (`y[i]`, `y[i+1]` and one reference to `x[j]` are scalar replaced) and two multiply-accumulates, giving a *uMinII* of 1.5. Thus, unroll-and-jam introduces more computation into an innermost loop body without a proportional increase in memory references, giving better performance.

For clustered architectures, unroll-and-jam (or unrolling) can also be performed to spread the parallelism in an innermost loop across clusters [12, 19, 20]. This transformation is analogous to a parallel loop where different iterations of the loop run on different processors. On a clustered VLIW architecture, each unrolled iteration can be executed on a separate cluster.

Consider the following loop that is unrolled by a factor of 2.

```
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 clusters and value cloning is applied [13].

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, there are two loop-carried dependences between the statements. If the code for iteration `j` is executed on one cluster and the code for iteration `j+1` on another, the loop schedule requires 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 < n; j++) {
  a[j] = b[j] + 1;
  c[j] = a[j-1] + 1;
}
```

This loop can be aligned as follows.

```
c[1] = a[0] + 1;
for (j = 1; j < n-1; j++) {
  a[j] = b[j] + 1;
  c[j+1] = a[j] + 1;
}
a[n-1] = b[n-1] + 1;
```

Now, we can unroll the `j`-loop and schedule iteration `j` on one cluster and iteration `j+1` on another cluster without dependences between clusters.

Unfortunately, alignment is not always possible. 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 [12], alignment conflicts can be used to determine the intercluster communication in a loop. The method presented in the next section predicts the number of edges that cause alignment conflicts before unroll-and-jam and unrolling are applied. This prediction serves as an estimate of the impact of communication on the final loop performance.

4. Method

To generate a good software pipeline for unroll-and-jammed loops on clustered VLIW machines a compiler should be able to, before unroll-and-jam is performed, not only determine which dependences introduce intercluster communication, but also predict whether the communication increases the schedule length of the software pipelined code. Our method uses *uMinII*, including the effects of intercluster data communication, as a metric to guide unroll-and-jam for clustered VLIW machines. We perform unroll-and-jam on the loop levels that will create the most intracluster and/or intercluster parallelism and determine the unroll-and-jam amounts that will give excellent loop performance.

The rest of this section is organized as follows. Section 4.1 gives our heuristic approach for picking loops to unroll. Section 4.2 describes a method to compute *uMinII* for unroll-and-jammed loops. Finally, Section 4.3 gives our method for computing unroll-and-jam amounts to achieve a lower *uMinII*.

4.1. Determining Loops to Unroll

In our method unroll-and-jam is applied to achieve both intracluster and intercluster parallelism. The first step of our approach is to determine the loop levels for unrolling or unroll-and-jam. For improving ILP in a single cluster, we seek the loop level, l_a, that will have the most data reuse after unroll-and-jam is applied. In other words, our algorithm chooses the loop level that carries the most dependences that can become amenable to scalar replacement after unroll-and-jam is applied. To obtain intercluster parallelism, we unroll the loop level, l_p, that contains the fewest dependences that may result in intercluster data communication after unrolling.

To aid in the discussion that follows we define U_a as the unroll factor for l_a and U_p as the unroll factor for l_p. Hence, the unroll factor of the entire loop nest is $U_a \times U_p$. Note that l_a and l_p can be the same loop.

4.2. Computing *uMinII*

To compute the *uMinII* of a loop, we must determine both the unit *RecII* (*uRecII*) and the unit *ResII* (*uResII*). Callahan, *et al.*, have shown that unrolling and unroll-and-jam do not increase the *uRecII* of a loop [5]. Thus, we need only compute the *uRecII* once. If the innermost loop is unrolled, then the *uRecII* remains unchanged. If an outer loop is unrolled, then the *uRecII* decreases by a factor of the unroll amount.

On an architecture with hardware loop support, FU_f fixed-point units, FU_m memory/address units and support for FU_c intercluster copies per cycle, the *uResII* is

$$\frac{\max\{\lceil \frac{F}{FU_f} \rceil, \lceil \frac{M}{FU_m} \rceil, \lceil \frac{C}{FU_c} \rceil\}}{U_a \times U_p}.$$

Here F is the number of fixed-point operations in the loop body, M is the number of memory references and C is the number of intercluster data transfers. F is defined as

$$f \times U_a \times U_p,$$

where f is the original number of fixed-point operations in the loop. The computation of M is outlined in detail by Carr and Kennedy [8].

When computing C it is assumed that the unroll-and-jammed loop is partitioned in such a way that U_a copies of loop body derived from unroll-and-jamming loop l_a are placed in a single cluster. Then each of the U_p copies of this statement group is executed in distinct clusters. There are two reasons for this assumption. First, by distributing the copies of the same statement group in separate clusters, a balanced work load across clusters is likely obtained. Second, this partitioning scheme keeps many operations involving data reuse within a single cluster, which limits intercluster communication.

4.2.1. Unrolling a Single Loop

When computing the intercluster copies caused by unrolling a single loop l ($l = l_a = l_p$), we consider the dependence graph $G_l = (V_l, E_l)$ consisting of all dependences where the distance vector associated with the dependence, $d(e)$, is of the form $\langle 0, \ldots, 0, d_l, 0, \ldots, d_n \rangle$ such that the l^{th} entry of $d(e)$, $d_l(e)$, is not 0. E_l is partitioned into two groups: E_l^C and E_l^I. E_l^C is the set of unalignable dependences carried by l whose source and sink are variant with respect to l. E_l^I is the set of dependences whose references are invariant with respect to l. The communication cost due to edges in E_l^C is denoted C_l^C and the communication cost due to edges in E_l^I is denoted C_l^I. For simplicity of presentation, we assume that each array reference has at most one incoming dependence edge. See [8] for details on handling multiple incoming edges.

Computing C_l^C: For each edge $e = (v_0, w_0) \in E_l$ there are $U_p \times U_a$ edges $e_0, e_1, \ldots, e_{U_p \times U_a - 1} \in E_l'$ after unroll-and-jam. For each $e_m = (v_m, w_n) \in E_l'$, $n = (m + d_l(e)) \bmod (U_p \times U_a)$ [5]. We examine the first U_a edges created by unrolling to determine the communication cost per cluster for l, denoted $uC_l(e)$ for each edge $e \in E_l'$. Since the sources of the first U_a edges, $v_0, v_1, \ldots, v_{U_a - 1}$, will be located in the first cluster, communication exists if and only if any sink is not in the first cluster, i.e., if any $n \geq U_a$. This implies that $uC_l(e)$

is the number of edges where $n = (m + d_l(e)) \bmod (U_p \times U_a) \geq U_a$, $m = 0, 1, \ldots, U_a - 1$. Hence,

$$C_l^C = \sum_{e \in E_l^C} uC_l(e) \times U_p.$$

To derive $uC_l(e)$, we break down the computation into three common cases. These cases arise from the fact that dependences in loops are usually very simple. For each case we give our conclusion and an example. For a detailed proof of each case, see [17].

Case 1: If $d_l(e) \bmod (U_a \times U_p) = 0$, then $uC_l(e) = 0$. Consider the following loop where $d_l(e) = 4$:

```
for (i = 1; i < 4*N, i++)
    a[i] = a[i-4];
```

To generate code for a 4-cluster machine, we may unroll the loop by a factor of 4, giving

```
for (i = 1; i < 4*N, i += 4) {
  a[i]   = a[i-4];
  a[i+1] = a[i-3];
  a[i+2] = a[i-2];
  a[i+3] = a[i-1];
}
```

Each statement may be executed on a different cluster without incurring a communication cost due to the original dependence.

Case 2: If $d_l(e) \bmod (U_a \times U_p) \neq 0$ and $U_a = 1$, then $uC_l(e) = 1$. Consider the following loop where $d_l(e) = 1$:

```
for (i = 1; i < 3*N; i++)
  a[i] = a[i-1];
```

After unroll-and-jam by a factor of 3, where $U_p = 3, U_a = 1$, we have

```
for (i = 1; i < 3*N; i+=3) {
   a[i]   = a[i-1];
   a[i+1] = a[i];
   a[i+2] = a[i+1];
}
```

If each of these statements is assigned to a different cluster, each cluster needs one intercluster data transfer.

Case 3: If $d_l(e) \bmod (U_a \times U_p) \neq 0$, $U_a > 1$ and $U_a \geq d_l(e)$, then $uC_l(e) = d_l(e)$. Consider the following loop where $d_l(e) = 2$:

```
for (i = 1; i < 6*N; i++)
  a[i] = a[i-2];
```

After unroll-and-jam by a factor of 6 (assume $U_p = 2, U_a = 3$), we have

```
for (i = 1; i < 6*N; i += 6) {
  a[i]   = a[i-2];
  a[i+1] = a[i-1];
  a[i+2] = a[i];
  a[i+3] = a[i+1];
  a[i+4] = a[i+2];
  a[i+5] = a[i+3];
}
```

If the first three statements are executed on one cluster and the last three on another cluster, each cluster needs two intercluster copies.

From the derivation of $uC_l(e)$ we can make the following observation: when unrolling a single loop l, the communication cost per cluster caused by dependences that are variant with respect to l does not change much with respect to the unroll factor. In practice most dependences fall into the above cases.[1] To compute $uC_l(e)$ for variant references in general, see [17].

Computing C_l^I: For each dependence whose source and sink are invariant with respect to l, there are $U_a \times U_p$ references in the loop body after unroll-and-jam/unrolling. $U_a \times U_p - 1$ memory references are eliminated by scalar replacement. When partitioning the loop body into U_p separate clusters, a memory operation is executed in one cluster and the other $U_p - 1$ clusters require a copy operation. This function remains constant as U_a changes, giving

$$C_l^I = \sum_{e \in E_l^I} (U_p - 1).$$

4.2.2. Unrolling Multiple Loops

When $l_a \neq l_p$, we must consider the effects of multiple loops. Unroll-and-jam of l_a will potentially increase the number of loop-carried dependences causing communication when l_p is unrolled or unroll-and-jammed to spread computation across clusters. For references invariant with respect to l_p, the communication costs can be computed as

[1]In our experiments, all dependences are handled by these cases.

described in the previous section. However, the same is not true for variant references.

To compute the communication cost for variant references, we consider the case when l_p is the innermost loop and the case when it is not. When l_p is the innermost loop, the two types of dependences that can cause communication are

1. innermost-loop-carried dependences, and
2. outer-loop-carried dependences that become carried by the innermost loop after unroll-and-jam.

In the first case, the method presented in the previous section accurately computes the communication cost. For the second case, only dependences with a distance vector of the form $\langle 0,\ldots,d_i,\ldots,0,\ldots,d_n\rangle$, where $d_i \neq 0$ and i is l_a, are considered. We call this set of dependence E_i^U. Consider any $e \in E_i^U$. After performing unroll-and-jam on i by a factor of U_i, $(U_i - d_i(e))^+$ dependences have a zero entry in i^{th} component of the distance vector and are carried by the innermost loop or are loop independent [8].

For each edge $e \in E_i^U$ made innermost by unroll-and-jam, we need to compute the communication cost caused by unrolling l_p (the innermost loop). To derive this communication cost, we use the convention that $C_n(e)$, where n is l_p, is the communication cost caused by the dependences with distance $d_n(e)$ after unrolling the innermost loop $U_n - 1$ times. Since unrolling the innermost loop can not increase data reuse, the only reason for unrolling the innermost loop is to create intercluster parallelism.

From the previous section, we have

$$C_n(e) = uC_n(e) \times U_n.$$

Thus, if the set of unalignable innermost-loop-carried dependences, is denoted E_n^C, then the communication cost when l_p is the innermost loop is

$$\sum_{e\in E_n^C} U_iC_n(e) + \sum_{e\in E_i^U} (U_i - d_i(e))^+C_n(e).$$

When l_p is not the innermost loop, we also only consider the dependences that can be made innermost. This set of dependences is denoted E_{ij}^U and the distance vector of each of these edges is of the form $\langle 0,\ldots,d_i,\ldots,0,d_j,0,\ldots,d_n\rangle$, where i is l_a and j is l_p, or vice versa. In this case both i and j must be unroll-and-jammed enough to make $d_i = 0$ and $d_j = 0$, giving a communication cost of

$$\sum_{e\in E_{ij}^U} (U_i - d_i(e))^+(U_j - d_j(e))^+.$$

$C_n(e)$ is not needed here since the innermost loop is not unrolled.

Example. To demonstrate intercluster copy prediction for unrolling multiple loops, consider the following loop:

```
for (j = 1; j < N; j++)
  for (i = 1; i < N; i++)
    {
      a[i][j]=a[i-1][j]
      b[i][j]=a[i][j-1]+a[i-1][j-1]
    }
```

Let e_1 denote the edge from `a[i][j]` to `a[i-1][j]`, e_2 denote the edge from `a[i][j]` to `a[i][j-1]`, and e_3 denote the edge from `a[i][j]` to `a[i-1][j-1]`. Note that $d(e_1) = \langle 0,1\rangle$, $d(e_2) = \langle 1,0\rangle$ and $d(e_3) = \langle 1,1\rangle$. If $U_i = 2$, $U_j = 2$ for a 2-cluster machine, we have

$$\begin{array}{lcl} E_n^C &=& \{e_1\} \\ E_j^U &=& \{e_2,e_3\} \\ C_n(e_1) &=& 1\times 2 = 2 \\ C_n(e_2) &=& 0 \\ C_n(e_3) &=& 1\times 2 = 2 \end{array}$$

Therefore,

$$\begin{array}{rcl} C = && U_j \times C_n(e_1) + (U_j - d_j(e_2))^+ \times C_n(e_2) + \\ && (U_j - d_j(e_3))^+ \times C_n(e_3) \\ &=& 2\times 2 + 1\times 0 + 1\times 2 \\ &=& 6 \end{array}$$

4.2.3. Register Pressure

Unroll-and-jam/unrolling can increase the number of registers needed in the innermost loop body. Carr and Kennedy have presented a method to compute the number of registers required by scalar replacement for an unroll-and-jammed loop before unroll-and-jam is applied [8]. Since they do not consider unrolling an innermost loop, we extend their work to include the effects of inner-loop unrolling.

In computing register pressure R, the method proposed in [8] partitions the reference set of a loop into the following three sets that exhibit different memory behavior when unroll-and-jam is applied.

- $V^\emptyset$ is the set of references without an incoming dependence,
- V_r^C is the set of memory reads that have a loop-carried or loop-independent incoming dependence, but are not invariant with respect to any loop, and
- V_r^I is the set of memory reads that are invariant with respect to a loop.

The number of registers required by each reference set is represented by $R^\emptyset$, R_r^C, and R_r^I respectively, giving $R = R^\emptyset + R_r^C + R_r^I$. In this paper, we only discuss the changes needed for R_r^C. $R^\emptyset$ will be 0 since the references in $V^\emptyset$ are not amenable to scalar replacement. The computation of R_r^I is described elsewhere [17].

For each reference $v \in V_r^C$ such that the edge associated with v is carried by the innermost loop, unrolling the innermost loop by U_n will create U_n dependence edges amenable to scalar replacement. From Callahan, et al. [5], $(U_n - d_n(e_v))^+$ of these edges have distances of $\lfloor d_n(e_v)/U_n \rfloor$, and $\min(U_n, d_n(e_v))$ of them have distances of $\lfloor d_n(e_v)/U_n \rfloor + 1$. Therefore, for each dependence in the innermost loop, the number of registers required by scalar replacement after unrolling the innermost loop is

$$\begin{aligned} R_n &= (U_n - d_n(e_v))^+ \times \left(\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor + 1 \right) + \\ &\quad \min(U_n, d_n(e_v)) \times \left(\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor + 2 \right) \end{aligned}$$

Since Carr and Kennedy show that

$$R_r^C = \sum_{v \in V_r^C} (\prod_{1 \le i < n} (U_i - d_i(e_v))^+) \times (d_n(e_v) + 1)$$

when the innermost loop is not unrolled, we have

$$R_r^C = \sum_{v \in V_r^C} \prod_{1 \le i < n} (U_i - d_i(e_v))^+ R_n$$

when the innermost loop is unrolled.

In practice dependence distances are almost always 0 or 1, allowing us to simplify the computation of register pressure. If $d_n(e_v) < U_n$, we have

$$\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor = 0,$$

and

$$(U_n - d_n(e_v))^+ = U_n - d_n(e_v).$$

Thus, R_n becomes

$$(U_n - d_n(e_v)) \times 1 + d_n(e_v) \times 2 = U_n + d_n(e_v),$$

giving

$$R_r^C = \sum_{v \in V_r^C} \prod_{1 \le i < n} (U_i - d_i(e_v))^+ (U_n + d_n(e_v)).$$

4.3. Computing Unroll Amounts

To find the best unroll amounts for a particular loop on a particular target architecture, we solve the following integer-optimization problem.

objective function: $\min \ uMinII$

constraints: $R_r^C + R_r^I \le R_m$
$U_a, U_p \ge 1$

where R_m is the number of physical registers in the target machine.[2]

To solve this problem, we can bound the search space and do an exhaustive search to find the best unroll amounts. Carr and Kennedy have shown that bounding the space by R_m in each dimension is sufficient. If an exhaustive search is too expensive, a heuristic search, such as described in [17], may be used.

5. Experimental Results

We have implemented the algorithm described in Section 4 in Memoria, a source-to-source Fortran transformer based upon the DSystem [1]. We evaluate the effectiveness of our transformations on a simulated architecture, called the URM [16], 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, including two full applications and 48 DSP kernels, contains typical DSP applications: FIR filter, correlation, Reed-Solomon decoding, lattice filter, LMS filter, etc. Out of these 119 loops, unroll-and-jam or loop unrolling is applicable to the 71 loops that do not contain branches or function calls in the innermost-loop body. Our results are reported over these 71 loops.

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.

Section 5.1 reports the speedups achieved by Memoria on four clustered VLIW architectures modeled with the URM. Section 5.2 presents the speedups achieved by Memoria on the Texas Instruments TMS320C64x. Finally, Section 5.3 analyzes the accuracy of our communication cost model.

[2]Note that registers will need to be reserved to allow for the increase in register pressure caused by software pipelining.

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 [11], software pipelining [18] and Chaitin/Briggs style register assignment [4]. 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 register bank has 48 integer registers. All functional units are homogeneous and have instruction timings as shown in Table 1. 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 1. Operation Cycle Counts

Table 2 shows the speedup obtained by our method on the benchmark suite. The speedup is measured using the actual unit *II* (*uII*) of the software pipelined loop before and after loop transformations. The "# Improved" row shows the number of loops that gain improvement via unroll-and-jam and/or unrolling.

We have observed a 1.39 – 1.68 harmonic mean speedup in *uII* achieved by our algorithm over the original loops. The median speedup ranges from 1.52 – 1.78. The speedups for individual loops range from 0.7 to 14.4, with more than 50 loops, or 70% of loops to which unrolling is applicable, seeing improvement by our method.

As can be seen in the range of speedups, some loops have very large speedups and some loops actually show a degradation. The loops with the largest speedups are doubly nested and compute a reduction. Unroll-and-jam improves the performance of this type of loop particularly well. The two loops that degrade in performance use index arrays, making accurate dependence analysis nearly impossible. Because the dependence analysis is inexact, our prediction scheme does not correctly predict the communication needed by the loop.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Harmonic Mean	1.39	1.68	1.40	1.43
Median	1.52	1.78	1.60	1.60
# Improved	50	69	50	51

Table 2. URM Speedups: Transformed vs. Original

Although the architecture with 8 functional units arranged in 4 clusters achieves the best overall speedup, the performance of each individual loop is lower than on the other architectures. This is because this architecture has enough functional units to capture parallelism exploited by unrolling, but requires a significant number of cycles for intercluster data transfers since only one such transfer can be initiated in a single cycle.

Using a fixed unroll amount, as is done in previous work, may cause performance degradation when communication costs are dominant. Table 3 shows the performance difference between our method and Huang's method [12] when the methods use different unroll amounts. The row marked "# of Loops" shows how many of the loops have different unroll amounts under both methods on each clustered architectures. The row labeled "Harmonic Mean" gives the harmonic mean speedup in *uII* obtained by our method. The row labeled "Harmonic (Fixed)" shows the harmonic mean speedup in *uII* obtained by Huang's method. Our algorithm gives a better harmonic mean speedup in *uII* than Huang's method on each architecture. The degradations seen on the 4-cluster 8-wide machine are due to the dependences with indeterminate dependence distances caused by index arrays.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Harmonic Mean	1.00	0.91	1.00	1.07
Harmonic (Fixed)	0.88	0.84	0.88	0.95
# of Loops	9	4	9	21

Table 3. URM Speedups: Our Algorithm vs. Fixed Unroll Amounts

5.2. TMS320C64x Results

We have also evaluated our algorithm 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 data transfers are 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 convert the unrolled code generated by the Memoria into C by hand. Then, both the original and transformed versions of the code are compiled and run on the C64x. When using the C64x compiler, we choose the highest optimization level (-o3) [23]. Since the unrolling algorithm in the C64x compiler is a subset of Huang's algorithm, we turned off unrolling in the TI compiler to examine the effect of our unrolling algorithm on our set of loops. We compare our algorithm to Huang's later in this section.

The results obtained on the C64x are summarized in Table 4. The C64x compiler fails to find a profitable schedule for two loops, and thus generates code for these loops without applying software pipelining. Additionally two loops that contain a division operation cannot be software pipelined since the C64x compiler treats division operations as function calls that disable software pipelining. Therefore, Table 4 gives the results for 67 loops in our benchmark suite.

	Speedup
Harmonic Mean	1.70
Median	2.00
# Improved	55

Table 4. TI C64x Speedups: Transformed vs. Original

The harmonic mean speedup in *uII* across the entire benchmark suite is 1.7. The individual speedups for the loops ranged from 1.0 to 4. The harmonic mean speedup improvement is better than that seen on the URM. The C64x supports more intercluster copies which, in turn, reduces the overhead of the copy operations.

On the 9 loops where our algorithm produces a different result than Huang's algorithm, Huang's algorithm gets better performance than ours. This is because the C64x supports SIMD operations and our performance model does not considers these effects. Our model will choose not to unroll these loops because of communication. However, unrolling allows the C64x compiler to detect the SIMD operations and improve intracluster parallelism. Since Huang's algorithm always unrolls irrespective of communcation cost, it blindly exposes the SIMD operations. The solution to this problem is to model SIMD operations in our performance model.

5.3. Accuracy of Communication Cost Prediction

To evaluate the accuracy of our algorithm in predicting intercluster communication, we compare the number of intercluster data transfers due to cross-cluster dependences predicted by our communication cost model against the actual number of cross-cluster dependences found in the transformed loops. For each loop after unroll-and-jam/unrolling and scalar replacement are applied, Memoria counts the intercluster true dependences whose sinks are not killed by a definition in the path from the source to the sink. The input dependences whose sources and sinks reside in distinct clusters are also recorded.

The result shows our communication cost model makes an exact prediction for most of the loops in our benchmark suite. For the 2-cluster machines, only 5 out of 71 loops show a misprediction. For 4-cluster machines, 7 loops have a misprediction. In each case, the misprediction is by one or two dependences and occurs in the loops that contain index arrays.

Many heuristic algorithms may not be able to derive a partition that limits copies to the extent that our model predicts. However, this prediction serves as a lower bound for these heuristics to attempt to achieve.

6. Conclusions

This paper presents a new method for predicting the amount of intercluster data transfers caused by data dependences in loops run on clustered VLIW architectures. This method is part of an integer-optimization problem used to predict the performance of a software-pipelined loop and to guide unroll-and-jam or loop unrolling for clustered VLIW architectures.

We have implemented our algorithm and run an experiment on DSP benchmarks for four different simulated, clustered VLIW architectures, and the TI TMS320C64x. Our results show that the prediction of intercluster data dependences that cause communication is accurate. In addition, out of the 71 DSP benchmarks used, 70%-97% of the loops can be improved via unroll-and-jam/unrolling by a harmonic mean speedup of 1.4 – 1.7 for the five different architectures.

Clustered VLIW embedded processors are increasing in use, making it important for compilers to achieve high ILP with low communication overhead. We believe that the work presented in this paper is an important step in generating high performance code on clustered architectures. We believe that high-level loop transformations should be an integral part of compilation for clustered VLIW machines.

Acknowledgments

This research was partially supported by NSF grants CCR-9870871 and CCR-0209036. StarCore provided support in the initial stages of this work. Additionally, the authors would like to thank Texas Instruments for providing the benchmark suite used in this experiment.

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. Aletá, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Austin, Texas, Dec. 2001.

[3] F. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[4] 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.

[5] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[6] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliw's: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 292–300, Portland, OR, December 1-4 1992.

[7] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1996.

[8] 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.

[9] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, Jan. 1994.

[10] J. Codina, J. Sanchez, and A. Gonzalez. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

[11] 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.

[12] 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.

[13] D. Kuras, S. Carr, and P. Sweany. Value cloning for architectures with partitioned register banks. In *Proceedings of 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 31st International Symposium on Microarchitecture (MICRO-31)*, pages 103–114, Dallas, TX, December 1998.

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

[17] Y. Qian. *Loop Transformations for Clustered VLIW Architectures*. PhD thesis, Department of Computer Science, Michigan Technological University, Houghton, MI, August 2002.

[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.

[24] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 160–169, Austin, Texas, Dec. 2001.