Loop Transformations for Architectures with Partitioned Register Banks

Xianglong Huang
Department of Computer Science
University of Massachusetts – Amherst
Amherst MA 01003-4610
xlhuang@cs.umass.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, processors can often exploit considerable instruction-level parallelism (ILP), and thus significantly improve performance, at the cost of substantially increasing register requirements. These increasing register requirements, however, make it difficult to build a high-performance embedded processor with a single, multi-ported register file while maintaining clock speed and limiting power consumption.

Some digital signal processors, such as the TI C6x, reduce the number of ports required for a register bank by partitioning the register bank into multiple banks. Disjoint subsets of functional units are directly connected to one of the partitioned register banks. Each register bank and its associate functional units is called a *cluster*. Clustering reduces the number of ports needed on a per-bank basis, allowing an increased clock rate. However, execution speed can be hampered because of the potential need to copy "non-local" operands among register banks in order to make them available to the functional unit performing an operation. The task of the compiler is to both maximize parallelism and minimize the number of remote register accesses needed.

Previous work has concentrated on methods to partition virtual registers amongst the target architecture's clusters. In this paper, we show how high-level loop transformations can enhance the partitioning obtained by low-level schemes. In our experiments, loop transformations improved software pipelining by 27% on a machine with 2 clusters, each having 1 floating-point and 1 integer register bank and 4 functional units. We also observed a 20% improvement on a similar machine with 4 clusters of 2 functional units. In fact, by performing the described loop transformations we were able to show improvements of greater than 10% over schedules (for un-transformed loops) generated with the unrealistic assumption of a single multi-ported register bank.

LCTES 2001, Snowbird, Utah, USA

Categories and Subject Descriptors

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

1. INTRODUCTION

Embedded systems in general, and digital signal processing (DSP) applications in particular require maximum performance within significant constraints for power consumption and chip cost. With aggressive instruction scheduling techniques, like software pipelining [20, 23, 3], DSP processors can exploit considerable instruction-level parallelism (ILP), and thus significantly improve performance. Unfortunately, such an increase in ILP leads to a large demand for machine registers. To support the amount of parallelism attainable with software pipelining on DSP architectures, the number of ports required for a single register bank would severely hamper access time [11, 15], power, and chip cost. One way to ameliorate this problem is to partition the machine register banks such that a subset of the machine's functional units is directly connected to each register bank. These functional unit/register bank subsets are called *clusters*. Partitioned register banks are one mechanism for providing high degrees of ILP while maintaining a high clock rate, an acceptable power budget, and a reasonable cost. Texas Instruments, for example, already produces several DSP chips that have partitioned register banks to support high ILP [27].

In one sense, partitioned register banks seem to be a natural architectural feature to support software pipelining, since, using multiple register banks, we can build ILP architectures that provide a multitude of registers without substantially increasing cycle time. Unfortunately, partitioned register banks also inhibit ILP as some mechanism is required to allow functional units access to "non-local" values (i.e. values contained in a different cluster). One approach to provide non-local register values is to add extra instructions to move the data to the register bank of the target functional unit. Another approach is to provide a communication network to allow access to non-local values. In either case, a compiler must deal not only with achieving maximal parallelism via

aggressive scheduling, but also with data placement among a set of register banks.

A compiler for an ILP architecture with partitioned register banks must decide for each operation, not only where the operation fits in a software-pipelined instruction schedule, but also in which cluster(s) the operands of that operation will reside. This, in turn, will determine which functional unit(s) can perform the operation. Obtaining an efficient assignment of operations to functional units is not an easy task as two opposing goals must be balanced. One, achieving near-peak performance, requires spreading the computation over the functional units equally, thereby maximizing their utilization. The second goal, minimizing costs resulting from copy operations, requires placement of all operands for an operation in register bank(s) associated with a single functional unit.

This paper focuses on high-level loop transformations, such as loop alignment [6, 7] and unroll-and-jam [5, 10], that can, as a prelude to current software pipelining techniques, enhance performance on architectures with partitioned register banks. We use a two-part approach. First, we transform the loop for better performance on a single cluster using techniques to improve ILP. Second, we transform parallel loops as in shared-memory compilation [6, 7] so that we can spread the computation across clusters with little communication.

We begin this paper with a discussion of previous work in the area of generating code for partitioned-register bank architectures. Next, we present a brief introduction to software pipelining and an overview of our low-level partitioning scheme. Then, we present our high-level loop optimization techniques and an experiment showing the speedups obtained with them. Finally, we present our conclusions.

2. PREVIOUS WORK

Considerable recent research has addressed the problems of architectures with partitioned register banks. Most approaches use a graph to represent the intermediate code generated from the program. Then from the relationship represented by the graph, these methods attempt to partition the operations effectively. We give an overview of some of these partitioning techniques in this section.

Ellis [14] 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 [22]. 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 showed, for their best heuristic, an average degradation of roughly 19% on an 8-wide machine grouped as two clusters of 4 functional units and 2 buses.

Nystrom and Eichenberger [21] present an algorithm that first performs partitioning with heuristics that consider modulo scheduling. 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. [16, 17], describe our experiments with register partitioning in the context of whole programs and software pipelining. Our basic approach, described in [16] and summarized in Section 3.2, 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 [22], we found a degradation in execution performance of 10% on average when compared to an unrealizable monolithic-register-bank architecture with the same level of ILP. Experiments with software pipelining showed 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 [24, 25] have independently 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 uses unroll-and-jam [12] and loop alignment [7] to improve modulo scheduled loops [18].

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 [20, 23]. 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 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 [23] provides a detailed discussion of an implementation of modulo scheduling, and in fact our implementation is based upon Rau's method.

3.2 Register Assignment

In partitioned register bank architectures a distinct set of registers is associated with each functional unit (or cluster of functional units). An example of such an architecture is the Texas Instruments C6x [27]. 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:

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 [19],
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 to banks, perform "standard" Chaitin/Briggs graph coloring register assignment for each register bank [8],

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 [17] 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

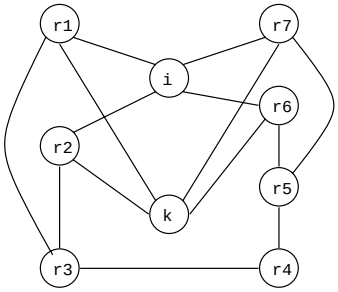

Figure 2: Register Component Graph

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.

[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

4. LOOP OPTIMIZATION

In this section, we show how four loop transformations – scalar replacement [9, 13], unroll-and-jam [10, 12], loop unrolling and loop alignment [6, 7] – can be used to enhance software pipelining on partitioned register bank 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 with an overview of our code generation strategy and then give an overview on how to improve intracluster ILP with loop transformations. Finally, we show how to enhance data-independent parallelism across multiple clusters.

4.1 Code Generation Strategy

The overall code generation strategy that we will use for a particular loop on a partitioned register bank architecture is as follows:

1. apply unroll-and-jam to improve intracluster ILP
2. determine the loop alignment conflicts (sources of register bank communication) for each loop in a nest
3. unroll the loop with the fewest alignment conflicts by a factor of the number of clusters to improve intercluster ILP,
4. perform scalar replacement
5. use the method from Section 3.2 to generate the final code.

The techniques used to improve intracluster ILP are from previous work [13, 12]. The method to improve intercluster ILP represents new work.

Our goal with these loop transformations is to improve performance. While code size is an issue in certain embedded systems applications, we do not consider it here. This issue could be handled within our framework by restricting loop unrolling.

4.2 Intracluster Parallelism

Previous work has shown that when the ratio of memory operations to floating-point operations that a machine can perform at peak speed, called *machine balance*, is close to the ratio of memory operations issued to floating-point operations issued in a loop, called *loop balance*, loop performance is very good [12]. In the rest of this section, we show by example how scalar replacement and unroll-and-jam can lower loop balance to be closer to machine balance in loops dominated by memory operations. In our code generation scheme for partitioned register banks, this transformed loop will then be transformed again to improve parallelism across clusters.

4.2.1 Scalar Replacement

Scalar replacement is a transformation that replaces references to array variables with references to sequences of temporaries in order to effect register allocation of array values. For example, in the loop

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

the references to `c[i][j]` can be moved out of the innermost loop and replaced with a scalar temporary as follows

```
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 result is a loop with a balance lowered from 3 to 2 (assuming a multiply-accumulate instruction). On most architectures the second loop would run faster.

4.2.2 Unroll-and-Jam

Before scalar replacement is applied, we can enhance its effectiveness and improve ILP with unroll-and-jam. Unroll-and-jam is a transformation that can be used to improve the performance of memory-bound loops by lowering loop balance. Additional computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, if we unroll-and-jam the `j`-loop in our matrix multiply example by 2, after scalar replacement 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;
  }
```

The balance of this loop is $\frac{3}{2}$, an improvement of a factor of 2 over the original loop. To guide unroll-and-jam, assuming scalar replacement will follow, we use the method described in [12].

4.3 Intercluster Parallelism

To create data-independent parallelism across clusters, we can adapt transformations designed for loop parallelism on shared memory multiprocessors. The goal in generating code for shared memory multiprocessors is to find a loop that carries no true, anti or output dependences and convert it into a parallel loop where each iteration of the loop is independent from the others. In the context of partitioned register bank architectures, we can partition the registers such that independent iterations of the parallel loop are executed on different clusters. While the startup overhead for loop parallelism on shared-memory machines can eliminate speedup, in our context, no such overhead exists. This makes these loop transformations attractive for our code generation strategy.

Consider the following parallel loop.

```
parallel for (i = 0; i < n; i++)
   a[i] = b[i] + c[i]
```

For shared-memory multiprocessors, each iteration can be executed on a separate processor in parallel. On partitioned-register-file architectures, each iteration could be scheduled on a separate cluster and run in parallel without communication (assuming value cloning is done). We accomplish the scheduling by unrolling the loop and allowing the low-level partitioning algorithm to find the data-independent parallelism amongst the original loop body and its copies.

When a loop carries a dependence, communication between register banks may be needed. To determine which dependences cause communication, we consider the effects of loop alignment on parallelism [6]. To illustrate how alignment and unrolling work together consider the following loop:

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

There is a true dependence from the first statement to the second statement carried by the i-loop. If the code for the first statement is executed on one cluster and the code for the second statement on another, there will either be communication between clusters through copying of registers, or through a store to and load from memory.

To limit communication we can keep the source and sink of the dependence on the same cluster and look for data independent parallelism elsewhere. That parallelism can be found by using loop alignment and unrolling. After alignment of the i-loop we get[2]

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

[2]Note that the pre- and post-loop to execute the first and last iterations of the i-loop have been removed.

Now, the dependence between the first and second statements is loop independent. If we unroll-and-jam the aligned i-loop we get copies of the innermost loop that are data independent (assuming value cloning of induction variables and loop invariants is done [19]).

```
for (i = 1; i < n-1; i+=2)
  for (j = 1; j < n-1; j++) {

   /* code for cluster 1 */

    a[i][j] = a[i][j+1] + 1;
    b[i][j] = a[i][j];

    /* code for cluster 2 */

    a[i+1][j] = a[i+1][j+1] + 1;
    b[i+1][j] = a[i+1][j];
  }
```

The independent sections of code can now be scheduled on different clusters with low communication cost.

Actually aligning a loop, however, is unnecessary if we are not going to generate a shared-memory parallel loop. Consider the following unrolled and unaligned loop:

```
for (i = 0; i < n; i+=2)
  for (j = 1; j < n-1; j++) {

    /* first loop body */

    a[i][j] = a[i][j+1] + 1;
    b[i][j] = a[i-1][j];

    /* second loop body */

    a[i+1][j] = a[i+1][j+1] + 1;
    b[i+1][j] = a[i][j];
  }
```

We can schedule the first statement of the first loop body and the second statement of the second loop body on the same cluster and the other two statements on a different cluster. This would accomplish the same thing as alignment as there would be no dependences between clusters. Therefore, rather than actually applying alignment, we use loop alignment to identify the best loop to unroll for partitioning.

Unfortunately, alignment is not always possible. It 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. Note that putting the source and sink of such a dependence in the same cluster will restrict intercluster parallelism. As an example, consider the following loop.

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

Alignment cannot change all of the dependences carried by the i-loop into loop-independent dependences because one of the loop-carried dependences has a distance of 2 and the other has a distance of 1. Aligning the distance 1 loop-carried dependence will leave the other loop-carried dependence as loop carried. Thus after unrolling, the original loop body and its copy are not completely independent. There will either be communication or limited parallelism. Our strategy in the face of alignment conflicts is to unroll the loop with the fewest alignment conflicts (lowest communication cost). We use a slight modification of the algorithm in [7] to compute the number of alignment conflicts in a loop.

5. EXPERIMENTAL EVALUATION

We have implemented the method described in Section 4 in Memoria, a Fortran source-to-source transformer based upon the DSystem [1]. We create a transformed version of the original code with Memoria by applying unroll-and-jam and loop unrolling to improve parallelism as described. After the transformed version of the code is created we apply scalar replacement, constant propagation, global value numbering, partial redundancy elimination, operator strength reduction and dead code elimination to both versions of the code. Finally, both the transformed and original code are run through Rocket [26] to perform partitioning, software pipelining [23] and register assignment [8].

Benchmark	# of Loops
tomcatv	5
swim	11
su2cor	49
hydro2d	61
mgrid	2
applu	52
turb3d	19
apsi	12
Total	211

Table 1: Benchmark Suite

Operation	Cycles
integer copies	1
float copies	1
loads	2
stores	4
integer mult	5
integer divide	12
other integer	1
other float	2

Table 2: Operation Cycle Counts

We ran our experiments on a set of 211 loops from the Spec95 benchmark suite as shown in Table 1. We report results for 4 different partitioned-register-bank architectures with the following configurations:

	8 FUs		16 FUs	
Clusters	*2*	*4*	*2*	*4*
% II	-2.63	-19.85	-19.58	-35.06

Table 3: Partitioned Transformed Code vs. "Ideal" Transformed Code

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 and 48 floating-point registers. All functional units are homogeneous and have instruction timings as show in Figure 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.

Table 3 shows the effect of partitioning on the transformed code in terms of the average percentage change in initiation interval in the row labeled "**% II**". The negative numbers indicate a degradation in II. We include these numbers for completeness since previous work (including our own) has typically evaluated partitioning schemes by measuring how close they could come to the ideal of a single register bank with enough read/write ports to support all of the machine's functional units. It is important to note that such a single-register-bank architecture is not realizable. The degradations reported here are consistent with our previous work[17].

Table 4 reports the average percent change in II achieved by our loop transformation algorithm when compared with the code before loop transformations on each partitioned architecture. Positive numbers indicate an improvement in II. Only a couple of loops in our test suite had enough outer loop data reuse for optimization for ILP to be beneficial. So, the improvements reported in Table 4 are almost exclusively as a result of optimization for partitioned architectures. The average percent improvements ranged from 19.7–35.8%. Our method achieved the smallest improvement on the architecture with 8 functional units having 4 clusters of size 2. This is due to contention for the single copy unit since having 4 clusters increases the number of interbank copies.

Table 4 also shows the percent change in intercluster copies after unrolling. For the architectures with 2 clusters, the number of copies in the loop decreased by 7%. For the architectures with 4 clusters, the change in the number of copies was negligible. This suggests that unrolling is indeed creating data-independent parallelism.

Table 5 shows how well our transformed loops performed in relation to the original loop's ideal schedule (the schedule on

	8 FUs		16 FUs	
Clusters	*2*	*4*	*2*	*4*
% II	27.2	19.7	29.8	35.8
% Copies	7.0	0.4	7.2	-2.0

Table 4: Transformed vs. Original Partitioned Loop

	8 FUs		16 FUs	
Clusters	*2*	*4*	*2*	*4*
% II	21.7	10.8	20.6	27.8
# Improved	150	133	147	164

Table 5: Transformed vs. Original "Ideal" Schedule

an architecture with a single cluster). The average percent improvement in II over the original code run on the ideal architecture was 10.8–27.8%. Most of the loops in our test suite, 65–78%, not only significantly improved over the original loop schedule on the partitioned architecture, but also improved over the original loop schedule on the ideal architectures with one register bank. The "**# Improved**" row shows how many of the 211 loops tested actually allowed us to get better results than that available (for un-transformed loops) with the "ideal" case of a single register bank. The loops where the II did not improve with unrolling were cases where there was a tie or a small degradation. These loops did not have enough data-independent parallelism.

Some of the loops that had a degradation contained inner-loop recurrences and were unrolled for parallelism. However, many of the loops having inner-loop recurrences showed an improvement. We could find no conclusive explanation for this behavior. We believe it is due to the heuristic nature of the compilation method.

6. CONCLUSIONS AND FUTURE WORK

To make efficient use of processors with partitioned register banks, compilers need to address several difficult issues. In this paper we have described an algorithm that uses loop unrolling, unroll-and-jam and loop alignment to improve software pipelining for partitioned-register-bank architectures. These high-level transformations allow us to obtain data-independent code sections that can be scheduled on functional units having separate register banks with much less communication. The code we generated will take better advantage of partitioned-register-bank architectures.

Our results show that by using high-level loop transformations, we can expect to see improvements of 20–36% in the actual II of a loop over the un-transformed code scheduled on the same architecture. In addition, our transformation system can expect to achieve a 11–28% improvement over the un-transformed code scheduled on the ideal architecture without partitioned register banks.

In the future, we will develop a metric for computing unroll amounts that combines the effects of intracluster and intercluster parallelism. In addition, we are looking into the effects of loop fusion on partitioned-register-file architectures. As a result of this research, increasing levels of ILP supported by partitioned register banks will be more attractive. By applying high-level loop transformations, our methods reduce the communication cost, expose more parallelism and increase the efficiency of loops. We believe that high-level transformations must be an integral part of compiling for partitioned-register-bank architectures.

Acknowledgments

This research was partially supported by NSF grant CCR-9870871.

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

[6] J. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on the Principles of Programming Languages*, Munich, West Germany, Jan. 1987.

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

[8] 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.

[9] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990.

[10] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[11] 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.

[12] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

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

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

[15] J. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architecures. In *Twenty-Ninth Annual Symposium on Micorarchitecture (MICRO-29)*, pages 324–335, Dec. 1996.

[16] 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.

[17] 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.

[18] X. Huang. High-level loop transformations for architectures with partitioned register banks. Master's thesis, Michigan Technological University, Department of Computer Science, August 2000.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] Texas Instruments. *Details on Signal Processing*, issue 47 edition, March 1997.