Register Assignment for Software Pipelining with Partitioned Register Banks [*]

Jason Hiser
Steve Carr
Philip Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
{*jdhiser,carr,sweany*}*@mtu.edu*

Steven J. Beaty
Metropolitan State College of Denver
Department of Mathematical and Computer Sciences
Campus Box 38, P. O. Box 173362
Denver, CO 80217-3362
beatys@mscd.edu

Abstract

Trends in instruction-level parallelism (ILP) are putting increased demands on a machine's register resources. Computer architects are adding increased numbers of functional units to allow more instructions to be executed in parallel, thereby increasing the number of register ports needed. This increase in ports hampers access time. Additionally, aggressive scheduling techniques, such as software pipelining, are requiring more registers to attain high levels of ILP. These two factors make it difficult to maintain a single monolithic register bank.

One possible solution to the problem is to partition the register banks and connect each functional unit to a subset of the register banks, thereby decreasing the number of register ports required. In this type of architecture, if a functional unit needs access to a register to which it is not directly connected, a delay is incurred to copy the value so that the proper functional unit has access to it. As a consequence, the compiler now must not only schedule code for maximum parallelism but also for minimal delays due to copies.

In this paper, we present a greedy framework for generating code for architectures with partitioned register banks. Our technique is global in nature and can be applied using any scheduling method

[*]This research was supported by NSF grant CCR-9870871 and a grant from Texas Instruments.

(e.g, *trace scheduling [13], modulo scheduling [22]) and any register assignment technique. We present an experiment using our partitioning techniques with modulo scheduling and Chaitin/Briggs register assignment [9, 6]. Our results show that it is possible to generate good code for partitioned-register-bank architectures.*

1. Introduction

With aggressive instruction scheduling techniques and significant increases in instruction-level parallelism (ILP), modern computer architectures have seen an impressive increase in performance. Unfortunately, large amounts of ILP hardware and aggressive instruction scheduling techniques put large demands on a machine's register resources. With these demands, it becomes difficult to maintain a single monolithic register bank. The number of ports required for such a register bank severely hampers access time [8, 14]. Partitioned register banks are one mechanism for providing high degrees of ILP with a high clock rate (Texas Instruments already produces several DSP chips that have partitioned register banks to support high ILP [24].) Unfortunately, partitioned register banks may inhibit achieved ILP, since some mechanism is required to allow functional units access to "non-local" values, i.e. registers contained in a different partition. One way to provide non-local register values is to add extra instructions to move the data to the register bank of an available functional unit in order to allow execution. Another approach is to provide some 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.

So, a compiler for an ILP architecture with partitioned register banks must decide, for each operation, not only where the operation fits in an instruction schedule, but also in which register partition(s) the operands of that operation will reside, which, in turn, will determine which functional unit(s) can efficiently perform the operation. Obtaining an efficient assignment of operations to functional units is not an easy task as two opposing goals must be balanced. Achieving near-peak performance requires spreading the computation over the functional units equally, thereby maximizing their utilization. At the same time the compiler must minimize communication costs resulting from copy operations introduced by distributing computation.

Most of the limited work in the areas of code generation for ILP architectures with partitioned register banks has dealt with entire functions, sometimes using global scheduling techniques, but often considering only local scheduling. It is now well accepted that extensive loop optimization is necessary to take advantage of significant program ILP. Perhaps the most popular of loop optimization techniques currently in use in the ILP compiler community is software pipelining [18, 22, 3]. Therefore, while our register partitioning method is applicable to entire programs, we will concentrate on software pipelined loops for the experimental work in this paper. There are at least two excellent reasons to restrict ourselves to software pipelined loops. One is that because most of a program's execution time is typically spent in loops, loop optimization is particularly important for good code generation. But perhaps more importantly, from the perspective of this paper is that software pipelining both leads to the greatest extraction of parallelism expected within a program and also places enormous requirements on an ILP architecture's register resources. Considering the combination of maximal parallelism and large register requirements, then, we would expect partitioning of software pipelined loops to be a particularly difficult problem, as well as an important one to solve.

Register banks can be partitioned such that one register bank is associated with each functional unit or by associating a cluster of functional units with each register bank. In general, we would expect that cluster-partitioned register banks would allow for "better" allocation of registers to partitions (fewer copies would be needed and, thus, less degradation when compared to an ideal ILP model) at the expense of adding additional complexity to assigning registers within each partition as additional pressure is put on each register bank due to increased parallelism.

Previous approaches to this problem have relied on building a directed acyclic graph (DAG) that captures the precedence relationship among the operations in the program segment for which code is being generated. Various algorithms have been proposed on how to partition the nodes of this "operation" DAG, so as to generate an efficient assignment of functional units. This paper describes a technique for allocating registers to partitioned register banks that is based on the partitioning of an entirely different type of graph. Instead of trying to partition an operation DAG, we build an undirected graph that interconnects those program data values that appear in the same operation, and then partition this graph. This graph allows us to support retargetability by abstracting machine-dependent details into node and edge weights. We call this technique *register component graph partitioning*, since the nodes of the graph represent virtual registers appearing in the program's intermediate code.

Since the experimental work included here relies solely on software pipelined loops, we first digress, in Section 2, to briefly describe software pipelining. Section 3 then looks at others attempts to generate code for ILP architectures with partitioned register banks. Section 4 describes our general code-generation framework for partitioned register banks while Section 5 outlines the greedy heuristic we use in register partitioning. Finally, Section 6 describes our experimental evaluation of our greedy partitioning scheme and Section 7 summarizes and suggests areas for future work.

2. 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 execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelines where speed-up is achieved by overlapping execution of different operations.

Allan et al. [3] provide an good summary of current software pipelining methods, dividing software pipelining techniques into two general categories called *kernel recognition* methods [1, 2, 4] and *modulo scheduling* methods [18, 25, 22]. The software pipelining used in this work is based upon modulo scheduling 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, *MinII,* required between initiating execution of successive loop iterations. Once that minimum initiation interval is determined, instruction scheduling attempts to match that minimum schedule while respecting resource and dependence constraints. After a schedule has been found, code to set up the software pipeline (prelude) and drain the pipeline (postlude) are added. Rau [22] provides a detailed discussion of an implementation of modulo scheduling, and in fact our implementation is based upon Rau's.

3. Other Partitioning Work

Ellis described the first solution to the problem of generating code for partitioned register banks in his dissertation [11]. His method, called BUG (bottom-up greedy), is applied to a scheduling context at a time (*e.g.,* a trace). His method is intimately intertwined with instruction scheduling and utilizes machine-dependent details within the partitioning algorithm. Our method abstracts away machine-dependent details from partitioning with edge and node weights, a feature that is extremely important in the context of a retargetable compiler.

Capitanio et al. present a code-generation technique for limited connectivity VLIWs in [8]. They report results for two of the seven software-pipelined loops tested, which, for three functional units, each with a dedicated register bank show degradation in performance of 57% and 69% over code obtained with three functional units and a single register bank.

Janssen and Corporaal propose an architecture called a Transport Triggered Architecture (TTA) that has an interconnection network between functional units and register banks so that each functional unit can access each register bank [17]. They report results that show significantly less degradation than the partitioning scheme of Capitanio et al. However, their interconnection network actually represents a different architectural paradigm making comparisons less meaningful. Indeed it is surmised [15] that their interconnection network would likely degrade processor cycle time significantly, making this architectural paradigm infeasible for hardware supporting the high levels of ILP where maintaining a single register bank is impractical. Additionally, chip space is limited and allocating space to an interconnection network may be neither feasible nor cost effective.

Farkas, et al.[12] propose a dynamically scheduled partitioned architecture. They do not need to explicitly move operands between register banks as the architecture will handle the transfer dynamically. Thus, comparisons are difficult.

Ozer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [21]. They state that UAS is an improvement over BUG since UAS can perform schedule-time resource checking while partitioning. This allows UAS to manage the partitioning with the knowledge of the bus utilization for copies between partitions. Ozer's experiments compare UAS and BUG on multiple architectures. They use architectures with 2 register partitions and look at clustering 2 general-purpose functional units with each register bank or clustering a floating-point unit, a load/store unit and 2 integer units with each register bank. Copies between register banks do not require issue slots and are handled by a bus, leading to an inter-cluster latency of 1 cycle. They report experimental results for both 2-cluster, 1-bus and 2-cluster, 2-bus models. In both cases, their results show UAS performs better than BUG. Ozer'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 busses. This degradation is with respect to the (unrealistically) ideal model of a single register bank for an 8-wide machine. We found somewhat similar results for 20 whole programs when we applied the greedy algorithm described here [16]. In a 4-wide machine with 4 partitions (of 1 functional unit each) we found a degradation of roughly 11% over the ideal of a 4-wide machine with a single register bank. Exact comparison of our results with Ozer et al. is complicated by the different experimental parameters, including:

- Ozer et al. have 2 clusters in an 8-wide machine while we looked at 4 clusters in a 4-wide machine. Its difficult to say how this might effect the results.

- our model included general function units while theirs did not. This should lead to slightly greater degradation for us, since the general functional-unit model should allow for slightly more parallelism

- Ozer et al. used global scheduling in their experiments, while we used local scheduling only. This should make their schedules more aggressive and certainly overcomes any additional parallelism available due to our general functional-unit model.

- our model includes a latency of two cycles for copies of integer values from one partition to another, and a latency of three cycles for accessing non-local floating point values. Their model assumes a 1-cycle latency for all inter-cluster communication. Of course this penalizes our partitioning somewhat as well.

Taken together, Ozer et al.'s results and ours suggest that one might reasonably expect 10-20% degradation for in execution time when partitioning by 2 or 4 clusters. However, there are enough differences in the experimental techniques to make more significant evaluation of the relative merits of UAS vs. our partitioning method difficult at best. More in-depth comparison of these two methods is on-going and will be presented in a different forum.

Nystrom and Eichenberger present an algorithm for partitioning for modulo scheduling [20]. They perform partitioning first with heuristics to allow modulo scheduling to be effective. Specifically, they try to prevent inserting copies that will lengthen the recurrence constraint of modulo scheduling. If copies are inserted off of critical recurrences in recurrence-constrained loops, the initiation interval for these loops may not be increased if enough copy resources are available. Since Nystrom and Eichenberger deal only with software pipelined loops, their work is most closely associated with that we discuss here. Because of that, we defer further discussion of their work and comparison with ours to Section 6.3 after we describe our experimental results.

4. Register Assignment with Partitioned Register Banks

A good deal of work has investigated how registers should be assigned when a machine has a single register "bank" of equivalent registers [9, 10, 6]. However, on architectures with high degrees of ILP, it is often inconvenient or impossible to have a single register bank associated with all functional units. Such a single register bank would require too many read/write ports to be practical [8]. Consider an architecture with a rather modest ILP level of six. This means that we wish to initiate up to six operations each clock cycle. Since each operation could require up to three registers (two sources and one destination) such an architecture would require simultaneous access of up to 18 different registers from the same register bank. An alternative to the single register bank for an architecture is to have a distinct set of registers associated with each functional unit (or cluster of functional units). Examples of such architectures include the Multiflow Trace and several chips manufactured by Texas Instruments for digital signal processing [24]. Operations performed in any functional unit would require registers with the proper associated register bank. Copying a value from one register bank to another could be expensive. The problem for the compiler, then, is to allocate registers to banks to reduce the number copies from one bank to another, while retaining a high degree of parallelism.

This section outlines our approach to performing register allocation and assignment for architectures that have a partitioned register set rather than a single monolithic register bank that can be accessed by each functional unit. Our approach to this problem is to:

1. Build intermediate code with symbolic registers, assuming a single infinite register bank.
2. Build data dependence DAGs (DDD)s and perform instruction scheduling still assuming an infinite register bank.
3. Partition the registers to register banks (and thus preferred functional unit(s)) by the “Component” method outlined below.
4. Re-build DDDs and perform instruction scheduling attempting to assign operations to the “proper” (cheapest) functional unit based upon the location of the registers.
5. With functional units specified and registers allocated to banks, perform “standard” Chaitan/Briggs graph coloring register assignment for each register bank.

4.1. Partitioning Registers by Components

Our method requires building a graph, called the *register component graph*, 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 can build the register component graph with a single pass over either the intermediate code representation of the function being compiled, or alternatively, with a single pass over scheduled instructions. We have found it useful to build the graph from what we call an “ideal” instruction schedule. The ideal schedule, by our definition, uses the issue-width and all other characteristics of the actual architecture except that it assumes that all registers are contained in a single monolithic multi-ported register bank. Once the register component graph is built, values that are not connected in the graph are good candidates to be assigned to separate register banks. Therefore, once the graph is built, we must find each connected component of the graph. Each component represents registers that can be allocated to a single register bank. In general, we will need to split components to fit the number of register partitions available on a particular architecture, essentially computing a cut-set of the graph that has a low copying cost and high degree of parallelism among components.

The major advantage of the register component graph is that it abstracts away machine-dependent details into costs associated with the nodes and edges of the graph. This is extremely important in the context of a retargetable compiler that needs to handle a wide variety of machine idiosyncrasies. Examples of such idiosyncrasies include operations such as `A = B op C` where each of `A`, `B` and `C` must be in separate register banks. This could be handled abstractly by weighting the edges connecting these values with negative value of “infinite” magnitude, thus ensuring that the registers are assigned to different banks. An even more idiosyncratic example, but one that exists on some architectures, would require that `A`, `B`, and `C` not only reside in three different register banks, but specifically in banks `X`, `Y`, and `Z`, and furthermore that `A` use the same register number in `X` that `B` does in `Y` and that `C` does in `Z`. Needless to say, this complicates matters. By pre-coloring [9] both the register bank choice and the register number choice within each bank, however, it can be accommodated within our register component graph framework.

4.2. A Partitioning Example

To demonstrate the register component graph method of partitioning, we show an example of how code would be mapped to a partitioned architecture with 2 functional units, each with its own register bank. For simplicity we assume unit latency for all operations.
For the following high-level language statement:

```
xpos = xpos + (xvel*t) + (xaccel*t*t/2.0)
```

the hypothetical intermediate code and corresponding register component graph appear in Figure 2. An optimal schedule for this code fragment, assuming a single multi-ported register bank is shown in

load r1, xvel	load r2, t
mult r5, r1, r2	load r3, xaccel
load r4, xpos	mult r7, r3, r2
add r6, r4, r5	div r8, r2, 2.0
mult r9, r7, r8	
add r10, r6, r9	
store xvel, r10	

Figure 1. Optimal Schedule for Example Code

```
load  r1,xvel
load  r2,t
load  r3,xaccel
load  r4,xpos
mult  r5,r1,r2
add   r6,r4,r5
mult  r7,r3,r2
div   r8,r2,2.0
mult  r9,r7,r8
add   r10,r6,r9
store xvel,r10
```

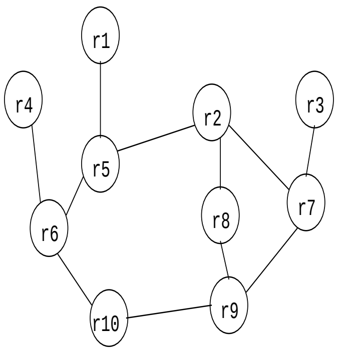

Figure 2. Code and Corresponding Register Graph

Figure 1. It requires 7 cycles to complete. One potential partitioning of the above graph (given the appropriate edge and node weights) is the following:

$$P_1 : \{r1, r2, r4, r5, r6\}$$

$$P_2 : \{r3, r7, r8, r9, r10\}$$

Given the unit latency assumption we can generate the schedule in Figure 3, which takes a total of 9 cycles, an increase of 2 cycles over the ideal case. Notice that two values (r2 and r6) required copying to generate the schedule of Figure 3.

load r1, xvel	load r3, xaccel
load r2, t	
move r22, r2	
mult r5, r1, r2	mult r7, r3, r22
load r4, xpos	div r8, r22, 2.0
add r6, r4, r5	mult r9, r7, r8
move r66, r6	
	add r10, r66, r9
	store xvel, r10

Figure 3. Schedule for the Register Graph Partitioning Heuristic

5. Our Greedy Heuristic

The essence of our greedy approach is to assign heuristic weights to both the nodes and edges of the RCG. We then place each symbolic register, represented as an RCG node, into one of the available register partitions. To do this we assign RCG nodes in decreasing order of node weight. To assign each RCG node, we compute the "benefit" of assigning that node to each of the available partitions in turn. Whichever partition has the largest computed benefit, corresponding to the lowest computed cost, is the partition to which the node is allocated.

To discuss the heuristic weighting process we need to digress for a moment to briefly outline some important structures within our compiler. The compiler represents each function being compiled as a control-flow graph (CFG). The nodes of the CFG are basic blocks. Each basic block contains a data dependence DAG (DDD) from which an ideal schedule is formed. This ideal schedule, defined in Section 4.1, is a basically a list of instructions, with each instruction containing between zero and N operations, where N is the "width" of the target ILP architecture. The nodes of the DDDs are the same operations that are scheduled in the ideal schedule.

In assigning heuristic weights to the nodes and edges of the RCG, we consider several characteristics of the basic blocks, DDDs, and ideal schedules for the function being compiled. These include, for each operation of the DDD or ideal schedule:

Nesting Depth : the nesting depth of the basic block in which the operation is scheduled.

DDD Density : defined to be the number of operations (DDD nodes) in the DDD divided by the number of instructions required to schedule those operations in the ideal schedule.

Flexibility : defined to be the "slack" available for a DDD node. This slack, computed as scheduling proceeds, is the difference between the earliest time a node could be scheduled (based upon the scheduling of all that DDD node's predecessors) and the latest time that the DDD node could be scheduled without requiring a lengthening of the ideal schedule. This is used both to identify nodes on a DDD critical path (such nodes will have zero slack time) and to weigh nodes with less flexibility as more important. (In our actual computation, we add 1 to Flexibility so that we avoid divide-by-zero errors).

Defined : the set of symbolic registers assigned new values by an operation.

Used : the set of symbolic registers whose values are used in the computation of an operation.

To compute the edge and node weights of the RCG, we look at each operation, O, of instruction, I, in the ideal schedule and update weights as follows:

- for each pair (i, j) where $i \in Defined(O)$ and $j \in Used(O)$, compute wt as follows

  if Flexibility(O) is 1, $wt = 200$ otherwise $wt = 0$

  $wt = \frac{(20+10^{NestingDepth(O)}+wt)*\ DDD\ Density(O)}{Flexibility(O)}$

  Either add a new edge in the RCG with value wt or add wt to the current value of the edge (i, j) in the RCG. This additional weight in the RCG will reflect the fact that we wish symbolic registers i and j to be assigned to the same partition, since they appear as defined and used in the same operation.

In addition, add wt to the RCG node weights for both i and j.

- for each pair (i, j) where $i \in Defined(O_1)$ and $j \in Defined(O_2)$ and O_1 and O_2 are two distinct operations in I, compute wt as follows

  if Flexibility(O) is 1, $wt = -500$ otherwise $wt = 0$

  $wt = \frac{(-50-10^{NestingDepth(O)}+wt)*\ DDD\ Density(O)}{Flexibility(O)}$

  Either add a new edge in the RCG with value wt or add wt to the current value of the edge (i, j) in the RCG. This additional weight in the RCG will reflect the fact that we might wish symbolic registers i and j to be assigned to different partitions, since they each appear as a definition in the same instruction of the ideal schedule. That means that not only are O_1 and O_2 data-independent, but that the ideal schedule was achieved when they were included in the same instruction. By placing the symbolic registers in different partitions, we increase the probability that they can be issued in the same instruction.

Once we have computed the weights for the RCG nodes and edges, we choose a partition for each RCG node as described in Figure 4. Notice that the algorithm described in Figure 4 does indeed try placing the RCG node in question in each possible register bank and computes the benefit to be gained by each such placement. The RCG node weights are used to define the order of partition placement while the RCG edge weights are used to compute the benefit associated with each partition. The statement

ThisBenefit -= $NumberOfRegsAssignedToRB^{1.7}$

adjusts the benefit to consider how many registers are already assigned to a particular bank. The effect of this is to attempt to spread the symbolic registers somewhat evenly across the available partitions.

At the present time both the program characteristics that we use in our heuristic and the weights assigned to each characteristic are determined in an ad hoc manner. We could, of course, attempt to fine-tune both the characteristics and the weights using any of a number of optimization methods. We could, for example, use a stochastic optimization method such as genetic algorithms, simulated annealing, or tabu search to define RCG-weighting heuristics based upon a combination of architectural and program characteristics. We reported on such a study to fine-tune local instruction-scheduling heuristics in [5].

6. Experimental Evaluation

To evaluate our framework for register partitioning in the context of our greedy algorithm we software pipelined 211 loops extracted from Spec 95. To partition registers to the clusters, we used the greedy

```
Algorithm Assign RCG Nodes to Partitions
{
        foreach RCG Node, N, in decreasing order of weight(N)
                BestBank = choose-best-bank (N)
                Bank(N) = BestBank


        choose-best-bank(node)
                BestBenefit = 0
                BestBank = 0
                foreach possible register bank, RB
                        ThisBenefit = 0
                        foreach RCG neighbor, N, of node assigned to RB
                                ThisBenefit += weight of RCG edge (N,node)
                        ThisBenefit -= $NumberOfRegsAssignedToRB^{1.7}$
                        If ThisBenefit $>$ BestBenefit
                                BestBenefit = ThisBenefit
                                BestBank = RB
                return BestBank
}
```

Figure 4. Choosing a Partition

heuristic described in Section 5, as implemented in the Rocket compiler [23]. We actually evaluated the greedy algorithm with two slightly different models of 16-wide partitioned ILP processors, as described in Section 6.1. Section 6.2 provides analysis of our findings, and Section 6.3 compares our results with the similar work of Nystrom and Eichenberger [20]

6.1 Machine Models

The machine model that we evaluated consists of 16 general-purpose functional units grouped in N clusters of differing sizes, with each cluster containing a single multi-ported register bank. We include results here for N of 2, 4, and 8 equating to 2 clusters of 8 general-purpose functional unit each, 4 clusters of 4 functional units and 8 clusters of 2 units. The general-purpose functional units potentially make the partitioning more difficult for the very reason that they make software pipelining easier and thus we're attempting to partition software pipelines with fewer "holes" than might be expected in more realistic architectures. Within our architectural meta-model of a 16-wide ILP machine, we tested two basic machine models that differed only in how copy operations (needed to move registers from one cluster to another) were supported. These are:

Embedded Model in which explicit copies are scheduled within the functional units. To copy a register value from cluster A to cluster B in this model requires an explicit copy operation that takes an instruction slot for one of B's functional units. The advantage of this model is that, in theory, 16 copy operations could be started in any one execution cycle. The disadvantage is that valuable instruction slots are filled.

Copy-Unit Model in which extra issue slot(s) are reserved only for copies. In this model each of N clusters can be thought to be attached to N busses and each register bank would contain $log_2 N + \frac{48}{N}$ ports, $\frac{48}{N}$ to support the $\frac{16}{N}$ functional units per cluster with the additional $log_2 N$ ports being devoted to copying registers to/from other cluster(s). The advantage of this model is that inter-cluster communication is possible without using instruction slots in the functional units. The disadvantage is the additional busses required, and the additional ports on the register banks. Our scheme of adding N busses for an N-cluster machine, but only $log_2 N$ additional ports per register bank is based upon the fact that the additional hardware costs for busses is expected to be minimal compared to the hardware cost for additional ports per register bank.

Both machine models used the following operation latencies.

- Integer copies take 2 cycles to complete for both the embedded-model and the copy-unit model.
- Floating copies take 3 cycles to complete.
- Loads take 2 cycles
- Integer Multiplies take 5 cycles
- Integer divides take 12 cycles
- Other integer instructions take 1 cycle
- Floating Point Multiplies take 2 cycles
- Floating divides take 2 cycles
- Other floating point instructions take 2 cycles
- Stores take 4 cycles

  Having inter-cluster communication have a latency of two cycles for integers and three for floats is, of course, unrealistically harsh for the the copy-unit model. However, since those are the latencies used in the embedded model (where they are somewhat more realistic) we decided to use those latencies as well in our copy-unit models for consistency.

6.2 Results

When considering the efficiency of any register partitioning algorithm an important consideration is how much parallelism was found in the pipeline. Consider a 16-wide ILP architecture. In a loop where the ideal schedule executes only four operations per cycle in the loop kernel, the limited parallelism makes the problem of partitioning considerably easier than if parallelism were eight operations per cycle. Not only would additional instruction slots be available with the more limited parallelism, but the copy-unit model would also “benefit”, since the fewer operations would presumably require fewer non-local register values than a pipeline with more parallelism. We can get a good estimate of the parallelism that software pipelining finds in loops by measuring the Instructions Per Cycle (IPC). Perhaps a more generic term would be Operations Per Cycle, but IPC has been an accepted term in the superscalar literature, where instructions and operations have a 1 to 1 relationship. So, to reduce confusion, we’ll use IPC as well.

	Two Clusters		Four Clusters		Eight Clusters	
Model	Embedded	Copy Unit	Embedded	Copy Unit	Embedded	Copy Unit
Ideal	8.6	8.6	8.6	8.6	8.6	8.6
Clustered	9.3	6.2	8.4	7.5	6.9	6.8

Table 1. IPC of Clustered Software Pipelines

	Two Clusters		Four Clusters		Eight Clusters	
Average	Embedded	Copy Unit	Embedded	Copy Unit	Embedded	Copy Unit
Arithmetic Mean	111	150	126	122	162	133
Harmonic Mean	109	127	119	115	138	124

Table 2. Degradation Over Ideal Schedules — Normalized

Table 1 shows the IPC for the loops pipelined for our 16-wide ILP architecture. The "Ideal" line shows that, on average we scheduled 8.6 operations per cycle in the loop kernel. This is of course with no clustering, and so it does not change for the different columns in the table. The "Ideal" row is included merely for comparison and to emphasize the point that the 16-wide ideal schedule is the same no matter the cluster arrangement. The IPC for the "Clustered" row was computed slightly differently for the embedded copies than for the copy-unit model. In the embedded model, the additional copies are counted as part of the IPC, but not in the copy-unit model, where we assume additional communication hardware obviates the need for explicit copy instructions.

The main purpose of Table 1 is to show the amount of parallelism that we found in the loops. Table 2 shows the degradation in loop kernel size (and thus execution time) due to partitioning, for all 211 loops we pipelined. All values are normalized to a value of 100 for the ideal schedule. Thus, the entry of 111 for the arithmetic mean of the embedded model of two clusters indicates that when using the embedded model with two clusters of 8 functional units each, the partitioned schedules were 11% longer (and slower) than the ideal schedule. Similarly, the entry of 133 for the arithmetic mean of the copy-unit model, eight cluster architecture shows a degradation of 33% over ideal. We include both arithmetic and harmonic means since the arithmetic mean tends to be weighted towards large numbers, while the harmonic mean permits more contribution by smaller values.

One use of the Table 2 is to compare the efficiencies of the embedded vs. the copy-unit models. Looking at the data in Table 2, we see that both the embedded copy model and the copy-unit model required degradation of about 25% for the 4-cluster (of 4 functional units) model. We should not be too surprised that for the two-cluster machine, the embedded copies model significantly outperformed the copy-unit model. The fewer non-local register copies needed for the two partitions can be incorporated relatively easily in the 8-wide clusters, leading to an arithmetic mean of 111. On the other hand, the non-local accesses needed in the copy model, where only 1 port per cluster was available ($log_2 2$), overloaded the communication capability of the copy-unit model, leading to a 50% degradation. On the other hand, the 8-cluster machine showed just the opposite effect, as we might imagine. Here we have 3 ports per partition for the two functional units included in the cluster, leading to a better efficiency (33% degradation) than trying to include the many copies required for supporting 8 partitions in the two-wide clusters in the embedded model.

Another use of Table 2 is to get some appreciation of the level of schedule degradation we can expect from a relatively "balanced" clustering model such as the 4 clusters of 4-wide general purpose functional units. The roughly 20-25% degradation that we see is, in fact, greater than what has been previously shown possible for entire programs [21, 16], and that is expected, since software pipelined loops will generally contain more parallelism than non-loop code, but the additional degradation we have found still seems within reason.

6.3 Comparison with Nystrom and Eichenberger

Nystrom and Eichenberger perform their experiments on architectures with clusters of 4 functional units per register bank. Each architecture, in turn, has 2 or 4 register banks. The four functional units are either general purpose or a combination of 1 floating-point unit, 2 integer units and 1 load/store unit. Copies between register banks do not require issue slots and are handled by either a bus or a point-to-point grid. Note that the bus interconnect needs to be reserved for 1 cycle for a copy and copies between more than one pair of register banks can occur during the same cycle with some restrictions. Nystrom and Eichenberger's partitioning algorithm includes iteration, which differentiates it from our model. It also is designed specifically for software pipelined loops, in that the chief design goal of their approach is to add copies such that maximal recurrence cycle(s) in the data dependence graph are not lengthened if at all possible. Again, this is in contrast to our current greedy method which does not consider recurrence paths directly. Thus, our method is not as directly tied to software pipelined loops, but is easily adapted to entire functions.

Nystrom and Eichenberger report only the percentage of loops that required no change in the II after partitioning. This represents that percentage of loops that showed 0% degradation, using our terminology. Figures 5, 6, and 7 show similar data for our 2-cluster, 4-cluster and 8-cluster machines. Those

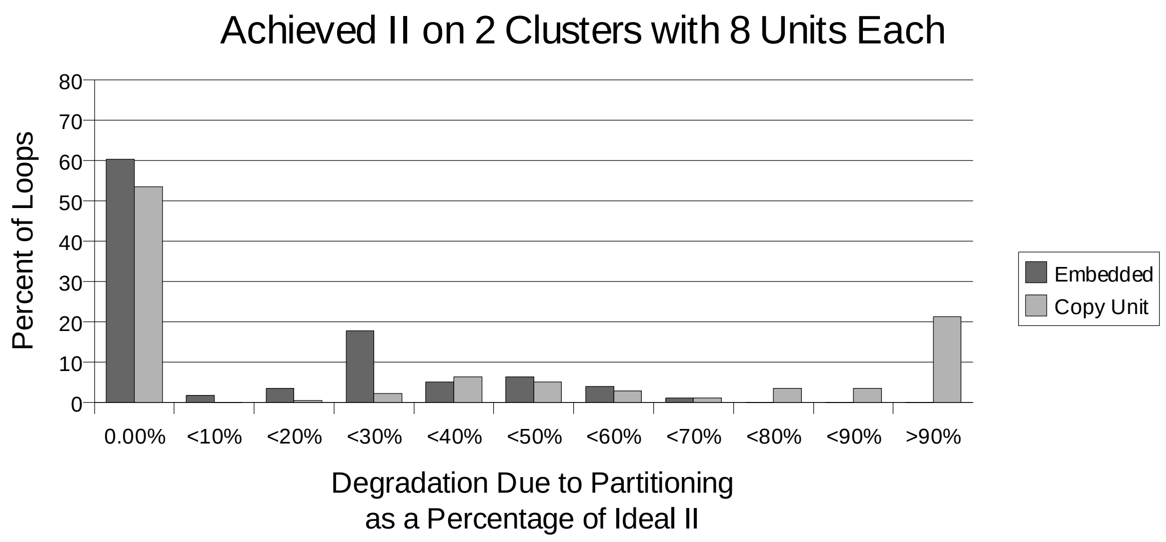

Figure 5.

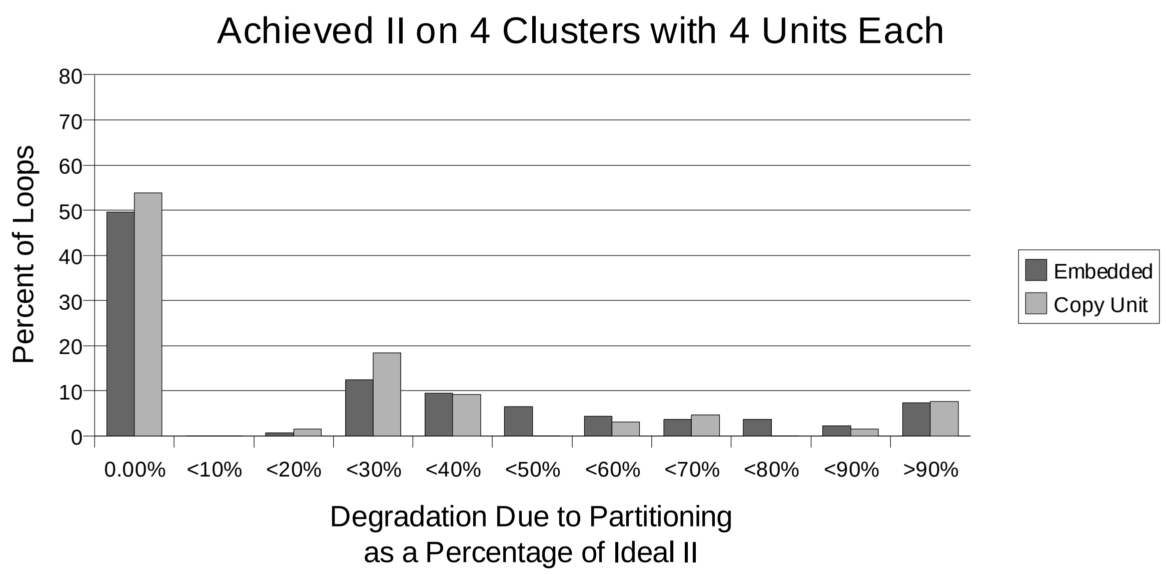

Figure 6.

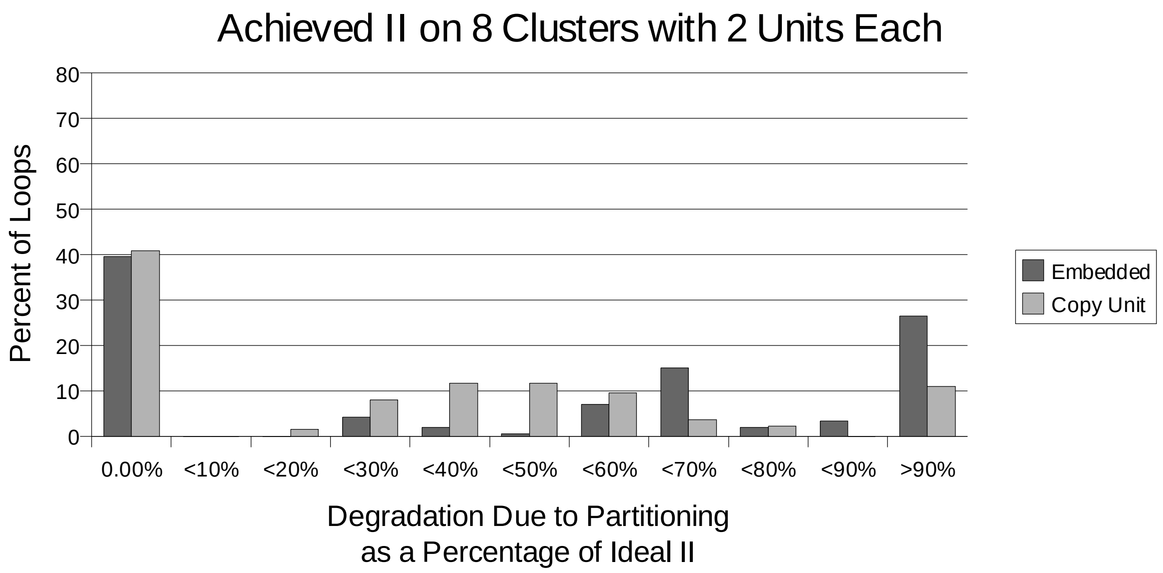

Figure 7.

figures show histograms that group the loops (as a percentage of the 211 loops pipelined) based upon the level of degradation. Looking at the 2-cluster loops, we can see that roughly 60% of the loops required no degradation. The 4-cluster model scheduled about 50% of the loops the loops with no degradation and the 8-cluster about 40%. This is significantly less than Nystrom and Eichenberger's reported results of roughly 98% of loops with no degradation for a 4-cluster machine.

How can we explain the large difference between our findings and Nystrom and Eichenberger's, at least in terms of number of pipelined loops requiring no degradation. While there are certainly a large number of differences in our experimental assumptions, perhaps the largest uncertainty is due to pipelining different loops. Nystrom and Eichenberger indicate that they used 1327 loops obtained from HP Labs. These loops were all innermost loops, with no procedure calls, no early exist, and fewer than 30 basic blocks. The loops came from Fortran77 code to which load-store elimination, recurrence back-substitution, and IF-conversion had already been applied. Our test suite consisted of 211 loops, extracted from Fortran 77, that were all single-block innermost loops. It is possible that, by using only single-block loops, our test suite exhibited more parallelism, thus make the modulo scheduling easier, but the partitioning a harder problem. Since Nystrom and Eichenberger do not indicate the average IPC for their loops, it is difficult to assess the difference due to the different test suites. In addition, each of the following differences between our experimental setup and that of Nystrom and Eichenberger make direct comparisons difficult.

- Nystrom and Eichenberger's partitioning algorithm calls for iteration. In that sense, our greedy algorithm can be thought of as an initial phase before iteration is performed. Nystrom and Eichenberger do include results for non-iterative partitioning, and they found that it led to degradation in about 5% of the loops, compared to 2% when they iterated.

- Our longer latency times for copies may have had a significant effect on the number of loops that we could schedule without degradation. We used latency of 2 cycles for integer copies and 3 for floating point values, while Nystrom and Eichenberger used latency of 1 for all non-local access.

- We use a slightly different modulo scheduling technique than Nystrom and Eichenberger. We use "standard" modulo scheduling as described by Rau [22] while they use Swing Scheduling [19] that attempts to reduce register requirements. Certainly this could have an effect on the partitioning of registers.

In addition to the above differences in our experimental model, Nystrom and Eichenberger limit themselves to what we're calling the copy-unit model, while we report results for the embedded model as well. And, while they report only the percentage of loops without degradation, we compute an overall average degradation as well.

Going beyond the experiments, our greedy partitioning method is easily applicable to entire programs, since we could easily use both non-loop and loop code to build our register component graph and our greedy method works on a function basis. Nystrom and Eichenberger restrict themselves to loops only and their method includes facets that makes it difficult to apply directly to entire functions.

7. Conclusions and Future Work

In this paper we have presented a robust framework for code generation for ILP architectures with partitioned register banks. The essence of our framework is the *register component graph* that abstracts away machine specifics within a single representation. This abstraction is especially important within the context of our retargetable compiler for ILP architectures.

Our framework and greedy partitioning method are applicable to both whole programs and software pipelined loops. Previously, we have shown that on whole programs for an 8-wide VLIW architecture with 8 register banks, we can expect roughly a 10% degradation in performance over an 8-wide machine with 1 monolithic register bank [16]. In this paper, we give experimental results for a set of software pipelined loops on an architecture with 4 register banks and 4 functional units per bank. Our results show that we could expect on the order of a 25% degradation for software pipelined loops over a 16-wide architecture with 1 monolithic register bank. The IPC for these loops averaged over 8.5.

Comparing our technique with the more aggressive Nystrom and Eichenberger method (they iterate on partitioning and we don't), we get no degradation for roughly 50% of our software pipelined loops on a 4-cluster machine with 4 functional units per cluster. They get no degradation for 98%. However, the lack of IPC numbers in their work makes it difficult to make a straightforward comparison. In addition, we view our technique as more robust, since it already is set up for entire functions, while their basic algorithm is based upon the assumption that they are generating code for a single loop only.

In the future, we will investigate fine-tuning our greedy heuristic by using off-line stochastic optimization techniques. We will also investigate other loop optimizations that can increase data-independent parallelism in innermost loops. Finally, we will investigate a tiled approach that will allow specialized heuristics for loops [7].

With trends in machine design moving towards architectures that require partitioned register banks, compilers must deal with both parallelism and data placement. If these architectures are to achieve their full potential, code generation techniques such as the ones presented in this paper are vital.

References

[1] 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.

[2] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Optimization Technique. In *Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science, #300*, pages 221–235, Atlanta, GA, March 1988.

[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] S. Beaty, S. Colcord, and P. Sweany. Using genetic algorithms to fi ne-tune instruction scheduling. In *Proceedings of the Second International Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 1996.

[6] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, 1991.

[8] 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.

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[10] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*.

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

[12] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. A mulicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pages 149–159, Research Triangle Park, NC, December 1997.

[13] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[14] 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.

[15] S. Freudenberger. Private communication.

[16] J. Hiser, S. Carr, and P. Sweany. Register Assignment for Architectures with Partitioned Register Banks, 1999. Submitted to International Conference on Parallel Architectures and Compilation Techniques (PACT 99) for publication.

[17] J. Janssen and H. Corporaal. Partitioned register files for TTAs. In *Twenty-Eighth Annual Symposium on Micorarchitecture (MICRO-28)*, pages 301–312, Dec. 1995.

[18] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[19] J. Llosa, A. Gonzalez, E.Ayguade, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of PACT '96*, pages 80–86, Boston, Massachusets, Oct. 1996.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] N. J. Warter, S. A. Mahlke, W.-M. Hwu, and B. R. Rau. Reverse if-conversion. *SIGPLAN Notices*, 28(6):290–299, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.