Register Assignment for Architectures 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*

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 [3, 9]. 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 [16].) Unfortunately, partitioned register banks may inhibit achieved ILP. An instruction may be able to be scheduled in a particular cycle, but if its data resides in a register bank that is not accessible to the available functional unit, extra instructions must be inserted to move the data to the register bank of an available functional unit in order to allow execution. Therefore, 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)

*This research was supported by NSF grant CCR-980781 and a grant from Texas Instruments.

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.

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 VLIW 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.

2. Previous Work

Ellis described the first solution to the problem of generating code for partitioned register banks in his thesis [7]. 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 [3]. They

report results for two of the seven 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. A major restriction of their implementation is that it only deals with straight-line loops (i.e. there are no conditional branches in the loop). Our implementation is performed on a function basis and imposes no such restrictions. In contrast to the work of both Ellis and Capitanio et al., our partitioning method considers global context and, thus, we feel it provides a distinct advantage. Also, the machine-independent nature of our approach is a distinct advantage for a retargetable compiler, as the machine dependent details can easily be represented in the node and edge weights within the register graph.

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 [11]. They report results that show significantly less degradation than either 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 [10] 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.

Capitanio, et al. [4] present a different algorithm for assigning registers to banks based upon an interconnection network like that proposed by Janssen and Corporaal. They call their method hypergraph coloring. In their model, graph nodes, which represent registers, are connected by hyperedges, edges that can connect more than two nodes. Their algorithm colors nodes such that no edge contains more nodes of the same color (representing a register bank) than the number of ports per register bank. When, during coloring, a node cannot be colored within their restrictions, an operation that accesses the register represented by the uncolorable node is deferred to a later instruction, or a new instruction is inserted and the code rescheduled. The algorithm terminates when all nodes have been colored. To evaluate hypergraph coloring, the authors applied their algorithm to architectures consisting of

1. 8 functional units and 2 register banks,
2. 8 functional units and 4 register banks, and
3. 4 functional units and 2 register banks.

They report, for several benchmarks, the number of code transformations required to perform the coloring and the degradation in performance they caused. They observed no degradation over 5% from an that of an ideal VLIW. Again though, while the performance numbers look good, we consider the architectural paradigm, like the TTA considered by Janssen and Corporaal, to be too expensive to implement efficiently in hardware with the high levels of ILP for which partitioned register banks are actually necessary.

Farkas, et al.[8] 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 [14]. 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, et al., present an experiment comparing UAS and BUG on multiple architectures. They use architectures with 2 register partitions. They look at clustering 2 general-purpose functional units with each register file or clustering a floating-point unit, a load/store unit and 2 integer units with each register file. Copies between register files 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.

Nystrom and Eichenberger present an algorithm for partitioning for modulo scheduling [13]. 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 critcal recurrences in recurrence-constrained loops, the initiation interval for these loops may not be increased if enough copy resources are available.

Nystrom and Eichenberger perform their experiments on architectures with clusters of 4 functional units per register file. Each architecture, in turn, has 2 or 4 register files. 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 files do not require issue slots and are handled by either a bus or a point-to-point grid. Note that bus interconnect needs to be reserved for 1 cycle for a copy and copies between more than one pair of register files can occur during the same cycle with some restrictions. For an architecture with 2 register files, they show that they can achieve no degradation in initiation interval over a single unified register-file machine for 99% of the loops that they schedule. Since there are 4 functional units

per register file, the number of copies needed to give good parallelism is likely not too high.

3. 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 [5, 6, 2]. 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 [3]. 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 [16]. 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.

3.1. 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 [5] both the register bank choice and the register number choice within each bank, however, it can be accommodated within our register component graph framework.

3.2. 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.

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

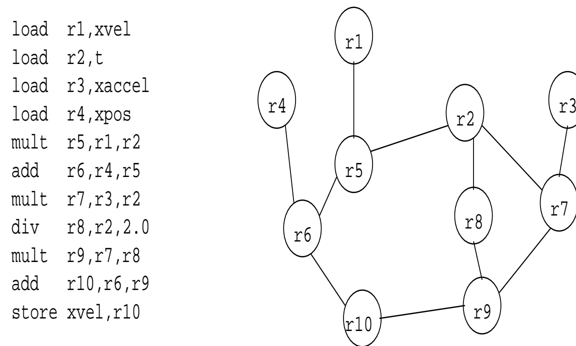

Figure 2. Code and Corresponding Register Graph

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 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

4. 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 descreasing 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 3.1, 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 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. To schedule them in the same instruction of the actual (post-partitioning) schedule, they must be in different register partitions.

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 [1].

5. Experimental Evaluation

To evaluate our framework for register partitioning in the context of our greedy algorithm we compiled 20 C programs using the greedy heuristic described in Section 4, as implemented in the Rocket compiler [15]. To run our partitioned programs we used a simulator of a 4-wide ILP architecture. We actually evaluated the greedy algorithm with two slightly different models of 4-wide partitioned ILP processors, as described in Section 5.1. The benchmarks we used are listed in Section 5.2. Section 5.3 provides

```
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

analysis of our findings.

5.1 Machine Models

The machine model that we evaluated consists of four general-purpose functional units, each with its own set of registers. This is somewhat unrealistic given today's ILP architectures that support 4-wide instructions with two multi-ported register banks, one for integers and one for floating-point values. But we deliberately chose a model designed to lead to more potential copies on the assumption that if our framework works well for a more difficult partitioning problem it should behave well for real architectures as well. Thus, we have 4 partitions. The general-purpose functional units also potentially make the partitioning more difficult for the very reason that they make scheduling easier and thus we're attempting to partition schedules with fewer "holes" than might be expected in more realistic architectures. Within our architectural meta-model of a 4-wide ILP machine with 4 register banks, we tested two basic machine models that differed only in how copy operations (needed to move registers from one bank to another) were supported.

Embedded Model is a 4-wide ILP machine in which explicit copies are scheduled within the four functional units. To copy a register value from bank A to bank B in this model requires an explicit copy operation that takes an instruction slot for functional unit B. The advantage of this model is that, in theory, 4 copy operations could be started in any one execution cycle. The disadvantage is that valuable instruction slots are filled.

Copy-Unit Model is a 4-wide ILP machine that includes 1 extra issue slot reserved only for copies. In this model each functional unit can be thought to be attached to a bus and so during any cycle, exactly one register can be copied from any bank to any other bank. The advantage of this model is that the additional copy slot actually makes this a non-homogeneous 5-wide machine so none of the 4 "original" functional units need be allocated to copy a register. The disadvantage is that only one register value can be copied per cycle.

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
- Ohter floating point instructions take 2 cycles
- Stores take 4 cycles

5.2 Benchmarks

The 20 C programs are listed below, along with a short description of each and minimal execution metrics. For each program we list the number of functions included in the program and the number of instructions required to run the program on the "ideal" 4-wide ILP machine, that being one with a single register bank, and the latencies listed in the Section 5.1.

KillCache KillCache is a program designed to make small caches perform poorly. It creates random data, manipulates it in a way that inhibits temporal and spatial locality of memory, and print the results. The single function in this program executes approximately 44,000K instructions.

MergeSort The MergeSort program allocates space for a moderately sized array, initializes the array to a set of randomly generated numbers, performs a merge sort on the array, and prints the unsorted and sorted arrays. The 4 functions in this program execute approximately 31K instructions.

Dice The Dice program simulates rolling a pair of dice using a random number generator. It's function is to ensure that the random number generator provides a good distribution of numbers over for large number of calls. The 4 functions in this program execute approximately 48K instructions.

Pascal The Pascal program recursively calculates and prints out each number in the first 10 rows of Pascal's triangle. The 3 functions in this program execute approximately 23K instructions.

IsPrime The IsPrime program determines which of the first 1000 integers are prime. The two functions execute approximately 47K instructions.

ListTest This program creates, appends, destroys, and prints lists. It demonstrates the power of our algorithm on small, modular code. The 6 related functions take approximately 555K instructions.

MatrixMult This program initializes two arrays, multiplies them together, and prints out the resulting array. This single function related to executing this program takes 227K instructions.

Malloc This program allocates an array using malloc, initializes it, and prints out the array. Only one function is used to implement this program and it executes a total of 9000K instructions.

BinarySearch This program allocates an array, fills it with random numbers, sorts it, and then repeatedly searches the array using a binary search. The 3 necessary functions for this execute a total of 644K instructions.

Hanoi This program recursively solves the 5 disk Towers of Hanoi problem. The 2 related functions take a total of 0.6K instructions to execute.

8Queens This program enumerates every solution to the 8 Queens on a chess board problem. The three functions used to implement this algorithm take a total of 248K instructions to execute.

BubbleSort This program runs the bubble sort algorithm. The 3 functions that comprise this program take 80K instructions to execute.

Nsieve This program is the Sieve of Eratosthenes program by Al Aburto. The 3 functions used to implement this algorithm take 81M instructions to execute.

Heapsort This program is the Heapsort program for variable sized arrays by Al Aburto. The two functions that comprise this program execute 25M instructions.

Dmxpy This program performs floating point calculations on several global arrays. The single function takes approximately 4K instructions to execute.

Gauss This program solves the Gaussian elimination problem for a moderate array. The 10 functions that comprise this program take approximately 91K instructions to execute.

TSP This program solves the Traveling salesman problem for a small set of cities. The exponential algorithm takes approximately 11K instructions to execute the 4 functions used.

Livermore The Livermore loops, which includes kernels taken from scientific code. The 3 functions take 1307K instructions to execute.

Whetstone This program is the whetstone benchmark written in C. It takes approximately 981K instructions to execute the 5 functions.

Frac This program finds rational approximation to floating point value and is written by Robert Craig, AT&T Bell Labs - Naperville. The 2 functions used to do this take approximately 0.5K instructions.

5.3 Results

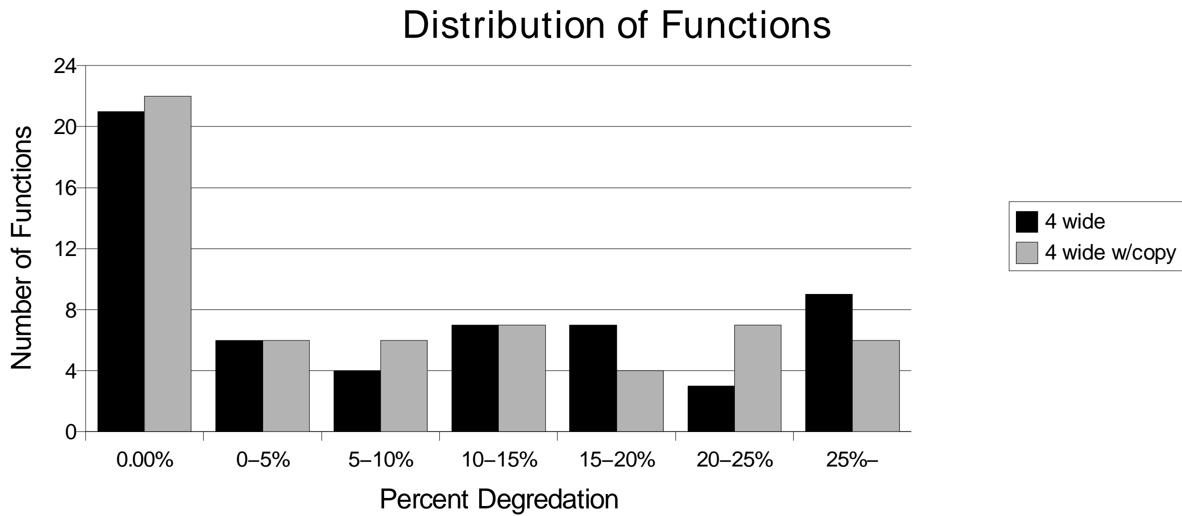

Figure 5. Histogram

Table 1 shows the amount of additional execution incurred by our partitioning as a percentage of the execution time required for an ideal schedule, i.e. one without partitioning. Note that, while four programs (Pascal, IsPrime, ListTest, and Nsieve) required no additional execution time to support partitioning, additional execution time as high as 32.4% of the ideal execution cycles (for MergeSort under the Embedded-Copies model) was required for others. Note also that, while the additional execution time required was quite often the same for both machine models (11 times), the Copy-Unit model did

Program	Embedded Copies	Copy Unit
KillCache	18.9	18.9
MergeSort	32.4	22.0
Dice	3.9	3.9
Pascal	0.0	0.0
IsPrime	0.0	0.0
ListTest	0.0	0.0
MatrixMult	1.3	1.3
Malloc	22.2	22.2
BinarySearch	19.9	13.7
Hanoi	2.5	2.5
8Queens	18.9	15.0
BubbleSort	14.2	16.0
Nsieve	0.0	0.0
Heapsort	28.9	18.9
Dmxpy	14.9	14.9
Gauss	20.5	16.4
TSP	7.1	7.1
Livermore	13.4	9.4
Whetstone	5.3	5.1
Frac	29.0	29.2
Arithmetic Mean	12.7	10.8
Harmonic Mean 1	6.9	6.6
Harmonic Mean 2	3.1	3.1

Table 1. Degradation Over Ideal Schedules

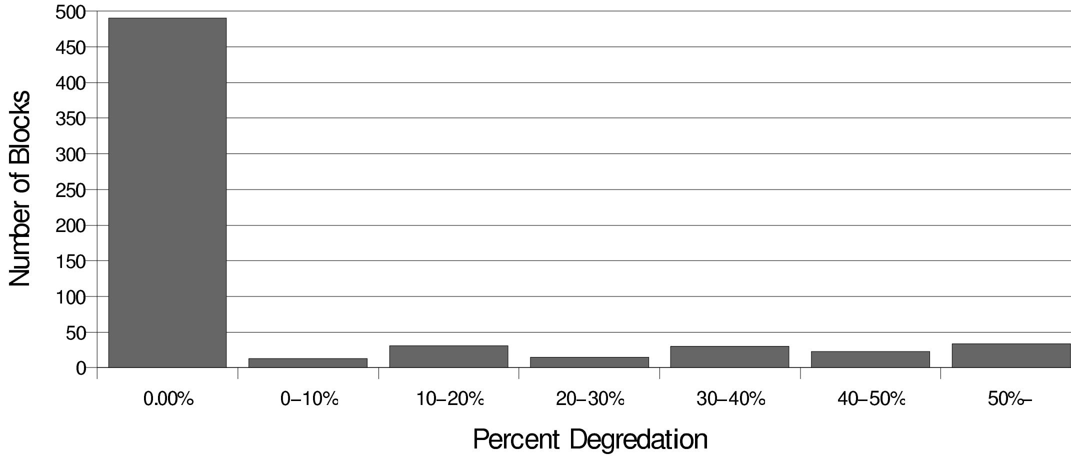

Figure 6. Histogram 2

perform better in 7 cases, while the Embedded-Copy model gave better partitioned execution time results in only 2 cases.

The last three rows of Table 1 show the arithmetic mean of the data, and two different modified harmonic means. Harmonic Mean 1 was computed by ignoring any data point with 0.0% degradation, while Harmonic Mean 2 was computed by converting any 0.0% values to be 1.0%. These different ways of calculating the mean demonstrate that, even for the arithmetic mean, partitioning with our method requires roughly 10% additional execution time. We feel these results are quite promising given the difficult model (general-purpose functional units, 2 or 3 cycle copy latency, and 1 register bank per functional unit) chosen to test our framework.

A different view of the data is provided by Figure 5 and Figure 6. Figure 5 shows that, of the 57 functions compiled, 21 required no additional execution time at all to support partitioning, and the largest degradation of any function was 33.3%. Figure 6 shows that 490 of the 636 basic blocks compiled required no additional execution time, and that, while 2 basic blocks showed 100% degradation (going from 2 to 4 instructions), the largest degradation for any block of at least 10 instructions was 78%.

6. Conclusions and Future Work

This paper describes a framework for generating code for instruction-level parallel (ILP) processors with partitioned register banks. At the heart of this framework is a heuristic-based approach to partitioning the register component graph for each function. The performance of our register graph partitioning is

significantly better than another study of partitioning for a similar model [3]. In addition, the flexibility of our register component graph allows us to easily represent, and generate code for, partitioned register banks with "idiosyncrasies" that often appear in special-purpose architectures such as DSP chips. Other strong features of our register component graph are that the framework is independent of scheduling and register allocation algorithms and that any approach to computing the cut-set of the register component graph can be easily inserted.

Preliminary results suggest that for a partitioned ILP architecture with 4 register partitions and general-purpose functional units, we can achieve execution times roughly 10% greater than that we could expect if we could support an "ideal" ILP machine with a single multi-ported register bank.

With the demands for increased performance, ILP architectures will have to become increasingly more aggressive. To support high levels of ILP partitioned register banks will become increasingly attractive. This work represents an early step in limiting the overhead due to such register bank partitioning.

References

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

[2] 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*.

[3] 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.

[4] A. Capitanio, N. Dutt, and A. Nicolau. Toward register allocation for multiple register file vliw architectures. Technical Report TR94-6, Computer Science Department, University of California at Irvine, Irvine, CA, June 1994.

[5] 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*.

[6] 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*.

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

[8] 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.

[9] 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.

[10] S. Freudenberger. Private communication.

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

[12] C. Lee, C. Park, and M. Kim. An efficient algorithm for graph partitioning using a problem transformation method. In *Computer-Aided Design*, July 1993.

[13] 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.

[14] 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.

[15] 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.

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