

Global Register Partitioning*

Jason Hiser

Department of Computer Science
University of Virginia
Charlottesville VA 22903
jdh&d@virginia.edu

Steve Carr

Philip Sweany
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
{carr,sweany}@mtu.edu

Abstract

Modern computers have taken advantage of the instruction-level parallelism (ILP) available in programs with advances in both architecture and compiler design. Unfortunately, large amounts of ILP hardware and aggressive instruction scheduling techniques put great demands on a machine's register resources. With increasing ILP, it becomes difficult to maintain a single monolithic register bank and a high clock rate. To provide support for large amounts of ILP while retaining a high clock rate, registers can be partitioned among several different register banks. Each bank is directly accessible by only a subset of the functional units with explicit inter-bank copies required to move data between banks. Therefore, a compiler must deal not only with achieving maximal parallelism via aggressive scheduling, but also with data placement to limit inter-bank copies. Our approach to code generation for ILP architectures with partitioned register resources provides flexibility by representing machine dependent features as node and edge weights and by remaining independent of scheduling and register allocation methods. Experimentation with our framework has shown a degradation in execution performance of 10% on average when compared to an unrealizable monolithic-register-bank architecture with the same level of ILP.

1. Introduction

Computer architects continually strive to build computers with ever greater performance. One technique to increase performance in a processor is to provide for multiple low-level instructions to be executing in parallel. Inclusion of such instruction-level parallelism (ILP) in mod-

ern processors has allowed architectures to provide greatly increased performance over that attainable with simple sequential processing.

However, one impediment to the addition of ILP to a processor is that the additional ILP puts a large demand on a computer's register resources. In order for operations to be executed in parallel, their source and destination registers must be accessible in parallel. While this is not major problem for processors with ILP levels of two to four, it soon becomes difficult to support such parallelism in a single register bank. For example, to support 16 three-address operations in parallel would require a 48-ported register bank. Such a multi-ported register bank would make it difficult to maintain a high clock rate, and would severely hamper register access time [2].

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 [12].) 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.

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 comparison to one functional unit per register bank, 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 hardware complexity.

Many previous approaches to this problem have relied on

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

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 global 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. In addition, unlike some of the other approaches to register partitioning, our register component graph easily supports a global partitioning scheme that considers an entire function’s register needs when partitioning.

2. Previous Work

Ellis described the first solution to the problem of generating code for partitioned register banks in his thesis [5]. 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 [2]. They report results for two of the seven loops tested, which, for three functional units, each with a dedicated register bank showed 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.

Desoli [4] describes a partitioning method used at HP Labs. Like BUG, Desoli’s method combines scheduling and

partitioning. It works from a scheduling DAG and uses a heuristically determined initial partitioning to start a gradient descent process. They report good results for up to four clusters, often showing better code than was possible with a single cluster. This seeming contradiction is due to reshaping of the scheduling DAG during the process. As such, it makes the results difficult to compare with other partitioning methods. Desoli reports that increase in compile time was “not excessive”, with compilation for partitioned code requiring 1.5 to 2.0 times, on average, the time necessary for code generation for a single cluster.

Ozer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [10]. UAS maintains two lists, a data-ready list holding operations ready to be scheduled, and a cluster-priority list that represents the potential for copies needed by each item on the data-ready list. They state that UAS is an improvement over BUG since UAS can, using the two lists, perform schedule-time resource checking while partitioning, allowing UAS to manage the partitioning with the knowledge of the bus utilization for copies between partitions. Ozer’s study of entire programs shows, for their best heuristic, an average degradation of roughly 19% on an 8-wide machine grouped as two clusters of four functional units and two busses. We shall compare Ozer, et al.’s work with our own in more detail in Section 5.4

Nystrom and Eichenberger present an algorithm for partitioning for modulo scheduling [9]. 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. We discuss our results in partitioning of software pipelined loops in [6]. In contrast, the method we describe here is not restricted to software pipelined loops and is applicable across an entire function.

Our previous work [7] investigated the applicability of a general k -way partitioning algorithm of Lee, Park and Kim (LPK) [8] to register partitioning. Our experiments in [7] were designed to be as comparable as possible with those of Capitanio et al. [2]. As such, we chose a 3-wide LIW with one register bank per functional unit, the same model used in [2]. Our results with the LPK algorithm, for a small set of programs that is a superset of the source code used in [2], showed performance degradation of roughly 30%. While this seems to be an improvement over what Capitanio et al. found, we were far from satisfied. In addition, using the LPK algorithm led to extremely long compile times. These long compile times, in combination with what we viewed as poor performance results, motivated the work presented in this paper.

3. Register Assignment with Partitioned Register Banks

On architectures with high degrees of ILP, it is often inconvenient or impossible to have a single register bank associated with all functional units. 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). 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*¹, whose nodes represent register operands (symbolic registers) and whose arc weights indicate the “affinity” that two registers have to be placed in the same register bank.

We could 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

¹The register component graph was originally described in [7].

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.

Consider a “typical” ILP architecture with multiple functional units, partitioned among clusters, where each cluster has its own register bank(s). In this case, we would want registers used within a single operation (and thus computed on a specific functional unit) to be placed in the same cluster if possible. To do this, we would add arcs, annotated with some appropriate positive weight, from the destination register to each source register. If for some reason we wanted to indicate that two registers should be in different banks, we could add a negative weighted edge between the two. This might well be the case for two register that represent the definitions of different operations scheduled within the same instruction in the ideal instruction schedule.

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 $A = B \text{ op } C$ 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 [3] 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 two functional units, each with its own register bank. For simplicity we assume unit latency for all operations. We shall add two types of edges to our graph for this hypothetical architecture:

- For each operation, we shall add an arc between the destination register (defined by the operation) and each source register.²

²We could also add arcs between multiple source registers but choose not to since our code generation scheme assumes that the location of the destination register will define the functional unit used.

- For each pair of registers R_i and R_j such that R_i and R_j are the destination registers of two operations scheduled in the same instruction of the ideal schedule, we add an arc between R_i and R_j .

Consider the following high-level language statement:

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

Hypothetical intermediate code for this statement appears in Figure 1. We have numbered the operations simply for identification when we use this example again later in this paper.

The DDD for this single-block “program” is shown in Figure 2. Each of the nodes is decorated with a number that is the number of the intermediate statement (of Figure 1) represented by that DDD node. In addition, each node includes two integers, separated by a comma. They represent the earliest and latest times that that DDD node can be scheduled without changing the longest path in the DDD. You can ignore those numbers for now. We will look at them again, when we revisit Figure 1 in the context of our greedy partitioning heuristic in Section 4.

One possible optimal schedule for the DDD of Figure 1, assuming a single multi-ported register bank is shown in Figure 3. The corresponding register component graph appears in Figure 4. For simplicity of this discussion at this stage, please ignore the edge weights of Figure 4. We will come back to this figure and consider the edge weights when we discuss our greedy heuristic approach to partitioning (in Section 4.) For now, assume that we use an “oracle” to assign edge weights and partition nodes. An optimal schedule, such as the one of Figure 3 requires seven cycles to complete. One potential partitioning of the above graph (given 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 5, which takes a total of nine cycles, an increase of two cycles over the ideal case. Notice that two values (r2 and r6) required copying to generate the schedule of Figure 5.

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 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.

```
( 1) load  r1,xvel
( 2) load  r2,t
( 3) load  r3,xaccel
( 4) load  r4,xpos
( 5) mult  r5,r1,r2
( 6) add   r6,r4,r5
( 7) mult  r7,r3,r2
( 8) div   r8,r2,2.0
( 9) mult  r9,r7,r8
(10) add   r10,r6,r9
(11) store xpos,r10
```

Figure 1. Low-level operations

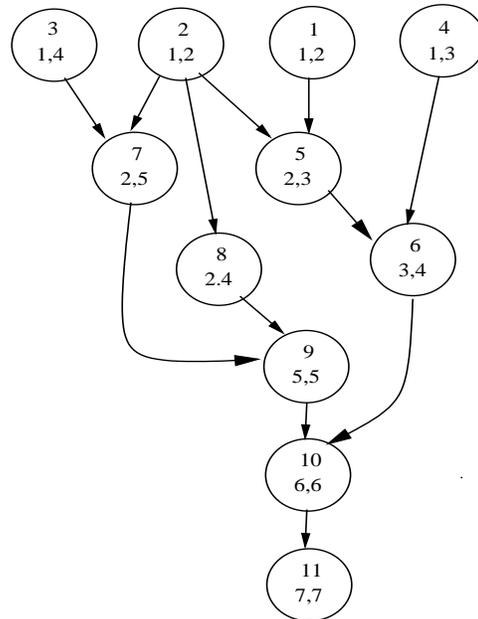


Figure 2. DDD

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 xpos, r10	

Figure 3. Optimal Schedule for Example Code

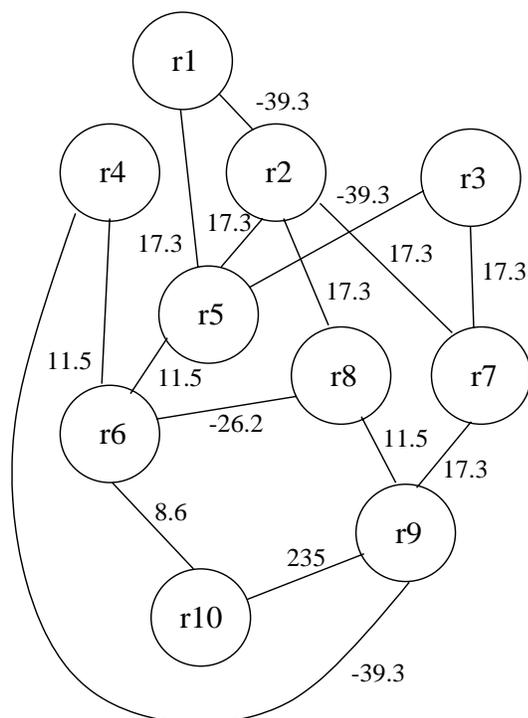


Figure 4. Register Component Graph

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 xpos, r10

Figure 5. Post-Partitioning Schedule

Whichever partition has the largest computed benefit, corresponding to the lowest computed cost, is the partition to which the node is allocated.

Before discussing the heuristic weighting process, we 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 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 basic block, DDD operation or ideal schedule:

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

DDD Density : 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 : 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. Note that the two comma-separated integers per DDD node in Figure 2 represent these very early and late scheduling times for each node. 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 one 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 j is live-in to this DDD, then $X = O$; otherwise Let X be the operation that defines j

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

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

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:

Let Flex = minimum of Flexibility(O_1) and Flexibility(O_2)

if Flex is 1, $wt = -500$ otherwise $wt = 0$

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

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. Again, we wish to add the magnitude of the edge weights to both i and j . This time we use the absolute value of the edge weight.

Once we have computed the weights for the RCG nodes and edges, we choose a partition for each RCG node as described in Figure 6. Notice that the algorithm described in Figure 6 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.

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 -= NumberOfRegsAssignedToRB1.7
      If ThisBenefit > BestBenefit
        BestBenefit = ThisBenefit
        BestBank = RB
    return BestBank
}

```

Figure 6. Choosing a Partition

4.1. Greedy Partitioning for Our Example

The previous discussion of our code generation using the greedy partitioning heuristic outlined above can be summarized by the following steps:

1. Build a Data-Dependence DAGs (DDD) for the scheduling contexts to be considered. (This could be a basic block at a time if local scheduling is used, or larger DDDs if global scheduling and/or software pipelining are considered.) Since the partitioning algorithm described here works at a function level, one would need to build all the relevant DDDs for an entire function before moving on to the next step.
2. Compute ideal schedule(s) for the DDD(s) of step 1.
3. Using the ideal schedule for each DDD of the function, construct a Register Component Graph (RCG) as described in Section 3, adding edge and node weights as described above for the greedy algorithm.
4. Sort the RCG nodes by weight
5. Using the algorithm of Figure 6, partition the RCG nodes (symbolic registers) to the partitions available on the target architecture.
6. Using the register partitioning of step 5, assign each node for each DDD of the function to a functional unit (or at least a cluster) based upon the register partition assigned to the destination register for that operand. (If the node represents a store and thus has no destination

register, use the source register to assign the node to a cluster.)

7. Once each DDD's node have been assigned to a cluster, use the *used* and *defined* values of the DDD nodes to determine where registers need to be copied from one partition to another. Add DDD nodes for each such register copy.
8. Schedule all of the function's DDDs again, with the new copy nodes included.

To demonstrate how the greedy heuristic is used in the code generation scheme described above, we shall return to our example "program" of:

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

Again, refer to the intermediate code description found in Figure 1, the ideal schedule of Figure 3 and the RCG of Figure 4. Again, we shall assume a simple partitioned architecture with two functional units, each with its own register bank, nd unit latency for all operations. This time we will pay attention to the edge weights in the RCG of Figure 4. These weights were computed, using the greedy heuristic described above.

Given the RCG of Figure 4 the nodes weights for the RCG can be computed by summing the absolute values for each edge incident to the node. (Of course, this would have been done while we were building the RCG, but to save space and reduce clutter in Figure 4, we do not include node weights there.) When we do that, and then sort the nodes by decreasing order of node weight, we get the sequence

9, 10, 7, 2, 5, 6, 1, 3, 8, 4

That is the order in which the RCG nodes (registers) are assigned to partitions. For our example, following the algorithm of Figure 6 we get the following partition assignment:

$$P_1 : \{r1, r3, r7, r8, r9, r10\}$$

$$P_2 : \{r2, r4, r5, r6\}$$

Having determined a partitioning, we can now assign a function unit for each node of the original DDD, Figure 2. In our compiler we use a "definition-compute" rule that a DDD node is assigned to the functional unit matching the partition of the destination register. Store instructions are mapped to the functional unit of the register being stored. Thus, given our partitioning, the functional unit assignment for DDD nodes is:

$$F_1 : \{N1, N3, N7, N8, N9, N10, N11\}$$

$$F_2 : \{N2, N4, N5, N6\}$$

Now, having assigned both registers and DDD nodes to a "cluster", it is necessary to add nodes to the DDD to represent register values copied from one partition to the other. Three such nodes are necessary, labeled 101, 102, and 103 in the new post-partitioning DDD shown in Figure 7. They are to copy register one from partition one to partition two (node 101), to copy register two from partition two to partition one (node 102), and to copy register six from partition two to partition one (node 103). Notice that Figure 7, in addition to including the three new nodes, decorates each DDD node with the partition in which it is to be computed. With our updated DDD in place, we now re-compute the instruction schedule, yielding the schedule of Figure 8. Notice that it requires eight instructions to schedule the post-partition DDD, a degradation of one instruction over the ideal schedule of Figure 3 but an improvement of one instruction over the schedule (with copies) of Figure 5. So, even though the schedule of Figure 8 requires three copies compared of two for Figure 5, we are still able to get a better schedule, showing that minimizing copies is not always the best solution when partitioning registers.

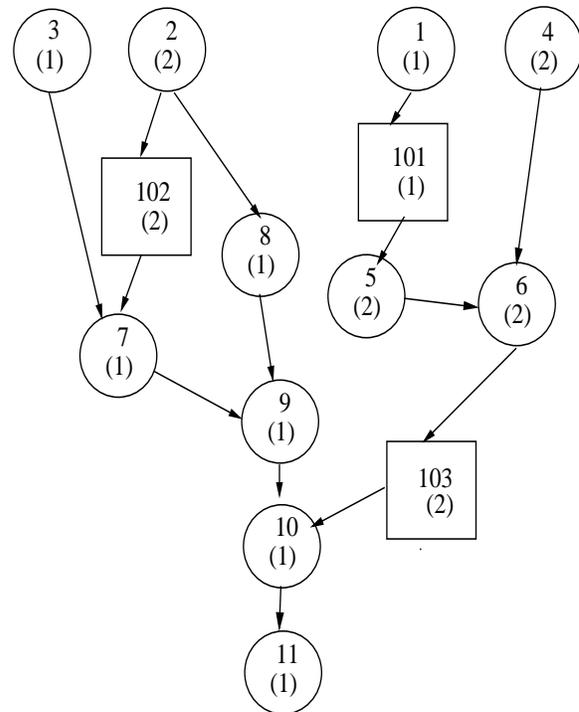


Figure 7. DDD after Partitioning

4.2. Choosing Weights for Our Greedy Heuristic

At the present time both the program characteristics that we use in our heuristic and the weights assigned to each

load r1, xvel	load r2, t
move r11, r1	load r4, xpos
load r3, xaccel	move r22, r2
mult r7, r3, r22	mult r5, r11, r2
div r8, r22, 2.0	add r6, r4, r5
mult r9, r7, r8	move r66, r6
add r10, r66, r9	
store xpos, r10	

Figure 8. Schedule after Greedy Partitioning

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 [11]. 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 Table 1. Section 5.3 provides 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 multiported 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 four 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 four 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, four 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 one 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 four "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 and loads take two cycles, integer multiplies take five cycles, integer divides take 12 cycles, all other integer operations take one cycle, floating-point copies take three cycles, all other floating operations take two cycles and stores take four cycles.

5.2 Benchmarks

We tested our greedy partitioning method on the 20 C programs listed in Table 1. These 20 programs, containing 57 functions and 2407 lines of (uncommented) C code. We chose programs to include both integer (most of the benchmarks) and floating point (Dmxdpy, Gauss, Livermore, Whetstone, Frac) computation. In consideration of the fact that our greedy partitioning method works at a function level, we wanted to include benchmarks with a range of function sizes. While the average number of lines of C code in the 57 functions is 42, the range was from two lines of code (in one function of Hanoi) to 238 (in one function of Livermore.)

5.3 Results

Table 2 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

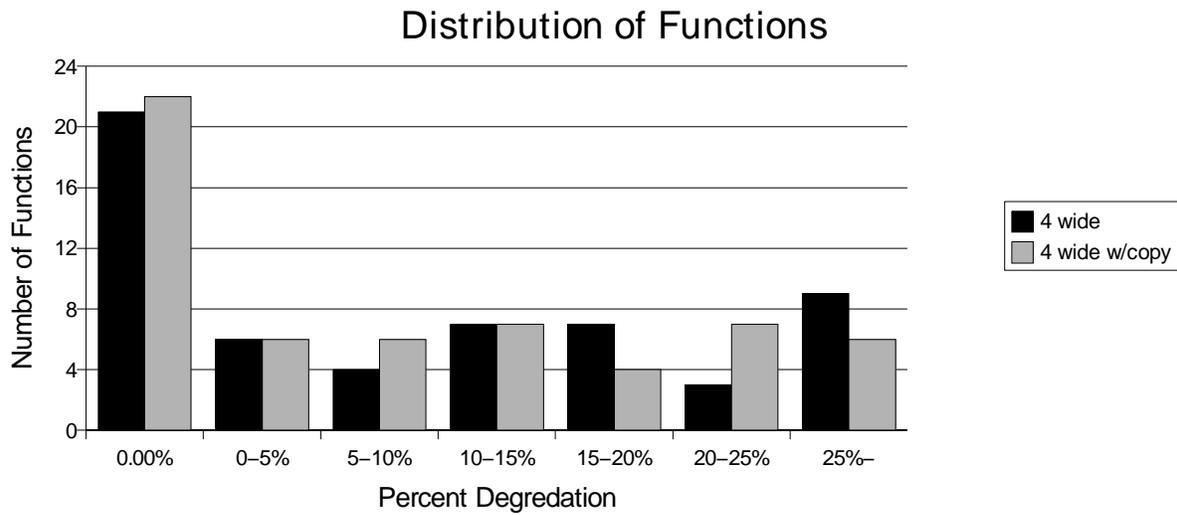


Figure 9. Histogram

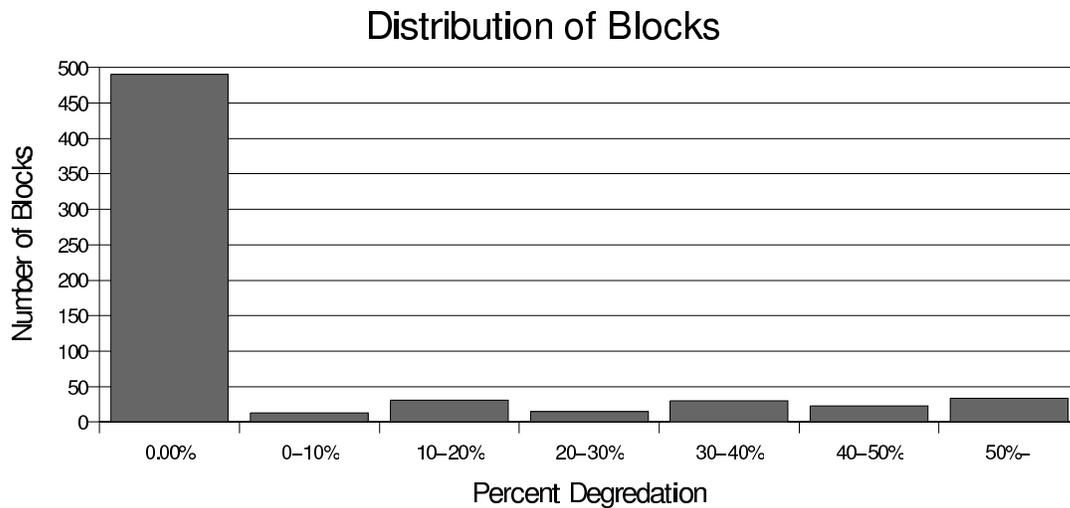


Figure 10. Histogram 2

Benchmark	Description
<i>KillCache</i>	inhibits spatial and temporal locality
<i>MergeSort</i>	performs a merge sort
<i>Dice</i>	simulates rolling a pair of dice
<i>Pascal</i>	Pascal's triangle
<i>IsPrime</i>	prints primes less than 1000
<i>ListTest</i>	tests basic list operations
<i>MatrixMult</i>	matrix-matrix multiplication
<i>Malloc</i>	dynamic array manipulation
<i>BinarySearch</i>	performs a binary search
<i>Hanoi</i>	solve the Towers of Hanoi problem
<i>8Queens</i>	all solutions to 8 Queens problem
<i>BubbleSort</i>	performs a bubble sort
<i>Nsieve</i>	the Sieve of Eratosthenes
<i>Heapsort</i>	perform a heapsort
<i>Dmxy</i>	matrix-vector multiplication
<i>Gauss</i>	solves the Gaussian elimination problem
<i>TSP</i>	solves the Traveling salesman problem
<i>Livermore</i>	the Livermore loops
<i>Whetstone</i>	the whetstone benchmark
<i>Frac</i>	floating point to (approx) rational

Table 1. Benchmarks

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
Dmxy	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

Table 2. Degradation Over Ideal Schedules

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 perform better in seven cases, while the Embedded-Copy model gave better partitioned execution time results in only two cases. The inclusion of small benchmarks (e.g., 8queens and hanoi) shows that even for simple code, the potential for significant degradation due to partitioning is possible.

There seemed no obvious relationship between function size and performance of our greedy heuristic. Some small functions (12 lines for IsPrime, and the many small functions of ListTest) showed no degradation over the ideal schedules at all. Other small programs, such as Mergesort, whose largest function was 38 lines long, and malloc, whose one function is 19 lines long, showed significant degradation. On the other end of the function size scale, the 124-line function of Nsieve showed no degradation whatever, while the 238-line function of Livermore showed a significant degradation. We also noted no discernible pattern, with respect to performance of our greedy algorithm with respect to integer or floating point computation. So, it appears that our benchmarks did, in fact, test enough different kinds of C programs to be relevant, and, in fact, our greedy heuristic seems to work reasonably well for various types and sizes of C functions.

The last row of Table 2, the arithmetic mean of the 20 degradations listed above, shows that on average partitioning with our method requires roughly 10% additional execution time. We consider our results excellent given the difficult model (general-purpose functional units, two or three cycle copy latency, and one register bank per functional unit) chosen to test our framework.

A different view of the data is provided by Figure 9 and Figure 10. Figure 9 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 10 shows that 490 of the 636 basic blocks compiled required no additional execution time, and that, while two basic blocks showed 100% degradation (going from two to four instructions), the largest degradation for any block of at least 10 instructions was 78%.

5.4 Comparison to Ozer et al.

Taking into consideration both the results presented here, and Ozer et al.'s results, one might reasonably expect 10-20% degradation in execution time when partitioning by two or four clusters. However, there are enough differ-

ences in the experimental techniques to make more significant evaluation of the relative merits of UAS vs. our partitioning method difficult at best. Exact comparison of our results with Ozer et al. is complicated by the different experimental parameters, including:

- Ozer et al. have two clusters in an 8-wide machine while we looked at four clusters in a 4-wide machine. It is 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.

6. Conclusions

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.

Our results suggest that for, a partitioned ILP architecture with four 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. Taking into consideration both the results presented here, and Ozer et al.’s results, one might reasonably expect 10-20% degradation in execution time when partitioning by two or four clusters.

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.

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 important 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] A. Capitano, 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.
- [3] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN ’82 Symposium on Compiler Construction*, pages 98–105, Boston, MA, June 1982.
- [4] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. HP Labs Technical Report HPL-98-13, HP Labs, Jan. 1998.
- [5] J. R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1984.
- [6] J. Hiser, S. Carr, and P. Sweany. Register Assignment for Software Pipelining with Partitioned Register Banks. In *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 211–217, May 2000.
- [7] S. Jang, S. Carr, P. Sweany, and D. Kuras. A code generation framework for VLIW architectures with partitioned register files. In *Proceedings of the Third International Conference on Massively Parallel Computing Systems (MPCS)*, pages 61–69, April 1998.
- [8] C. Lee, C. Park, and M. Kim. An efficient algorithm for graph partitioning using a problem transformation method. *Computer-Aided Design*, pages 611–619, July 1993.
- [9] 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.
- [10] 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 31st International Symposium on Microarchitecture (MICRO-31)*, pages 308–316, Dallas, TX, December 1998.
- [11] P. H. Sweany and S. J. Beaty. Overview of the Rocket re-targetable C compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.
- [12] Texas Instruments. *Details on Signal Processing*, issue 47 edition, March 1997.