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

Jason Hiser[†] Steve Carr[‡] Philip Sweany[‡] Steven J. Beaty[§]

Abstract

Many techniques for increasing the amount of instruction-level parallelism (ILP) put increased pressure on the registers inside a CPU. These techniques allow for more operations to occur simultaneously at the cost of requiring more registers to hold the operands and results of those operations, and importantly, more ports on the register banks to allow for concurrent access to the data. One approach of ameliorating the number of ports on a register bank (the cost of ports in gates varies as N^2 where N is the number of ports, and adding ports increases access time) is to have multiple register banks with fewer ports, each attached to a subset of the available functional units. This reduces the number of ports needed on a per-bank basis, but can slow operations if a necessary value is not in an attached register bank as copy operations must be inserted. Therefore, there is a circular dependence between assigning operations to functional units and assigning values to register banks. We describe an approach that produces good code by separating partitioning from scheduling and register assignment. Our method is independent of both the scheduling technique and register assignment method used.

1. Introduction

With aggressive instruction scheduling techniques and significant increases in instruction-level parallelism (ILP), modern computer architectures have seen impressive performance increases. Unfortunately, these factors put large demands on a machine's register resources by requiring many data accesses per cycle. The number of ports required for such register banks severely hampers access time [3, 7] and cost. Partitioned register banks are one mechanism for providing high degrees of ILP while maintaining a high clock rate and a reasonable cost. Texas Instruments, for example, already produces several DSP chips that have partitioned register banks to support high ILP [14]. Unfortunately, partitioned register banks may inhibit ILP as some mechanism is required to allow functional units access to "non-local" values, i.e.: values contained in a different partition. One approach to provide non-local register values is to add extra instructions to move the data to the register bank of the target functional unit. Another approach is to provide a communication network to allow access to non-local values. In either case, a compiler must deal not only with achieving maximal parallelism via aggressive scheduling, but also with data placement among a set of register banks.

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

Some of the work in the areas of code generation for ILP architectures with partitioned register banks has dealt with global scheduling techniques, but most consider only local scheduling. It is 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 [9, 12, 1]. Therefore, while our register partitioning method is applicable to entire programs, we will concentrate on software pipelined loops for the work in this paper. There are at least two 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

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

[†] Department of Computer Science, University of Virginia, Charlottesville VA 22903, *jdh8d@virginia.edu*.

[‡] Department of Computer Science, Michigan Technological University, Houghton MI 49931-1295, {*carr,sweany*}*@mtu.edu*.

[§] 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*

generation. Two, and perhaps more importantly from the perspective of this paper, is that software pipelining leads both to the greatest extraction of parallelism within a program and also places significant requirements on the register resources. Considering the combination of maximal parallelism and large register requirements, we 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 compared to an ideal ILP model.) This comes at the expense of adding to the difficulty of assigning registers within each partition as additional pressure is put on them by 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 a program segment. Various algorithms have been proposed on how to partition the nodes of this "operation" DAG, to generate an efficient assignment of functional units. In contrast, our approach allocates registers to partitioned register banks based on the partitioning of an undirected graph that interconnects those program data values that appear in the same operation. This graph allows us to support retargetability by abstracting machine-dependent details into node and edge weights. We call this technique *register component graph* (RCG) partitioning, because the nodes of the graph represent virtual registers appearing in the program's intermediate code.

The experimental work included here studies software pipelined loops. A good summary of software pipelining can be found elsewhere [1]. In this paper, we begin in Section 2 with a look at others' attempts to generate code for ILP architectures with partitioned register banks. Section 3 describes our general code-generation framework for partitioned register banks while Section 4 outlines the greedy heuristic we use in register partitioning. Finally, Section 5 describes our experimental evaluation of our partitioning scheme and Section 6 summarizes and suggests areas for future work.

2. Other Partitioning Work

Ellis [6] describes an early solution to the problem of generating code for partitioned register banks in his dissertation. His method called BUG (bottom-up greedy), is applied to a scheduling context at a time (*e.g.,* a trace) and is intimately intertwined with instruction scheduling, utilizing machine-dependent details within the partitioning algorithm. Our method abstracts away machine-dependent details from partitioning with edge and node weights, a feature extremely important in the context of a retargetable compiler.

Capitanio et al. [3] present a code-generation technique for limited-connectivity VLIWs. Their results (for two of the seven software-pipelined loops they tested) for three functional units, each with a dedicated register bank, show degradation in performance of 57% and 69% over code obtained for 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 [8]. They report results that show significantly less degradation than the partitioning scheme of Capitanio et al., however their interconnection network represents a different architectural paradigm making comparisons less meaningful. Indeed it is surmised 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.

Ozer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [11]. They state that UAS is an improvement over BUG since UAS can perform schedule-time resource checking while partitioning allowing UAS to manage the partitioning with the knowledge of the bus utilization for copies between partitions. 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.

Nystrom and Eichenberger [10] present an algorithm that first performs partitioning with heuristics that consider later modulo scheduling. 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. Our greedy partitioning method is not limited to just software pipelined loops but is easily applicable to entire programs, since we could easily use both non-loop and loop code to build our register component graph. Nystrom and Eichenberger restrict themselves to loops and their method includes facets that makes it difficult to apply to larger entities.

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 "bank" of equivalent registers [4, 5, 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 as it 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. As 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 (12 for reading and 6 for writing, although reading and writing may be separated in the execution cycle) 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 [14]. Operations performed in any functional unit require registers with the proper associated register bank, and copying a value from one register bank to another is expensive. The problem for the compiler, then, is to allocate registers to banks to reduce the number of copies, while retaining a high degree of parallelism.

Our approach to this problem is as follows.

1. Build intermediate code with symbolic registers, assuming a single infinite register bank.
2. Build data dependence graphs (DDGs) and perform software pipelining still assuming an infinite register bank.
3. Partition the registers to register banks (and thus preferred functional unit(s)) by the "Component" method outlined below.
4. Re-build DDGs 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" Chaitin/Briggs graph coloring register assignment for each register bank.

3.1. Partitioning Registers by Components

Our method builds 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 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 find it useful to build the graph from what we call an "ideal" instruction schedule which by our definition, uses all the characteristics of the actual architecture, except that it assumes that all registers are in a single multi-ported register bank. 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. Once the graph is built, we find each connected component of the graph as each represents registers that can be allocated to a single partition. In general, we need to split components to fit the number of register partitions available on a particular architecture, computing a cut-set of the graph that has a low copying cost and high degree of parallelism among components.

A 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, such as when `A = B op C` where each of `A`, `B` and `C` must be in separate register banks. This situation could be handled abstractly by weighting the edges connecting these values with negative values 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 [4] 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 start with the following software pipelined inner loop from matrix multiply, assuming a single cycle latency for add, and a two-cycle pipeline for multiply. In addition we'll assume that all loads are cache hits and also use a two-cycle pipeline. An ideal schedule for this inner loop is shown in Figure 1. The corresponding register component graph appears in Figure 2. The loop kernel requires 4 cycles to complete.[1]

[1] we assume the availability of a load command that allows for either pre-increment or post-increment of the address operand.

load r1, a[i,k]	add r4,r4,r5
load r2, b[i,k++]	mult r5,r7,r6
load r7, a[i,k]	add r4,r4,r3
load r6, b[i,k++]	mult r3,r1,r2

Figure 1. Ideal Schedule

One potential partitioning of the graph in Figure 2 (given the appropriate edge and node weights) is the following:

$$P_1 : \{r1, r2, r3, r4, i1, k1\}$$

$$P_2 : \{r5, r6, r7, i2, k2\}$$

In the above partition we assume that both i (a read-only variable) and k (an induction variable) are cloned to have one copy in each of the partitions. This assumption allows us to produce the partitioned code of Figure 3, which requires 5 cycles for the loop kernel, a degradation of 20% over the ideal schedule of Figure 1. The additional instruction is necessary to copy the value of r5 from partition 2 to partition 1, where it is known as r55.

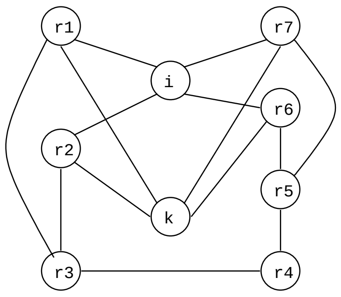

Figure 2. Register Component Graph

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 computed by 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.

load r1, a[i1,++k1]	copy r55, r5
load r2, b[i1,k1++]	mult r5,r7,r6
add r4,r4,r55	nop
add r4,r4,r3	load r7, a[i2,++k2]
mult r3,r1,r2	load r6, b[i2,k2++]

Figure 3. Partitioned Schedule

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), each node representing a basic block. Each basic block contains a data dependence graph (DDG) 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.

In assigning heuristic weights to the nodes and edges of the RCG, we consider several characteristics for each operation of the DDG or ideal schedule:

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

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

Flexibility : defined to be the "slack" available for a DDG 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 DDG node's predecessors) and the latest time that the DDG node could be scheduled without requiring a lengthening of the ideal schedule. This is used both to identify nodes on a DDG critical path (such nodes will have a 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 $wt = 0$

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

  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, as they are 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 $wt = 0$

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

  That is: 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 in Figure 4 tries placing each RCG node in question in all possible register banks and computes the benefit 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 in an 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. Attempts to fine-tune both the characteristics and the weights using any of a number of optimization methods could be made. 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.

5. Experimental Evaluation

To evaluate our method for register partitioning in the context of our greedy algorithm, we software pipelined 211 loops extracted from SPEC 95 suite. To partition registers to the clusters, we used the greedy heuristic described in Section 4, as implemented in the Rocket compiler [13]. We evaluated the greedy algorithm with two slightly different models of 16-wide partitioned ILP processors, as described in Section 5.1. Section 5.2 provides analysis of our findings.

```
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

5.1. Machine Models

The machine model we evaluated consists of 16 general-purpose functional units grouped in N clusters of differing sizes, each cluster containing a 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: they do not force the assignment of particular operations to certain functional units. 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.

The Embedded Model in which explicit copies are scheduled within the functional units. To copy a register value from cluster A to cluster B 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.

The 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, as are 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.

Operation	Cycles
integer copies	2
float copies	3
loads	2
stores	4
integer mult	5
integer divide	12
other integer	1
other float	2

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, those are the latencies used in the embedded model (where they are somewhat more realistic) and we decided to use those latencies as well in our copy-unit models for consistency.

5.2. Results

When considering the efficiency of any register partitioning algorithm an important consideration is how much parallelism is 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, as the fewer operations would presumably require fewer non-local register values than a pipeline with more parallelism. 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; to reduce confusion, we'll use IPC as well.

Table 1 shows the average IPC for the loops pipelined for our 16-wide ILP architecture. The "Ideal" row 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 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. We include both arithmetic and harmonic means as 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. We see that both the embedded copy model and the copy-unit model created a degradation of about 25% for the 4-cluster (of 4 functional units) model. It is not too surprising 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. The 8-cluster machine showed just the opposite effect, as we might imagine. Having 3 ports per partition for the two functional units included in the cluster, leads 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.

Table 2 can also be used to gain some appreciation of the level of schedule degradation expected in a relatively "balanced" clustering model such as the 4 clusters of 4-wide general purpose functional units. The roughly 20-25% degradation is in fact greater than what has been previously shown possible for entire programs [11]. This is expected, as software pipelined loops will generally contain more par-

	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. Average 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

allelism than non-loop code; the additional degradation we have found still seems reasonable.

6. Conclusions and Future Work

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

Our method and greedy partitioning algorithm are applicable to both whole programs and software pipelined loops. Our results here show that on the order of a 25% degradation for software pipelined loops over a 16-wide architecture with one register bank can be expected.

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.

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] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3), September 1995.

[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] 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*.

[5] 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*.

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

[7] 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.

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

[9] 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*.

[10] 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.

[11] 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.

[12] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, December 1994.

[13] 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.

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