# A Code Generation Framework for VLIW Architectures with Partitioned Register Banks *

*Saurabh Jang*
Motorola Corporation
1301 East Algonquin Road
Schaumburg, IL 60196
*jang@eis.comm.mot.com*

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

## Abstract

*Modern computers are taking increasing advantage of the instruction-level parallelism (ILP) available in programs with advances in both machine and compiler design. Unfortunately, large amounts of ILP hardware and aggressive instruction scheduling techniques put large demands on a machine's register resources. With large amounts of ILP, it becomes difficult to maintain a single monolithic register bank and a high clock rate. The number of ports required for such a register bank severely hampers access time [2, 8]. 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.*

*This paper describes our approach to partitioning values among available register banks. Our method provides flexibility for our retargetable compiler by representing machine dependent features as node and edge weights and by remaining independent of scheduling and register allocation methods. Preliminary experimentation with our framework has shown a degradation in execution performance of 33% on average when compared to an unrealizable monolithic-register-bank VLIW architecture with the same level of ILP. This compares very favorably with other approaches to the same problem.*

*This research was supported by a grant from Texas Instruments.

## 1. Introduction

Many of today's computer applications require computation power that cannot be easily obtained using computer architectures that provide little parallelism and simple instruction timing. Because of this need for power, several varieties of parallel architectures continue to be investigated.

One alternative to simple processors is building computers that provide greater instruction-level parallelism, increasing computation during each machine cycle. Such instruction-level parallel (ILP) architectures allow parallel computation of the lowest level machine operations such as memory loads and stores, integer additions, and floating-point multiplications within a single instruction sequence. To effectively use parallel hardware, compilers must identify the appropriate level of parallelism. For ILP architectures, effective hardware usage requires that the single instruction stream be ordered such that, whenever possible, multiple low-level operations can be in execution simultaneously. This ordering of machine operations to effectively use an ILP architecture's increased parallelism is typically called *instruction scheduling*.

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 [2, 8]. 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 [14].) 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 VLIW compiler must decide, for each operation, not only where the operation fits in an instruction schedule, but also in which register partition(s) the operands of that operation will reside, which, in turn, will determine which functional unit(s) can efficiently perform the operation. Obtaining an efficient assignment of operations to functional units is not an easy task as two opposing goals must be balanced. Achieving near-peak performance requires spreading the computation over the functional units equally, thereby maximizing their utilization. At the same time the compiler must minimize communication costs resulting from copy operations introduced by distributing computation.

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 new 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 [6]. 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 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. or our preliminary results. However, their interconnection network actually represents a different architectural paradigm making comparisons less meaningful. Indeed it is surmised [9] 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. [3] 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 al-

gorithm 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.[7] 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.

## 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 [4, 5, 1]. 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 [2]. 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 [14]. 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 a new 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 data dependence DAGs (DDD)s 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 [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 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**

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. If we run the partitioning algorithm of Lee, *et al.* [12], on the above graph for $k = 2$, one potential partitioning (given the appropriate edge and node weights) is the following:

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

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

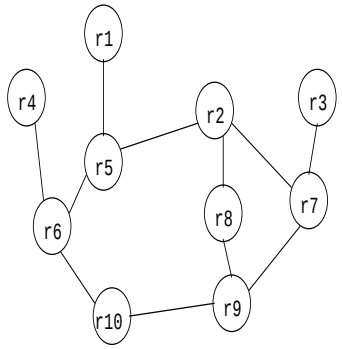


**Figure 2. Code and Corresponding Register Graph**

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

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**

### 3.3. Cut-Set Heuristics

Given the register component graph described above, in general we need to find a minimal cost cut-set. Since finding an optimal cut-set is NP-complete [10], we need to focus on heuristic solutions. To initially evaluate our framework, we chose a graph partitioning algorithm by Lee, Park and Kim [12] to find the components in our graph. This algorithm is mentioned by Capitanio et al. [2] and is not specific to our register partitioning problem.

The Lee, Park, Kim (LPK) algorithm finds a heuristic solution to the uniform $k$-way partitioning problem. The

uniform $k$-way partitioning is as follows: Given a graph $G$ with costs on its edges and sizes on its nodes, partition the nodes of $G$ into $k$ subsets of equal size such that the total cost of the edges cut is minimized. Since equality cannot be achieved in general, the technique attempts to find "balanced" partitions. A partition is called "balanced" when the total sum of size differences of all pairs of subsets is a minimum.

A key advantage of the LPK algorithm is that they transform the k-way partitioning problem on the original graph to the problem of finding a maximum k-cut-set of a modified graph derived from the original graph. The maximum k-cut-set problem requires one to come up with a k-way partitioning of the nodes of a graph that maximizes the cut-set of the partitioning.

#### 3.3.1. Transforming the Register Component Graph

From the original register component graph $G$, we come up with a modified graph $G^*$ by using the following scheme:

Given the nodes $\nu_i$ and $\nu_r$ of $G$,

- if there is an edge or multiple edges between them, then insert a single edge between them in $G^*$ with the cost

  $Size(\nu_i) \cdot Size(\nu_r) \cdot R - \sum W(\nu_i, \nu_r)$

  where $\sum W(\nu_i, \nu_r)$ is the sum of the edges weights for all edges connecting $\nu_i$ and $\nu_r$.

- if there is no edge connecting them, then insert an edge between them in $G^*$ with the cost

  $Size(\nu_i) \cdot Size(\nu_r) \cdot R$

  The factor $R$ is chosen such that $R \geq \sum_{i=1}^{E} W(e_j)$.

Clearly, $G^*$ is a completely-connected graph with the same number of nodes as $G$. Note that the transformed graph is no longer a multi-graph since now there is only one edge connecting each pair of nodes. We chose the factor $R$ to be equal to $\sum_{i=1}^{E} W(e_j)$ for this work. Intuitively the weights derived by the above formulae make sense, because nodes that are unconnected in the original graph make a greater contribution to the cut-set costs of the modified graph if they are placed in different partitions. Thus, the bias will be towards keeping unconnected nodes in the original graph in separate partitions, lowering the cut-set costs (and hence communication costs) for the register component graph.

#### 3.3.2. The Partitioning Algorithm

Before describing the steps of the algorithm we need to introduce the concept of *gain* of a node. Given a partitioning of $n$ graph nodes into $k$ subsets ($P_1$, $P_2$, .., $P_k$), the gain of a node with respect to a partition(subset) is defined as the amount by which the cut-set cost of the partitioning would increase if the node were to be moved from its current partition to that partition. More formally, given the edge-cost matrix $C = (c_{ij})$ where $c_{ij}$ is the cost of the edge between the nodes $\nu_i$ and $\nu_j$ in the transformed graph, then the gain of each node $\nu_a$ in $P_i$ over $P_j$ is defined as:

$$g(\nu_a, P_j) = \sum_{\nu_i \in P_i} c_{ai} - \sum_{\nu_j \in P_j} c_{aj}, \forall \nu_a \in P_i$$

The algorithm starts with all of the nodes being in an initial subset and all other $(k-1)$ subsets being empty. The algorithm consists of a series of passes. In each pass we start with the previous partitioning and try to improve it by iteratively choosing a single node (called the candidate node) to be moved to a partition different from its existing one. The candidate element is the one with the highest gain that has not yet been moved in this pass. Once a node is moved in a given pass, it is not allowed to move again. When all the $n$ nodes have been move candidates, the pass is over. Out of the $n$ partitions produced during the pass, the one with the highest cut-set cost is chosen as the starting partition for the next pass. Passes are performed until little or no improvement in cut-set cost can be obtained.

After moving a node, the gains of all elements that have not yet been moved during the pass must be re-calculated according to the following formulae, which are a result of a straightforward application of the gain definition provided above:

$$g(\nu_a, P_j) = g(\nu_a, P_j) - 2c_{ar}, \forall \nu_a \in P_i$$

$$g(\nu_b, P_i) = g(\nu_b, P_i) + 2c_{br}, \forall \nu_b \in P_j$$

$$g(\nu_c, P_i) = g(\nu_c, P_i) + c_{cr}, \forall \nu_c \in P_l, 1 \leq l (\neq i, j) \leq k$$

$$g(\nu_c, P_j) = g(\nu_c, P_j) - c_{cr}, \forall \nu_c \in P_l, 1 \leq l (\neq i, j) \leq k$$

The complexity of each pass of the algorithm is $O(kN^2)$. The number of passes is usually small, and ranged from 1 to 5 for our benchmarks.

#### 3.3.3. Node and Edge Weights

Using the LPK algorithm, then, we can heuristically calculate a cut-set for the register component graph and, thus, allocate registers to specific register banks. But the LPK algorithm requires that both the edges and nodes of the register component graph be weighted. In building the register component graph described here, each edge's weight is determined by two factors:

- Using profile information, we add a positive frequency value, $F$, to each edge, where $F$ is the execution frequency of the operation that is responsible for this edge being included in the graph.

| Program | Instructions | Cycles |
|---|---|---|
| 8q | 111 | 140 |
| bubble | 100 | 117 |
| gauss | 106 | 133 |
| livermore | 106 | 137 |
| matrix mult | 112 | 163 |
| nsieve | 111 | 130 |
| whetstone | 121 | 114 |
| average | 110 | 133 |

**Table 1. Normalized Instructions and Cycles**

- Using the schedule generated for an "ideal" VLIW, we add a negative weight edge between the register component graph nodes representing destination registers of two operations scheduled in the same instruction. This tends to force destination registers for operations scheduled at the same time into different components in the register component graph.

The weight for each register component graph node, GN, is the sum of the frequency value, $F$, for each operation in which GN appears as a destination.

## 4. Experimental Evaluation

We have implemented the above algorithm in Rocket, a retargetable optimizing compiler for C and Fortran [13]. Our preliminary results are summarized in Table 1 which lists the results we obtained by applying the LPK algorithm to the register component graphs built for each benchmark. The machine model chosen included 3 general-purpose functional units, each with its own register bank. This 3-wide partitioning was chosen in part to compare our results with those presented in [2]. In our machine model, we assumed that all integer computations within a functional unit required 1 cycle, while all floating point computation within a functional unit required 2 cycles. To sufficiently penalize register copies between banks we doubled these times for all inter-bank copies, leading to 2 cycles for copying an integer value and 4 cycles for a floating point value. All loads and stores required 4 cycles. We compare the code generated for our 3-wide partitioned VLIW with that for an "ideal" VLIW of the same width but a single register bank by normalizing the values to 100 for the ideal model. Thus, the value of 140 cycles for 8q indicates that, on our partitioned 3-wide VLIW, 8q required 40% more cycles than would be required on a 3-wide architecture with a single register bank. The table lists both the number of instructions and number of execution cycles as normalized values.

Comparing our work with that of Capitanio, et al. [2] is not straightforward since their algorithm is applied to only single block loops and ours is applied to whole programs. Further they assume that each operation can complete in one cycle, including inter-bank copies. Thus, our model penalizes such copies more than Capitanio's does. They present results for two loops on many different VLIW configurations. For the 3 functional unit versions of machines, their two loops degrade from 70 to 110 (57.1%) instructions and from 51 to 86 (68.6%) instructions when going from the ideal model to a machine with 3 register banks. Our results in Table 1 compare very favorably with their results.

Our results, while limited, lead to several observations. Foremost is that, while the partitioning algorithm chosen does a relatively good job of limiting copies (an average of 10% additional copies over the benchmark suite) it does not perform nearly as well when execution cycles are considered. In fact, only one of the benchmarks, whetstone, showed a smaller "degradation" for cycles than copies. This is certainly counter-intuitive. If anything, we would expect instructions to show larger degradation than execution time. This is because the imperfect ability to fill all three operation slots for each instruction should lead to "holes" in the ideal schedule that could accommodate some of the necessary copies between register banks "for free" in terms of execution cycles. The observation that, in fact, the execution time suffers much more than the number of instructions leads us to conclude that although the LPK algorithm may do a good job of minimizing the edges removed from the register component graph, it unfortunately tends to sequentialize the code generated by assigning registers that could represent destinations of independent operations to the same register bank. An additional negative feature of the partitioning algorithm used to date is that it requires considerable computation time. Thus, we plan that future work will investigate alternatives to the particular partitioning used in this experiment.

While our partitioning algorithm heuristics leave room for improvement, we believe that our framework modeling the problem is appropriate. It allows us to represent machine dependent features as costs on the nodes and edges of the graph and is independent of instruction-scheduling and register-allocation implementations.

## 5. Conclusions and Future Work

This paper describes a framework for generating code for VLIW 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 compares favorably to another study of partitioning for a similar model [2]. 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, while giving low increases in copy instructions, suggest the opportunity for further improvements in the register graph partitioning technique to increase parallelism. These opportunities will be fully investigated in future work.

Given the significant degradation in performance that this and other register partitioning techniques [2] have shown when compared to an ideal model with a single register bank, one might think that an interconnection network between the functional units and the register banks (thereby allowing full access to each register for each functional unit) as in the case of TTAs is a better approach. However, although simulation results of algorithms tuned to such hypothetical architecture give much better performance (less degradation), the cost, performance and feasibility of such an architecture with high levels of ILP presents serious problems.

Our initial work has led us to pursue several areas of future research to investigate potential improvements to the actual algorithms used within our framework, including:

1. Investigation of alternate graph partitioning schemes such as greedy algorithms and various stochastic methods (e.g., tabu search, simulated annealing and genetic algorithms) for improving the partition when long compile times are not prohibitive (e.g., compilation of DSP code). A multifaceted approach with different heuristics for ideal code with few or many "holes" appears to be very promising.

2. Use of value cloning to eliminate copies. We plan to look into the duplication of read-only values and induction variables across functional units to minimize copies and increase parallelism.

3. Our current model applies an "owner-computes" rule so that operations are performed in the functional unit associated with the register bank where the destination register resides. Further investigation into situations where relaxing this rule is beneficial would be of interest as well.

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] 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*.

[2] 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.

[3] 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.

[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] 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.

[8] 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.

[9] S. Freudenberger. Private communication.

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, 1979.

[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] 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.