RC 21884 (98461) 16 November 2000 Computer Science

IBM Research Report

Minimizing Inter-File Tranfers in Architectures with Separate Address Registers

Mayan Moudgill
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Ayan Zaks
IBM Research Center Haifa
Matam, Haifa 30195, Israel

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Minimizing Inter-file Transfers in Architectures with Separate Address Registers

Mayan Moudgill*
IBM T.J. Watson Research Center
P.O. Box 218 Yorktown Heights, NY 10598
mayan@watson.ibm.com

Ayal Zaks
IBM Research center in Haifa
Matam, Haifa 30195, Israel
zaks@il.ibm.com

ABSTRACT

In this paper, we consider instruction selection in architectures where the general-purpose register file is replaced by separate address and integer register files, each feeding a separate execution unit. In these architectures, load and store operations use address registers to compute the location being accessed. Further, values in address registers can be manipulated in only a limited number of ways. In general, a value may need to be transferred from an address register to an integer register, operated on by the integer unit, and then transferred back. In this paper, we describe an optimal polynomial time algorithm to partition operations between address and integer units that minimizes the number of inter-file transfers.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: *Processors-code generation*, compilers, optimization; C.0 [Computer Systems Organization]: General—*Instruction* set design

General Terms

Algorithms, Languages, Performance

Keywords

Min cut, Max flow

1. INTRODUCTION

Most processors have distinct tied-point and floating-point units, and use distinct register files to hold fixed-point and floating-point values. This has several benefits:

. It allows for a more compact encoding; or, alternatively, it allows for more registers for the same number of bits. The floating point instructions use the floating point registers (*fpr*), and the fixed point instructions use the fixed point registers, also called general-purpose registers (*gpr*). Thus, with 5 bits per register, an instruction can address a total of 64 different registers (32 *gpr* + 32 *fpr*).

- It reduces the number of ports. The floating point registers are connected to the floating unit, while the general purpose registers are connected to the fixed point unit. Thus, an architecture that can issue a floating point and a fixed point instruction every cycle needs only 3 ports (2 read + 1 write) per register file, instead of having one 6 ported register file.

The drawback of having distinct register files is that when a Boating point value needs to be operated on by a fixed point instruction, the floating point value has to be transferred from an *fpr* to a *gpr*, and vice-versa. In some processors, such transfers are not supported directly, and must be implemented by storing the value and loading it from memory. Fortunately, this does not happen very often.

It may be possible to split the general purpose register file further, into an address register file and an integer register file. The address register file will be used to compute addresses when accessing memory. Thus, memory access instructions such as a load with displacement instruction would use an address register for the base value, while other fixed point instructions such as multiply would use the integer registers. We further assume that, other than explicit inter-file transfers, all instructions use and set either only address registers or only integer registers.

Splitting the general purpose register file into address and integer register files would have similar benefits as having a floating point register file. It reduces the number of ports and allows for a simpler implementation that reduces both power and area. At our design point [4], having two register files of 16 registers and 3 ports (each) instead of one 32 register file with 6 ports saves 10% power and reduces the area by 50% [I].

Unfortunately, it is expected that the number of transfers between the address and integer files will occur more often than between general-purpose and floating-point register files. Generating addresses may require copying an address

*The work was done while the authors were working at T.J. Watson Research Center.

to an integer register, operating on it to compute a new address, copying the value back to the address register file, and then using it.

In order to minimize the number of transfers, a split file architecture will selectively duplicate instructions so that the new addresses can be computed without transferring to the integer register file. Thus, both *addi* and *adda* instructions might exist, where the former uses integer registers, and the latter uses address registers. However, typically the address register manipulation instructions would be a small subset of the integer register manipulation instructions.

In this paper we describe a polynomial time algorithm to partition values between address and integer registers using the minimum number of integer-address register transfers possible. This allows us to investigate the impact of duplicating additional instructions on the number of transfers required.

2. EXISTING WORK

Recent work [12, 14] extends the floating point unit of a superscalar machine with additional restricted integer capability, and then off-loads integer instructions from the integer and extended floating point units. Their architecture, unlike ours, does not exploit the inherent partition between address operations and integer operations. Further, they do not attempt to come up with algorithms to minimize the number of inter-unit value transfers, but instead focus on reducing the total execution time in a superscalar, multi-issue machine.

There is another body of work focusing on clustered architectures [8, 7, 12, 13, 5, 2]. These architectures divide the available registers and functional units into multiple groups called clusters. Typically, the registers in a cluster communicate mostly with units in the same cluster, and have only restricted communication with other clusters. This allows fewer register ports. In such architectures, the clusters tend to be relatively homogenous. Further, such architectures are multiple issue machines. Compiler work in this area focuses on distributing operations between units so as to balance the usage of the clusters and minimize total cycle time.

A class of architectures that has a separate address unit are the decoupled access/execute architectures [3, 9]. These architectures split (or possibly duplicate) operations between an address (or access) unit and an execute unit. The goal of these architectures is to allow address operations to be moved ahead of the other computations, so that data can be loaded before it is actually needed, thereby hiding memory latency. Consequently compiler work in this area has not focused explicitly on minimizing the number of inter-file tranfers.

3. THE ALGORITHM

3.1 Target ISA

The instruction set architecture we shall consider is a generic RISC ISA, modified appropriately to support the use of separate integer (*ir*) and address (*ar*) register files. The modifications are as follows:

la, lua	load/load-with-update an *ar* value
sta, stua	store/store-with-update an *ar* value
adda, addia	add an *ar* to an *ar* / a constant
subfa, subfia	subtract an *ar* from an *ar* / a constant
cmpa, cmpia	compare an *ar* with an *ar* / a constant
cmpla, cmplia	compare logical *ar* with *ar* / a constant
mta, mfa	move an *ir* to/from an *ar*

Table 1: address instructions

- The base address used by all load and store operations must be in an *ar*. This register is also being defined by load-with-update and store-with-update operations.
- The value stored or loaded from memory can be in an *ar*, using the *sta* and *la* instructions.
- Move instructions *mta, mfa* (referred to as *ma* in general) must be used to transfer values from an *ir* to an *ar* and vice versa, respectively.
- Basic arithmetic operations can be performed on *ar* values: addition, subtraction and comparison of two *ar* values, or one *ar* and a constant immediate value.

All operations that could use integer registers in the generic ISA can still do so, with the exception that memory operations must use an *ar* for the base address. This restriction was made to simplify the presentation; our framework can easily accommodate operations that cannot use integer registers, as we will show.

Instructions that use and/or set address registers will be called *ar* instructions, to distinguish them from instructions that use the *ir* registers, which will be called *ir* instructions. The *ar* and *ir* instruction opcodes are suffixed with a and i respectively. The set of *ar* instructions being considered is summarized in Table 1.

The above set of instructions is used only to illustrate the algorithm. It will be obvious how to extend the algorithm to handle extensions such as indexed memory operations, or the further duplication of *ar* instructions.

3.2 Initial Problem

First, we shall consider instruction selection for code within a basic block. We assume that we are given the data dependence graph for the straight-line codes sequence containing abstract operations such as loads, stores and arithmetic operations. We shall assume that each operation can be realized using one instruction. There are two decisions to make:

1. Assign each operation to either an *ir* or *ar* instruction.
2. Insert *ma* instructions at appropriate places so that *ir* instructions use only *ir* values and *ar* instructions use only *ar* values.

The two problems are closely related: solving one forces a solution for the other. As we shall see, both problems can be solved together, using a unified network flow approach.

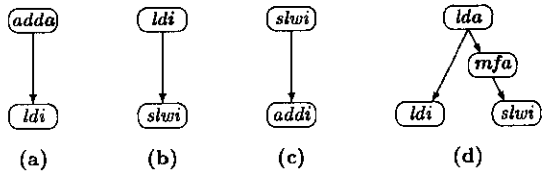

Figure 1: instruction selection

3.3 Developing the Solution

Let us consider some of the possible choices, and their ramifications.

- The base address for a memory operation, such as *ldi*, must be in an *ar*. If this base is computed using an *ar* instruction, such as an *adda*, no transfer is required (Fig. 1(a)).
- An operation that can only be executed by an *ir* instruction, such as a shift-left instruction *slwi*, should have its inputs computed using *ir* instructions (e.g *ldi*) to avoid a transfer (Fig. 1(b)).
- An operation that can only be executed by an *ir* instruction, such as a shift-left instruction *slwi*, should be used by *ir* instructions (e.g *addi*) to avoid a transfer (Fig. 1(c)). The analogue holds for operations that must define *ar* registers, such as *ldui* w.r.t. the base address register.
- If the result of an operation is used by both an *ir* -only operation and an *ar* -only operation, then it does not matter which instruction is selected for it; in either case, a transfer will be required (Fig. 1(d)).

These examples show that when minimizing the number of inter-file transfers, the constraints for address and integer registers flow in both directions:

- if a node defines a register as an *ir* or *ar*, all it's successor nodes should try to use that register as an *ir* or *ar*, respectively.
- if a node uses a register as an *ir* or *ar*, all its predecessor nodes should try to define that register as an *ir* or *ar*, respectively.

3.4 The Abstract Problem

Let $G = (N, E)$ denote the data dependence graph, where the nodes N represent abstract operations such as loads, stores and arithmetic operations, and the directed edges E represent def-use data dependencies. To simplify the presentation, some nodes are split into two or more sub-nodes. This happens when the set of neighbors of a node can be partitioned, such that the constraints and decisions related to the node can be applied to each part independently. There are two situations where nodes are split: memory operations and *ir* -only/*ar* -only operations.

Nodes that represent memory operations (loads and stores) are split into two sub-nodes: one representing the value being loaded or stored, and the other representing the address used and perhaps defined by the operation. The edges in E connected to the original node are re-connected to the appropriate sub-node. The two sub-nodes are not connected to each other, because for our purposes they are independent of each other: the value can be either in an *ir* or *ar*, independent of the address which must be in an *ar* (see Sastry et al. [14] for more details). Denote by $G' = (N', E')$ the graph obtained after splitting the memory nodes.

Recall that only a subset of operations were duplicated and can be executed on both address and integer units. Let $N_d \in N'$ denote these dual operations (including the value sub-nodes of memory nodes), and let $N_a \in N'$ denote all the address sub-nodes of memory nodes, which must be assigned to *ar* registers. Finally, let N_i denote all the remaining nodes - all the *ir* -only operations that must be implemented using an *ir* instruction, so that $N' = N_a \cup N_d \cup N_i$. Note that our framework can easily deal with *ar* -only operations as well, by including them in N_a. We chose not to include such operations to simplify the presentation.

Now consider a node v in N_i. The registers used by v and the register defined by v are all constrained to be *ir*. Moreover, the decision of placing *mfa* or *mta* instructions before or after v, respectively, affects the used and the defined registers independently. Therefore we can split v into several sub-nodes, one for each used or defined register[1]. Denote these sets of sub-nodes by N_{idef} and N_{iuse}. Abusing the notation, we redefine N_i to be $N_{idef} \cup N_{iuse}$. The nodes in N_a are also split in a similar way into N_{adef} and[2] N_{ause}. Let $G" = (N", E")$ denote the graph obtained after splitting the N_a and N_i nodes.

Now consider a (maximum) connected component $G_c = (N_c, E_c)$ of $G"$, disregarding the direction of the edges $E"$. If $N_c \cap N_i = \emptyset$, all the nodes N_c can be implemented using *ar* instructions; there is no need for any *ma* instruction. Likewise, if $N_c \cap N_a = \emptyset$, all N_c nodes can be implemented using *ir* instructions. It may happen that both conditions hold, in which case we are free to choose whether to implement N_c using only *ir* or only *ar* instructions. The interesting situations occur when N_c contains some N_i and some N_a nodes. In such cases, the use of some inter-file transfers is required.

Let $G_c = (N_c, E_c)$ be a connected component of $G"$, where $N_{ca} = N_c \cap N_a \neq \emptyset$ and $N_{ci} = N_c \cap N_i \neq \emptyset$. Focusing on each connected component separately in not necessary, but can reduce space consumption and simplifies the presentation. In this setting, problems 1 and 2 described in subsection 3.2 can be expressed in the following abstract form.

[1]In the inter-block setting, if v uses a register which has several reaching definitions, we can introduce a separate sub-node for each reaching definition. However, if the register defined by v has several uses, we cannot introduce a separate sub-node for each use because all the uses can share the same *ma*.

[2]Note that N_{adef} nodes correspond to load/store-with-update operations.

The Abstract Problem

Given a directed graph $G_c = (N_c, E_c)$ and two disjoint sets $N_{ca}, N_{ci} \subset N_c$, find a partition $N_c = N'_{ci} \cup N'_{ca}$ such that:

1. $N'_{ci} \supseteq N_{ci}$, $N'_{ca} \supseteq N_{ca}$, and
2. $|\{N'_{ci} \cap P(N'_{ca})\}| + |\{N'_{ca} \cap P(N'_{ci})\}|$ is minimized, where $P(N) = \{p : \exists n \in N, (p,n) \in E_c\}$.

The first condition corresponds to the decision of assigning each operation to either *ir* (N'_{ci}) or *ar* (N'_{ca}). The second condition counts the number of *ma* instructions inserted. Each *ir* instruction which defines a value used by an *ar* instruction must be followed by an *mta*, and the analogue situation requires an *mfa*.

3.5 The Min Cut Problem

The basic intuition for solving our problem, is that at least one *ma* instruction is required along every undirected path connecting an N_{ci} node and an N_{ca} node in G_c. Indeed, the abstract problem can be solved by transforming it into a classical (s,t) min cut problem. Specifically, we shall use the minimum node cut problem in undirected networks, which is the following problem (see, for example, [11, 10, 16]).

The Min Cut Problem

Given an undirected graph $G_u = (V_u, E_u)$ and two vertices $s, t \in V_u$, find the minimum subset $C_u \subset V_u \setminus \{s,t\}$ that separates s from t. That is, every path between s and t must contain a node in the cut set C_u. Such a cut set exists iff s is not connected by an edge to t.

The basic idea is to introduce two nodes s,t, where s is connected to all N_{ci} nodes and t is connected to all N_{ca} nodes, and compute a minimum node (s,t) cut C_u of the underlying undirected graph. Such a cut set exists, because s is not connected directly to t. Now all nodes on one "side" of C_u (explicitly defined below) will use *ir* instructions, and all the other nodes will use *ar* instructions. The cut set C_u will indicate where to place *ma* instructions.

3.6 Differences between the two Problems

Before describing the actual transformation, let us consider a few differences between our abstract problem and the classical min cut problem: the difference between an *ma* and a cut-node, and the fact that we require a partition in addition to the cut set. Both differences will be handled by the same transformation, as explained in subsection 3.7.

3.6.1 Cut nodes vs. ma *instructions*

In the min cut context, choosing a node v to be a cut node disconnects all a,v,b paths, for any pair of nodes a,b which are neighbors of v. In the original directed graph, a and b can each be a predecessor or successor of v. (The terms predecessor and successor always refer to the original directed graph G_c).

In our problem, inserting an *ma* after node v affects a subset of successors of v, disconnecting all a,v,b paths where a and/or b are successors of v. Paths where both a and b are predecessors of v are not disconnected by placing an *ma* after v.

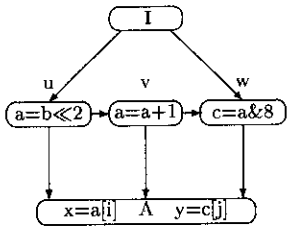

Figure 2: cut-nodes = {u,w}, *ma* needed for {u,v,w}

Note also that in our case there are two types of cut nodes: those that require an *mta*, and those that require an *mfa*. In contrast, the cut nodes of the min cut problem are all the same.

3.6.2 Partition requirement

Our goal is to partition all the nodes into two parts, in order to determine which operations will become *ir* instructions and which will become *ar* instructions. Only nodes of the cut set C_u are to have *ma* instructions, because this is the objective function of our optimization problem.

The cut set C_u separates s from t, but does not necessarily provide the required partitioning property. There are examples where more nodes require *ma* instructions, in addition to the nodes of a minimal cut (see figure 2 for an example).

The partition requirement states that only cut nodes are allowed to have successors from both parts. In addition, the predecessors of any node must all be of the same type or be cut nodes. This is because all instructions use either only address or only integer registers.

3.7 The Transformation

We now describe the transformation from the abstract problem to the min-cut problem, which handles the previously elaborated difficulties. The transformation involves two basic steps: the first step takes care of nodes that cannot serve as cut nodes, and the second step bridges the gap between cut nodes and *ma* instructions. These two steps provide a solution to the partition requirement described above, as explained in subsection 3.8.

3.7.1 First Step

The N_{iuse}, N_{ause} and compare nodes cannot serve as cut nodes, because an *ma* cannot be inserted after them. We must therefore make sure that such nodes will not be included in any minimal cut set. The first step addresses this problem as follows. The N_{iuse} and N_{ause} nodes are eliminated from the graph by merging them with nodes s and t, respectively[3]. Notice that this merge does not introduced an

[3]Note that we do not eliminate N_{idef} and N_{adef} nodes, because each such node is a candidate for placing *mta* and *mfa* instructions, respectively.

(s,t) edge, because $N_{ca} \cap N_{ci} = \emptyset$. The compare nodes are retained in the graph, in order to participate in the partition that will determine their assignment to an *ir* or *ar* instruction. The following second step guarantees that compare nodes will not belong to any minimal cut.

3.7.2 *Second Step*

The second and more important step, involves adding an edge (u,w) to E_u for every pair of nodes u,w that have a common successor $v \notin \{s,t\}$. This bridges the gap between cut nodes and *ma* 's: choosing node v to be a cut node will not disconnect paths between pairs of predecessors of v, similar to inserting an *ma* after v. Also note that a node cannot be a direct successor of both s and t (because $N_{ca} \cap N_{ci} = \emptyset$), and therefore this step does not introduced an (s,t) edge.

3.8 The Partition

After performing the above two steps, we are ready to find a min node (s,t) cut C_u in the underlying undirected graph, and define the required partition as follows. Let S denote the set of all nodes (in $V_u \setminus (C_u \cup \{s,t\})$) that can reach s without reaching C_u first, and T denote all nodes that can reach t without reaching C_u first. Clearly, S and T are disjoint, and no edge (v,u) connects them (i.e. $v \in S$ and $u \in T$), because that would contradict the fact that C_u is an (s,t) cut. The remaining nodes $R = V_u \setminus (S \cup T)$ are "isolated" from both s and t: they are connected only to cut nodes and to themselves.

The desired partition sets N'_{ci} and N'_{ca} are now built as follows. First, every node of S is included in N'_{ci} and every node of T in included in N'_{ca}. Next, each cut node $c \in C_u$ is assigned to N'_{ca} or N'_{ci} according to its predecessors: if c has a predecessor that belongs to S, then c is assigned to N'_{ci}; an *mta* will be inserted after it. The analogue is performed if c has a predecessor that belongs to T. Note that the two conditions cannot hold for the same cut node, because any pair of predecessors are connected with an edge.

We are left with the R nodes and with cut nodes whose predecessors are all cut nodes themselves or R nodes. There may be several ways to assign these nodes to N'_{ci} and N'_{ca}. One way is simply to assign all of them to N'_{ca}. This completes our partition.

3.8.1 *Proof of correctness*

We now claim that the partition produced complies with the requirements:

1. For every node $v \in V_c$ and any pair of predecessor nodes u,w of v which are not cut nodes, $u,w \in N'_{ci}$ or $u,w \in N'_{ca}$.
2. For every node $v \in V_c \setminus C_u$, all the successors of v belong to N'_{ci} or they all belong to N'_{ca}.

Proof of claim:

Part 1. If $v \in S$ then v is reachable from s and so are the predecessors of v, so $u,w \in S$. A similar argument holds if $v \in T$. If $v \in C_u$ and some predecessor of v was in S or in T, then all the predecessors of v which are not cut nodes belong to S or they all belong to T (respectively). If $v \in R$ then the predecessors of v which are not cut nodes also belong to R, and thus all belong to N'_{ca}.

Part 2. Let $v \in V_c \setminus C_u$. If $v \in S$ or $v \in T$, all the successors of v belong to N'_{ci} or they all beolng to N'_{ca}, respectively. If $v \in R$ then the successors of v either belong to R or to C_u. We claim that they were all assigned to N'_{ca}. Indeed if v were to have a successor $c \in N'_{ci}$, then $c \in C_u$ (because $R \subseteq N'_{ca}$). But recall that a cut node is assigned to N'_{ci} only if it has a predecessor $p \in S$. A contradiction is reached: v should belong to S instead of R because it is connected to $p \in S$. Therefore all successors of v belong to N'_{ca}, completing the proof. □

```
bar () {                    foo(int x) {
int * a;                    ...
foo(a);                     }
}
```

Figure 3: calling convention

4. PROCEDURAL ANALYSIS

The basic algorithm presented in the previous section in the setting of straight-line code, can easily be extended to multiple blocks of a procedure. The algorithm as described can compute the minimal number of *ma* instruction needed for the entire procedure, their placement and an associated partition of the procedure's operations to *ir* and *ar* instructions.

In order to retain our basic framework, we assume that the control flow graph has been transformed so that no block has multiple predecessor and multiple successor blocks. This allows us to consider placing *ma* instructions inside existing basic blocks only, without compromising the quality of the result.

When considering inter-file transfer minimization for procedures, another reasonable objective is to minimize the number of *ma* instructions that will be executed while running the program. This can also be computed by our algorithm, using block frequencies gathered from profiling or other sources, and a weighted variant of the min cut problem: each node is assigned a weight (the number of times it is expected to be executed), and the objective is to find an (s,t) cut set C_u with minimum total weight.

It is also possible to incorporate other factors into the weight assigned to each node. In general, the heavier the weight assigned to a node, the less likely it will be included in the cut set and have an *ma* placed after it.

5. INITIAL RESULTS

We have implemented our algorithm in a C optimizing compiler [6]. In C programs, a pointer that is passed as an argument may actually be treated by the called function as an integer (see Figure 3). To get around this problem, all parameters are passed in the integer register file. Similarly, all return values are returned in the integer register file. This causes additional *ma* instructions to be introduced

benchmark	description	size	mfa	mta	ma/size
compress	Spec92	4154	64	25	2.14%
spell	AIX utility	2607	53	61	4.37%
sed	AIX utility	9753	289	98	3.97%
yacc	AIX utility	22572	198	239	1.94%
prof	AIX utility	8071	128	55	2.27%
xlisp	OO Lisp	17807	611	263	4.91%
m88ksim	Spec95	37356	719	706	3.81%

Table 2: static measurements

benchmark	size	mfa	mta	ma/size
compress	138183	5784685	94465	4.25%
spell	47691	77185	1336824	2.96%
sed	14261	11056	481235	3.45%
yacc	51054	81397	622176	1.38%
prof	7907	49197	79237	1.62%
xlisp	6062540	70402286	166382237	3.91%
m88ksim	271745	1358864	4423817	2.13%

Table 3: dynamic measurements

at function calls.

In this section we present experimental results gathered by running our algorithm on several benchmarks. We computed both the minimal number of *ma* instructions needed, and the optimal placement of *ma* instructions taking into consideration frequency estimates. We used the heuristics of Ball and Larus [15] in order to estimate the basic block frequencies. The static measurements are presented in table 2, and table 3 contains the dynamic measurements.

6. REMARKS AND FUTURE WORK

It is interesting to note that the problem cannot be solved by considering the two sub-problems separately, one sub-problem for placing a minimum number of *mta* instructions and the other for placing the minimum number of *mfa* instructions. The two sub-problems are inter-connected, and one must find the total minimum of *ma* instructions together.

The algorithm presented in this paper computes the optimum placement of inter-transfer instructions. However, its objective function relies on adequate costs assigned to the nodes of the graph.

The preliminary results indicate that the algorithm proposed in this paper makes it possible to use split address and integer register files with an additional overhead of less than 5% additional instructions. These additional instructions are *mfa* and *mta* instructions that are introduced to copy values between the address and integer files.

Future work will reduce this number even further. Since the algorithm presented in this paper is optimal, we shall not be looking for better heuristics. Instead, we shall focus on the following two areas:

1. Make use of type information wherever possible to pass pointer values to functions in address registers; this will cut down on the number of *ma* 's introduced because of function calls.
2. Add additional *ar* instructions so as to reduce the necessity of transferring from the integer unit. Two prime candidates are *lia* (load immediate value) and *slwia* (shift left word by immediate value).

A simple analysis suggests that, even without these improvements, using separate address and integer register files in combination with the algorithm described in this paper will save at least 15% energy in our implementation. Each instruction has, on the average, about 2 register accesses (reads and/or writes). Each access uses 10% less energy, because of the simpler register files. Consequently, even after taking into account the additional 5% instructions, our implementation realizes about a 16% energy saving.

7. REFERENCES

[1] D.Meltzer. private communication. 2000.

[2] E.Nystrom and A.E.Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st annual international symposium on Microarchitecture*, pages 103–114, 1998.

[3] J.E.Smith, G.E.Dermer, B.D.Vanderwarm, S.D.Klinger, C.M.Rozewski, D.L.Fowler, K.R.Scidmore, and J.P.Laudon. The zs-1 central processor. In *Proceedings of the 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–204, Oct 1987.

[4] J.Glossner, J.Moreno, M.Moudgill, J.Derby, E.Hokenek, D.Meltzer, U.Shvadron, and M.Ware. Trends in compilable dsp architecture. In *Proceedings of the 2000 IEEE Workshop on Signal Processing Systems (SIPS) Design and Implementation*. IEEE, October 2000.

[5] J.Hiser, S.Carr, P.Sweany, and S.J.Beaty. Register assignment for software pipelining with partitioned register banks. In *Proceedings of the International Symposium on Parallel and Distributed Systems*, 2000.

[6] J.H.Moreno, M.Moudgill, K.Ebcioglu, E.Altman, B.Hall, R.Miranda, S.K.Chen, and A.Polyak. Simulation/evaluation environment for a vliw processor architecture. *IBM Journal of Research and Development*, 41(3), may 1994.

[7] J.Janssen and H.Corporaal. Partitioned register file for ttas. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 303–312. ACM, November 1995.

[8] J.L.Cruz, A.González, M.Valero, and N.P.Topham. Multiple-banked register file architectures. In *The 27th Annual International Symposium on Computer architecture*, pages 316–325. ACM, June 2000.

[9] N.P.Topham, A.Rawsthorne, C.E.McLean, M.J.R.G.Mewissen, and P.Bird. Compiling and optimizing for decoupled architectures. In *Proceedings of the Supercomputing '95*, Dec 1995.

[10] R.K.Ahuja, T.L.Magnanti, and J.B.Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, New Jersey, 1993.

[11] S.Even. *Graph algorithms.* Computer Science Press, Maryland, 1979.

[12] S.Palacharla and J.E.Smith. Decoupling integer execution in superscalar processors. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 285–290. ACM, November 1995.

[13] S.Rixner, W.J.Dally, B.Khailany, P.Mattson, U.J.Kapasi, and J.D.Owens. Register organization for media processing. In *6th International Symposium on High-Performance Computer Architecture*, January 2000.

[14] S.S.Sastry, S.Palacharla, and J.E.Smith. Exploiting idle floating-point resources for integer execution. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 118–129. ACM, June 1998.

[15] T.Ball and J.Larus. Branch prediction for free. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, June 1993.

[16] T.H.Cormen, C.E.Leiserson, and R.L.Rivest. *Introduction to Algorithms.* MIT Press, 1990.