FULLY DISTRIBUTED REGISTER FILES FOR HETEROGENEOUS CLUSTERED MICROARCHITECTURES

A Dissertation
Presented to
The Academic Faculty

By

Santithorn Bunchua

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
July 2004

FULLY DISTRIBUTED REGISTER FILES
FOR HETEROGENEOUS CLUSTERED MICROARCHITECTURES

Approved by:

Dr. D. Scott Wills, Advisor

Dr. Linda M. Wills, Advisor

Dr. Douglas M. Blough

Dr. Hsien-Hsin S. Lee

Date Approved: July 9, 2004

ACKNOWLEDGEMENT

This dissertation could not be completed without influences from several individuals, to whom I am grateful for their contributions, direct or indirect, to the completion of this research.

Prof. Scott Wills, my advisor, has continually supervised and guided this research from the beginning. I have learned a lot more than research work through his vision and enthusiastic approach to work and life. He has also arranged for my financial support throughout my PhD study in computer architecture.

Prof. Linda Wills, my co-advisor, has also supervised and guided this research with her dedication and in-depth knowledge on computer architecture and code generation. Without you, Scott and Linda, I would not be able to complete this work.

Prof. Douglas Blough and Porf. Hsien-Hsin Lee, my reading committee members, have provided several useful comments and recommendations to improve the quality of this research.

Prof. John Cressler, Prof. Bonnie Heck, and Prof. Milos Prvulovic have dedicated their valuable time to participate in my dissertation committee, and have provided useful comments and feedbacks for this research.

In addition, I would like to thank all PICA and EASL research group members for giving me a wonderful time and experiences; especially Krit Athikulwongse and Jongmyon Kim, for a countless number of productive and enjoyable discussions; Soojung Ryu, for sharing her insight on the register allocation topic; Steve Nugent, for developing GENESYS and sharing his insight on its inner working; and all others, Jinsung, Lewis, Brett, Nidhi, Hongkyu, Peter, Cory, Murat, Chris, Cameron, Mark, Antonio, William, Tarek, Roy, and Sam, in no particular order.

Most importantly, I would not be here without continued support and unconditioned love and encouragement from my family – Kirti and Pensri, my parents, and Santinath, my sister; and Voravannee, for her love and care for me in the past, present, and future to come.

Finally, all these would not be possible without *Your* will. Please let me be *Your* effective instrument, and continue to guide me along the path *You* have chosen for me.

Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Conventional processor design utilizes a centralized operand repository (a central register file) and transport network (a zero-cycle fully connected bypass network) to deliver operands to and from functional units. This centralized design works well with a small number of functional units but cannot scale to a large number of functional units. As more functional units are integrated into a processor, the number of ports on a central register file grows linearly while area, delay, and energy consumption grow even more rapidly. Physical properties of a bypass network scale in a similar manner.

In this dissertation, a fully distributed register file organization is presented to overcome this limitation by relying on small register files with fewer ports and localized operand bypasses. Unlike clustered microarchitecture explored in other research, each cluster features a small single-issue functional unit coupled with a small local register file. Several clusters are used, and each of them can be different. All local register files are interconnected through a register transfer network that supports efficient multicast communications. Microarchitecture and techniques to support distributed register file operations are presented for both dynamically scheduled and statically scheduled processor organizations. These include the eager and multicast register transfer mechanism in the dynamic approach and the global data routing with multicasting algorithm in the static approach. Although a distributed register file architecture requires additional cycles to execute a program, it is compensated by significant savings obtained through smaller area, faster operand access time, and lower energy consumption. With

faster operating frequency and more efficient hardware implementation, overall performance improvement can be achieved.

In addition, the fully distributed register file organization is applied to an ILP-SIMD processing element, which is the major building block of a massively parallel media processor array. The results show reduction in die area, which can be utilized to implement additional processing elements. Consequently, overall performance is improved through a higher degree of data parallelism through a larger processor array.

In summary, the fully distributed register file architecture permits future processors to scale to a large number of functional units. This is especially desirable in high-throughput processors such as wide-issue processors and simultaneous multithreaded processors. Moreover, localized communication is highly desirable in the transition to future deep submicron semiconductor technologies since long wire is becoming a critical issue in processes with extremely small feature sizes.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Technological advances in the past decade have created enormous opportunity for computer designers to design and build processors with increasingly higher performance. Several key technologies contribute to this rapid rate of advancement, with technology scaling as one of the most important. Through evolution in lithography, material sciences, and logic design, semiconductor devices can be scaled down into very deep submicron regions. Aggressive technology scaling allows for extremely integrated components to be realized using transistor devices that are both faster and consume less energy.

With smaller and less energy consuming transistors, computer designers can implement more complex functionality and even integrate more components into a single chip substrate. Faster transistors enable faster computation. However, as feature size is scaled below 0.25μm, wire delays start to dominate signal delays. It is predicted that in the 100nm generation, only 16% of the die area can be reached in a single clock cycle and even less as feature sizes are further decreased [1]. In next generation processors, long global interconnect should be avoided in favor of short local interconnect. Architectural solutions to long global interconnect issues are needed for future technology generations in the long term because conventional interconnect scaling cannot achieve the required performance as predicted by the ITRS roadmap [2]. Interconnect scaling is therefore a critical issue for future processor design.

In modern processor designs, several structures are shared and, thus, require global interconnect. Shared structures related to operand transport deserve special attention due to their scalability issues as processors become increasingly parallel. These structures are depicted in Figure 1 and include a central register file, one or more reservation stations (which can be a part of a reorder buffer or a register update unit, depending on implementations), and an operand bypass network. Furthermore, implementation costs of these components grow rapidly as the processor issue width increases, and these structures are considered critical paths in modern processor designs further limiting the increase in the processor clock frequency [3][4][5].

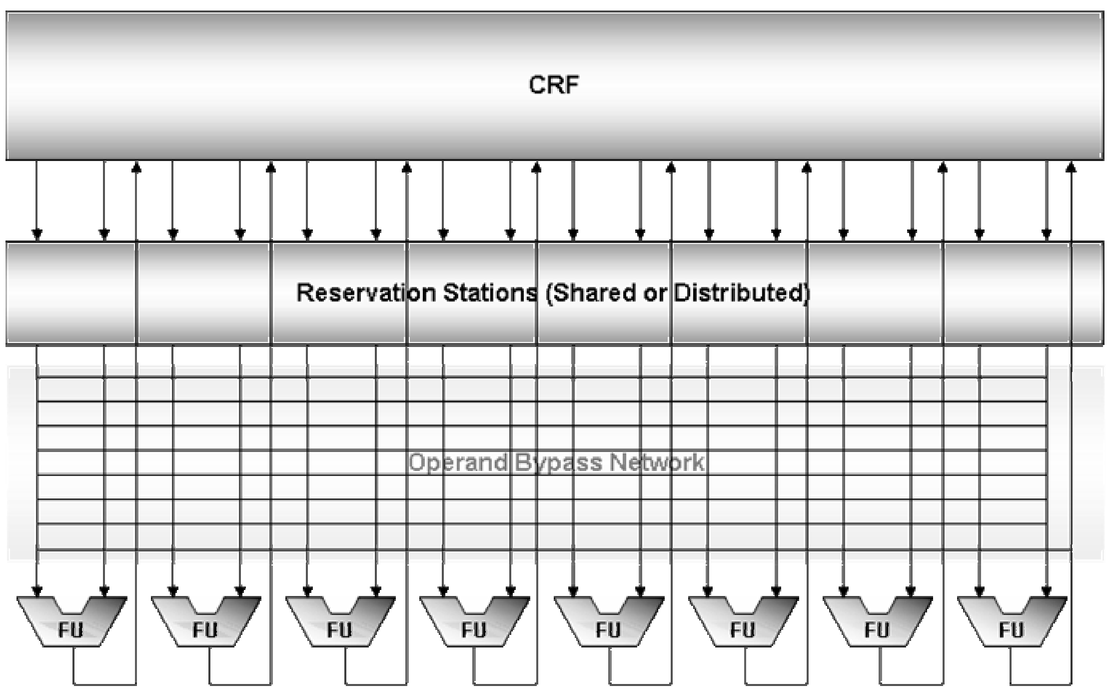


Figure 1 Operand transport in ILP processors

Several approaches have been devised to address this inefficiency. In the physical level, new interconnect and low-k dielectric materials [2][6] are being sought that have better performance than current materials. Novel approaches, like optical or radio frequency interconnects [7][8][9], are also promising but are still in the research phase. In the logic circuit level, circuit designers employ clever techniques when designing operand transport circuits. These include multiplexed read/write ports (to reduce the total number of access ports) [10], multiple replicated register files to distributed access port usage [11][12][13], and register caches [13]. Despite all these developments, scalability is a critical issue with a large number of functional units. Moreover, conventional interconnect scaling is insufficient and cannot achieve required performance in the long term. Therefore, an architectural solution is needed that can efficiently transport operands to and from a large number of functional units by utilizing local communications as much as possible.

1.2 Problem Statement

As processors strive to support higher degrees of instruction level parallelism (ILP) and thread-level parallelism (TLP), a large number of functional units need to retrieve operands and write back results concurrently in each execution cycle. The number of operands that have to be transported grows linearly with the total number of functional units. The hardware implementation cost of the current operand transport design, which utilizes a central register file and a large and complex operand bypass network, grows even more rapidly. The problem becomes especially critical in future deep submicron processes, which are interconnect-limited, since the register file and operand bypass structure involve a large number of long interconnects.

To be able to scale to extremely small feature sizes and highly parallel processor organizations, a shared register file must be replaced with distributed structures having localized communications. This form of distributed register file structures needs to be studied to determine its implications as employed in modern processor architectures. The research presented in this dissertation studies this implication with the following approaches:

- Introduce the distributed register file organization to modern ILP processor architectures including dynamically scheduled, statically scheduled, and embedded media processor architectures.
- Devise mechanisms to improve operational efficiency and decrease overhead of the distributed register file organization.
- Identify factors that have significant impact on the performance and the implementation of distributed register file architectures.
- Model the new architectures through detailed execution-driven simulators and evaluate execution performance of the new approaches.
- Employ hardware implementation cost models to evaluate implementation costs of the new architectures and determine overall performance and efficiency from the previously obtained execution performance results.

Currently there are three broad categories of processor architectures that are of interest. The first is the dynamically scheduled architecture based on RISC and superscalar principles. Processors in this category have been widely successful commercially because they can achieve very high performance through ILP exploitation without recompilation of application software. The second category is the statically scheduled architecture aimed for high performance applications and includes VLIW and EPIC. Processors in this category typically require recompilation for applications to work well with each processor generation. This requirement is somewhat relaxed in the recent EPIC architecture. The final category is the statically scheduled embedded processor architecture, which is more specialized towards intended applications and has less restrictions in terms of backward compatibility than general purpose VLIW or EPIC architectures due to the nature of its applications. To more fully understand the implication of distributed register files, the subject should be studied on these three categories of architectures using a common framework.

Although the concept of distributed register files is simple, its application is not straightforward. The following issues need to be addressed:

Execution with distributed registers

> Typical applications are written with code generated based on the assumption that registers are global and shared resources. The execution of such code with distributed register files breaks this assumption and requires special processing. The mismatch between this assumption in the code generator and the actual microarchitecture can create difficulties while executing applications. It is crucial to identify these difficulties and provide efficient mechanisms to address them.

Register file and functional unit organizations

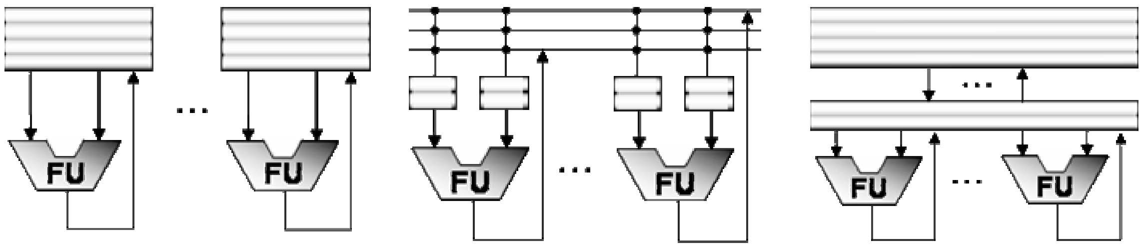
There can be several ways to partition and distribute registers. With the primary objective of localized communications, the organization of registers and functional units and their interconnection should be carefully designed to avoid global communications as much as possible.

Overall performance and efficiency

Transitioning from global to distributed structures normally impose several additional issues such as synchronization, consistency, and redundancy. Mechanisms to deal with these issues typically incur overhead. It is important to confine overhead associated with distributed register files to within a small manageable level so that overall performance and efficiency improvements are realized.

1.3 Related Work

Conventional processor architectures utilize a centralized multi-port register file as a fast operand repository for all available functional units within a datapath. However, for a non-trivial number of functional units, this conventional design does not work well since its physical properties (area, delay, and energy consumption) scale poorly with the number of functional units. A recent study [14] by Rixner, *et al*, analyzes scaling properties of various register file organizations. The results show that the implementation die area of a central register file grows as n^3, access time delay as $n^{3/2}$, and energy consumption as n^3, with n being the total number of functional units. Figure 2 shows other register file organizations that have been evaluated including SIMD, distributed register file (DRF), and hierarchical organizations. The stream organization, which provides efficient staging of streaming data, is beyond the scope of this dissertation.



(a) (b) (c)

Figure 2 Register organizations (Rixner, *et al*): (a) SIMD organization, (b) DRF organization, and (c) Heirarchical organization

Note that a distributed register file organization presented in this dissertation falls under the SIMD category in this particular study. The DRF organization is used in Imagine media processors [15][16] and is discussed briefly in the subsequent chapter.

A multiple replicated register file organization is used to distribute register read ports to multiple register files while keeping contents of all register files synchronized. This organization is used in the context register matrix of Cydrome's Cydra 5 [17] and the clustered datapath of DEC's Alpha processors (starting from 21264 onwards) [11][12][13]. Each functional unit can read from a connected register file only, while a computed result is written to all register files, which normally incurs extra latency (e.g. 1-cycle delay in Alpha when a value is replicated to a remote register file). Moreover, complete replication of register file contents is inefficient in terms of both area and energy consumption.

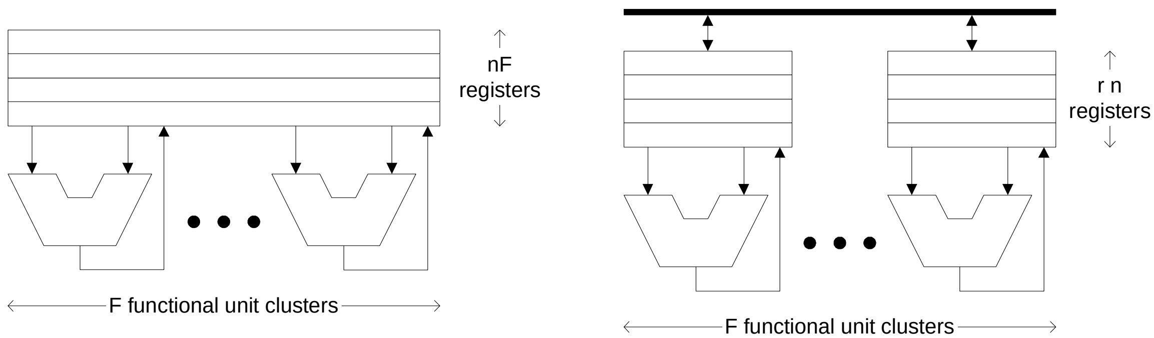
The WSRS (register write specialization/register read specialization) architecture [18] extends a replicated register file scheme by partitioning a register file into several subsets. Each cluster can only write to some subsets (write specialization) and read from some subsets (read specialization). Multiple copies of each subset are available, and their contents are synchronized by always writing a new value to all copies.

Register file partitioning without replication has been used in the transport triggered architecture (TTA) [19]. A central register file is partitioned into one or more smaller register files with fewer access ports. A transport network connects all functional units and all register files together. A functional unit can access any register file through a transport network restricted only by network and register port contentions. A global register namespace is used in which each register is assigned a unique number visible at the ISA level. This approach reduces the number of access ports on each register file, but all register communications still involve long latency signal transmission.

Various types of interconnection networks for operand transport are characterized and evaluated in [20][21][22]. Other register file organizations and techniques related to dynamically and statically schedule processor architectures are presented in their respective chapters.

1.4 Comparison of Central and Distributed Register File Designs

As processors deploy more concurrent functional units, the register file needs more registers and more access ports to support additional parallelism. These additional registers and ports create adverse effects on register file performance, in terms of die area, access delay, and energy consumption. The distributed register file organization is a viable structure to overcome these limitations of the central register file structure. In this section, the performance of the distributed register file is demonstrated and compared to the central register file using Rixner's register file model [14].



(a) (b)

Figure 3 Register file organization: (a) central register file and (b) distributed register file

The central and distributed register file organizations are illustrated in Figure 3. In the central organization, the number of ports (p) is related to the number of functional unit clusters (F) by $p = 3F$, assuming each functional unit cluster needs to read two operands and write one result. On the other hand, the distributed organization has one register file for each functional unit cluster with each register file having four ports. Three ports are connected to the attached functional unit, and the fourth port is a read/write port for transferring data between register files through the provided register transfer network. In this analysis, the register transfer network is a single bus connected to all register files. Other parameters used in the model are similar to [14]. The related parameters are provided in Table 1.

Table 1 Parameters for Rixner's register file model

Parameter	Empirical Value	Description
w	3	Register cell width (wire tracks) without ports
h	4	Register cell height (wire tracks) without ports
b	32	Data width (bits)
C_{word}	0.33	Ratio of a register cell's word select transistor capacitance to the capacitance of a minimum-sized inverter
C_{bit}	0.22	Ratio of a register cell's bit line transistor capacitance to the capacitance of a minimum-sized inverter
C_w	0.05	Ratio of capacitance of one track of wire to the capacitance of a minimum-sized inverter
E_0	12	Energy required to charge a minimum-sized inverter (in fJ)
v_0	1350	Wire propagation velocity in tracks per FO4-inverter-delays
α	0.25	Activity factor (probability that a node changes from 0 to 1 on a given cycle)
f_{cyc}	1/20	Clock frequency (in 1/FO4-inverter-delay)
F	1 – 10	Number of functional unit clusters
n	16	Number of registers required for each functional unit cluster

Equations (1), (2), and (3) model the implementation die area of a central register file (A_{CRF}), a distributed register file with a single register transfer bus (A_{DRF}), and a distributed register file with F register transfer buses ($A_{DRF,FullBus}$) respectively.

$$A_{CRF} = nFb(w+3F)(h+3F) \tag{1}$$
$$A_{DRF} = rnFb(w+4)(h+4) + Fb(w+4) \tag{2}$$
$$A_{DRF,FullBus} = rnFb(w+4)(h+4) + F^2b(w+4) \tag{3}$$

In the above equations, r is the distributed register demand factor, which represents the increase in register demands when a register file is distributed. In this analysis, the r values of 1, 2, and 3 are used and compared. The results are shown in Figure 4.

As shown in Figure 4, the area of a central register file grows a lot faster than the area of a distributed register file. Assuming a moderate register demand factor, a distributed register file results in significant reduction in area when three or more functional unit clusters are used. Moreover, the use of a fully connected network (e.g. multiple buses in this analysis) incurs only 1-5% increase in area over a single bus organization, which is negligible compared to the significant improvement over the central organization.

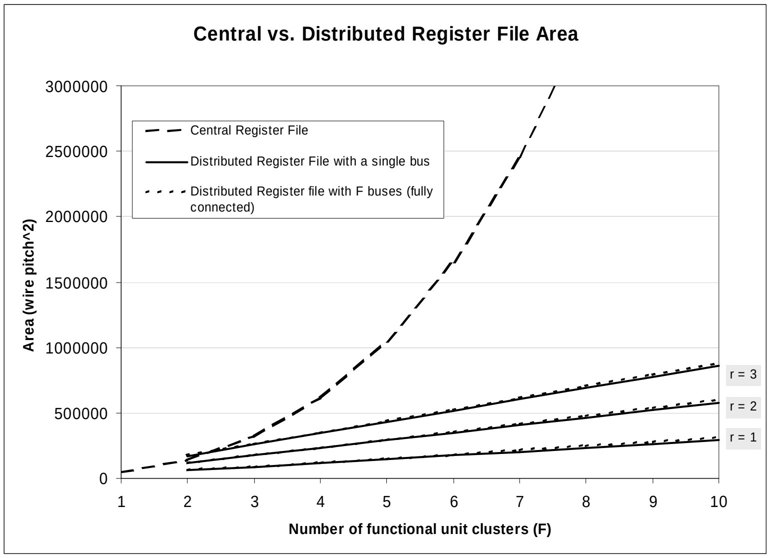


Figure 4 Area of central vs. distributed register files

In the following, delay or register read access time is modeled using Equations (4) and (5). The results are plotted as shown in Figure 5.

$$D_{CRF} = \log_4\left[b(C_{word} + (w+3F)C_w)\right] + \frac{b(w+3F)}{v_0} + \log_4\left[nF(C_{bit} + (h+3F)C_w)\right] + \frac{nF(h+3F)}{v_0} \quad (4)$$

$$D_{DRF} = \log_4\left[b(C_{word} + (w+4)C_w)\right] + \frac{b(w+4)}{v_0} + \log_4\left[rn(C_{bit} + (h+4)C_w)\right] + \frac{rn(h+4)}{v_0} \quad (5)$$

Similar to the area comparison, significant reduction in delay is evidenced as the number of clusters increases beyond two. The delay for the distributed register organization is constant because register files size are fixed for each cluster. The delay through the register transfer network is not modeled because it is considered to overlap with the execution stage of other functional units in this particular architecture.

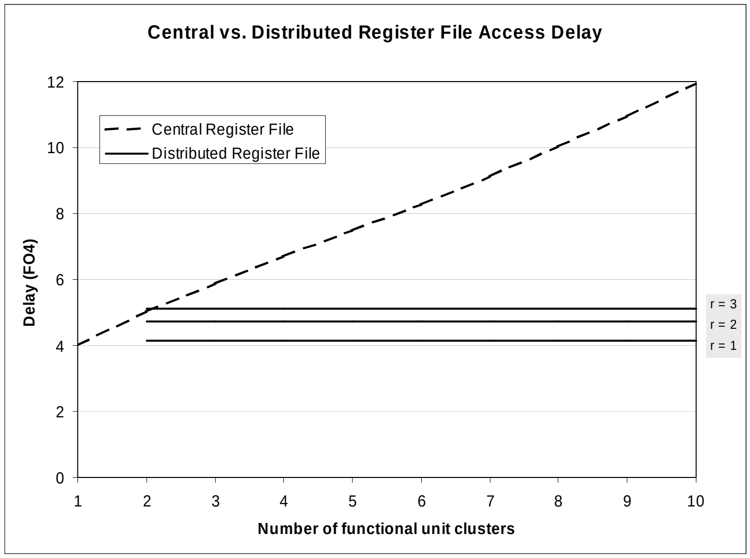


Figure 5 Access delay of central vs. distributed register files

Finally, the energy consumption is modeled as in Equations (6) and (7), and the results are shown in Figure 6. The results also show significant reduction in energy consumption for the distributed organization when the number of clusters is three or more.

$$P_{CRF} = 3F\left[(C_{word} + (w+3F)C_w)bE_0 f_{cyc} + (C_{bit} + (h+3F)C_w)nFb\alpha E_0 f_{cyc})\right] \tag{6}$$

$$P_{DRF} = 4F\left[(C_{word} + (w+4)C_w)bE_0 f_{cyc} + (C_{bit} + (h+4)C_w)rnb\alpha E_0 f_{cyc}\right] + (w+4)bFC_w\alpha E_0 f_{cyc} \tag{7}$$

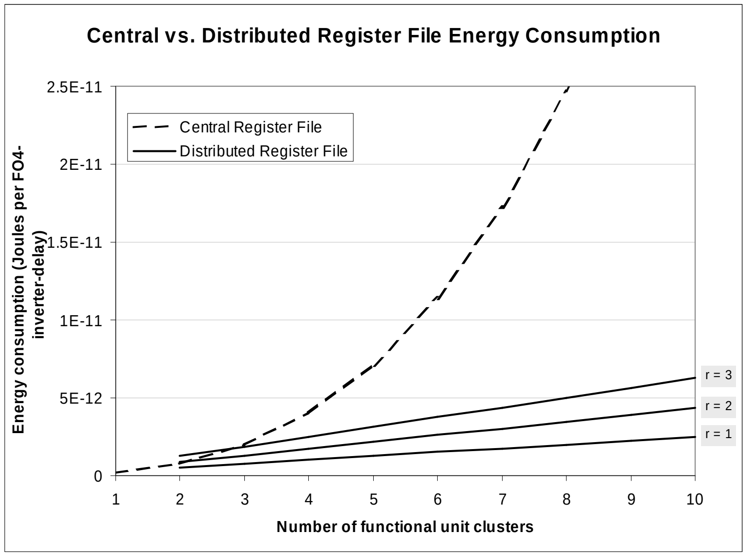


Figure 6 Energy consumption of central vs. distributed register files

In summary, the distributed register file organization has impressive circuit-level properties in terms of area, delay, and energy consumption when compared to the central register file organization. Considering the register demand factor of 1-3, comparable performance is obtained on the two-cluster organization. As the number of clusters increases beyond two, significant improvement is achieved.

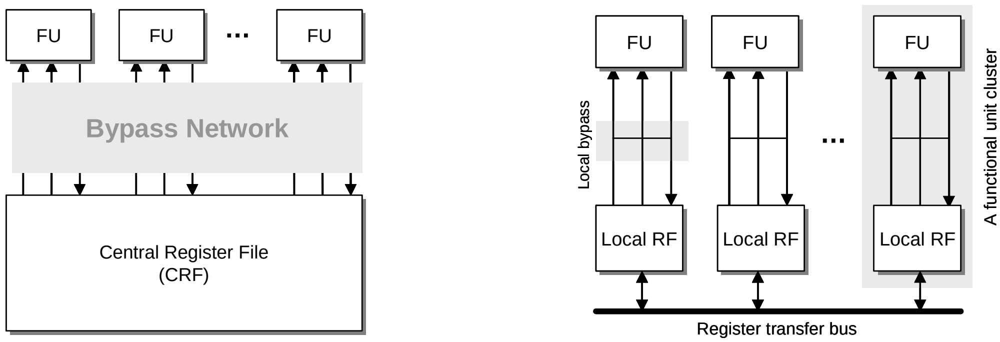
Besides circuit-level performance, the distributed register file has effects on the performance at the architecture and code generation levels. These issues are explored in the following sections which discuss various architectural mechanisms to support the distributed register file organization.

1.5 Overview of a Fully Distributed Register File Architecture

In a distributed register file architecture, a central register file, which is shared by all functional units, is replaced with several smaller register files distributed among all functional units. A register file that is directly attached to a functional unit is termed a *local register file* for that functional unit. A value in a local register file can be accessed with minimum delays while an access to a value in a remote register file incurs extra cycles to transfer a register value into a local register file before it can be accessed. There are several ways to organize distributed register files according to the following key parameters:

- the number of functional units associated with each local register file,
- the number of concurrent accesses for each local register file, and
- the interconnection among register files.

The above design space roughly determines the size of each local register file and the number of ports required on each. The more functional units sharing the same local register file, the more registers are needed in that local register file to hold intermediate operand values and the results of all connected functional units. Similarly, the more concurrent accesses allowed, the more access ports are needed. Typically, a functional unit with its associated local register file is referred to as a *cluster* in the literature. Having more than one functional unit that can operate simultaneously in a cluster increases register file complexity (size, number of ports, access time, and energy consumption) and instruction issue complexity. In the *fully* distributed register file architecture, a cluster consists of only one functional unit with one local register file as shown in Figure 7.



(a) **(b)**

Figure 7 (a) central register file datapath vs. (b) fully distributed register file datapath

All functional units that can execute in parallel are organized into distinct functional unit clusters, each with a local register file. These functional unit clusters can have different configurations, i.e., they are not necessarily uniform. The type and multiplicity of functional unit clusters, however, can have a significant impact on execution performance. In this organization, each local register file can have a minimum size and a minimum number of access ports (generally two read and one write ports), plus a number of extra ports for transferring values among register files through one or more register transfer buses.

The register transfer bus is used to transfer values between local register files when a value in a remote register is required in the local functional unit cluster. A broadcast-capable network, such as a bus or a crossbar, is used to facilitate eager transfer and multicast transfer mechanisms for reducing execution cycle penalty. These are described in subsequent sections. One extra bidirectional read/write port is added to a local register file for each register transfer bus to which it is connected. Without these extra transfer ports, normal functional unit operations may be blocked by register transfer operations as register file access ports are occupied. Multiple buses can be used to increase the number of concurrent register transfers or provide connection within subsets of clusters to reduce interconnect length and fan-outs. For example, two buses can be used: one for integer clusters, and the other for floating-point clusters.

Because accesses to remote data values incur extra transfer cycles, the total number of cycles required to execute a particular program in a distributed register file environment is greater than in a central register file environment. Therefore, an important design goal of a distributed register file architecture is to reduce these extra cycle penalties to a minimum through careful optimization of architectural parameters and distributed register file operations. The important functions required to support a distributed register file architecture are the following:

- functional unit (or cluster) assignment – assigns or dispatches each machine instruction to a particular functional unit;
- data routing – transfers register values between local register files by scheduling register transfer operations.

(Since there is only one functional unit in each cluster, the terms functional unit and cluster can be used interchangeably in the context of the fully distributed register file architecture.) These functions can be performed either dynamically at run-time or statically at compile-time.

A dynamic approach is suitable for dynamically scheduled processor architectures, such as superscalar processors. With this approach, register references in existing code, which assumes the presence of a central register file, are transformed into local register references at run-time through special hardware units. Since the transformation is performed automatically in hardware, all existing applications can be deployed immediately without recompilation. Additionally, the transformation can automatically adapt to different hardware configurations and changing run-time conditions. These benefits allow maximum binary compatibility with existing applications and high execution performance.

On the other hand, code can be generated specifically for distributed register file architectures. The idea is well-suited for most existing statically scheduled processor architectures, such as VLIW and EPIC. For a static approach, compilers need to be extended to generate code that is ready to execute in a distributed register file environment. The advantages of this approach include simple hardware implementations and the global knowledge about the code so that good decisions can be made for data routing and code scheduling.

1.6 Contribution Summary

The major contribution of this dissertation is the exploration of the fully distributed register file organization in various mainstream processor architectures, namely, superscalar, VLIW, and embedded media processors. Each of these three categories of processors present unique challenges and issues that need to be efficiently addressed when incorporating the aforementioned register file organization. In this section, contributions are summarized according to the order presented in this dissertation.

1.6.1 Dynamically Scheduled Fully Distributed Register File Architecture

- Design and definition of the fully distributed register file organization for dynamically scheduled processor architectures
- Design and evaluation of basic distributed register file operations including dependence-based cluster assignment, local register mapping, and demand-driven register transfer
- Design and evaluation of eager and multicast transfer mechanisms to reduce execution cycle penalty incurred by basic distributed register file operations
- Evaluation of the execution cycle penalty and its relationship with application characteristics and distributed register file parameters including number of register transfer buses and local register file sizes
- Comparison of area, delay, and energy consumption of a distributed register file organization and a conventional central register file organization

1.6.2 Statically Scheduled Fully Distributed Register File Architecture

- Development of simple and efficient algorithms to support the generation of code for a processor with a distributed register file organization
- Development of a code generation framework in the form of a code retargeting tool to transform legacy code into distributed register file code
- Development of evaluation methodology using a common framework with the dynamically scheduled distributed register file architecture evaluation
- Performance evaluation and comparison with the dynamic approach

1.6.3 Fully Distributed Register Files for ILP-SIMD

- Design and definition of the fully distributed register file organization for ILP-SIMD processing elements using the static approach
- Enhancement to the static code retargeting tool to support ILP-SIMD ISA
- Performance evaluation using an ILP-SIMD simulator
- Implementation cost evaluation using technology model-based system simulation

CHAPTER 2

DYNAMICALLY SCHEDULED DISTRIBUTED REGISTER FILE PROCESSOR ARCHITECTURES

2.1 Summary

Current operand transport design for superscalar processors involve a large centralized register file and an operand bypass network. Physical implementations of these structures do not scale well (in terms of area, delay and power) as the number of functional units integrated onto a processor complex is increased especially in future deep submicron semiconductor processes, which are interconnect-limited. Although area and energy consumption of these structures are small compared to the entire high-performance superscalar chip, their delay time is highly critical.

A fully distributed register file organization is used to address this problem. A central register file is replaced with local register files distributed to all functional units (i.e. one local register file per functional unit). Each functional unit can access local operands with minimum delay through its associated local register file and a local bypass path. An operand in a remote register file can be transferred into a local register file through a register transfer bus, which incurs an extra processor cycle. With this organization, register file access time is reduced by 23% and 41% for 4- and 8-way machines, respectively, with significant savings in terms of area and energy consumption.

The dynamic approach is presented to support distributed register files entirely in hardware. Therefore, all existing code can be executed on the presented architecture without modification. Additional functions performed during instruction dispatches

include dependence-based functional unit assignment, local register mapping, and register transfer operation dispatch. These functions convert register operands names in the original instruction stream to appropriate local register names and schedule remote register transfer as needed. Because of these register transfer operations, extra cycles are needed to complete the execution. Eager and multicast transfer mechanisms are used to reduce this execution cycle penalty by 27% on average. Overall, a fully distributed register file with a register transfer bus results in IPC value that is 84% (78%) of a 4-way (8-way) architecture with a central register file. An additional register transfer bus reduces IPC penalty by 23-28% while the third bus produces no significant benefits. The variation in the local register file size has only slight impact on performance.

2.2 Introduction

Dynamically scheduled microarchitectures, or more specifically superscalar architectures, are used in almost every commercial high-performance general-purpose processor due to its effectiveness in exploiting instruction-level parallelism (ILP) without software recompilation. The continuing trend for superscalar processors is to aggressively increase fetch and issue width and also clock frequency, which is possible thanks to the rapid development of semiconductor technology. These two objectives, however, cannot be easily achieved simultaneously. Wider issue width increases hardware complexity, which consequently lengthens processor cycle time. Although aggressive pipelining can be used to address this problem, the pipelining of operand transport stages (register file read and write stages) require extra levels of operand bypass network, which is costly in terms of cycle time and does not scale well as the number of functional units and pipeline stages increase.

Typically, operand movement in a superscalar processor relies on a large central register file and an operand bypass network. The register file size and the number of access ports, both of which grow linearly as issue width increases, largely determine its cost (area, delay, and energy consumption) [14][23][24]. Although area and energy consumption of a central register file and an operand bypass network are just a small fraction of the total die area and energy consumption of a modern superscalar processor chip, their delays are critical. Moreover, delay time of these structures is dominated by interconnect delay, which has become a critical problem in modern deep submicron technologies, since interconnect delay does not scale as well as gate delay [1][2][3][4][5].

Since the delay of a register file grows as the number of registers and the number of associated functional units increase, an organization with several small register files each serving a small number of functional units are desirable especially for a wide superscalar architecture [14]. This form of register file organization is called partitioned or distributed register files. A functional unit can access data from the attached local register file with minimum latency, while data from a remote register file incur extra latency to transfer into a local register file prior to being consumed. To retain backward compatibility with existing instruction set architectures (which assume a central register file), a distributed register file architecture must dynamically manage data transfer among local register files in addition to dynamically scheduling instructions. These extra latencies required to transfer data among local register files normally result in more execution cycles when compared to a similar architecture with a central register file. The ultimate objective of a distributed register file architecture design is, therefore, to obtain a realizable wide superscalar architecture with clock frequency that is high enough to overcome the execution cycle penalties caused by incomplete interconnection between register files and functional units.

In this contribution, a fully distributed register file organization that provides several small local register files, one for each individual functional unit is explored. Although physical properties of such configuration are highly favorable, impacts on execution performance is expected to be quite high due to its extremely distributed nature. This execution performance impact is characterized in detail, and mechanisms to reduce this shortcoming are presented.

2.3 Related Work

The register file requirement for wide superscalar processors have been studied in [4], which shows that a central register file does not scale well in wide superscalar processors due to the large number of registers and ports required. To address this problem, the MultiCluster architecture has been proposed, which utilizes a distributed register file organization. A two-cluster configuration of the MultiCluster architecture is shown in Figure 8.

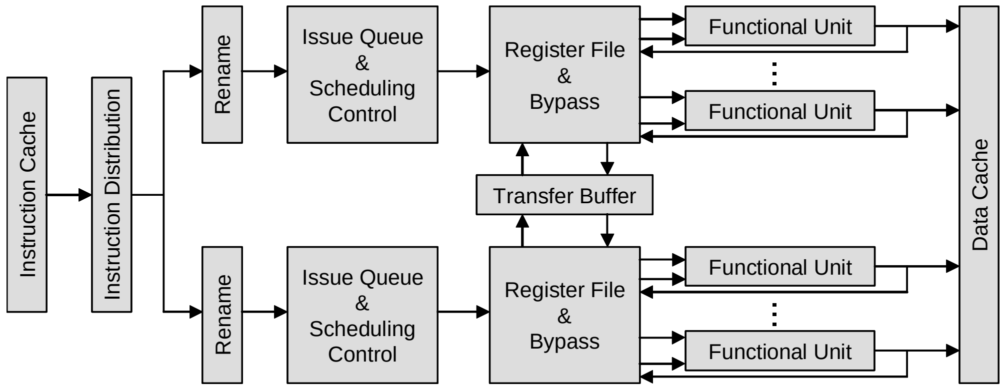
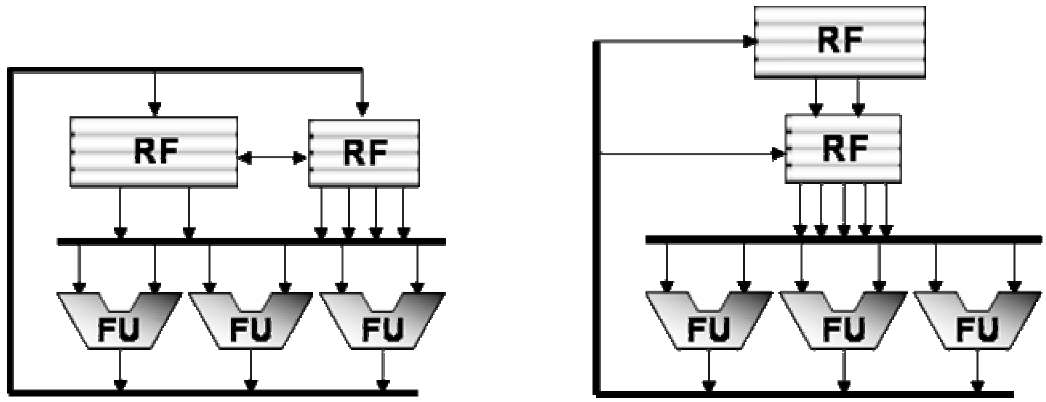


Figure 8 Dual-cluster Multicluster architecture

In the MultiCluster architecture, there are two or more clusters of functional units, each with its own register file. Registers are explicitly assigned to clusters based on register names. For example, in a dual-cluster architecture, even-number registers can be in the first cluster, and odd-number registers in the other cluster. By examining register references in an instruction, the instruction distribution unit can dispatch an instruction to the cluster with the needed register operands. If an instruction involves registers from multiple clusters, it is dispatched to all related clusters. The cluster that carries out the computation is designated the master cluster while the other clusters become slave clusters. Operations are issued to slave clusters when register values need to be exchanged with the master cluster through the transfer buffer. With explicit register to cluster binding, instruction distribution hardware is greatly simplified. However, by transferring register values during execution pipeline stages, extra latencies are inevitable for every instruction that is multiply issued. Consequently, performance of the MultiCluster architecture is highly dependent on the capability of compilers to generate code with effective register assignment.

Multiple replicated register files have been used to overcome cycle time problem in Alpha 21264 [11] and 21364 [12] processors, and later in the 21464 design [13] (which has been canceled prior to manufacturing). Alpha 21264 is a 4-way out-of-order superscalar processor implemented in 0.35 μm technology. Typically, an integer register file with 8 read ports and 4 write ports is required for a 4-way integer unit. However, an 80-register 12-port register file is difficult to implement and would exceed cycle time target. The integer unit of a 21264 is then divided into two clusters with one register file per cluster. The content of these two register files are kept identical by always writing a new value to both register files. An extra cycle is required when broadcasting an updated register value to the other cluster.

Multiple-banked register file architectures [27][28] employ multiple register files to address long register access latency but without requiring register files to contain duplicate copies of one another. Register files can be arranged in single- or multi-level configurations as shown in Figure 9. All register banks can have different size and access ports and, thus, different access latencies. A multi-level configuration or a register file cache is similar to a multi-level cache memory. The lower register file can be accessed in one cycle and contains only a subset of all registers while the upper register file stores all available registers but has multi-cycle latency. Two caching policies are explored: non-bypass caching and ready caching. These two caching policies leverage the fact that most operands are accessed only once. Alpha 21464 architecture employ a form of register caches in its integer and floating-point execution units that stores copies of the last 8 cycles of generated results, act as local bypass results multiplexers, and align result writeback to the main register file avoiding contention for the 8 write ports due to instruction latency [13].



(a) single-level **(b) multi-level (register file cache)**

Figure 9 Multiple-banked register files

Recent work on a distributed register file architecture extends the register renaming mechanism to support distributed register files [30][32]. With this approach, distributed register file operations are supported entirely in hardware without the need to provide multiple identical copies of register files. Values are transferred through register transfer operations, which are automatically dispatched as needed. These register transfer operations are treated similar to normal operations, thus, simplifying subsequent pipeline stages and exception recovery. However, an instruction dispatch unit becomes more complex with additional functions of cluster assignment and register transfer operation dispatch. These functions need to be designed carefully to minimize register communications, which is the major source of execution cycle penalties, and minimize workload imbalance among functional units, which can significantly affect performance through low functional unit utilization.

A dependence-based dispatch scheme has been proposed in a clustered dependence-based microarchitecture [5] to reduce the complexity of a large centralized issue window and long bypass buses. This dependence-based dispatch policy is used as the basis for a cluster assignment policy in this research, and is extended to address the workload imbalance problem through the eager and multicast transfer mechanism, which performs register communications ahead of time without incurring additional overhead similar to the eager writeback mechanism for writeback caches [29].

Several other cluster assignment algorithms, both static and adaptive (including a dependence-based scheme), have been studied on a uniform cluster configuration (all clusters have an identical configuration) [30][31]. Static cluster assignment algorithms are fixed policies that do not change during run-time while adaptive policies base their decisions on collected run-time information. In addition, a value prediction scheme has been proposed to predict remote operand values so that execution can continue even before values are received from remote clusters [32]. If the predicted value is incorrect, all affected instructions need to be cancelled and re-issued, which incurs several extra recovery cycles. In addition, it requires a prediction table, which can be quite large.

An instruction replication technique is proposed to reduce inter-cluster communications [33]. An instruction can be dispatched to additional clusters that will later need its result. This technique is appropriate in a uniform cluster configuration. However, it requires a history table to determine operand flows in various clusters; otherwise, it can only evaluate the demand for replication within a single dispatch group. Moreover, the proposed scheme increases dispatch bandwidth and assumes global broadcast of all computed results as a mean to communicate register operands among clusters, which increase hardware complexity.

In the following section, the fully distributed register file design for dynamically scheduled processors is presented, which features multiple local register files, one for each functional unit. Performance is then evaluated through cycle-accurate simulations of standard benchmark applications. Finally, the merit of the design is assessed through the performance results and implementation cost savings in terms of area, delay, and energy consumption.

2.4 Microarchitecture

The microarchitecture for dynamically scheduled fully distributed register file design (D-DRF) [34] is shown in Figure 10. It uses the register mapping scheme, which maps *architectural* register names (register operand names specified in incoming instructions) to local register names dynamically in the instruction dispatch stage, to support distributed register file operations. The local register mapping table (LRMT) keeps track of the current mapping and is consulted and updated by the instruction dispatch unit. Because no changes are needed at the instruction set architecture (ISA) level, compatibility with all existing code is maintained.

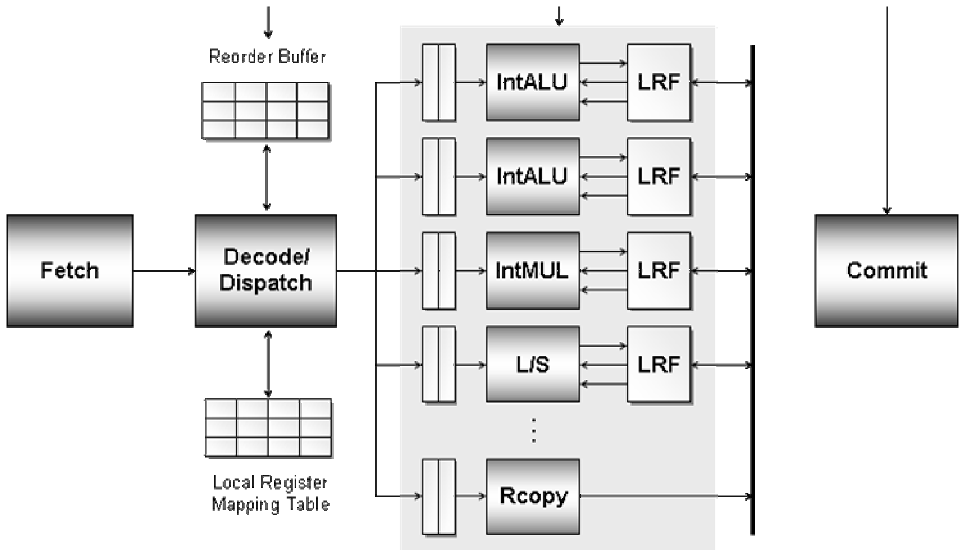


Figure 10 Fully distributed register file microarchitecture for dynamically scheduled processors

All functional units that can execute in parallel are organized into distinct *functional unit clusters*, each with a local register file. These functional unit clusters can have different configurations, i.e., not necessarily uniform. The type and multiplicity of functional unit clusters, however, can have significant impact on execution performance.

Each functional unit cluster is comprised of an instruction issue queue, a functional unit, and a local register file. An instruction issue queue is a simple FIFO queue for storing instructions awaiting execution in the current functional unit cluster. Instructions are issued sequentially from an instruction issue queue to the associated functional unit without the need for complex dependency check logic.

Each local register file has four access ports. Three ports (two reads and one write) are connected to the associated functional units and are used to supply operands and receive the result of the computation. The fourth port is a bidirectional read/write port connected to a dedicated register transfer bus and is used to transfer values among connected local register files. Dedicated register transfer ports are provided so that register transfers can be overlapped with normal executions of all functional units. These networks must be capable of broadcast operations to support the eager and multicast transfer mechanisms, which will be discussed later. Multiple networks can be used to provide connections within disjoint groups of functional unit clusters to reduce interconnect length and capacitive loads. For example, two buses can be used: one for integer functional unit clusters, and the other for floating-point functional unit clusters.

To execute a conventional instruction stream on a fully distributed register file architecture, the instruction dispatch unit has been extended to perform three additional functions. The first function is functional unit assignment, which determines the most appropriate functional unit cluster to which a particular instruction is dispatched. The assignment algorithm should recognize that operands are not ubiquitously accessible as in a central register file architecture, and an instruction needs to be dispatched carefully to minimize performance degradation. Once the functional unit is determined, the second function is to map source and destination register operand names (architectural register names) to local register names. This mapping process removes name dependences (as in the register renaming process in out-of-order superscalar architectures) and eliminates the need for future dependency checks since instructions are executed sequentially within each functional unit cluster. The third additional function is the dispatch of register transfer operations, which are necessary when some operands are located in remote register files. These three additional functions of the instruction dispatch unit are discussed in the following sections in more detail.

2.4.1 Functional Unit Cluster Assignment

One of the most important decisions is to determine the functional unit cluster where each instruction will be executed, which consequently specifies the local register file that can be accessed. This problem is widely known as the cluster assignment problem. If dependent instructions are assigned to different clusters, register communications through register transfer network is required, which may increase the total execution cycles. On the contrary, if too many instructions are assigned to the same cluster while leaving other clusters idle (workload imbalance scenario), these instructions may compete for the limited issue bandwidth of a single cluster and result in more execution cycles. The optimal assignment is the one that results in the smallest number of total execution cycles. Since this is an NP-complete problem, heuristics that can be efficiently implemented in hardware must be used. Several cluster assignment heuristics have been explored [30][31]. These heuristics try to address one or both of the following simpler problems: minimization of remote register communications and minimization of workload imbalance among functional unit clusters.

In D-DRF, a simple dependence-based assignment algorithm [5][30] is used. It assigns a given instruction to a functional unit cluster that possesses most of the required operands. If multiple functional unit clusters qualify, the one with the smallest load is selected. If there is more than one such candidate clusters, one unit is randomly chosen. The detail of this algorithm is shown in Figure 11. This dependence-based approach tries to minimize the number of extra instructions that are required to transfer values from remote register files i.e. minimize the remote register communication objective. The workload imbalance problem, however, is addressed by the eager and multicast transfer mechanisms. Moreover, some data routing operations are inevitable due to the non-uniform characteristic of functional units, which implicitly helps reduce the workload imbalance problem.

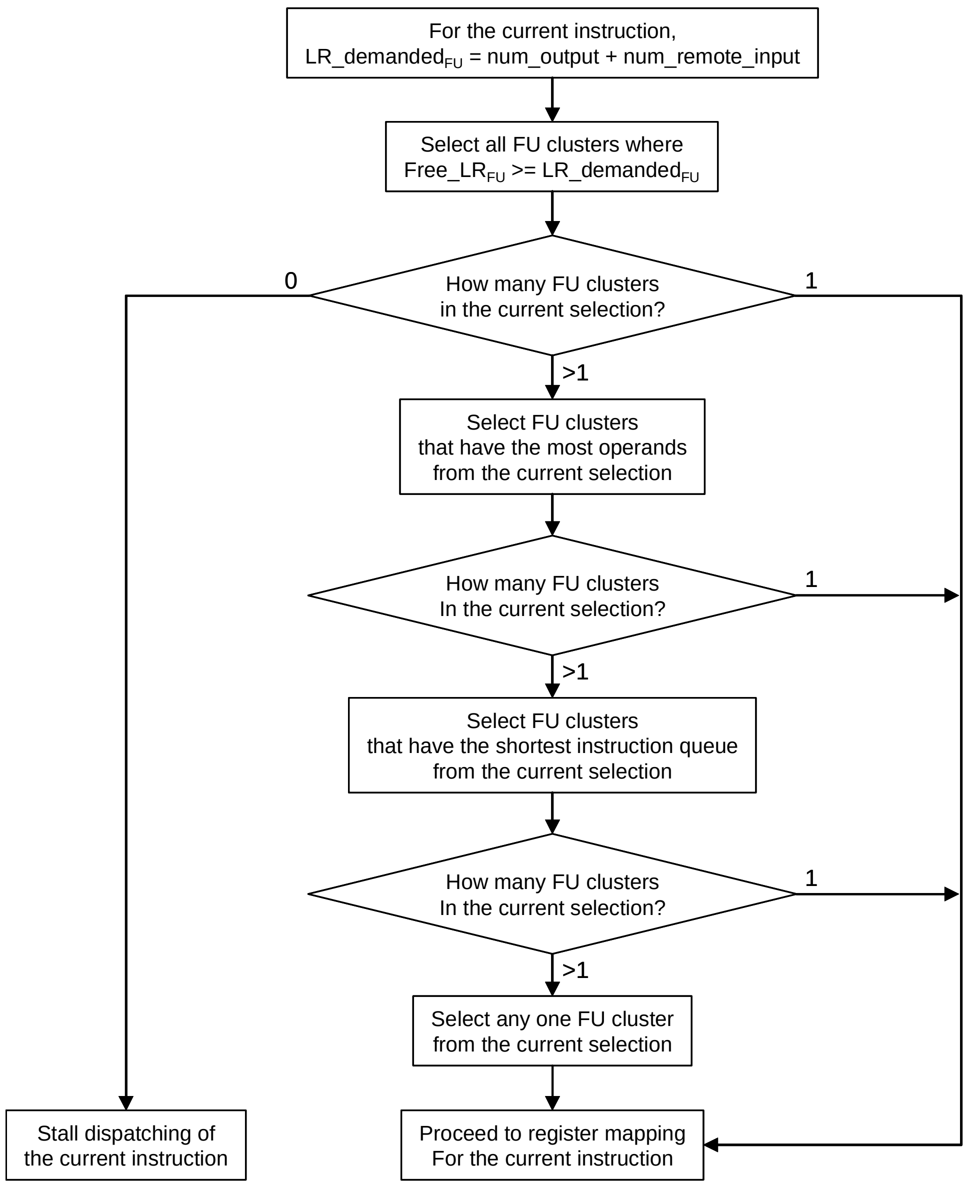


Figure 11 Dependence-based functional unit cluster assignment algorithm

The functional unit cluster assignment algorithm begins with the check for free local register files in each cluster. If there is not enough free local registers to hold all input and output operands of the current instruction, instruction dispatch is stalled. If several functional unit clusters have enough capacities in their local register file, the one that contains most of the required operands are chosen. If several functional unit clusters have the same maximum number of required operands, the one with the lightest load, as determined by the length of its instruction issue queue, is selected. The final tie-breaking rules is provided, which randomly select a cluster from a set of candidate clusters. After the functional unit cluster is determined for a given instruction, its register operands, both inputs and outputs, must be mapped to appropriate local registers in the respective functional unit cluster.

2.4.2 Local Register Mapping

After an instruction has been assigned a functional unit cluster, all of its register operand names must be mapped to local register names in the assigned cluster. The mapping information is obtained from the Local Register Mapping Table (LRMT). Additionally, the local register free lists are provided to indicate which local registers are free. The structure of LRMT is shown in Figure 12.

	FU Cluster 0			**FU Cluster 1**				**FU Cluster *m***		
	Valid	**Temp**	**LR**	**Valid**	**Temp**	**LR**		**Valid**	**Temp**	**LR**
R0	1	0	4	1	1	2		0	x	x
R1	1	0	2	0	x	x	…	0	x	x
R2	0	X	x	1	0	1		0	x	x
R*n*	1	0	3	0	x	x		1	1	1

Figure 12 The Local Register Mapping Table (LRMT)

LRMT is similar to a distributed set of register renaming tables, one for each functional unit cluster. Three pieces of information are stored for each mapping in each cluster: the valid bit, the temporary bit, and the mapped local register name. The valid bit indicates the validity of the entry. The temporary bit indicates the status of the map entry for freeing local registers that are eagerly transferred. The use of the temporary bit will be discussed when the eager and multicast transfer mechanisms are explained. Finally, the mapping from an architectural register name to a local register name is provided in the mapped local register name field (the LR column). Note that by providing multiple tables, one for each cluster, an architectural register name can be mapped to local registers in one or more clusters simultaneously.

Besides the mapping of register operand names, a local register has to be allocated for each destination register if an instruction produces one or more register results. The allocation process makes use of the free lists, which provide information about all free local registers. Free local registers in the assigned functional unit cluster are picked from the free list of that cluster, and the mapping from destination register names to those newly allocated local register names are inserted into LRMT. After all register names are mapped to local register names, an instruction is inserted into the instruction issue queue of the assigned cluster.

2.4.3 On-Demand Register Transfers

Although the functional unit assignment algorithm tends to dispatch an instruction to the functional unit cluster with the most operands, the assigned functional unit cluster often does not have all the required operands in its local register file. In this case, those missing operands have to be obtained from remote register files. The transfer of operand values whenever the need arises is called *on-demand register transfer*.

During an instruction dispatch, zero or more register transfer operations can be dispatched to transfer operand values required by that particular instruction. These register transfer operations are dispatched to instruction queues of register transfer units, which manage the arbitration of their associated register transfer networks. Although the dispatch of register transfer operations is straightforward, the issue of these operations requires dependency resolution for both source and destination operands. For example, the register transfer operation,

$$\texttt{rcopy}\quad C_{src}:R_{src}\quad C_{dst1}:R_{dst1},\ C_{dst2}:R_{dst2},\ \ldots$$

transfers a value of a local register R_{src} from a functional unit cluster C_{src} to local registers R_{dst1}, R_{dst2}, … in functional unit clusters C_{dst1}, C_{dst2}, …, respectively. This operation can be issued only when the dependency on its source operand is resolved, i.e., after the previous operation that defines $C_{src}:R_{src}$ has completed its execution. Moreover, any subsequent operations that make use of its destination operands cannot be issued until this register transfer operation has been issued. The dependency check logic is much less complicated than in conventional superscalar processors since dependency needs to be checked only between the register transfer unit and other functional units, not among all available functional units. An alternative approach to dedicated register transfer units is to dispatch register transfer operations to source and destination clusters with source clusters arbitrating for register transfer bus accesses. While this approach has simpler dependency check logic, instruction dispatch is more complex than the former approach.

The automatic insertion of register transfer operations lengthens the dependency chains of those operations involved with operands being transferred. If these extra operations require extra execution cycles to execute, the performance will degrade in terms of total execution cycles. It is possible that some register transfer operations will not cause extra cycle penalties, for example, when idle cycles are present due to cache misses, branch mispredictions, or other stalls in the execution pipeline.

2.4.4 Eager and Multicast Register Transfers

Execution cycle penalty is a major drawback of distributed register file architectures. This penalty is more severe in a non-uniform cluster configuration because some operand transfers are mandatory no matter how clever the functional unit cluster assignment algorithm is. For example, if two functional unit clusters are provided, one for simple integer operations and the other for integer multiplication, the use of an operand value, first in a simple integer operation, and later in a multiply operation, always requires a remote register transfer since these two operations cannot be executed in the same functional unit cluster. Therefore, the penalty of a non-uniform cluster configuration tends to be higher than that of a uniform configuration.

To address this problem with a non-uniform cluster configuration and the execution cycle penalty problem in general, eager and multicast register transfer mechanisms are employed. The main idea of the eager transfer mechanism is to transfer register values in advance without creating extra execution cycle penalty. Therefore, when a value is needed by a functional unit, it can be readily accessed from the local register file.

The second technique, multicasting, is an efficient mean to transfer a register value to multiple functional unit clusters in a single network transaction by utilizing broadcast through a bus or a crossbar network. Although a bus connecting several register files incurs longer delay than local point-to-point interconnects due to longer propagation delay and larger fan-out delay, a register transfer transaction is designed to be performed concurrently with normal computing operations, thus having a full clock cycle to complete its task. Furthermore, each cluster is small (with only one functional unit), and, with careful optimization, multiple buses can be used to connect subsets of functional unit clusters without loss of functionality. Therefore, a broadcast network, such as a bus or a crossbar, is a viable and efficient approach for the target architecture.

Multicasting can also be viewed as a form of eager transfer since an operand is transferred to some other functional unit clusters besides the intended destination required by an on-demand transfer. Multicasting is always used in both on-demand and eager register transfer. To dispatch an eager register transfer operation, the following components need to be determined:

- source registers,
- source/destination clusters, and
- a dispatch cycle.

In each cycle, there can be several candidate source registers. All registers that are not assigned as destination registers of any in-flight instructions comprise the set of candidate source registers. Depending on the number of register transfer network available, only a subset of these candidates must be chosen for eager transfers. Since most values are used soon after they are defined (temporal locality of operand values

[35]), the most recently defined register is chosen as the source register to be eagerly transferred on each available register transfer network

Next, the source and destination clusters need to be determined. Since a dedicated register transfer port is provided, any functional unit cluster that has a valid mapping of the source register can be chosen as the source cluster without blocking any access ports required for normal operations. Multiple destination clusters can be chosen by means of multicast. With multicast, the source register value is transferred to all local register files that are connected to the same register transfer network and that still have free local registers. All local registers that are destinations of multicast register transfers will have their temporary bit set in LRMT unless it is the primary intended destination of an on-demand register transfer. The temporary bit is cleared when the local register is later used as a source operand. Local registers with temporary bits set can be freed when a free register is needed but there is none available in the free list.

Finally, the decision needs to be made whether eager transfer operations will be dispatched in the current cycle. If too many eager transfer operations are dispatched, they can interfere with normal operations and cause unnecessary dependency in execution pipelines. In the current design, eager transfer operations are dispatched whenever the issue queue of the register transfer unit is empty. This approach ensures that eager transfer operations are immediately issued in the next cycle, and never delay the issue of other normal operations.

2.5 Performance Evaluation

The total number of execution cycles to complete a particular program on a distributed register file architecture is always equal to or higher than on a traditional central register file architecture because of extra operand transfer cycles. To quantify this performance impact, the D-DRF architecture is modeled and simulated. Its performance results are compared with a central register file superscalar architecture. The performance impact of local register file size and register transfer buses are also evaluated.

2.5.1 Simulation Methodology

Architectural models for both distributed and central register file architectures are implemented using SimpleScalar 3.0c [36]. The models are implemented based on the detailed out-of-order processor model (*sim-outorder*) provided with SimpleScalar. Both 4- and 8-way machine configurations are simulated with parameters as shown in Table 2. In addition, changes are made to emulate the distributed reservation station organization as opposed to the centralized combined reservation stations, register renaming, and reorder buffer structure. Distributed register file operations including functional unit cluster assignment, on-demand register transfer, and eager and multicast register transfer are implemented in the simulator. Note that functional unit classes are non-overlapped, and enough functional units are provided to minimize the possibility of a resource hazard (10 functional unit clusters in a 4-way configuration, and 18 functional unit clusters in an 8-way configuration). As a consequence, a large penalty is expected for distributed register file simulations. Therefore, the simulation results are considered conservative with ample room for optimization.

Table 2 SimpleScalar simulation parameters

Parameter	4-way	8-way
Fetch/decode/commit width	4	8
IntALU/IntMUL/FpALU/FpMUL/Mem	4/1/2/1/2	6/2/4/2/4
Branch predictor	Combined predictor (1K entries) of a bimodal predictor (2K entries) and a 2-level predictor (1K 2-bit counters and 8-bit global history)	
Local register file size (per FU)	32	
Issue queue and load/store queue	8-entry issue queue per FU, 16-entry load/store queue	
I-cache L1	16KB direct-mapped, 32-byte lines, 1-cycle latency	
D-cache L1	16KB 4-way set associative, 32-byte lines, 1-cycle latency	
I/D-cache L2	256KB 4-way set associative, 64-byte lines, 10-cycle latency	
Memory	64-bit bus width, 50-cycle first chunk latency, 2-cycle inter-chunk latency	

Eighteen applications from SPEC CPU2000 and MediaBench [37] are chosen for simulations based on no particular preferences. These applications and their descriptions are shown in Table 3. All applications are compiled into PISA (Portable ISA) binaries using gcc 2.7.2.3 and the default compilation options as specified by each benchmark. All compilations and simulations are performed on an x86-based machine running Linux operating system. Three architectures are modeled and simulated:

- *Base* – a traditional superscalar architecture with a central register file,
- *Drf* – a dynamically scheduled distributed register file architecture, and
- *Drf+Eager* – *Drf* with eager and multicast transfer mechanisms.

All applications are simulated for the maximum of 100 million instructions. The first 50 million instructions are fast forwarded in SPEC CPU2000 applications so that some common initialization code is skipped.

Table 3 Benchmark applications used in the simulations

Applications	Descriptions
SPEC CINT2000	
164.gzip	Compression
176.gcc	C programming language compiler
181.mcf	Combinatorial optimization
197.parser	Word processing
256.bzip2	Compression
300.twolf	Place and route simulator
SPEC CFP2000	
168.wupwise	Physics / quantum chromodynamic
171.swim	Shallow water modeling
172.mgrid	Multi-grid solver: 3D potential field
173.applu	Partial differential equations
179.art	Image recognition / neural networks
183.equake	Seismic wave propagation simulation
MediaBench	
adpcm	4-bit ADPCM coder and decoder
epic	Efficient pyramid image coder and decoder
g721	G.721 (32kbps 4-bit) voice coder and decoder
gsm	GSM (13kbps) speech coder and decoder
jpeg	JPEG image compression and decompression
mpeg2	MPEG-2 video encoder and decoder

To measure execution performance, an average instruction-per-cycle (IPC) metric is used. An average IPC is computed according to the following equation:

$$Average\ IPC = \frac{Total\ number\ of\ instructions\ committed}{Total\ number\ of\ execution\ cycles}. \tag{8}$$

Based on the average IPC metric, the performance of a distributed register file architecture can be compared to a central register file architecture using an *IPC ratio* metric, which is defined as

$$IPC\ Ratio = \frac{IPC_{DRF}}{IPC_{CRF}}. \tag{9}$$

IPC_{DRF} is the average IPC when an application is running on a distributed register file architecture, and IPC_{CRF} is the average IPC when running on a central register architecture. *IPC ratio*, therefore, represents the execution performance of a distributed register file architecture when compared to a central register file architecture with a similar configuration. In addition, the effectiveness of the eager and multicast transfer technique is measured as its capability to reduce execution cycle penalty incurred by distributed register file operations. This *reduction in execution cycle penalty* is computed from the following equation:

$$Reduction\ in\ execution\ cycle\ penalty = \frac{cycle_{Drf} - cycle_{Drf+Eager}}{cycle_{Drf} - cycle_{Base}} = \frac{CPI_{Drf} - CPI_{Drf+Eager}}{CPI_{Drf} - CPI_{Base}} \qquad (10)$$

2.5.2 Average IPC Comparisons

In this experiment, all applications are simulated to obtain average IPC values when executing on *Base*, *Drf*, and *Drf+Eager* architectures. The average IPC comparisons for 4-way and 8-way machine configurations are shown in Figure 13 and Figure 14, respectively.

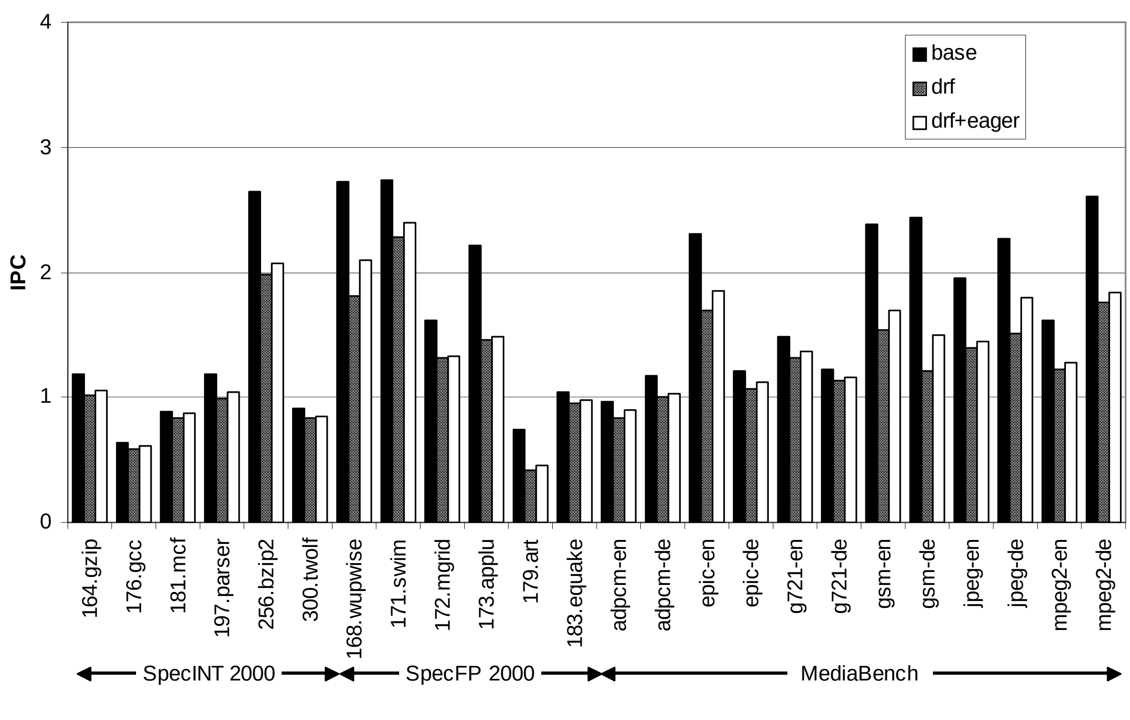


Figure 13 Raw IPC comparisons for a 4-way machine configuration

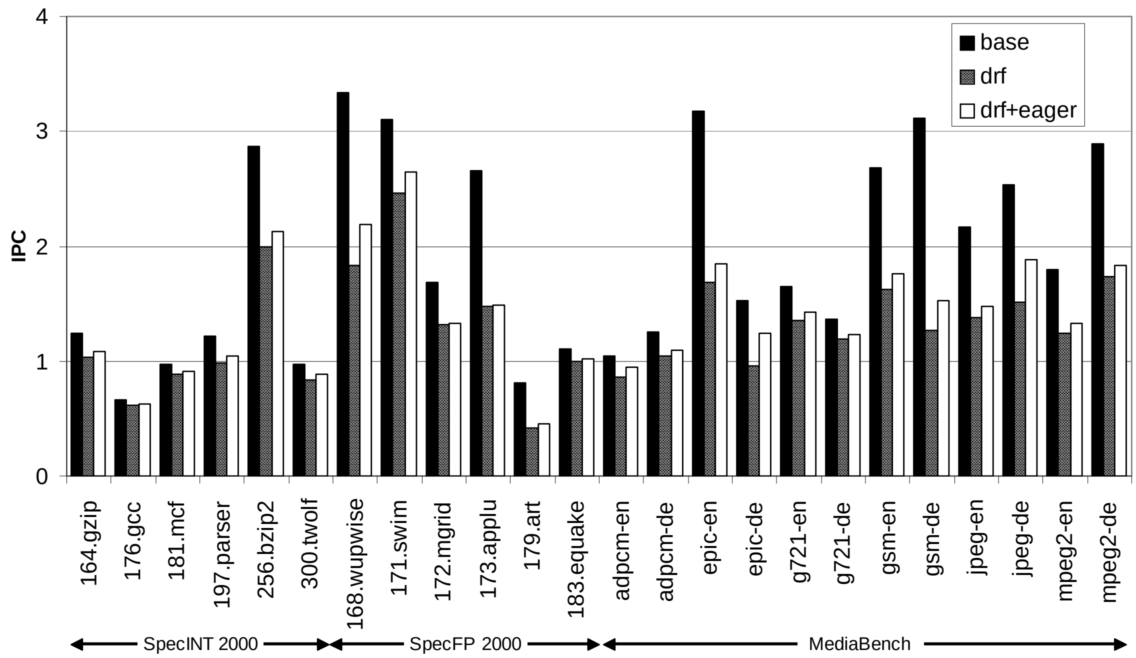


Figure 14 Raw IPC comparisons for an 8-way machine configuration

As seen from the results, the average IPC values for distributed register file architectures (*Drf* and *Drf+Eager*) are consistently lower than those for central register file architectures (*Base*). Moreover, the results for *Drf+Eager* show slightly higher IPC values than *Drf*. This result shows the effectiveness of the eager and multicast transfer mechanisms, which reduce the execution cycle penalty by 27% on average.

The average IPC ratios are summarized in Table 4. The performance penalty in the 8-way machine configuration is more than in the 4-way configuration. This is due to the larger number of functional unit clusters in the 8-way configuration. Comparing different sets of applications, SPEC CINT2000, which is a set of symbolic integer applications, incurs the least penalty when executing on a distributed register file architecture. SPEC CFP2000, which is a set of scientific floating-point applications, incurs the most penalty, while MediaBench, a set of multimedia applications (most of them integer), incurs moderate amount of penalty.

Table 4 IPC ratio results (in percentage)

Application	IPC Ratio	
	4-way	8-way
SPEC CINT2000	92%	90%
SPEC CFP2000	74%	69%
MediaBench	83%	76%

Penalty in the execution pipeline, caused by other factors, can be effectively utilized to hide penalty caused by distributed register file operations. This is because any stall or idle cycles can be used for on-demand or eager register transfers without incurring extra execution cycles. For example, memory latency in future processor generations has been continually growing as the gap between CPU speeds and DRAM speeds continue to grow. Figure 15 shows absolute changes in IPC ratio values for selected applications from SPEC CPU2000 when the memory latency is increased from 50 to 300 cycles. Note that IPC ratio values are in the range (0,1].

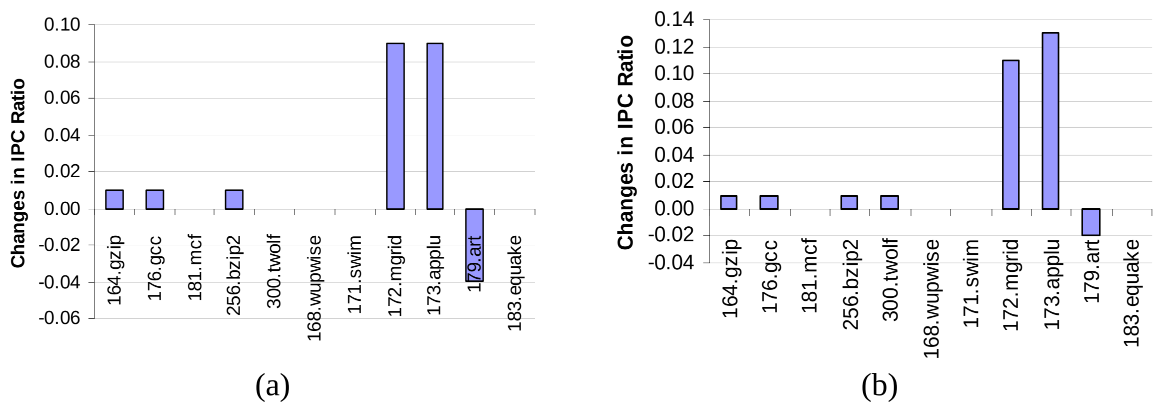


Figure 15 Absolute changes in IPC ratio values when the memory latency is increased from 50 to 300 cycles for (a) 4-way configurations, and (b) 8-way configurations

With increased memory latency, all applications, except 179.art, experience either less IPC penalty or the same level of IPC penalty caused by distributed register operations. Increased penalty is observed in 179.art due to excessive cache misses (>50%). While stall cycles from cache misses can be effectively used to absorb register communication penalties, register communications that depends on a memory operation cannot be issued during the stall cycles of that particular memory operation. Since a large percentage of memory operations in 179.art results in cache misses, dependency chains of these long-latency memory operations easily become the critical path (the longest collection of dependency chains that determines the lower bound on the total execution cycles). Therefore, 179.art becomes highly sensitive to extra register transfer operations that are needed between memory operations in the same dependency chain since they cannot be absorbed within memory stall cycles.

2.5.3 Relationships between IPC Penalty and Application's *Base* IPC

As seen in Table 4, IPC ratios and thus IPC penalty (1 - IPC ratio) of distributed register file architectures depend on two factors: the number of functional unit clusters and application characteristics. The more functional unit clusters, the more data routing operations are needed. The latter factor, application characteristics, is difficult to quantify precisely. Instead, the IPC of the baseline architecture (*base* IPC) is used here to approximate application characteristics. This relationship is shown in Figure 16. The plot shows that applications with high IPC tend to incur more IPC penalty when executed on a distributed register file architecture.

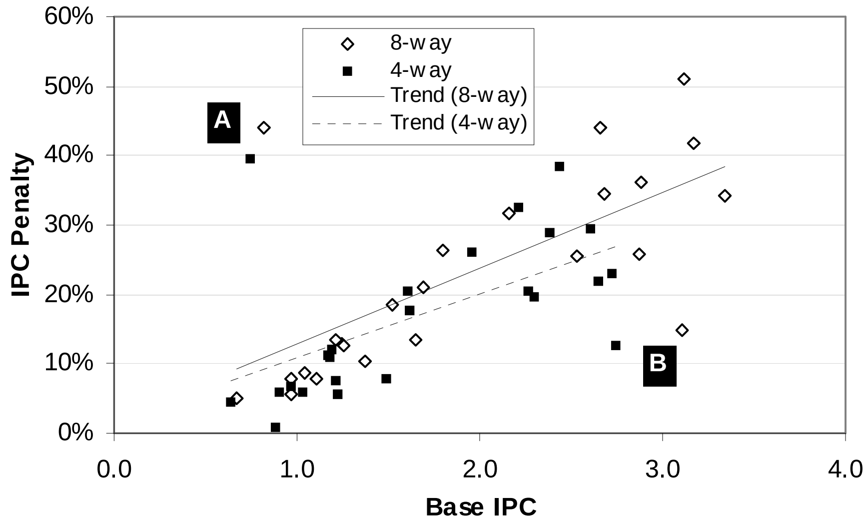


Figure 16 Relationships between IPC penalty and application's *Base* IPC

Two exceptions are observed from Figure 16. First, high IPC penalty is incurred for 179.art (A) due to its excessive L1 data cache misses (>50% miss rate) compared to less than 6% miss rate in other applications. This makes it highly sensitive to extra data routing operations. Second, low penalty is incurred for 171.swim (B) despite its higher *base* IPC. This is because the distance from the instruction that produces the value until the first instruction that uses the value (*Def-FirstUse* distance) is long for most of its data values. Long *Def-FirstUse* distances allow more data routing operations to be dispatched without incurring extra cycle penalties. The cumulative distribution plot of *Def-FirstUse* distance of this application is shallower than other applications as shown in Figure 17.

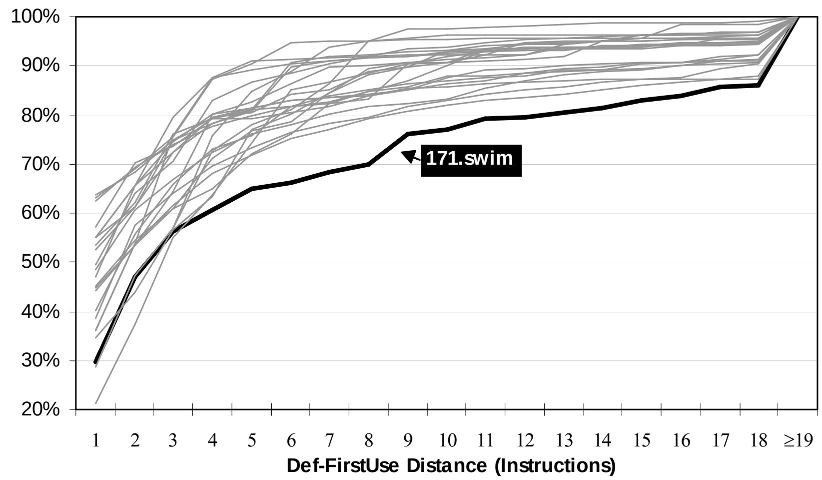
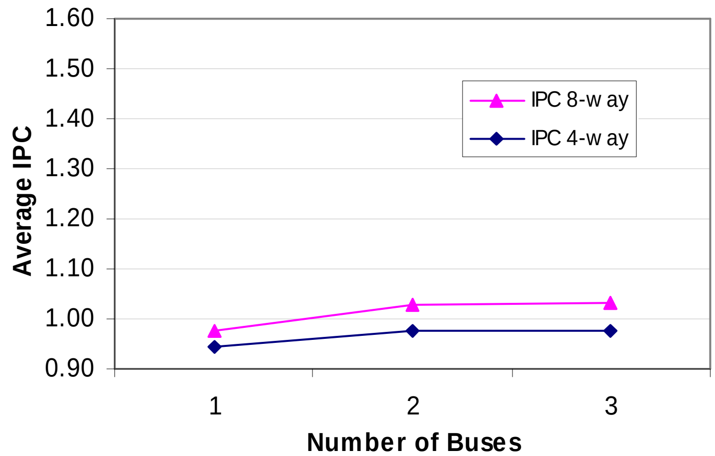


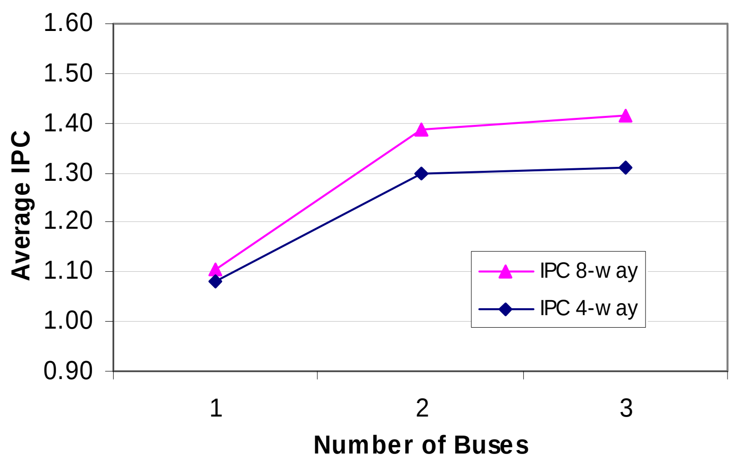
Figure 17 Cumulative distribution of the *Def-FirstUse* distance for all applications (171.swim is highlighted with a bold line)

2.5.4 Performance Impact of the Number of Register Transfer Buses

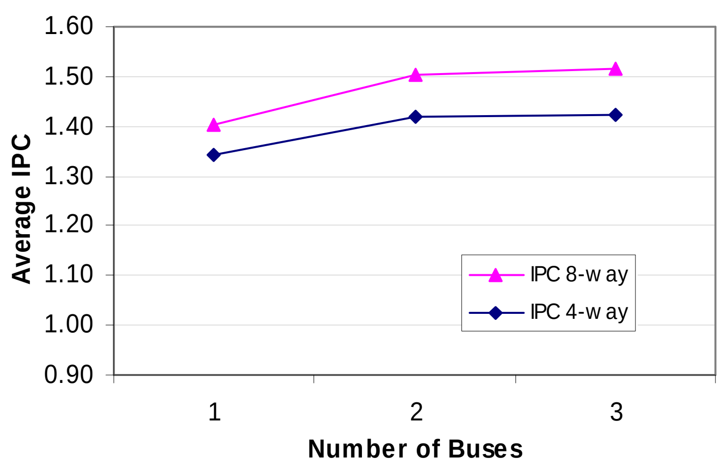
The number of register transfer buses specifies the number of register transfer operations that can be issued concurrently in each cycle. In all simulations discussed so far, only one bus is present. This can be a limiting factor if many register transfer operations are blocked waiting for the bus. The performance results with one, two, and three register transfer buses are shown in Figure 18.



(a) SPEC CINT2000



(b) SPEC CFP2000



(c) MediaBench

Figure 18 Performance impact for different number of register transfer buses

Considering MediaBench applications, with a single bus, the average number of register transfer operations per cycle that are ready but have to be stalled due to the unavailability of the transfer bus (*RcopyBlocked*) is 0.42 and 0.51 for 4- and 8-way configurations, respectively. When the number of buses is increased to two, the *RcopyBlocked* values are reduced to 0.14 and 0.20 for 4- and 8-way, respectively. A slightly larger performance gain is observed in an 8-way configuration since a larger number of register transfer operations are demanded by a more distributed configuration. Consequently, the average IPC also increases by 5.63% and 6.05% when the number of buses is increased to two and three respectively in a 4-way configuration, and by 7.28% and 8.08% respectively in an 8-way configuration. As can be seen, increasing the number of buses from two to three shows only a slight increase in performance.

Similar trends are observed for SPEC CPU2000 applications. A larger performance gain is observed for SPEC CFP2000 applications than for SPEC CINT2000 applications. This is because SPEC CFP2000 applications have higher IPC than SPEC CINT2000 applications, and applications with higher IPC have more register transfer operations blocked waiting for the transfer bus than applications with lower IPC. Note that an additional transfer bus requires an additional access port in local register files, which increases the implementation cost of local register files.

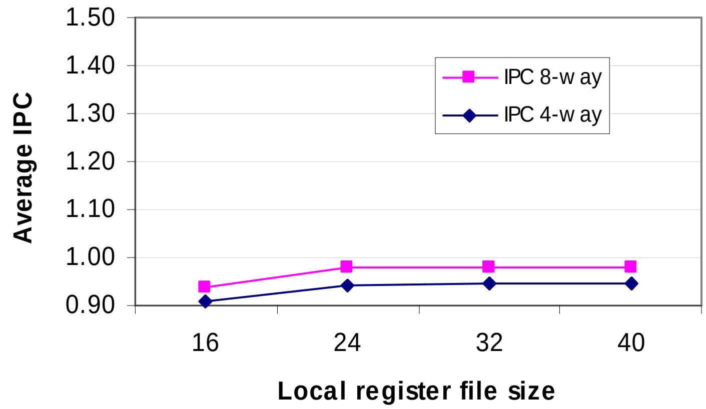
2.5.5 Performance Impact of the Local Register File Size

The size of each local register file is another important design parameter. If it is too big, implementation cost will be high. On the other hand, if it is too small, execution performance can be severely impacted due to stall in the pipeline waiting for registers to be freed.

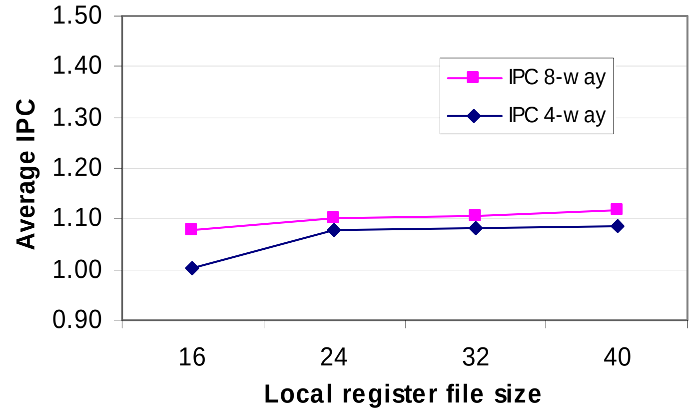
Figure 19 shows the average IPC when each local register file size is increased from 16 to 24, 32, and 40. As expected, performance is slightly increased as register file size is increased. However, no significant performance improvement is observed beyond 24 registers.

2.6 Implementation Cost Evaluation

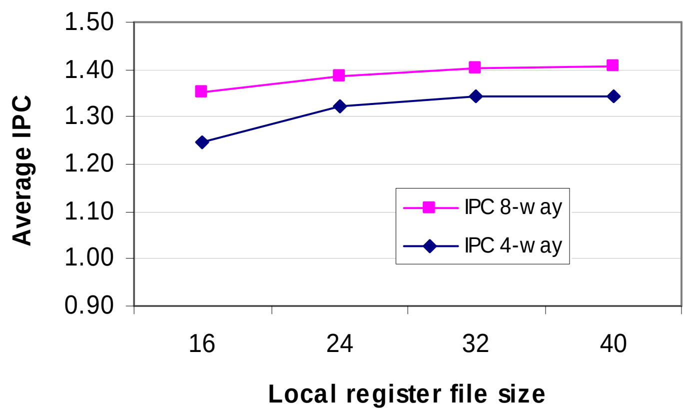
The primary motivation of a distributed register file is to reduce operand transport complexity as the number of functional units is increased. A central register file and a bypass network are two critical components that are cycle-time limited. They are especially critical in deep submicron technologies since they are wire-limited and wire does not scale as well as transistors. In this section, the complexity of these structures in a distributed register file configuration with four and eight functional unit clusters is estimated and compared to a traditional central register file configuration that allows concurrent accesses from four and eight functional units to a central register file, respectively.



(a) SPEC CINT2000



(b) SPEC CFP2000



(c) MediaBench

Figure 19 Performance impact of local register file size

Register file access time is estimated using CACTI 3.2 [24] with 100-nm technology parameter from Berkeley Predictive Technology Model [25][26]. All registers are 32-bit. For a 4-way configuration, an 80-register 12-port central register file is compared to four 4-port local register files, each with 32 registers. Similarly for an 8-way configuration, a 128 24-port central register file is compared to eight 4-port local register files with 32 registers. The results from CACTI, which show implementation cost of a distributed register file relative to a central register file, are shown in Table 5. The results show significant cost saving in terms of area, power, and operand access delay.

Table 5 Implementation cost of a distributed register file (data array only)

	4-way	**8-way**
Data Array Area	$0.35A_{CRF}$	$0.14A_{CRF}$
Data Array Energy	$0.34E_{CRF}$	$0.13E_{CRF}$
Access Time	$0.77D_{CRF}$	$0.59D_{CRF}$

As the number of functional unit clusters increase from four to eight, area is reduced by an even larger margin (from $0.35A_{CRF}$ down to $0.14A_{CRF}$) because a large number of extra access ports in an 8-way configuration demands a significantly large amount of die area while the number of ports does not increase in a distributed register file configuration. A similar result is obtained for register files' energy consumption and delay.

In modern high-performance processors, such as Alpha [11][12][13] and Itanium 2 [10], a conventional central register file design is considered the single most critical bottleneck for cycle time improvement. Assuming that the processor's cycle time scales proportionally to that of the register files [4] and no other factors contributes to the improvement in cycle time, performance improvement in terms of instructions per second (IPS) can be estimated. The following equation computes an improvement in IPS when a distributed register file architecture is implemented.

$$Increase\ in\ IPS\ (\%) = \frac{IPS_{DRF} - IPS_{CRF}}{IPS_{CRF}} = \frac{IPC\ Ratio}{CycleTime\ Ratio} - 1 \qquad (11)$$

The *CycleTime Ratio* is the ratio of the cycle time of a distributed register file architecture and the cycle time of a central register file architecture, and can be obtained from Table 5.

Using the above equation, which considers both IPC penalty and cycle time improvement, improvements in IPS are computed and summarized in Table 6 for both 4- and 8-way machine configurations. The results show speedup in all benchmarks except for SPEC CFP2000 in a 4-way configuration, which incurs slight performance degradation. Moreover, higher speedup is observed in a more distributed organization (8-way) since its register file implementation is more efficient.

Table 6 Performance improvements in terms of instruction per second (IPS)

	4-way	**8-way**
CINT2000	19%	53%
MediaBench	8%	29%
CFP2000	-4%	17%

For all cost evaluations in this section, note that the total number of registers in a distributed register file configuration increases by a larger proportion (128 to 256) than in a central register file configuration (80 to 128) when the issue width is increased from four to eight. In an actual implementation, architectural parameters, such as functional unit and register file size, should be optimized through more extensive simulations to achieve desirable performance and cost savings.

2.7 Conclusion

Current operand transport design, which utilizes a central register file and an operand bypass network, demands long interconnects, creating a critical problem in processor design in future deep submicron technology. To address this problem, a fully distributed register file architecture is presented, which employs multiple small register files distributed among functional units. These local registers are shown to reduce register file implementation die area by 86%, operand access delay by 41%, and energy consumption by 87% for a processor configuration with eight functional unit clusters. Processor cycle time can be potentially improved through this significant reduction in operand access delay.

The challenge in a distributed register file architecture is reducing the execution performance degradation. This is due to extra cycles required to transfer data values among local register files. Simple and effective eager and multicast transfer mechanisms are used to reduce execution cycle penalty by 27% on average. The results show average IPC penalties of 8-26% for a 4-way configuration, and 10-31% for an 8-way configuration, which are significantly less than the potential improvement in cycle time as mentioned above. The penalty is 8-10% for general symbolic applications (SPEC CINT2000). Considering both IPC penalty and cycle time improvement, overall performance speedup is observed for most applications.

CHAPTER 3

STATICALLY SCHEDULED DISTRIBUTED REGISTER FILE PROCESSOR ARCHITECTURES

3.1 Summary

Distributed register file operations can be supported in software by generating code specifically for distributed register files at compile-time. With this approach, hardware implementation is simplified, and code can be thoroughly analyzed to generate effective schedules. Drawbacks include increased code size and inability to exploit run-time conditions to reduce execution cycle penalty from distributed register file operations.

Code generation tasks related to distributed register files are performed in three phases. First, instruction are scheduled and, at the same time, assigned to available functional units. A dependence-based cluster assignment algorithm is used, which assigns a given instruction to a cluster with the most operands. In the second phase, producer-consumer chains are extracted, and data routing operations are scheduled. Finally, registers are allocated from distributed register files and assigned to all register operands.

Code generation techniques are implemented as a code retargeting tool, which transforms conventional code for a central register file to distributed register file code. Applications from Mediabench are retargeted and simulated. Experimental results show that code size is increased by 36% on average, and the average IPC ratio is 77% compared to a central register file architecture with a similar machine configuration.

3.2 Introduction

In contrast to the dynamic approach described in the previous section, the statically scheduled fully distributed register file architecture (S-DRF) requires no special processing at the microarchitectural level since it is taken care of at the software level. The presence of multiple local register files, one for each functional unit, is visible at the instruction set architecture (ISA) level, and register transfer operations are scheduled as machine instructions, which are treated in the same way as other ordinary machine instructions.

Three changes are needed at the ISA level. First, a functional unit cluster identification needs to be provided for every instruction so that each instruction can be correctly dispatched to the intended functional unit cluster. This is more critical in a distributed register file architecture because this functional unit identification implicitly specifies the particular local register file that can be accessed.

Second, local register namespaces are used for all register operand names. Since a particular instruction that is destined to a particular functional unit can only access registers from its attached local register file, register names are local to that particular functional unit cluster. For example, R_1 in functional unit cluster 1 is a different register from R_1 in functional unit cluster 2.

Third, a new instruction (rcopy) is introduced to transfer register values between local register files. The format of an rcopy instruction is as follows.

$$\text{rcopy} \quad C_{src}{:}R_{src} \quad C_{dst1}{:}R_{dst1},\, C_{dst2}{:}R_{dst2},\, \ldots$$

The first parameter is the source register, which can be uniquely specified with the combination of a functional unit cluster id (C_{src}) and a register name (R_{src}). The destination registers are specified as subsequent parameters. There can be multiple destinations, all in distinct local register files. Transfers to all destinations can be performed concurrently through multicasting. Typically, a maximum of three destinations are observed, which is affordable in terms of instruction encoding efficiency. However, most transfers use only one or two destinations.

In the following section, related research is described followed by the code generation framework for S-DRF. Performance evaluations are presented in subsequent sections.

3.3 Related Work

The earliest use of distributed register files and clustered microarchitectures is in VLIW processors. Several early VLIW machines have extremely wide issue width, which renders a central register file design infeasible. For example, the widest model of the Multiflow Trace machines [38,39] can schedule up to 28 operations per cycle (8 integer operations in each half of a cycle, 8 floating-point operations a cycle, and 4 branches a cycle). This would require integer and floating-point register files with 24 access ports, which are not realizable using the current technology of the late 1980s. To address this problem, Multiflow designed the Trace machines using the 7-wide cluster as the basic building block. Within a Trace 300 cluster, each functional unit (integer or floating-point) is connected to a local register file (4R/3W ports), and operands can be communicated among register files through a global bus. In Trace 500, which was designed using a newer technology, two functional units share a local register file (4R/4W ports) with twice the size of the local register file of Trace 300.

The MultiVLIW architecture [40][41] employs a similar approach with a single register transfer bus connecting all clusters. The architecture provides basic support for broadcast or multicast transfers through the register transfer bus using an instruction encoding as shown in Figure 20. For each cluster, two additional fields are used to specify the source and destination register for the register transfer bus. In each cycle, there can be only one source and possibly several destinations. This encoding scheme has two limitations. First, the encoding is inefficient since all but one of the OUT fields are unused in each transfer operation. Another limitation is that this encoding supports only a single transfer bus. Other network types are not supported.

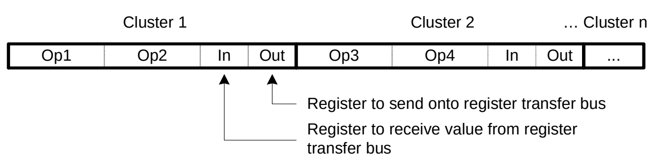


Figure 20 MultiVLIW instruction encoding with register transfers

Multiple-banked register files and distributed register files are explored in the context of VLIW processor architectures and modulo scheduling [42][43][44]. Recent work [45] applies an instruction replication technique to improve performance of loop scheduling in a clustered microarchitecture.

Stanford's Imagine [15][16] is a streaming media processing architecture that utilizes distributed register files in its SIMD arithmetic clusters. An Imagine processor is composed of eight arithmetic clusters, each of which operates in VLIW mode. The organization of an Imagine arithmetic cluster is shown in Figure 21. Local register files within an arithmetic cluster are further distributed among functional unit's inputs resulting in one register file per input port. Although this distribution creates tighter restrictions on operand placements, it is compensated by the global write mechanism, which permits a functional unit to write its result to any register file directly.

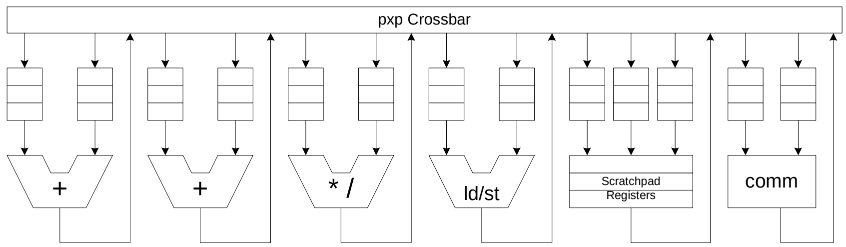


Figure 21 Imagine arithmetic cluster

In addition, several recent embedded and DSP processors employ a distributed register file organization. Examples include TI's C6x architecture [46], ADI TigerSharc [47], HP's Lx [48], and Equator's MAP 1000 [49].

The effectiveness of a clustered VLIW machines relies primarily on the capability of compilers to schedule operations to appropriate clusters. Since this problem is NP-complete, several heuristics have been used. In the following, cluster assignment algorithms based on greedy heuristics, graph partitioning, and search and iterative improvements methodology are described.

3.3.1 Greedy Heuristic Approach to Cluster Assignment

Bottom-up greedy (BUG) [50] is the first well-known cluster assignment algorithm for clustered microarchitectures. An enhanced version of BUG is used in the Multiflow compiler for the Multiflow Trace machines. BUG traverses the data flow graph (DFG) upwards from the exit node towards the entry nodes in the depth first search order. At each step, a set of candidate clusters is estimated based on the estimated set of candidate clusters of its previous (downstream) nodes. Once the entry node is reached, BUG works its way back along the DFG performing the final cluster assignment. The two-pass approach is used because a good assignment is effected by operand locations of both the upstream and downstream portions of the DFG.

BUG uses a form of greedy heuristic techniques that try to minimize remote register communications. Other algorithms based on greedy heuristics include unified assigned and scheduled (UAS) [51], communication scheduling [52], and combined cluster assignment, register allocation, and instruction scheduling (CARS) [53].

UAS performs cycle-based instruction scheduling and cluster assignment simultaneously in the same phase. At each cycle, operations are scheduled using a list scheduler with additional constraints of cluster and register transfer network (if required) availability. The algorithm considers both workload imbalance and remote register communication minimization.

Communication scheduling is based on UAS, but differs in its scheduling of register transfer operations. Register transfer operations are scheduled by incrementally allocating register transfer paths rather than scheduling in a single step at the consuming instructions. At the producing operation, the valid write path that is not in conflict with other operations in the same cycle is determined. Later, when the consuming operation is scheduled, the valid read path is chosen to construct a route from the producer to the consumer. If the two paths cannot meet at the same register file, one or more copy operations are scheduled between the two communicating operations. Backtracking and rescheduling are required in case copy operations cannot be scheduled, which significantly increases the algorithm's complexity.

Finally, CARS tries to address the phase coupling problem by perform all tasks simultaneously in a single phase including cluster assignment, instruction scheduling, and register allocation. This approach reduces overhead from phase separation and ordering, however, the complexity is significantly higher than other greedy-based methods.

3.3.2 Graph Partitioning Approach to Cluster Assignment

With this approach, a graph representation is created and partitioned into several components based on some objective function. Heuristics are available for partitioning graphs resulting in minimum or maximum cut set (the total value of all edges cut) and minimum size differences of all subgraphs (balanced partitioning). The clustering approaches based on graph partitioning are Limited-Connectivity VLIW (LC-VLIW) [54] and register component graph (RCG) [55]. These two approaches assume uniform cluster configurations but differ in the way their graph representations are constructed.

The LC-VLIW approach uses a data flow graph (DFG) with each node representing an operation and each edge a flow of data. For an architecture with k clusters, the DFG is partitioned into k subgraphs with minimum cut set and balanced subgraphs. The edges that have been cut represent all register transfer operations required in the final code schedule.

In the RCG approach, each node in a RCG represents each live range. An edge connects two nodes that form a source/destination pair of the same operation. The k-way graph partitioning is then performed on the RCG with minimum cut set and balanced subgraph objectives. The priority is to keep values required by a single operation in the same cluster to reduce register transfer overhead.

These graph partitioning approaches have moderate complexity. Common partitioning heuristics, such as [56], have time complexity of $O(k|V|^2)$, where k is the number of partitions and $|V|$ is the number of nodes in the graph.

3.3.3 Searching and Iterative Improvement to Cluster Assignment

Although complete solution-space searching is not feasible due to NP-completeness, partial searching with heuristics can be employed. These searching or iterative improvement approaches have higher complexity than previous approaches. Example of cluster assignment methods in this class are PCC [57] and FACTS [58][59].

PCC establishes an initial solution based on a greedy heuristic similar to BUG. First, a data flow graph (DFG) is partitioned based on its connectivity and a predetermined maximum partition size. These partitions are then assigned to available clusters. At each step of the assignment, a cluster that is least loaded is chosen so that the initial solution that has balanced workload among clusters can be expected. Once the initial solution is obtained, the iterative improvement phase is invoked by swapping partitions among available clusters and measured the expected schedule length. Schedule length is determined using a simplified list scheduler.

Another approach used in the FACTS framework involves modeling datapath constraints using a conflict graph and searching for a feasible allocation of resources that satisfies these constraints. A datapath conflict graph is constructed from a DFG with some additional nodes inserted between any two connected nodes of the DFG. These additional nodes include a node representing all valid register files between the two operation nodes and one or more constraint nodes, each representing an invalid path between an operation node and the register file node. With a datapath conflict graph, a graph coloring algorithm can be used to determine a valid cluster assignment for a particular data flow graph. The final assignment is obtained by repeating the following steps for each DFG until cluster assignments for all DFGs are determined: creating a datapath conflict graph, perform graph coloring, and backtrack if no valid coloring is found. This method is complicated by the requirement for backtracking and also the complexity of the graph coloring stage since the datapath conflict graph tends to be large.

3.4 Code Generation Framework

To generate code for a distributed register file architecture, two important tasks need to be performed. First, all instructions must be explicitly assigned to functional unit clusters. Second, register transfer instructions need to be scheduled to transfer register values appropriately to produce correct results. The optimal code schedule is, however, difficult to achieve in practice since code scheduling and code partitioning are both NP-complete problems. Several simple heuristics are used to generate good results efficiently.

The code generation technique for S-DRF is depicted in Figure 22 in the form of the retargeting process, which transforms the original assembly code into new assembly code with distributed register file support. Code is first analyzed to extract a control flow graph, a data flow graph, and register liveness information. These intermediate representations form the basis for the three major tasks of: 1) code scheduling and functional unit assignment, 2) data routing, and 3) register allocation. These three tasks are described in the following sections.

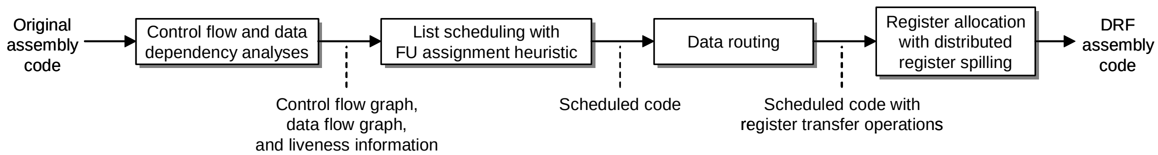


Figure 22 Code retargeting process

3.4.1 Code Scheduling and Functional Unit Assignment

A code scheduler is used to schedule or re-order instruction sequences to achieve high performance through parallelism. One of the most critical tasks of a code scheduler for a distributed register file architecture is functional unit assignment. The result of the assignment significantly impacts the performance of the final code schedule. If instructions are assigned mostly to some particular functional units, the length of the final code schedule may increase due to congestion among those functional unit resources. This is commonly known as a workload imbalance problem. However, if instructions are dispersed among too many functional units, too much data routing between local register files will be needed. Since code scheduling and partitioning are NP-complete problems, these two objectives, namely minimizing functional unit imbalance and minimizing extra data routing operations, are targeted simultaneously in most cluster assignment algorithms to approximate the optimum solution.

A simple solution is presented for S-DRF as shown in the simplified pseudo-code in Figure 23. This approach is based on the unified assign and schedule (UAS) algorithm [51], which performs functional unit assignment during the code scheduling phase. A list scheduling algorithm is used with a dependence-based functional unit assignment heuristic. More complex scheduling algorithms can be used to extract more ILP from the code without substantial modifications to the presented framework. The algorithm starts by creating a list of candidate instructions, which are all instructions in the basic block that have no dependency (or all dependencies resolved). Instructions in the list can be ordered based on some priority function such as the distance to the last instruction in the data dependency graph (which gives high priority to instructions in the critical path). Then, a candidate instruction is chosen from the list, assigned to an available functional unit, and scheduled into the current cycle. Instructions are scheduled until the list is exhausted or no more instructions can be accepted due to other dependencies or resource constraints, in which case, it advances to the next cycle and repopulates the list with any additional candidate instructions.

```
compute a data flow graph
compute priority functions
while ( unscheduled op exists )
        update a list of ready instructions
        if ( issue slot is available )
                inst = next ready instruction with highest priority
                {fu} = all free FUs that can execute inst
        if ( inst not valid or {fu} is empty )
                advance to the next cycle
                update resource usage information
        else
                chosenFu = cluster_assignment( inst, {fu} )
                schedule inst onto chosenFU in the current cycle
                update resource usage information
```

Figure 23 Code scheduling algorithm for S-DRF

A simple heuristic is used for functional unit assignment. To reduce data routing overhead, an instruction will be assigned to the available functional unit that has most of the instruction's operands. In other words, the functional unit to which most of the instruction's predecessor instructions are assigned is chosen. If there are many such candidate functional units, the one that is assigned to a predecessor instruction that has been scheduled most recently is chosen. This is to ensure that many cycles are available to schedule data routing operations in the data routing phase. For example, in Figure 24, two candidate functional units for instruction X are FU1 and FU2, both computing one operand of instruction X. FU1 has instruction A, which is one of X's predecessors, scheduled in time slot 3 while FU2 has instruction B, the other predecessor of X, in time slot 1. If X is assigned to FU1, three time slots are available to schedule data routing operations from FU2 to FU1. If, however, X is assigned to FU2, only one time slot is available. Therefore, the heuristic will assign X to FU1 since it provides more opportunity for data routing without adding an extra time slot. If more than one functional unit is qualified, the one with the lightest load is chosen. If there is still a tie, a functional unit can be randomly chosen. The last two tie-breaking rules are provided for completeness. Based on the empirical simulation results, they do not have significant impact on the final code schedule.

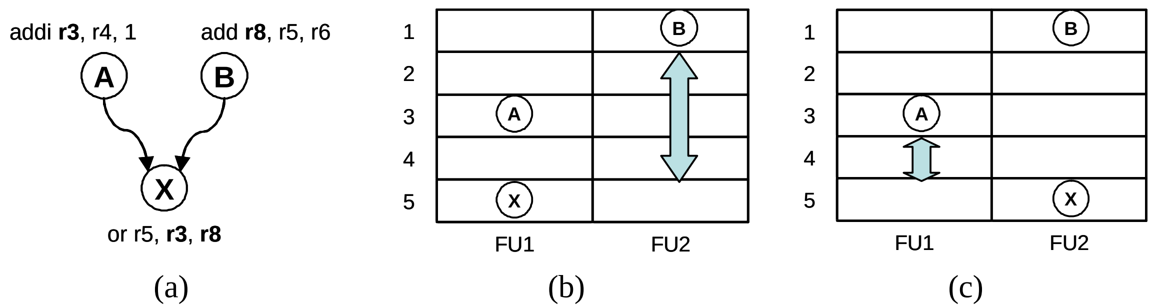


Figure 24 The effect of functional unit assignment to the range of candidate time slots for register transfer operations: (a) data dependence graph, (b) candidate range when X is assigned to FU1, and (c) candidate range when X is assigned to FU2

The functional unit assignment heuristic helps reduce extra data routing operations, which is one of the two objectives mentioned above. The other objective (to reduce functional unit imbalance) is achieved by performing the code scheduling phase before data routing and register allocation phases, and by assuming a central register file organization. Because of this, the schedule has the most flexibility in generating code with as much parallelism as possible without being limited by partial connectivity of register files and register allocation constraint. The functional unit assignment heuristic is illustrated in Figure 25.

```
function cluster_assignment

inst = the instruction to be assigned a functional unit
{fu} = all free FUs that can execute inst

pred(inst) = predecessor instructions of inst

{chosenFu} = all FUs in {fu} that have the most operands of inst
if ( more than one member in {chosenFu} )
        {chosenFu} = all FUs in {chosenFu} that have some
                pred(inst) scheduled in the most recent cycle
        if ( more than one member in {chosenFU}
                {chosenFu} = all FUs in {chosenFu} with the lightest load

return the first FU in {chosenFU}
```

Figure 25 Cluster assignment algorithm for S-DRF

3.4.2 Data Routing with Multicast Transfer

The purpose of the data routing phase is to insert register transfer operations as needed into the code schedule. Since the code has been scheduled prior to the data routing phase, no code movement is allowed, which greatly simplifies the overall approach.

The objective of the data routing algorithm is to incur the minimum number of extra cycles for register transfer operations. Since a dedicated register transfer unit is provided, register transfer operations can often be scheduled concurrently with normal operations so extra cycles are not needed. However, there are cases in which an extra cycle is needed, for example, when too many register transfer operations are required to be scheduled into a single cycle (resource constraint), or when a value is produced and is immediately used in the following cycle.

The data routing algorithm is shown in Figure 26. It begins by examining each register value and searching for a suitable cycle to schedule the needed register transfer operations. For each such value, a range of candidate time slots is established. These are all time slots from when the value is produced until it is first used. Once the range is established, all the overlapping basic blocks are annotated with all the functional units that consume the data value along the path. Then, a search is made starting from the producing instructions along the path to find an available time slot to schedule an rcopy instruction. If no cycle can be used, an extra cycle is inserted into the schedule.

When scheduling an rcopy instruction, if the annotation shows that multiple functional units need the data value, a single rcopy instruction can be used to transfer the data value to multiple destinations via multicasting.

```
foreach reg in { temporary register variables }

    // FU consumption annotation

    clear all annotation information
    foreach bb_use in { basic blocks that consume reg }
        fu = FU that consume reg in bb_use
        if ( there is a definition of reg in bb_use before any use of reg )
            {bb} = { ∅ }
        else
            {bb} = all basic blocks upstream from bb_use until
                a basic block that produces reg is reached
                (including bb_use and all producer basic blocks)
        add fu to annotation of all basic blocks in {bb}

    // schedule data routing operations

    foreach inst_def in { instructions that produce reg }
        fu_def = FU that produces reg
        {cycle} = all cycles following inst_def but before the instruction that
            consumes reg is reached or until a basic block with
            zero or multiple upstream/downstream is reached (not
            considering basic blocks with no annotation information)
        cycle = first cycle in {cycle} with a free rcopy slot
        if ( cycle is not valid)
            cycle = add a new cycle after inst_def
        schedule [ rcopy Cs:Rs Cd1:Rd1,Cd2,Rd2,… ] in cycle
            Cs is fu_def
            Rs,Rd1,Rd2,… is reg
            Cd1,Cd2,… is from the annotation information
        {bb} = all basic blocks from cycle until a basic block that
            produces reg (excluding inst_def) is reached
        remove Cd1,Cd2,… from annotation of all basic blocks in {bb}
```

Figure 26 Data routing algorithm for S-DRF

3.4.3 Register Allocation with Distributed Register Spilling

The final step is to perform register allocation separately on each local register file using a conventional graph-coloring algorithm [60,61]. For each local register file, the algorithm starts by creating an interference graph showing all register values that have overlapping live ranges. Then, a graph coloring algorithm is used to color this interference graph with the maximum number of colors being the total number of registers in the local register file being allocated.

Typically, when more registers are needed to be allocated than available in a particular local register file, values are spilled to memory and loaded back when they are later used. Spilling to memory, however, requires data to be transferred to the memory unit (if not already there), which may involve extra register transfer operations. Moreover, communications with memory can incur long latency.

To address this inefficiency, one approach is to increase the local register file size to reduce the need for spilling. An alternative is to utilize available capacities in other local register files and perform distributed register spilling to free up some local registers. The distributed register spilling approach results in better utilization of local register resources than the spilling to memory approach. The traditional method of spilling to memory is used as the last resort when all local register files are fully utilized.

3.5 Code Retargeting and Execution Performance Evaluation

In this section, the effectiveness of the code generation mechanism for S-DRF is evaluated through simulations. Code generation algorithms are implemented as a retargeting program, which transform an assembly code written for a central register file to a new assembly code suitable for distributed register file architectures. Performance is measured through cycle-accurate simulations implemented based on the simulation framework presented in the previous chapter. Therefore, performance results can be directly compared with the results of D-DRF evaluations to assess the effectiveness of the dynamic and static approaches to distributed register files.

3.5.1 Simulation Methodology

Figure 27 shows the simulation flow, which consists of two phases: code retargeting and performance simulations. In the code retargeting phase, a retargeting tool has been created in C++ to retarget assembly code generated by the PISA compiler suite into a new code for S-DRF simulations. All code generation techniques described in the previous sections are implemented and integrated into a single retargeting program based on the structure shown in Figure 22. The output of the retargeting program is the original assembly code, which is rescheduled and annotated with information pertaining to data routing and static scheduling. Data routing operations are generated as *nop* instructions with a special annotation bit for data routing, and are treated specially by the simulator to emulate the effect of the real data routing operations. Information about the static code schedule is embedded in the generated code using the concept of instruction group similar to the variable-length VLIW instruction format, such as used by the IA-64. An instruction group is a group of one or more independent instructions that can be executed together in the same time slot. Instruction groups can be distinguished by explicitly marking the last instruction in each group with a special annotation bit recognized by the simulator. With this approach, the generated code is not bound to a specific machine organization and is more efficient in terms of code size and operation encoding. Moreover, the code annotation mechanism help simplify the simulation process since existing tools (PISA assembler, linker, and binary loader) can be immediately used without modifications as illustrated in Figure 27.

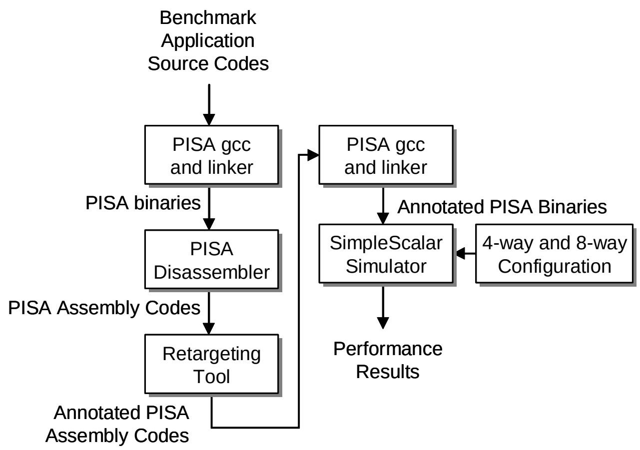


Figure 27 The simulation flow for statically scheduled distributed register file simulations

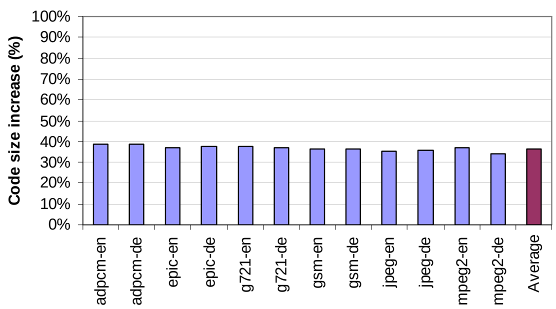
In the performance simulation phase, the retargeted code is simulated on a SimpleScalar simulator. Architectural models for both distributed and central register file architectures are implemented using SimpleScalar 3.0c [36] based on *sim-outorder* model with the out-of-order execution capability disabled. Additionally, the simulator has been extended to recognize instruction groups and data routing operations through special annotation bits generated by the retargeting tool.

Applications from MediaBench [37] are chosen for performance simulations. After the code retargeting process, all applications are simulated for the maximum of 100 million instructions. Performance results are measured using an instruction-per-cycle (IPC) metric and an IPC ratio metric, which is defined as $\frac{IPC_{DRF}}{IPC_{CRF}}$ (IPC_{DRF} is the average

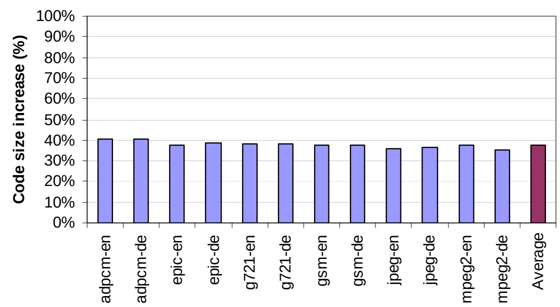
IPC when an application is running on a distributed register file architecture, and IPC_{CRF} being the average IPC on a central register file architecture.)

3.5.2 Code Retargeting Performance

Figure 28 shows code size increases for all applications after they have been retargeted through the code retargeting tool. The increase in code sizes is determined by the total number of register transfer operations that are added to code schedules. Most applications experience the similar degree of code size explosion because they share a large portion of code base (e.g. C runtime library). The average increase is 36% and 37% for 4- and 8-way target machines, respectively.



(a) 4-way



(b) 8-way

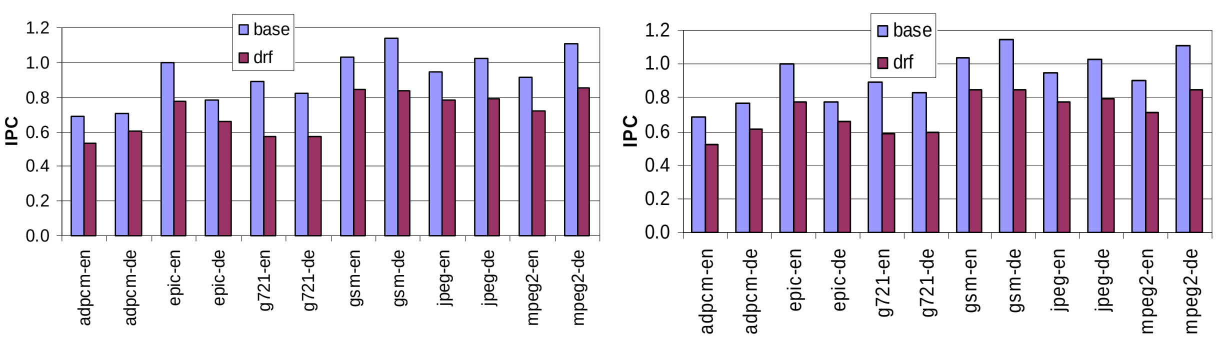
Figure 28 Code size increases after retargeting

The effectiveness of the scheduling and data routing algorithms are evaluated by measuring the number of extra cycles added to code schedules. While some register transfer operations can be scheduled in existing time slots, others need extra time slots to be created. For all applications retargeted, similar results are observed in both 4- and 8-

way target configurations. On average, 53% of all register transfer operations can be hidden or scheduled without incurring extra time slots.

3.5.3 Performance Impact of Distributed Register Files

Figure 29 shows the IPC results of the distributed register file architectures compared to the baseline architectures with a central register file. Performance degradation is observed in distributed register file architectures. However, there is no significant difference in IPC results between 4- and 8-way configurations. This is due to the limited capability of the basic block scheduling to extract more parallelism.



(a) 4-way **(b) 8-way**

Figure 29 IPC results of baseline and distributed register file architectures

The average IPC ratio is 77% compared to a central register file architecture. When all applications are simulated under a deterministic environment (perfect branch prediction, 32x bigger caches, and single-cycle cache and memory accesses), their IPC values are higher (the average of 0.87 compared to 0.69 in a realistic environment). However, the average IPC ratio remains the same at 77%. Therefore, these non-deterministic events do not affect the execution cycle penalty incurred by distributed register file operations in the current implementation.

3.5.4 Performance Impact of the Number of Register Transfer Buses

In previous experiments, a single register transfer bus is present. In this section, the number of register transfer buses is increased to two and three and performance measured. The average IPC results are shown in Figure 30.

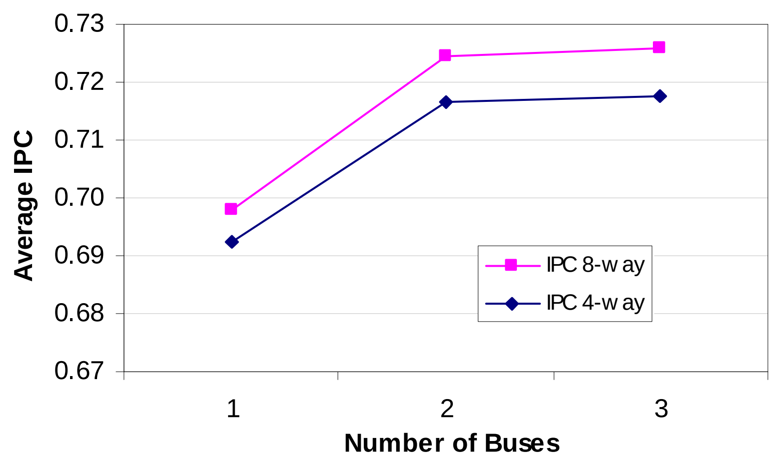


Figure 30 Performance Impact for Different Number of Register Transfer Buses

Varying the number of buses in S-DRF shows similar trend as in D-DRF. Performance gain, however, is not significant. Performance is increased by only 3.49% and 3.40% when the number of buses is increased to two and three respectively in a 4-way configuration, and 3.82% and 4.00% respectively in an 8-way configuration.

3.5.5 Performance Impact of the Local Register File Size

In this section, performance is evaluated for different local register file size. Evaluations are performed for local register file size of 8, 16, 24, and 32. The average IPC results are shown in Figure 31.

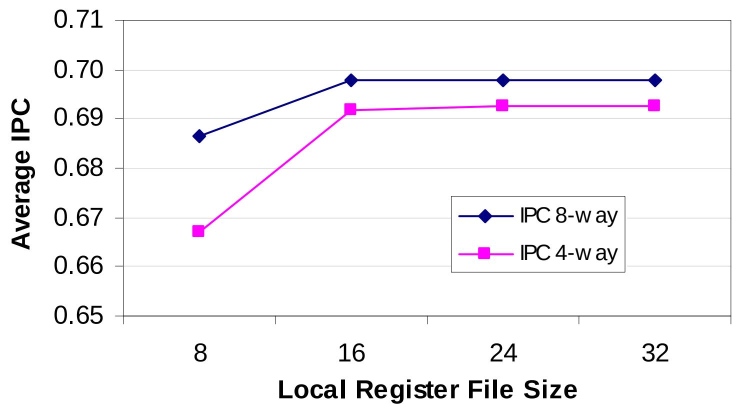


Figure 31 Performance Impact of Local Register File Size

Similar to the number of buses evaluations, performance differences when varying local register file sizes are insignificant. Slight performance gain is observed when increasing the size from 8 to 16 registers. However, no performance gain is observed beyond 16 registers.

3.5.6 Comparison of the Dynamic and Static approaches

It is interesting to compare the dynamic and static approaches. Through a common simulation framework, they can be directly and fairly compared in this research. From the average IPC results (Figure 13, Figure 14, and Figure 29), the dynamic approach shows better performance than the static approach for all applications simulated. This is mainly due to the limitation of the basic block scheduling algorithm used in the static approach and the inability of static code schedules to adapt to the processor's run-time conditions.

Without the effect of dynamic run-time conditions (perfect branch prediction and a near-perfect memory system), the average IPC of S-DRF is improved by 26% while the average IPC penalty incurred by distributed register file operations remain constant. Therefore, dynamic run-time conditions have no significant impact on the static approach in its current implementation.

In summary, the execution cycle penalty of S-DRF is worse than D-DRF in a 4-way configuration. They are comparable in an 8-way configuration. Execution performance of S-DRF, however, is lower than D-DRF in all applications. With the direct relationship between IPC penalty and application's *base* IPC as discussed in Chapter 2, the execution cycle penalty of statically scheduled architectures will be higher than those of dynamically scheduled architectures when the *base* IPCs are made comparable by improving the scheduling techniques. This outcome suggests the advantage of the dynamic distributed register file approach over the static approach.

3.6 Conclusion

Distributed register files can be supported in software by generating code specifically for a distributed register file architecture, which significantly simplifies hardware implementation. In this chapter, a code generation technique for distributed register files is presented with the main contribution in the global data routing technique with multicasting.

Applications from the MediaBench benchmark suite are retargeted and simulated on a simulator built based on SimpleScalar. The retargeting result shows that code size is increased by 36% on average due to the addition of register transfer operations. 47% of these additional operations incur extra scheduling cycles while the rest (53%) can be scheduled into existing scheduling cycles.

For a 4-way configuration, simulation results show 23% IPC penalty caused by distributed register file operations with the average IPC value of 0.67. Simulation in a deterministic environment shows an increase in IPC but the same degree of IPC penalty. No significant impact is observed when the number of register transfer buses and the local register file size is varied. No significant performance improvement is observed when the issue width is increased from 4 to 8. This is mainly because of the limitation of the basic block scheduling algorithm used in the current implementation. Comparing this result with the result of the dynamic approach shows that the dynamic approach is more effective than the static approach in distributed register file support since the dynamic approach results in lower IPC penalty while exhibiting higher IPC for all applications.

CHAPTER 4

DISTRIBUTED REGISTER FILES FOR ILP-SIMD ARCHITECTURES

4.1 Summary

Emerging portable multimedia applications demand extremely high computational throughput with small area and limited power. Data parallel processor designs, such as SIMPil, have been demonstrated to achieve these goals by exploiting aggressive data parallelism and novel streaming data retrieval mechanisms. ILP-SIMD further enhances performance compared to conventional data parallel architectures by exploiting instruction-level and control parallelism. It does so without significant increase in area and energy consumption, except in the central register file, which grows rapidly as the issue width is increased.

This chapter presents a distributed register file organization for ILP-SIMD which addresses the scalability issue of the central register file. By dividing functional units into clusters, each with a dedicated local register file, area and energy consumption can be reduced. Operands in a local register file can be accessed directly while remote operands must be transferred into a local register file through register transfer instructions before they can be consumed. These register transfer instructions cause extra execution cycles and can delay the execution of other operations. Register transfer instructions are statically scheduled by compilers and can be scheduled in the same cycle that the operands are produced. The latter condition significantly reduces execution cycle penalties and is enabled by the moderate target clock frequency of ILP-SIMD. For a 2-way configuration, although performance is slightly degraded (requiring 4% extra

execution cycles), the total area demand of a processing element can be reduced by 12%. This extra area can be utilized to increase the total number of processing elements by 13%, which directly improves overall performance through extra support for data-level parallelism.

4.2 Introduction

In the previous two chapters, distributed register files have been employed to address the operand transport problem of general purpose processors as the number of functional units is increased. A central register file and a large multi-stage operand bypass network, which are major components of a conventional operand transport system, scale poorly with an increasing number of functional units because of their rapidly growing demands for long interconnects. The consequences are long operand access time, high energy consumption, and large implementation die area. For high performance general purpose processors, area and energy consumption of operand transport system are not critical since they constitute only a small fraction of total energy consumption and die area of the whole processor chip. Long operand access time, however, can limit the maximum operating clock frequency and is one of the most important problems for this processor category.

Another category of processors aims at providing sufficient computational power for specific applications but imposes strict constraints on implementation die area and energy consumption. Examples are embedded processors and media processors used in portable devices, such as PDAs, mobile phones and digital cameras/camcorders. These processors are used in devices with small form factor and limited power sources.

Therefore, their complexity and operating clock frequency are typically limited by system design budget. Nevertheless, the demand for processing power is rapidly growing as emerging applications, such as streaming video and speech recognition, are being deployed.

A SIMD processing array with focal plane area I/O (SIMPil) has been developed for efficient processing of image and video streaming data [62][63][64]. SIMPil exploits data-level parallelism (DLP), which is abundant in many multimedia applications, to achieve high processing throughput with high efficiency in terms of implementation die area and energy consumption. ILP-SIMD [65] further enhances the architecture of SIMPil processing elements to exploit instruction-level parallelism (ILP) and control parallelism inherent in an instruction stream in addition to DLP. Speedup in processing throughput is achieved in ILP-SIMD but with high cost in register file implementation.

In this chapter, a distributed register file organization is explored to address the high cost in ILP-SIMD implementation. Background on ILP-SIMD is briefly discussed in the next section followed by the design of a distributed register file ILP-SIMD PE architecture based on the statically scheduled distributed register file architecture presented in Chapter 3. Execution performance is evaluated in the subsequent section followed by implementation cost evaluation using the GENESYS system simulation framework. The conclusion is then given, including a discussion of future work.

4.3 ILP-SIMD Architecture

ILP-SIMD enhances SIMD processing elements (PEs) through ILP exploitation. In a typical SIMD processor array, instructions are broadcast sequentially from an array control unit (ACU) to all PEs synchronously, with each PE being a small simple processor capable of executing one instruction at a time. By allowing each PE to execute more than one instruction each cycle, significant speedup can be achieved.

ILP-SIMD has been developed based on SIMPil, a SIMD pixel processor. In the following subsection, the architecture of SIMPil is presented followed by a brief discussion of some related enhancements to the basic SIMD architecture in the literature. The two flavors of ILP-SIMD, single control-flow (SCF) and control-parallel (CP), are discussed in subsequent subsections.

4.3.1 The SIMD Pixel Processor (SIMPil)

SIMPil is an experimental SIMD system with focal-plane area I/O for image and video processing. It benefits from efficient exploitation of data-level parallelism (DLP), short wire lengths, and specialized microarchitecture to provide a significant improvement in energy efficiency [62][63][64]. The SIMPil architecture consists of a 2-D array of SIMD processing elements (PEs). Each PE has a RISC-like datapath with specialized units for SIMD operations and area I/O data streams. The block diagram of the SIMPil processor array is shown in Figure 32.

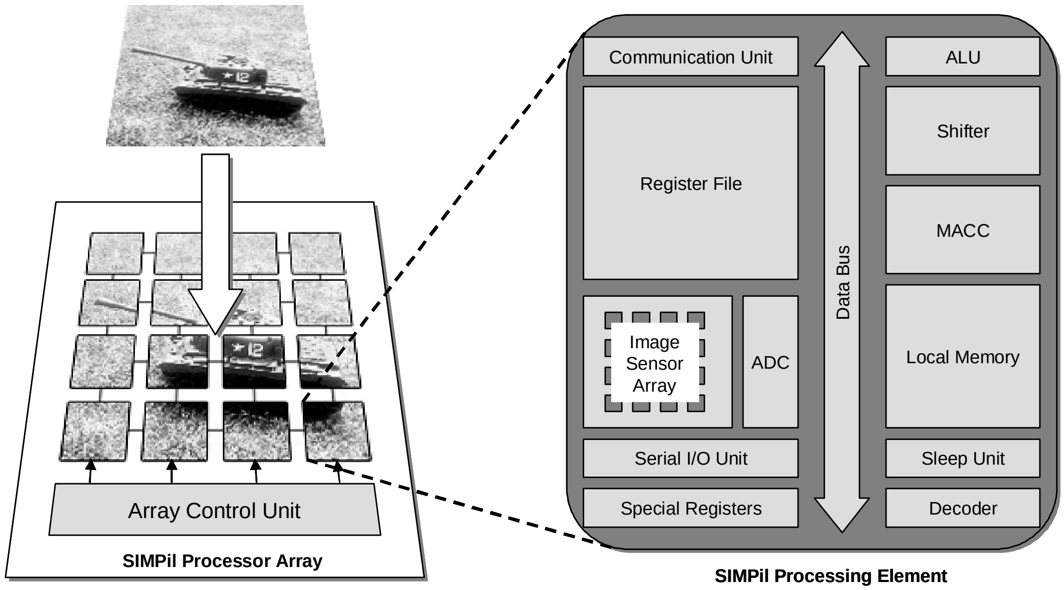


Figure 32 Block diagram of the SIMPil processor array and its processing elements

In each cycle, an instruction is broadcast from an array control unit (ACU) to all PEs. Each PE is a simple 16-bit processor with the following seven major functional units:

- ALU – computes basic arithmetic and logic operations,
- Barrel shifter – performs multi-bit logic/arithmetic shift operations,
- MACC – multiplies 16-bit values and accumulates into a 32-bit accumulator,
- Sleep – activates or deactivates a PE based on local information,
- Communication - communicates with a neighboring PE through a NEWS (north-east-west-south) network,
- Local memory – accesses fast memory array of up to 256 words, and
- Pixel unit – samples pixel data from the local image sensor array.

With local 4x4 pixel sensor arrays, each PE is associated with a specific portion (4x4 pixels) of an image frame. This form of area-I/O allows streaming pixel data to be retrieved and processed locally resulting in higher data bandwidth and higher energy efficiency than other mainstream media processors with global memory and data caches.

4.3.2 Architectural Enhancements for SIMD

Despite significant performance improvement from DLP, there have been continuous efforts to increase the performance further without significantly increasing implementation costs. This section describes some important developments, which improve SIMD performance by exploiting other kinds of parallelism in addition to DLP. These developments include Multiple SIMD (MSIMD), mixed mode SIMD/MIMD, superscalar SIMD, and ILP enhancement to SIMD PEs.

4.3.2.1 Multiple SIMD (MSIMD)

The early issue with SIMD machines is low utilization of its fixed-size processor array because different applications have different dataset sizes and characteristics. While few applications may be able to utilize all PEs in the array, many applications can make use of only a small portion of the array leaving many PEs idle.

MSIMD addresses the low utilization issue of a SIMD processor array by dividing a large processor array into several smaller arrays, each with its own ACU. These sub-arrays can operate independently on different applications hence increasing the overall array utilization especially in multi-user and multi-programming environment. In addition, sub-arrays can be combined into a larger array to handle applications with larger datasets. The array configuration is normally performed at the operating system level. For example, the early Connection Machine (CM-1 and CM-2) [66] contains four SIMD partitions that can be arbitrarily connected to four front-end computers through the nexus (a fully connected crossbar). Performance improvement is achieved by better array utilization through task parallelisms.

Another source of low utilization of a SIMD array is conditional branching, such as *if-then-else* and *case* statements. In such cases, different flows of control (e.g. *then* vs. *else* blocks) are dispatched sequentially, one after the other, to all PEs, with each PE properly discarding instructions from irrelevant control flows. With this serialized execution, the average utilization of a SIMD array is 50% at most for conditional blocks. Parallelism among different control flows can be exploited in MSIMD through dynamic partitioning of the processor array at instruction level.

The GPA machine is an MSIMD machine that can be partitioned dynamically [67]. Arbitrary non-overlapped partition can be formed at run-time by configuring crossbar switches connecting ACUs to PEs. Complexity of the network is reduced by using several small p x p crossbars (p is the number of controllers) to p instruction broadcast trees of controllers. As an example, PEs executing the *then* and *else* blocks of the same instruction stream can be connected to two different controllers with both partitions executing in parallel.

4.3.2.2 Mixed-Mode SIMD/MIMD

Control parallelism is considered a prominent feature of MIMD architecture where each PE independently executes its own instruction stream in parallel. Therefore, one approach to exploit control parallelism in SIMD is to allow a SIMD processor array to switch to MIMD mode as needed. Examples of mixed-mode SIMD/MIMD machines are OPSILA [68] and PASM [69]. These machines allow a processor array to switch between SIMD and MIMD mode of operations dynamically at instruction level and, therefore, combines the best of both worlds. Inter-PE synchronization and communication are simple and efficient in SIMD mode while control parallelism is the advantage of MIMD mode. Despite the flexibility of mixed-mode SIMD/MIMD machines, the synchronization overhead (during mode switching) is substantial, and application programming is non-trivial. Moreover, implementation cost is significantly higher than plain SIMD or MSIMD machines since each PE requires a dedicated controller and an instruction buffer to support both SIMD and MIMD modes of operation.

4.3.2.3 Superscalar SIMD

Another recent development is the Superscalar SIMD architecture [70], which is a dynamically reconfigurable SIMD machine capable of arbitrary non-overlap partitioning. Control parallelism within a program is exploited by dispatching instructions from different control flows simultaneously to all PEs. Each superscalar SIMD PE then selects an instruction stream to execute based on local information, independent of other PEs. Therefore, the reconfiguration process incurs much less overhead than in dynamically partitionable MSIMD, which forms partitions through a central control unit.

The Superscalar SIMD machine is composed of k ACUs and n PEs ($k << n$). Each PE receives k streams of instructions but selects only one instruction stream for execution based on local information. Control parallelism from *if-then-else* constructs is exploited by transforming the conditional block into two instruction streams (*then* and *else* streams) with a *fork* operation performed at the beginning of the block and a *join* operation at the end. When a *fork* is encountered, the parent ACU acquires an idle ACU and replicates its register file content. The two ACUs then start their executions in parallel, one for the *then* block, and the other for the *else* block. A fast register file replication mechanism is provided through the bit-interleaved register file, which interleaves bits from *k* register files together so that a copy operation can be performed in one cycle. Finally, when a *join* is encountered, the register file of the child ACU is merged back to the parent ACU. The merge operation is handled properly by examining the *busy* and *dirty* flags of each register in both register files.

The above *fork/join* mechanism and the bit-interleaved register file allow concurrent execution of the *then* and *else* blocks by removing dependencies on the ACU (or scalar) registers. Other dependencies exist that require synchronization between the concurrently executing partitions. These dependencies occur when there are communications between PEs or between PEs and their associated ACU.

4.3.2.4 ILP Enhancement to SIMD PEs

All the improvements to SIMD machines discussed above focus on improving the performance and the utilization of the SIMD array by improving the controller, the communication network, or the instruction distribution network. Besides, improvement can be applied to PEs to take advantage of ILP within an instruction stream. Early work includes Maspar MP-2 [71] (overlap integer and memory operations), CCSIMD [72] (overlap communication operations), and pipelined PE [73].

The most recent development includes Imagine media processors [15], which employs a SIMD engine with eight ILP-capable PEs. Streams of data are staged into the processor through a large stream register file and local distributed register files. ILP is exploited within each PE through multiple instruction issues with code scheduling performed statically by compilers [52]. Although Imagine can exploit both DLP and ILP, it has limited DLP capability (eight PEs) unlike full-scale SIMD architectures, such as SIMPil and ILP-SIMD.

ILP-SIMD architectures exploit ILP by issuing and completing multiple instructions simultaneously within a single cycle. As shown in Figure 33, the ACU is enhanced to broadcast multiple independent instructions to all PEs, and each PE is enhanced to execute all those instructions concurrently each cycle. By executing multiple instructions concurrently, functional unit utilization and overall instruction throughput are improved. Moreover, since instruction schedules are statically computed by compilers, additional hardware is minimal resulting in an increase in area and power efficiency.

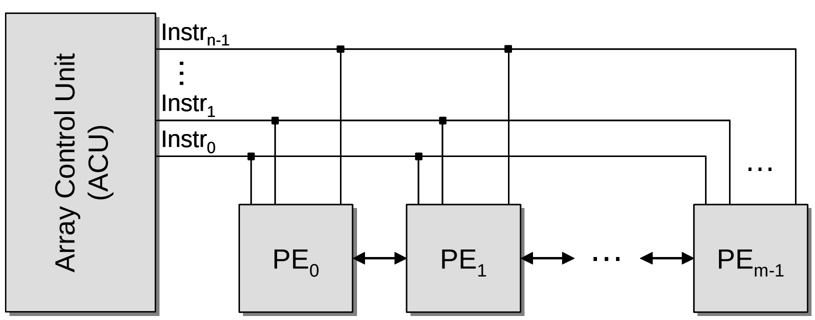


Figure 33 ILP-SIMD architecture

Two families of ILP-SIMD architectures include single control flow (SCF) and control parallel (CP), which differ by their control flow management mechanisms. SCF ILP-SIMD processes a single control flow at a time and is a direct derivation of its predecessor SIMD architectures. Control parallelism is exploited in CP ILP-SIMD by executing instructions from multiple control flows concurrently and completing only those instructions that belong to validated control flows. CP ILP-SIMD can outperform SCF ILP-SIMD in applications where control parallelism can be exploited.

4.3.3 Single Control-Flow ILP-SIMD (SCF)

SCF ILP-SIMD is a data parallel SIMD architecture capable of executing multiple instructions from a single control flow concurrently within a PE. Similar to SIMPil and other traditional SIMD architectures, instructions belonging to different control flows must be scheduled and executed sequentially. Control flow validations are managed by the masking unit. Conditional constructs in high-level languages, such as *if-then-else* or *case* statements, are translated into *sleep* and *wakeup* assembly instructions by compilers.

In an N-way SCF ILP-SIMD machine, there are N functional unit clusters. Each cluster is comprised of one or more functional units. With N clusters, the central register file has to provide $2N$ read ports and N write ports. The block diagram of SCF ILP-SIMD PE is shown in Figure 34.

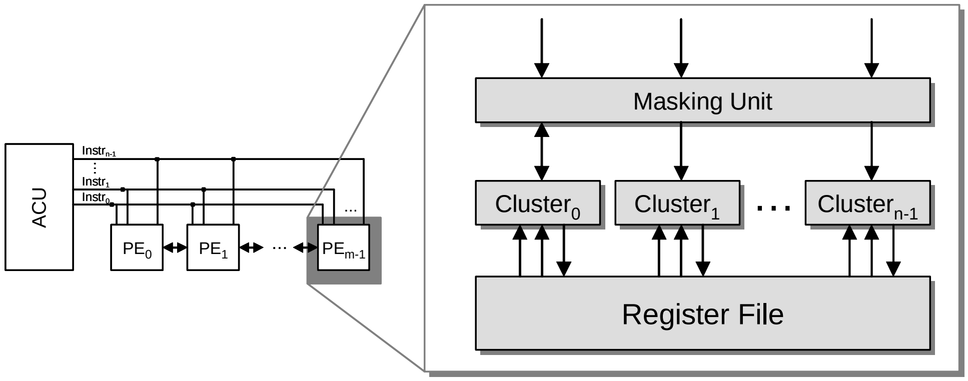


Figure 34 The block diagram of SCF ILP-SIMD processing element

4.3.4 Control-Parallel ILP-SIMD (CP)

CP ILP-SIMD extends SCF ILP-SIMD with the capability to exploit control parallelism. Instructions from different control flows can be dispatched concurrently to all PEs. However, only those instructions belonging to the control flow chosen by each PE are allowed to complete their executions. With this approach, different control flows can be executed concurrently by different PEs. Processor array utilization is improved since sequential execution of conditional flow of controls (through *sleep* and *wakeup* instructions) is no longer needed.

Flow control validation is performed using predication and the control flow unit. The compiler statically assigns tag and flag information to each instruction at compile time. The tag field indicates the control flow nesting level of the basic block to which an instruction belongs. The flag field is a one-bit field used to distinguish instructions that belong to the *then* clause from those that belong to the *else* clause. Instructions that belong to the same basic block have identical tag and flag values. Instructions from *then* and *else* clauses share the same tag but different flag values. Tag values are assigned to basic blocks in program order with the main program flow having the tag value of 0. The first basic block with a conditional instruction is assigned tag value of 1, followed by 2, and so forth.

Each PE maintains the current tag and flag values in its tag and flag predicate registers. Instructions from both *then* and *else* clauses are allowed to begin execution concurrently. The control flow unit examines tag and flag predicates in each instruction and allows only those with matching predicate values to write back to the register file. The control flow unit, therefore, is equivalent to multiple masking units as shown in Figure 35.

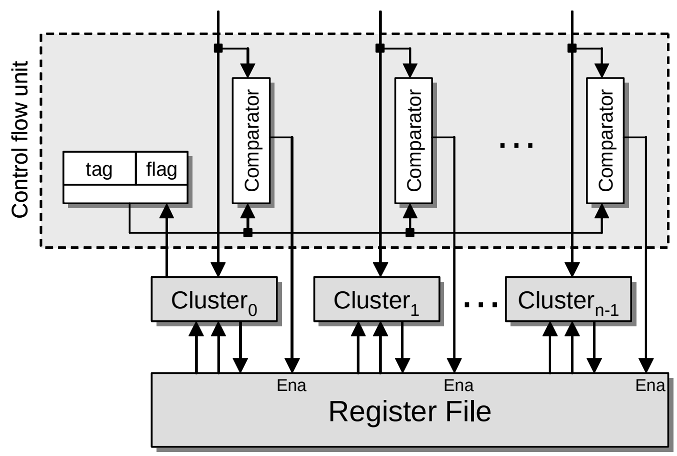


Figure 35 Control flow unit in CP ILP-SIMD architectures

4.3.5 Implementation Cost of ILP-SIMD

One of the most important issues in ILP-SIMD PE implementation is die area. A small PE is highly desirable so that a large number of PEs can be integrated onto a single chip. The total number of PEs directly translates to the degree of data parallelism that can be exploited. Since many image and video applications, which comprise the target application domain of ILP-SIMD, contain a significant amount of data parallelism, the total number of PEs (and thus the PE area) has a significant impact on the overall system performance. Power density is another important cost metric. However, it is not considered a critical issue in ILP-SIMD since the processor array can be clocked at moderate frequency while still sustaining high instruction throughput through DLP (and extra performance from ILP and control parallelism).

ILP-SIMD presents a simple mechanism to increase PE performance by exploiting ILP and control parallelism. This is desirable as long as the increase in PE area is minimal. To evaluate the implementation area of ILP-SIMD, hardware models of SIMPil PE and ILP-SIMD PE are simulated using the GENESYS system simulation tool [74] with macro cell capability. Technology-independent hardware description for each functional block (macro cell) is used by GENESYS to calculate functional performance of each unit and the whole PE. The model has been verified with the silicon implementation of a SIMPil PE in a 0.8µm process [75]. The final result is obtained for a 100nm technology through an appropriate technology model. Figure 40 shows the area of ILP-SIMD with various issue widths obtained from GENESYS. The values are normalized to the area of the baseline SIMPil PE implementation.

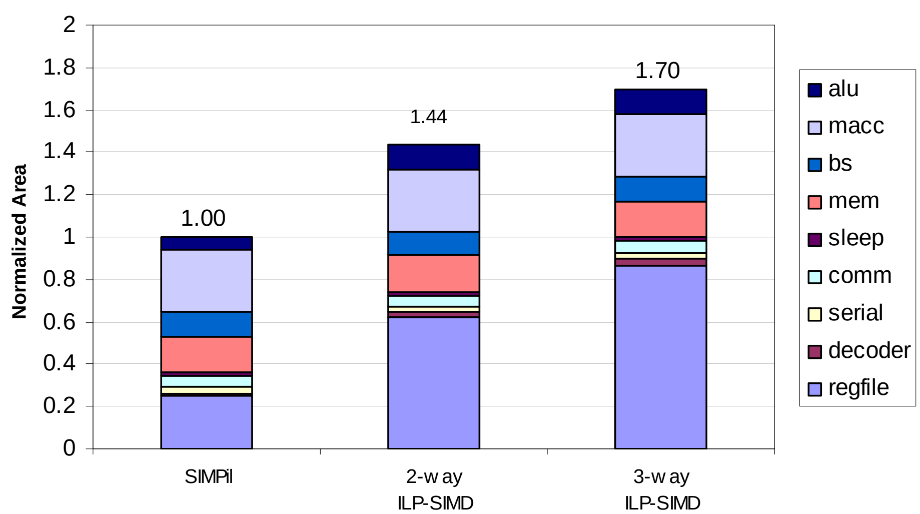


Figure 36 Implementation die area of 1-, 2-, and 3-way ILP-SIMD processing elements

ILP-SIMD PE requires 44% and 70% more PE area in 2- and 3-way configurations, respectively. The increase in area is mostly attributed to a large 24-register register file (6 ports in 2-way ILP-SIMD, and 12 ports in 3-way ILP-SIMD), which starts to dominate PE area as the issue width is increased to two or more.

4.4 Distributed Register Files for an ILP-SIMD PE

In ILP-SIMD, dynamic instruction scheduling hardware is not needed since instructions are scheduled statically by compilers. However, some additional hardware is still needed to support execution of multiple instructions concurrently within a PE. These include extra functional units and a large register file that can supply enough operands to all functional units that can be active simultaneously. While functional units in ILP-SIMD are relatively small, the multi-port register file dominates the area of a PE as the issue width is increased beyond one. In a 1-way SIMPil PE, the register file is the second-largest component (the largest being the MACC unit) and consumes 25% of the total PE area. In 2- and 3-way ILP-SIMD PEs, however, the register file becomes the largest component and consumes 44% and 51% of the total PE area, respectively.

In this section, a distributed register file (DRF) organization is employed to address the area scalability issue of a central register file in ILP-SIMD. This is referred to as DRF ILP-SIMD. First, the cluster organization and the register communication mechanism of DRF ILP-SIMD are explained based on SCF ILP-SIMD architectures. Then the code generation technique is presented.

4.4.1 Functional Unit Cluster Organization

To reduce high area demand of a central register file, a distributed register file (DRF) organization is used as shown in Figure 37 for both 2- and 3-way ILP-SIMD PEs. Functional units in a 2-way DRF ILP-SIMD PE are divided into two clusters, each with a dedicated local register file. These local register files are small with only 4 ports, and they are interconnected through a register transfer bus. A 3-way DRF ILP-SIMD PE is organized in a similar manner but with three clusters and three local register files.

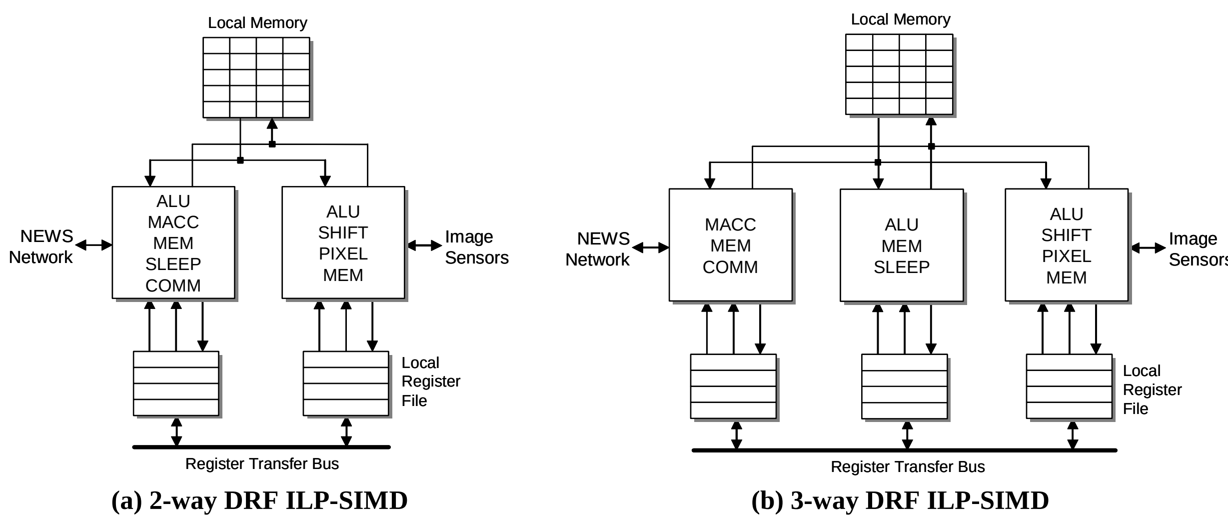


Figure 37 Datapath of ILP-SIMD with a distributed register file organization

A functional unit can only access registers from the local register file in the same cluster. Values in a remote register file must be transferred into the local register file through the register transfer bus before it can be accessed. These register transfer operations are scheduled statically by compilers using register transfer instructions. A single register transfer instruction can transfer a value from a source register to one or more destination registers in different clusters simultaneously through multicasting, which is inherently supported by a register transfer bus. A register transfer instruction can be issued in the same cycle as the instruction that produced the value to be transferred since the target clock frequency of ILP-SIMD is relatively low to limit the maximum power density. As a result, a register value can be consumed immediately in the following cycle after it is produced. Performance degradation is significantly reduced and is limited only by the number of buses available. This same-cycle transfer approach is similar to the approach used in Imagine. While Imagine requires a global register namespace, DRF ILP-SIMD split the transaction into local register write and remote register transfer, which can operate with a smaller local register namespace. The DRF ILP-SIMD approach results in more efficient instruction encoding (fewer bits to encode register operands) but demands more registers to hold intermediate results.

The composition of functional units into clusters is highly critical in ILP-SIMD. Since ILP-SIMD functional units are smaller and more specialized than in general purpose processors, placing these functional units in different clusters results in a highly distributed configuration, which induces a large number of register transfer operations. In Figure 37, functional units are carefully organized so that functional units that frequently communicate operands between each other are placed in the same cluster. Although some communication patterns are common, other communication patterns are application-specific. Therefore, the optimum functional unit cluster composition for one application, which can be determined empirically through simulations, may not be optimum for another application. In addition, all ILP-SIMD operations are deterministic, including local memory accesses and branch processing, unlike in high performance general purpose processors. As a result, additional delays in the execution pipeline incurred by register transfer operations cannot be effectively hidden as in general purpose processors, which are less deterministic.

4.4.2 Code Generation Framework

Compilers need to be extended to generate code for a distributed register namespace and schedule register transfer instructions as appropriate. First, instructions are explicitly assigned to clusters during the instruction scheduling phase. The algorithm, shown in Figure 38, is based on the unified assign and schedule (UAS) framework [51] but performs the scheduling of register transfer instructions globally in a separate data routing phase. Second, register transfer instructions are inserted into the code schedule by searching for empty slots starting from the cycle that produces the value until the cycle that first uses the value. If an empty slot is found, a register transfer instruction can be scheduled without adding an extra cycle to the code schedule.

```
while unscheduled ops exist
        update a list of data-ready ops and cluster usage information
        pick the highest priority op
        pick the non-busy cluster with the most operands
        if multiple clusters qualify
                pick the cluster with the shortest dependency chain
                to the current instruction
        if multiple clusters qualify
                pick one cluster randomly
        if a cluster is chosen
                assign op to the chosen cluster
                schedule op in the current cycle
        else
                advance to the next cycle
```

Figure 38 List scheduling algorithm with the cluster assignment heuristic (the assignment heuristic is in bold face)

Finally, registers must be allocated separately for each local register file. Because of the limited number of register transfer buses and the limited size of each local register file, the overall speedup for all applications in a distributed register file configuration is less than in a central register file configuration. However, the implementation is much smaller and extra area can be used to increase the total number of PEs in an ILP-SIMD processor array, which results in higher performance than an ILP-SIMD with a central register file organization.

4.5 Performance and Cost Evaluation

To evaluate the effectiveness of the distributed register file approach to ILP-SIMD, the architecture is simulated and its performance degradation quantified. The saving in implementation area is then evaluated to determine the overall merit of the approach.

4.5.1 Simulation Methodology

Performance of ILP-SIMD architectures (SCF, CP, and SCF with distributed register files) are evaluated through cycle-accurate simulations. Multimedia applications for the baseline SIMPil architecture are retargeted to ILP-SIMD machine code and simulated using ILP-SIMD simulators. Applications are selected to cover a wide spectrum of key tasks in the image and video processing domains. These applications and their brief descriptions are provided in Table 7.

Table 7 Benchmark applications for SIMPil and ILP-SIMD simulations

Application	Description
Inedge	Inside-edge detection
Medge	Edge detection through morphological operations with 3x3 masks
Median	Median filtering using a 3x3 window
Ring90	Two-stage image rotation with a skew-based rotation followed by a set of *n* fast 90-degree rotations
Skel	Skeletonization
SpatFilt	2D convolution-based filtering using a 3x3 filter mask
Vq	Image compression using vector quantization technique

The ILP-SIMD architectural model is built into the machine code retargeting tool and the simulator. Seven types of functional unit are modeled similar to the baseline SIMPil architecture. These include ALU, barrel shifter (SHIFT), multiply-accumulate (MACC), local memory (MEM), masking unit (SLEEP), communication unit (COMM), and pixel I/O unit (PIXEL). Unlike in SIMPil, two ALUs and a dual-access local memory, which allows one read and one write in a cycle, are provided to sustain higher instruction throughput. Register file sizes are big enough to eliminate the need for spilling in most cases and are determined based on simulation results. Sixteen registers are used in the baseline SIMPil architecture while 24 registers are used in both 2- and 3-way ILP-SIMD architectures. All registers and memory words are 16-bit wide.

In DRF SCF ILP-SIMD exploration, clusters of functional units and local register files are organized as shown in Figure 37. The retargeting tool is extended with algorithms for cluster assignment, global data routing, and DRF register allocation with spilling between local register files. The ILP-SIMD simulator is extended accordingly with the same cluster organization. The 2-way (2-cluster) architecture has the total of 24 local registers (12+12). Similarly, the 3-way (3-cluster) architecture has the total of 30 local registers (8+10+12).

In the following sections, performance improvement of SCF and CP ILP-SIMD (both 2- and 3-way configurations) are evaluated and compared to the baseline SIMPil architecture. Performance impact of a distributed register file organization is then evaluated based on SCF ILP-SIMD. Finally, implementation costs are analyzed for both central and distributed register file organizations.

4.5.2 Performance Speedup

In a DRF organization, performance (in terms of the total number of execution cycles) can be adversely impacted. An extra cycle is required to transfer an operand value from a remote register file into the local register file when the needed operand is produced in a different cluster. The performance impact of DRF SCF ILP-SIMD is shown in Figure 39, which compares speedup values of a central and distributed register file variances of SCF ILP-SIMD relative to the baseline SIMPil architecture. A DRF organization has lower speedup for all applications due to the aforementioned execution cycle penalty.

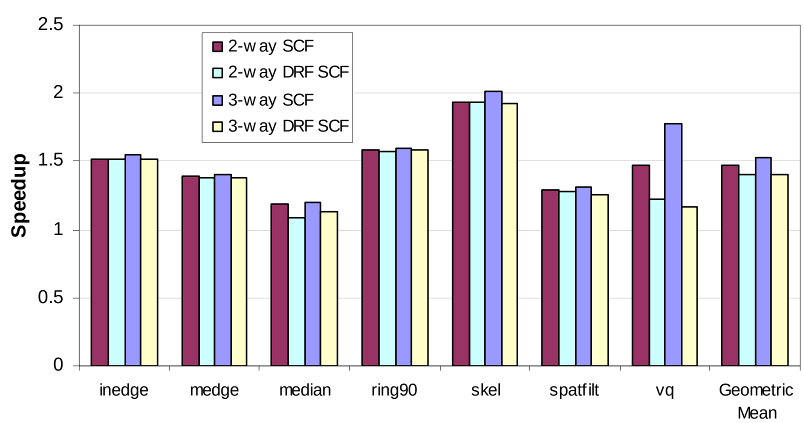


Figure 39 System performance of DRF SCF ILP-SIMD architectures

Considering all applications simulated, a DRF organization incurs 4% and 9% more execution cycles on average than a CRF organization for 2- and 3-way configurations, respectively. The performance of a 3-way DRF architecture, however, is not better than a 2-way DRF architecture. This is because functional units in a 3-way DRF architecture is more distributed and, thus, incur more register transfer penalty than its 2-way counterpart. This penalty is typically compensated by more concurrent executions in a wider configuration. However, ILP is not high enough to overcome the aforementioned penalty because of the limited number of functional units (both 2- and 3-way DRF architectures have the same number of functional units) and the limitation of the list scheduling algorithm, which cannot extract ILP beyond basic block boundaries.

Another implication of a DRF organization is the increase in the number of registers required during execution (register pressure). Extra registers are needed when a single operand value is replicated to multiple clusters. Therefore, the total of 30 registers is used in DRF architectures while 24 registers comprises a central register file in non-DRF architectures.

4.5.3 Implementation Die Area Evaluation

One of the most important issues in ILP-SIMD PE implementation is die area. A small PE is highly desirable so that a large number of PEs can be integrated onto a single chip. The total number of PEs directly translates to the degree of data parallelism that can be exploited. Since many image and video applications, which comprise the target application domain of ILP-SIMD, contain a significant amount of data parallelism, the total number of PEs (and thus the PE area) has a significant impact on the overall system performance. Power density is another important cost metric. However, it is not considered a critical issue in ILP-SIMD since the processor array can be clocked at moderate frequency while can still sustain high instruction throughput through DLP (and extra performance from ILP and control parallelism).

ILP-SIMD presents a simple mechanism to increase PE performance by exploiting ILP and control parallelism. This is desirable as long as the increase in PE area is minimal. To evaluate the implementation area of ILP-SIMD, hardware models of SIMPil PE and ILP-SIMD PE are simulated using the GENESYS system simulation tool [74] with macro cell capability. Technology-independent hardware description for each functional block (macro cell) is used by GENESYS to calculate functional performance of each unit and the whole PE. The model has been verified with the silicon implementation of a SIMPil PE in a 0.8μm process [75]. The final result is obtained for a 100nm technology through an appropriate technology model. Figure 40 shows the area of various ILP-SIMD configurations obtained from GENESYS. The values are normalized to the area of the baseline SIMPil PE implementation.

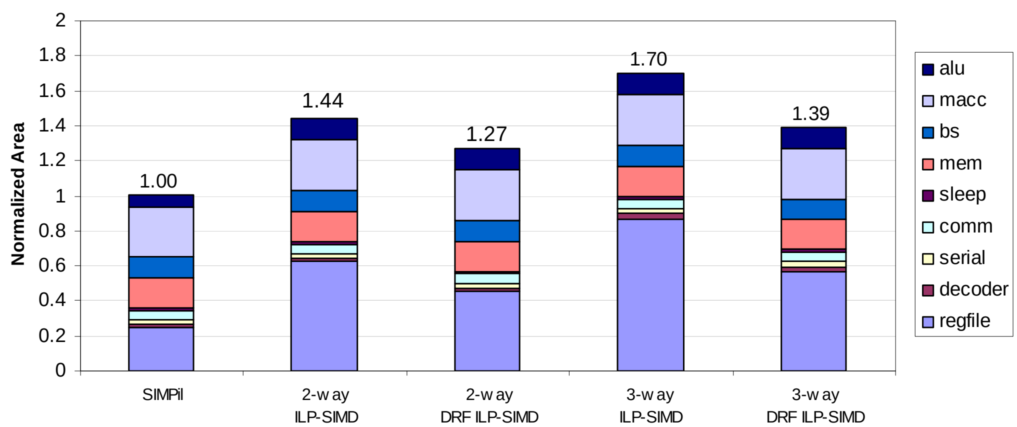


Figure 40 Implementation die area of ILP-SIMD normalized to the baseline SIMPil architecture

ILP-SIMD PE requires 44% and 70% more PE area in 2- and 3-way configurations, respectively. The increase in area is mostly attributed to a large 24-register register file (6 ports in 2-way ILP-SIMD, and 12 ports in 3-way ILP-SIMD). The DRF configuration reduces total register file size with distributed 4-port local register files. The increase in PE area in DRF ILP-SIMD PE is reduced to 27% and 39% for 2- and 3-way, respectively.

Although DRF ILP-SIMD has lower speedup compared to its non-DRF counterpart, the significant area saving is highly attractive and can be used to integrate more PEs into the processor array. In a 2-way configuration, DRF ILP-SIMD requires 4% more cycles on average for most applications; however, extra area obtained from smaller register files can be used to implement 13% more PEs, which directly increase performance through DLP. Similar argument can be made for a 3-way DRF ILP-SIMD (9% more cycles but with 22% more PEs). However, it is not as attractive since its speedup results are only slightly higher than a 2-way configuration. With more complex scheduling algorithms, the benefit from larger issue width can become substantial.

4.6 Conclusion

A distributed register file organization is an effective mechanism to reduce implementation cost in ILP-SIMD implementation. Distributed register operations are managed by compilers through static code scheduling and register transfer instructions. A simple and effective code generation technique for distributed register files has been developed, which results in 4% performance penalties on average in terms of total execution cycles for a 2-way configuration. These penalties are well compensated by a smaller PE implementation. For a given die area, this permits 13% additional processing elements to be implemented when compared to a conventional ILP-SIMD implementation. These extra processing elements, if implemented, will increase the degree of data parallelism that a processor array can support by approximately the same amount.

Based on the microarchitecture and the code generation technique presented, the 2-way distributed register file configuration achieves the best result by having the average speedup close to a central register file architecture with significant savings in area. Configurations with large issue widths result in significant performance degradation because of their highly distributed functional unit organization. In addition, the basic block scheduling algorithm is limited in extracting enough ILP to overcome performance loss from distributed register file operations. Future research and experiments include the exploration of more effective ILP scheduling algorithms, such as trace scheduling, for DRF ILP-SIMD, and the extension of the code generation and simulation framework to a DRF CP ILP-SIMD architecture.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this dissertation, the fully distributed register file architecture has been explored to address the scalability problem of a central register file as the number of functional units in a processor execution core is increased. A conventional design requires a central register file with a large number of access ports and a large multi-stage bypass network to deliver all required operands to all operations that are ready to execute. Implementation costs (die area and energy consumption) of these components grow rapidly as the number of operands to be transported concurrently, and thus the number of functional units, is increased. In addition, delay of these components is significant and becomes a cycle-time bottleneck in high-performance processor configurations.

The fully distributed register file organization has been explored in the context of three major processor architectures: dynamically scheduled (superscalar), statically scheduled (VLIW), and ILP-SIMD media processors. These three categories have some common requirements, yet they possess unique characteristics that affect the design of a distributed register file organization and supporting techniques. In this section, results from research presented in this dissertation are summarized. Future research directions are also given, which conclude this dissertation.

5.1 Summary of Results

The fully distributed register file organization and the proposed techniques have been evaluated by comparing their execution performance and hardware implementation costs to those of central register file architectures with similar configurations. In terms of execution performance, an IPC (instruction per cycle) metric, which is directly derived from a total execution cycle, is used. A distributed register file architecture always require more cycles to complete a particular program than a central register file architecture because additional cycles are required to transfer operand values among local register files. This execution cycle penalty is represented by an IPC ratio metric, which indicates the IPC of a distributed register file as a fraction of performance of a conventional central register file system.

The advantage of a distributed register file architecture over a central register file architecture is its superior physical properties, i.e., smaller die area, lower delay, and less energy consumption. Overall performance improvement is achieved when gains from these physical properties outweighs the execution performance drawback described above.

5.1.1 Dynamically Scheduled Fully Distributed Register File Architecture

In the dynamic case, distributed register files are supported entirely in hardware. Register operands in conventional machine code, which assumes a central register file model, are transformed into distributed register operand references at run-time. Moreover, register transfers among local register files are scheduled dynamically at run-time using *on-demand register transfer*. An *eager and multicast transfer* mechanism reduces execution cycle penalty by 27% on average.

With both on-demand and eager/multicast transfers, the IPC ratios are 92%, 74%, and 83% for SPEC CINT2000, SPEC CFP2000, and Mediabench, respectively, in a 4-way configuration. They are 90%, 69%, and 76% in an 8-way configuration. Higher penalty is observed in a wider configuration because it is more distributed. Moreover, different types of applications incur varying degrees of penalty. It has been demonstrated that applications with higher IPC (e.g. SPEC CFP2000 and Mediabench) typically have higher penalty than applications with lower IPC (e.g. SPEC CINT2000). Finally, performance can be slightly improved by varying the number of register transfer bus and the number of local registers.

By implementing a distributed register file organization, register file access time can be reduced by 23% and 41% in 4- and 8-way configurations, respectively, with significant reduction in the delay of a bypass network. As a result, processors can be clocked faster, which allow overall performance to be improved despite execution cycle penalty. Significant reduction is also achieved for area and energy consumption.

5.1.2 Statically Scheduled Fully Distributed Register File Architecture

In the static case, code is generated specifically for a distributed register file architecture. Therefore, the tasks of local register mapping and register transfer scheduling are performed as part of a compiler back-end at compile-time as opposed to a run-time hardware mechanism in the dynamic case. These tasks are carried out in separate phases. First, machine instructions are scheduled and assigned to clusters. Then, register transfer operations are scheduled using multicasting and global scheduling. Finally, local registers are allocated accordingly.

The code generation algorithms are implemented to retarget Mediabench applications from PISA machine code to a distributed register file code using a basic-block list scheduling algorithm for instruction scheduling. Code size was increased by 36% on average due to the addition of register transfer operations. 46% of these extra operations incur extra scheduling slots while the rest can be scheduled into existing scheduling slots.

Distributed register file code is simulated using a common simulation framework as in the dynamic case. An average IPC ratio for the static approach is 77% compared to 83% obtained by the dynamic approach. The absolute IPC result is also lower in the static case, 0.7 for static compared to 1.3 for dynamic. This is mostly attributable to the limited ILP extracted by a basic-block scheduling approach. With improved scheduling, IPC can be improved but, at the same time, execution cycle penalty will increase due to their direct relationship as observed in the dynamic experiments. This makes the dynamic approach slightly more effective than its static counterpart. Increasing the number of register transfer buses and the local register file sizes has only a small impact on performance.

5.1.3 Fully Distributed Register Files for ILP-SIMD

The static approach to a distributed register file was applied to the ILP-SIMD processing elements (PE). Unlike general purpose cores, ILP-SIMD PE architecture is deterministic. All memory operations take one cycle, and branches are handled by the array controller. Since the target clock frequency is moderate, cycle time is not a critical problem. Instead, a central register file was shown to dominate die area in PE implementation. Since it is desirable to integrate more PE into a processor array, area demand reduction is an objective of this contribution.

Since the operating frequency is moderated, register transfer operations can be scheduled in the same cycle that the operand is created. With this relaxed scheduling algorithm, a distributed register file architecture incurs 4% and 9% more execution cycle than a central register file architecture for 2- and 3-way configuration, respectively. Performance of a 3-way configuration, however, is lower than a 2-way configuration because functional units are more distributed and the same number of functional unit are used in both configurations to keep die area small.

Die area is evaluated through GENESYS. With a distributed register file, area demand is reduced by 12% in a 2-way configuration. This extra area can be used to integrate 13% more PE into a processor array to exploit a higher degree of DLP.

5.2 Future Research Directions

The research presented in this dissertation is the first to explore and evaluate a distributed register file organization in various major processor architectures including superscalar, VLIW, and media processors. In this section, a number of related and interesting future research directions are outlined.

5.2.1 Distributed Register File Organization

- Develop methodology to evaluate the optimum cluster configuration (functional unit composition, local register file size, and their interconnection) for generic and/or specific applications domains.
- Evaluate various types of interconnection network and topology that permit efficient register operand transport and that can scale to an extremely large machine configuration.

5.2.2 Dynamic Approach to Distributed Register Files

- Evaluate the dynamic approach in parallel, multithreaded, and multiprogramming workloads, which demand processor architectures with a large number of functional units, such as wide SMT superscalar processors.
- Develop efficient mechanisms for fast context switching, exception handling, and interrupt handling in a distributed register file environment.
- Develop mechanisms to establish eager and multicast group and dynamically adapt to changing workload characteristics to support a large machine configuration where a single broadcast group is not efficient.
- Analyze operand usage and transport characteristics at compile-time and provide this information as hints from compilers so that a processor can make more effective cluster assignment and operand transport scheduling.
- Develop approaches to distribute or reduce complexity in other centralized control units and memory elements, such as fetch unit, local register mapping unit, and commit unit.

5.2.3 Static Approach to Distributed Register Files

- Apply state-of-the-art techniques for statically scheduled code generations, such as trace scheduling, data speculation, and control speculation, to the code generation framework in the static approach.
- Enhance scheduling and cluster assignment algorithms with software pipelining for distributed register files.

REFERENCES

[1] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *IEEE Computer*, September 1997.

[2] "The International Technology Roadmap for Semiconductors", http://public.itrs.net, 2003.

[3] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[4] K.I. Farkas, "Memory-system Design Considerations for Dynamically-scheduled Microprocessors," *PhD Thesis*, University of Toronto, Ontario, Canada, January 1997.

[5] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[6] C.-H. Jan, *et al*, "90 nm Generation, 300mm Wafer Low k ILD/Cu Interconnect Technology," *Proceedings of the International IEEE Interconnect Technology Conference*, June 2003.

[7] P. Kapur and K.C. Saraswat, "Comparisons between Electrical and Optical Interconnects for On-Chip Signaling," *Proceedings of the International IEEE Interconnect Technology Conference*, June 2002.

[8] K.K. O, *et al*, "Wireless Communications using Integrated Antennas," *Proceedings of the International IEEE Interconnect Technology Conference*, June 2003.

[9] M.F. Chang, V. Roychowdhury, L. Zhang, H. Shin, and Y. Qian, "RF/Wireless Interconnect for Inter- and Intra-Chip Communications," *Proceedings of the IEEE*, vol. 89, no. 4, April 2001.

[10] E.S. Fetzer and J.T. Orton, "A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium Microprocessor," *Digest of Technical Papers IEEE International Solid-State Circuits Conference*, February 2002.

[11] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, March/April 1999.

[12] A. Jain, *et al*, "A 1.2GHz Alpha Microprocessor with 44.8GB/s Chip Pin Bandwidth," *Digest of Technical Papers IEEE International Solid-State Circuits Conference*, February 2001.

[13] R.P. Preston, *et al*, "Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading," *Digest of Technical Papers IEEE International Solid-State Circuits Conference*, February 2002.

[14] S. Rixner, W.J. Dally, B. Khailany, P. Mattson, U.J. Kapasi, and J.D. Owens, "Register Organization for Media Processing," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, January 2000.

[15] S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A.López-Lagunas, P.R. Mattson, and J.D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[16] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no. 2, March-April 2001.

[17] B. Ramakrishna Rau, D.W.L. Yen, W. Yen, and R.A. Towle, "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs," *IEEE Computer*, January 1989.

[18] A. Seznec, E. Toullec, and O. Rochecouste, "Register Write Specialization Register Read Specialization: A Path to Complexity-Effective Wide-Issue Superscalar Processors," *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, November 2002.

[19] J. Janssen and H. Corporaal, "Partitioned Register File for TTAs," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[20] M.B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures," *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[21] A. Terechko, E.L. Thenaff, M. Garg, J.V. Eijndhoven, and H. Corporaal, "Inter-cluster Communication Models for Clustered VLIW Procesors," *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[22] J.M. Parcerisa, J. Sahuquillo, A. González, and J. Duato, "Efficient Interconnect s for Clustered Microarchitectures," *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[23] J.M. Mulder, N.T. Quach, and M.J. Flynn, "An Area Model for On-Chip Memories and its Application," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 2, February 1991.

[24] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," *Compaq WRL Research Report 2001/2*, August 2001.

[25] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu, "New Paradigm of Predictive MOSFET and Interconnect Modeling for Early Circuit Design," *Proceedings of IEEE CICC*, June 2000.

[26] "Berkeley Predictive Technology Model," http://www-device.eecs.berkeley.edu/~ptm.

[27] J.L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham, "Multiple-Banked Register File Architectures," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[28] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, December 2001.

[29] H.S. Lee, G.S. Tyson, and M.K. Farrens, "Eager Writeback – a Technique for Improving Bandwidth Utilization," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[30] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically Scheduled, Superscalar Processors," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[31] R. Canal, J.M. Parcerisa, and A. González, "Dynamic Cluster Assignment Mechanisms," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, January 2000.

[32] J.M. Parcerisa and A. González, "Reducing Wire Delay Penalty through Value Prediction," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[33] A. Aggarwal and M. Franklin, "Instruction Replication: Reducing Delays due to Inter-PE Communication Latency," *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[34] S. Bunchua, D.S. Wills, and L.M. Wills, "Reducing Operand Transport Complexity of Superscalar Processors using Distributed Register Files," *Proceedings of the IEEE International Conference on Computer Design*, October 2003.

[35] M. Franklin and G.S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992.

[36] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, February 2002.

[37] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997.

[38] R.P. Colwell, W.E. Hall, C.S. Joshi, D.B. Papworth, P.K. Rodman, and J.E. Tornes, "Architecture and Implementation of a VLIW Supercomputer," *Proceedings of Supercomputer '90*, November 1990.

[39] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, "The Multiflow Trace Scheduling Compiler," *Journal of Supercomputing*, vol. 7, no. 1-2, March 1993.

[40] J. Sánchez and A. González, "Instruction Scheduling for Clustered VLIW Architectures," *Proceedings of the 13th International Symposium on System Synthesis*, September 2000.

[41] J. Sánchez and A. González, "Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[42] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Two-level Hierarchical Register File Organization," *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[43] J. Zalamea, J. Llosa, A. Ayguadé, and M. Valero, "Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures," *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, December 2001.

[44] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Hierarchical Clustered Register File Organization for VLIW Processors," *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[45] A. Aletà, J.M. Codina, A. González, and D. Kaeli, "Instruction Replication for Clustered Microarchitectures," *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[46] Texas Instruments, "TMS320C64x Technical Overview," January 2001.

[47] O. Wolfe and J. Bier, "TigerSharc Sinks Teeth into VLIW," *Microprocessor Report*, vol. 12, no. 16, December 1998.

[48] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[49] P.N. Glaskowsky, "MAP 1000 Unfolds at Equator," *Microprocessor Report*, vol. 12, no. 16, December 1998.

[50] J.R. Ellis, "Bulldog: A Compiler for VLIW Architectures," *MIT Press*, 1986.

[51] E. Özer, S. Banerjia, and T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures," *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[52] P. Mattson, W.J. Dally, S. Rixner, U.J. Kapasi, and J.D. Owens, "Communication Scheduling," *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[53] K. Kailas, K. Ebcioglu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors," *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[54] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[55] S. Jang, S. Carr, P. Seany, and D. Kuras, "A Code Generation Framework for VLIW Architectures with Partitioned Register Banks," *Proceedings of the Third International Conference on Massively Parallel Computing Systems*, April 1998.

[56] C.H. Lee, M. Kim, and C.I. Park, "An Efficient K-Way Graph Partitioning Algorithm for Task Allocation in Parallel Computing Systems," *Proceedings of the First International Conference on Systems Integration*, April 1990.

[57] G. Desoli, "Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach," *HP Laboratories Technical Report HPL-98-13*, February 1998.

[58] M. Bekooij, J. Jess, and J.V. Meerbergen, "Phase Coupled Operation Assignment for VLIW Processors with Distributed Register Files," *Proceedings of the 14th International Symposium on System Synthesis*, September 2001.

[59] M. Bekooij, B. Mesman, J.V. Meerbergen, and J. Jess, "Constraint Analysis for Operation Assignment in FACTS," *Proceedings of IEEE ProRISC Workshop on Circuits, Systems, and Signal Processing*, December 2000.

[60] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring," *Computer Languages*, vol. 6, no. 1, 1981.

[61] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon, "Coloring Heuristics for Register Allocation," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, July 1989.

[62] H.H. Cat, *et al*, "SIMPil: An OE Integrated SIMD Architecture for Focal Plane Processing Applications," *Proceedings of the Third Annual Conference on Massively Parallel Processing using Optical Interconnections*, October 1996.

[63] A. Gentile, "Portable Multimedia Supercomputers: System Architecture Design and Evaluation", *PhD Thesis*, Georgia Institute of Technology, Atlanta, GA, November 2000.

[64] A. Gentile and S. Wills, "Portable Video Supercomputing," to appear in *IEEE Transaction on Computers*, August 2004.

[65] K.S. Chung, "ILP-SIMD: An Instruction Parallel SIMD Architecture with Short-Wire Interconnects," *PhD Thesis*, Georgia Institute of Technology, Atlanta, GA, April 2000.

[66] L.W. Tucker and G.G. Robertson, "Architecture and Applications of the Connection Machine," *IEEE Computer,* August 1988.

[67] T. Bridges, "The GPA Machine: A Generally Partitionable MSIMD Architecture," *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, October 1990.

[68] F. Boeri and M. Auguin, "OPSILA: A Vector and Parallel Processor," *IEEE Transactions on Computers*, vol. 42, no. 1, January 1993.

[69] H.J. Siegel, *et al*, "The PASM Project: A Study of Reconfigurable Parallel Computing," *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks*, June 1996.

[70] D.E. Schimmel, "Superscalar SIMD Architecture," *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, October 1992.

[71] MarPar Computer Corporation, "The Design of the MasPar MP-2: A Cost Effective Massively Parallel Computer," Sunnyvale, CA.

[72] V. Garg and D.E. Schimmel, "Hiding Communication Latency in Data Parallel Applications," *Proceedings of the First merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, March 1998.

[73] J.D. Allen and D.E. Schimmel, "The Impact of Pipelining on SIMD Architectures," *Proceeding of the 9th International Parallel Processing Symposium*, April 1995.

[74] J.C. Eble, V.K. De, D.S. Wills, and J.D. Meindl, "A Generic System Simulator (GENESYS) for ASIC Technology and Architecture Beyond 2001," *Proceedings of the Ninth Annual IEEE International ASIC Conference,* September 1996.

[75] S.M. Chai, T.M. Taha, D.S. Will, and J.D. Meindl, "Heterogeneous Architecture Models for Interconnect-Motivated System Design," *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 6, December 2000.