

# Reducing Operand Transport Complexity of Superscalar Processors using Distributed Register Files

Santithorn Bunchua, D. Scott Wills, and Linda M. Wills  
*School of Electrical and Computer Engineering*  
*Georgia Institute of Technology*  
*Atlanta, GA 30332*  
*{keh,scott.wills,linda.wills}@ece.gatech.edu*

## Abstract

*A critical problem in wide-issue superscalar processors is the limit on cycle time imposed by the central register file and operand bypass network. In this paper, a distributed register file architecture that employs fully distributed functional unit clusters is presented. It utilizes a local register mapping table and a dedicated register transfer network to support distributed register operations. In addition, an eager transfer mechanism is developed to reduce penalties caused by incomplete operand transport interconnection. Distributed register files can be employed to reduce operand access time by a factor of two with associated average IPC penalties of 14% and 21% on 4- and 8-way superscalar architectures across a broad range of symbolic, scientific, and multimedia applications. The IPC penalties are only 3% and 10% for SpecINT 2000 applications.*

## 1. Introduction

In a traditional superscalar architecture, increasing the superscalar factor has direct impact on register file performance since a large central register file is required to supply all necessary operands to possibly all functional units in each cycle. A large register file is difficult to implement and can become a cycle time bottleneck because of its area, delay, and power requirements, in addition to long wire delays for transporting operands to all destinations.

The problem of long wire delay is further exemplified by aggressive technology scaling since interconnect delay does not scale as well as gate delay. Therefore, global and semi-global interconnect are undesirable. Unfortunately, long interconnects are needed to transport operands between a central register file and all functional units. Similarly, an operand bypass network, which is another delay-critical unit, has its delay dominated by wire delay and consists mostly of long interconnects.

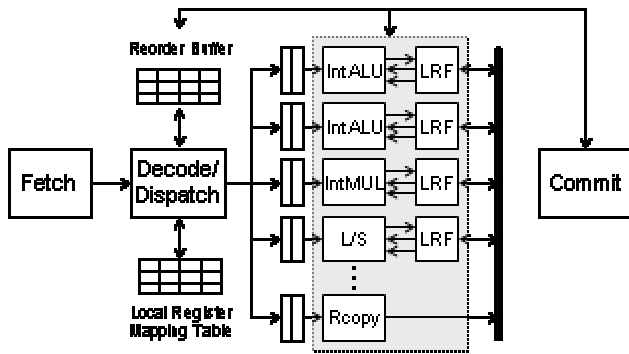
This paper presents a distributed register file organization that addresses this interconnect issue in operand transport in addition to its implementation complexity and cycle time constraint. The following section provides a brief discussion of related work. The architecture and the proposed mechanisms are described in Sections 3 and 4. Section 5 provides experimental results. The improvement in operand transport complexity is analyzed in Section 6. Conclusions are discussed in Section 7.

## 2. Related work

Early work on distributed register files and datapath clustering addresses operand transport complexity in very wide VLIW processors. It is explored in the context of superscalar processors when complexity and cycle time constraints become unmanageable as issue width and clock frequency increase.

The multicluster architecture [1] explicitly assigns registers to available clusters based on register names. If an instruction involves registers from multiple clusters, it is issued to all related clusters, and register values are transferred during its execution stage. Consequently, performance is highly dependent on the capability of compilers to generate effective register assignment.

Recent work extends the register renaming mechanism to support distributed register files. Register transfer operations are automatically inserted into an instruction issue queue as needed. Consequently, the instruction issue stage becomes more complex with extra functions of cluster assignment and register transfer operation issue. Based on this mechanism, [2] experiments with several cluster assignment algorithms, both static and dynamic, for quad-cluster machines. [3] proposes a value prediction scheme to reduce extra cycle penalties caused by register transfer operations. Operand values are predicted and used without waiting for the results of register transfer operations. If the predicted value is incorrect, all affected instructions need to be replayed. In addition, it requires a



**Figure 1. Distributed register file architecture**

prediction table, which can be large in wide-issue superscalar processors.

### 3. Architecture

Our fully distributed register file architecture is shown in Figure 1. 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 (not necessarily uniform). Local register files are connected through dedicated register transfer networks. Register transfer operations are dispatched similar to normal instructions and handled by a dedicated Rcopy unit.

Instructions are fetched and dispatched in-order to various issue queues in front of each cluster. The dispatch unit first determines the most suitable functional unit cluster for an instruction. It then allocates a new local register to hold the result and maps all input register operands to local registers according to the local register mapping table (LRMT). The LRMT keeps track of local registers that hold the current value of each architectural register. A write to an architectural register automatically invalidates all previous mappings of that specific register.

If there is no valid mapping in a chosen cluster, one or more register transfer operations are dispatched to transfer the value from remote register files. This operation is called an *on-demand register transfer*.

Functional unit assignment is highly critical to overall performance as explored in previously proposed cluster architectures ([2,3]). In our proposed scheme, a simple dependence-based assignment is used; therefore, only minimal extra circuits are required in the dispatch unit. For each instruction, the LRMT is examined to find a functional unit of an appropriate class that has the most operands in its local register file. If more than one functional unit qualifies, the least busy one is selected. If multiple units still qualify, any one of them can be selected.

Using a dependence-based scheme can raise an issue of workload imbalance among functional units. However, with the eager transfer mechanism described in the next section, variation of the functional unit assignment

scheme from a simple dependence-based one produces negligible difference in performance.

### 4. Eager and multicast transfer mechanisms

On-demand register transfers, as explained in the previous section, only issue register transfer operations as needed, i.e., only when an instruction that needs values from remote register files is dispatched. This usually delays the execution of the original instruction. (This delay may not occur if there are slacks in the execution pipeline due to other data or resource dependencies). The eager transfer mechanism reduces this delay penalty by issuing register transfer operations in advance. This form of proactive transfer can be effective if the distance between the value producing instruction and the value consuming instruction is large enough. In addition, it helps distribute workload to all connected clusters.

To issue an eager register transfer operation, the following components need to be determined: source registers, source/destination clusters, and an issue cycle. At each cycle, there can be several candidate registers. Since most values are used soon after they are defined (temporal locality [4]), the most recently defined register is chosen as the source register for eager transfer.

Next, the source and destination clusters need to be determined. Since a dedicated register transfer network is available, any functional unit cluster that has a valid mapping of the source register can be used. Multiple destination clusters can be chosen by means of multicast. Multicast transfer is employed to transfer the source register value to all local register files that are connected to the same register transfer network and that still have free local registers. Multicasting can be efficiently implemented with minimal implementation complexity, especially in a bus or crossbar network.

Multicasting is also used for all on-demand register transfers. Therefore, whenever an on-demand register transfer is issued, eager transfer operations are also embedded by transferring the value to all other local register files in addition to the originally intended destination local register file.

Finally, the decision needs to be made whether eager transfer operations will be issued in the current cycle. In the current implementation, an eager transfer instruction is issued whenever the issue queue of the register transfer unit is empty.

### 5. Experimental results

The distributed register file architecture and the eager and multicast transfer mechanisms are simulated using the SimpleScalar toolset [5]. Both 4-way and 8-way configurations are simulated with parameters as shown in Table 1. Note that functional unit classes are non-

**Table 1. 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	

overlapped and a large enough number of functional units are provided to minimize the possibility of 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. The results presented in this section, therefore, are considered conservative with ample room for optimization. Eighteen applications from SpecCPU 2000 and MediaBench are chosen. All applications are compiled to PISA binaries using gcc 2.7.2.3 and the default compilation options as specified by each benchmark. Applications are chosen based on no particular preference. All compilations and simulations are performed on an x86-based machine running Linux operating system. All simulations are run for 100 million instructions (the first 50 million instructions are skipped in all SpecCPU 2000 applications). Three architectures are simulated:

- *Base* – a traditional superscalar architecture
- *Drf* – a distributed register file architecture
- *Drf+Eager* – *Drf* with eager and multicast.

Simulation results comparing raw IPC for an 8-way machine configuration are shown in Figure 2. The IPC results show a performance penalty caused by incomplete interconnection among register files and functional units. This penalty is the result of functional unit assignment constraint and extra cycles incurred by register transfer operations. The results also show the effectiveness of the

eager and multicast transfer mechanisms, which can reduce the extra cycle penalty by 26% on average.

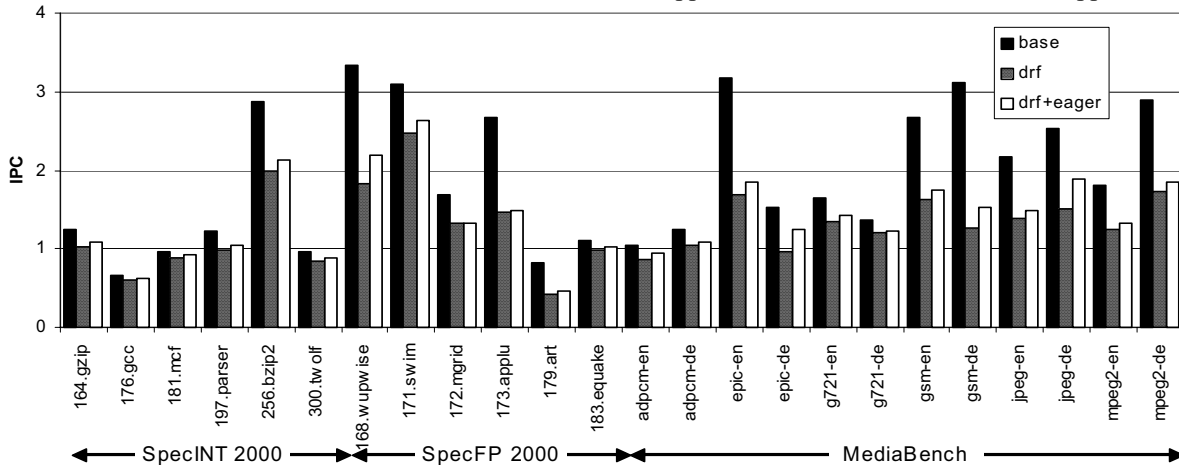
To quantify performance penalties, the average IPC penalty result is shown in Table 2. An IPC penalty is computed using the following equation.

$$IPC\ penalty = 1 - \frac{IPC_{Drf\ or\ Drf+Eager}}{IPC_{Base}}$$

Performance is shown to be highly dependent on application characteristics. Low IPC penalties are observed for SpecINT 2000 while significantly higher penalties are observed for SpecFP 2000 and MediaBench.

Typically, performance degradation can be from two sources: remote register communications and functional unit imbalance. These events can delay execution of some operations. However, performance degradation occurs only when these delays cause extra cycles in the whole program execution. In a superscalar execution, several run-time conditions can contribute to slack in the execution pipeline, such as speculative execution of a mispredicted branch, cache miss on load operations, idle functional units during normal execution, etc. These slacks allow execution delays caused by distributed register file operations to be hidden and to reduce overall performance penalties.

Application characteristics directly impact the amount of slack in superscalar execution. The IPC of an application running on the *Base* architecture can be used to approximate the amount of slack. Applications with



**Figure 2. IPC comparisons for an 8-way machine configuration**

**Table 2. IPC penalty (4-way and 8-way)**

	4-way		8-way	
	Drf	Drf+Eager	Drf	Drf+Eager
SpecINT2000	6%	3%	14%	10%
SpecFP2000	30%	26%	35%	31%
MediaBench	23%	17%	31%	24%
Overall	18%	14%	26%	21%

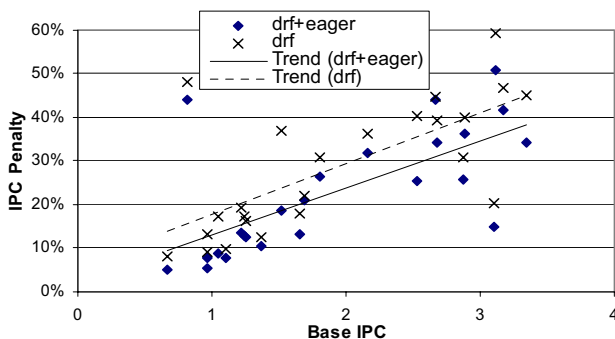
high IPC tend to make the execution pipeline busy with less slack. Therefore, applications from SpecINT 2000, which have low IPC, show only small penalty in contrast to applications from SpecFP 2000 or MediaBench. The relationship between IPC (*Base*) and IPC penalty is shown in Figure 3. Note that this observation only shows the trend of the relationship. Other run-time conditions can sometimes cause a low IPC application to suffer large performance degradation or vice versa.

## 6. Operand Transport Complexity

The primary motivation of a distributed register file is to reduce operand transport complexity as the superscalar factor is increased. Register file operations and a bypass network are two critical components that are cycle-time limited. They are especially critical in a deep submicron technology 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 is estimated and compared to a traditional central register file configuration.

Register file access time is estimated using CACTI [6]. A 128 32-bit register file with 16 read ports and 8 write ports is used for a central register file and is compared to a local register file with 32 32-bit registers, 2 read ports, 1 write port, and 1 read/write port. The results from CACTI show a 47.8% reduction in access time for a distributed register file across all technologies.

A bypass network delay is estimated using a methodology similar to [7]. Delay of a bypass network is dominated by delay of the wires, which run across the height of all functional units and a central register file. Delay is directly related to  $L^2$ , where  $L$  is wire length. In a distributed register file configuration, a bypass network can be eliminated or can be used to bypass only for a



**Figure 3. Relationship between Base IPC and IPC penalty (an 8-way machine configuration)**

single functional unit. The delay of a bypass network, therefore, becomes negligible (no more than 2% of the original delay in an 8-way machine configuration).

## 7. Conclusion

Operand transport complexity becomes a limiting factor in improving superscalar processor performance. A distributed register file architecture is presented to address this complexity. The architecture features heterogeneous functional unit clusters, a local register file per cluster, and a dedicated register transfer network. Using fully distributed and heterogeneous functional unit clusters allows for flexible processor configurations and optimal physical operand transport performance.

The overhead of register transfer operations can be reduced by 26% using the eager and multicast transfer mechanisms. The overall penalties over a wide range of applications are 14% and 21% on 4- and 8-way configurations (only 3% and 10% for SpecINT 2000). Analyses of operand transport complexity show a potential operand transport speedup, which exceeds performance overhead of distributed register operations and results in overall performance improvement over a traditional superscalar architecture with a central register file.

## Acknowledgement

This work was jointly supported by the National Science Foundation under NSF CAREER Grant CCR-0092552, DARPA Grant under contract # MDA972-99-1-0002, and MARCO under contract # 98-IT-674.

## References

- [1] K.I. Farkas, *et al.*, "The Multicluster Architecture: Reducing Cycle Time through Partitioning," *Proc. of the 30<sup>th</sup> Int. Symp. on Microarchitecture*, December 1997.
- [2] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically Scheduled, Superscalar Processors," *Proc. of the 33<sup>rd</sup> Int. Symp. on Microarchitecture*, December 2000.
- [3] J.M. Parcerisa and A. González, "Reducing Wire Delay Penalty through Value Prediction," *Proc. of the 33<sup>rd</sup> Int. Symp. on Microarchitecture*, December 2000.
- [4] M. Franklin and G.S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proc. of the 25<sup>th</sup> Int. Symp. on Microarchitecture*, December 1992.
- [5] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, February 2002.
- [6] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," *Compaq WRL Research Report 2001/2*, August 2001.
- [7] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. of the 24<sup>th</sup> Int. Symp. on Computer Architecture*, 1997.