ARCHITECTURAL ENHANCEMENTS FOR EFFICIENT OPERAND TRANSPORT IN MULTIMEDIA SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Hongkyu Kim

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

Georgia Institute of Technology
May, 2007

ARCHITECTURAL ENHANCEMENTS FOR EFFICIENT OPERAND TRANSPORT IN MULTIMEDIA SYSTEMS

Approved by:

Dr. D. Scott Wills, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda M. Wills, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Jeffrey A. Davis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Allen R. Tannenbaum
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Gabriel H. Loh
College of Computing
Georgia Institute of Technology

Date Approved: December 18, 2006

Dedicated to my grandmother, forever on my mind

We never had a chance to say goodbye to each other and
I never really had a chance to thank her for all she had done for me.
This is my tribute to her. Thank you, Grandma!

ACKNOWLEDGEMENTS

This dissertation could not be completed without the support of many people I had to express my gratitude. I would like to thank Dr. Scott Wills, my advisor, for his continuous encouragement and advice throughout my Ph.D. study. Dr. Linda Wills, my co-advisor, has also supervised and guided this research with her kind advice and attention to detail. It has been a great pleasure to have you, Scott and Linda, as my advisors during my stay at Georgia Institute of Technology.

I thank my committee members, Dr. Hsien-Hsin Lee, Dr. Jeffrey Davis, Dr. Allen Tannenbaum, and Dr. Gabriel Loh for their time, efforts, and suggestions. Their constructive comments have improved the quality of this research.

I wish to extend my thanks to all PICA and EASL research group members, both alumni and current, for their helps and friendship: Dr. Santithorn Bunchua, Dr. Peter Sassone, Dr. Soojung Ryu, Dr. Jongmyon Kim, Krit Athikulwongse, Cory Hawkins, Senyo Apewokin, Nidhi Kejriwal, and Brian Valentine, in no particular order.

I am indebted to my beloved parents, Moonam Kim and Youngsuk Kim, who have provided me with their dedicated love and sacrifices for my life. I also give my special thanks to my brother Hongjoon and his wife for their understanding.

Finally, I cannot fail to thank my lovely wife, Minah Cho, for her love, friendship, patience, and understanding through the final moment of my study. Without her unselfish devotion and endless support, e.g. lunch boxes, this dissertation could not have been accomplished.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Multimedia applications pose new challenges to computer architecture. Their tremendous communication demands severely burden the interconnect between functional units, which has become a bottleneck in high performance architectures. This dissertation addresses the critical challenge in multimedia processors: to efficiently transport operands among computational and storage components. It provides architectural enhancements that enable the high bandwidth, low latency communication demanded by multimedia applications.

This research analyzes multimedia workloads to characterize the communication patterns that occur in the execution of standard multimedia benchmarks. This empirical analysis indicates that most operands exhibit strong locality, enabling several optimizations of transport mechanisms, particularly to operand transport networks, storage structures, and instruction steering algorithms. This empirical study shows that an eight-entry local buffer with approximate information on operand lifetime is sufficient to suppress 81% of operand writes. In addition, chaining selected pairs of FUs based on producer-consumer information allows 50% of reads to be accessed through the shortest path.

These results guide the design and development of two efficient operand transport mechanisms: (i) a traffic-driven operand bypass network and (ii) a dynamic instruction clustering. The *traffic-driven operand bypass network* is designed using a novel, systematic design customization process for wide-issue architectures. It is driven by a technology model-based evaluation methodology on different execution engines,

resulting in a low cost, high performance bypass network targeted for multimedia applications. This technique places microarchitectural components exploiting the transport communication patterns, reorganizes each of the bypass paths based on the traffic rate, and maps inter-instruction communication on the local paths. The reduction in operand transport latency combined with a faster clock cycle achieves an instruction throughput gain of 2.9x over the broadcast bypass network at 45nm. In addition, the instruction throughput gain over a typical clustered architecture is 1.3x.

Dynamic instruction clustering groups dependent instructions into clusters during instruction execution, detects the operand lifetime, performs intra- and inter-cluster operand transport pattern analysis, and maps the clustered instructions to an efficient cluster execution unit. Two cluster execution unit implementations are explored: network ALUs and a dynamically-scheduled SIMD PE array. In the network ALUs, intermediate values within the inner loops are propagated among ALUs without distribution through global bypass buses. The reduction in operand transport latency results in a 35% IPC speedup over a conventional ILP processor. The dynamically-scheduled SIMD PE array supports DLP processing of the innermost loops in image processing applications. Data-parallel operations combined with localized operand communication produce an IPC speedup of 2.59x over a 16-way, four-clustered microarchitecture.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Traditionally, computer builders have focused primarily on the design of individual functional units (FUs) and storage components, since these elements consumed the majority of implementation resources (typically transistors). With advances in integrated circuit technology over the past few decades, transistor feature size (i.e., the minimum dimension of a transistor) has been continuously scaled down, making transistors both smaller and faster. This technology trend supports higher clock rates and increased integration of computational elements. Over the same period, advances in on-chip interconnect have not matched device improvements. Demands for faster clocks, larger chips, and increased transistor counts are contributing to an interconnect bottleneck in system performance [42].

Semiconductor industry projections (shown in Figure 1) from the International Technology Roadmap for Semiconductor (ITRS) [62] indicate a growing disparity between wire and gate delays as feature size shrinks; wire delay will contribute a growing fraction of signal delay and become a dominant component in processor cycle time. Interconnect issues are currently a dominant concern in the design of next-generation processors.

ITRS trends suggest the need to focus less on “transistor-centric” design and more on “interconnect-centric” techniques [18]. A focus on new interconnect organization and technology is yielding new techniques for high-performance interconnects. Examples

include 3D integrations, optical or radio frequency interconnects, and polylithic integrations [42]. Though these techniques are promising, the architectural response to these changes in technology is limited by the required compatibility with decades-old instruction set architectures (ISAs) that emphasize sequentially specified operations and a restricted register file-based operand namespace. New software-compatible architectures are needed that introduce novel interconnect strategies to deliver increased system performance.

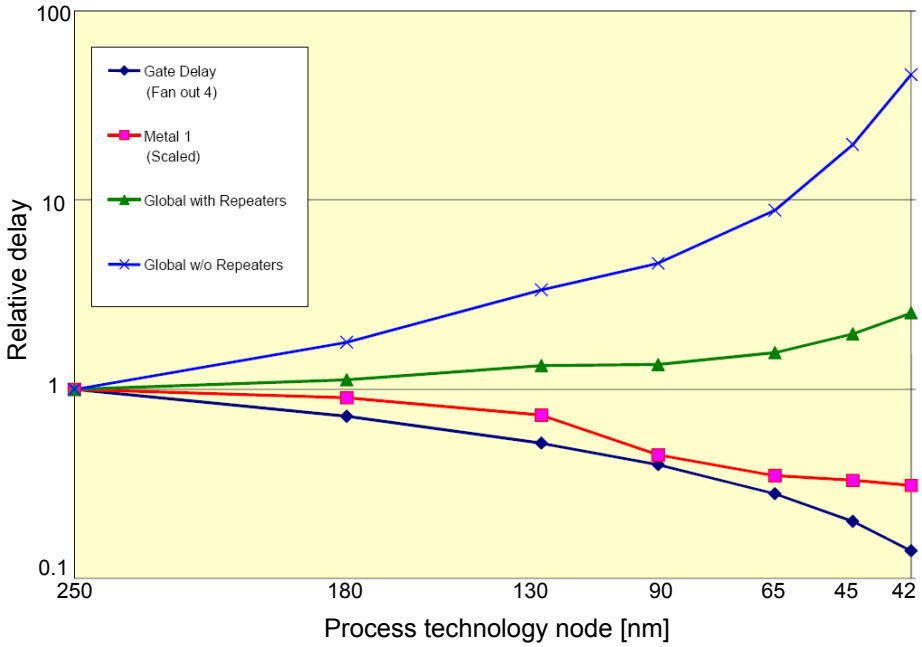


Figure 1: Delays for gate and wires versus feature size [62].

Computer architecture should also address application needs since it links applications and technologies. As workload requirements change, computer architects must produce innovative systems that can deliver needed performance and cost effectiveness [1]. Multimedia workloads have become increasingly important in general-purpose computing as well as embedded systems. It is predicted that media processing

will become the dominant force in computer architecture and microprocessor design in the near future [21].

Media-centric applications pose new challenges to processor architecture. Symbolic applications (e.g., office worksuites) that have dominated desktop and laptop computing are characterized by complex control flow, limited inherent parallelism, scalar processing of integer data types, and short data dependence distances. In contrast, multimedia-centric applications have several distinguishing characteristics: (i) real-time processing of continuous media data streams composed of large collections of small data elements, (ii) rich fine-grained parallelism (both instruction-level parallelism (ILP) and data-level parallelism (DLP)), (iii) high instruction reference locality in a small number of loops, and (iv) computationally intensive routines with highly predictable branches [17].

Multimedia processors exploit higher levels of application parallelism by employing large numbers of FUs and associated operand communication mechanisms. This exacerbates the interconnect problem in current processor architectures. Techniques to support parallel execution such as operand bypassing and instruction wakeup/select require a large number of non-local interconnects. Since they typically employ poorly scaling broadcast buses to distribute operands, these mechanisms are expensive to implement and often limit performance [50]. The critical challenge in multimedia processors is to efficiently transport operands among computational and storage components.

1.2 Research Approach Summary

This dissertation presents approaches to reduce the latency associated with operand movement within a parallel datapath, especially for multimedia applications. This research shifts the microarchitectural focus from operand computation to operand transport, which addresses the delivery of operands to FUs that require them. It specifically concentrates on the operand transport network, which carries operands between hardware resources.

The research approach exploits the unique properties of operand movement typically found in multimedia applications. In particular, these applications typically consist of the uniform processing of stream-oriented input data. In addition, execution is dominated by a small number of complex but deterministic data flow patterns within loop bodies. Loop kernels typically span tens (and occasionally hundreds) of instructions that are iterated over hundreds or thousands of times.

This research develops and evaluates dynamic execution techniques that recognize and exploit regular operand distribution patterns in multimedia applications to reduce the latency, storage requirements, and interconnect demands of operand transport. Additionally, this research develops lower cost transport mechanisms than traditional bypass networks for multimedia applications by converting global communication needs to local transport, exposing opportunities for lower latency. Cost analysis is performed with respect to expected VLSI implementations by evaluating the mechanisms across a range of future technology points using technology modeling.

Contribution 1: Characterization and modeling of operand usage and transport

To develop efficient operand transport mechanisms for existing ISAs, this research begins by studying the characteristics of operands in the execution of standard multimedia application benchmarks from MediaBench [39]. Recognition and understanding of operand usage and transport properties are important to efficiently control operand traffic. This research contribution strives to characterize the distributions and modes of operand movement between storage and FUs during the execution of application programs. Of particular interest are the temporal locality and spatial locality of operands.

Architectural techniques that exploit these operand transport characteristics are implemented and their effectiveness in reducing operand traffic is evaluated. These techniques include transport network configuration, storage organization, lifetime detection, and instruction steering strategy. Results of this contribution [34] show that (i) 25% of operand reads are accessed through the shortest paths with eight-entry local storage and bypass paths; (ii) 81% of operand writes to global storage are eliminated by applying dynamic register operand lifetime detection; and (iii) 50% of operands are read directly from local storage by adding dedicated bypass paths between heavily trafficked resources and by applying a novel instruction mapping scheme based on operand consumer information.

Operand transport characteristics extracted from this empirical analysis are the key to developing novel communication mechanisms. Two architectural techniques are presented in the following contributions.

Contribution 2: Customizing operand bypass network

This architectural contribution explores improved operand bypass networks. The bypass networks of ILP processors are targeted since their wiring demands are particularly high and the forwarding path delay of conventional broadcast-style buses is a limiting factor of processor performance. Technology modeling techniques for architectural evaluation are combined with cycle-accurate simulation to measure the operand transport cost in interconnect and buffering when representative platforms execute multimedia benchmark programs. Using technology modeling and operand characteristics from workload analysis, this technique provides a lower cost, higher performance bypass network, especially for multimedia applications.

Our technique places microarchitectural components to exploit the transport communication patterns, reorganizes each of the bypass paths based on the traffic rate, and maps inter-instruction communication on the local paths [35]. The reduction in operand transport latency combined with a faster clock rate achieves an instruction throughput gain of 2.9x over the broadcast bypass network using 45 nm technology. The total length of the bypass wires can be kept within 24% that of the broadcast bypass network. In addition, the instruction throughput gain over a typical clustered architecture is 1.3x with only 50% of the total bypass wire length of the clustered architecture.

Contribution 3: Dynamic instruction clustering

This architectural contribution is a dynamic execution mechanism that extracts more parallelism and reduces operand transport latency based on the operand transport pattern analysis. It exploits the regular operand transport patterns and the plentiful parallelism of multimedia applications to achieve greater instruction throughput. This

dynamic execution technique (i) dynamically groups data-dependent instructions into *clusters*, (ii) detects operand lifetime, (iii) recognize of intra- and inter-cluster operand access patterns, and (iv) maps the clustered instructions to a specialized cluster execution unit.

Two cluster execution unit implementations are presented and evaluated: network arithmetic and logic units (ALUs) and a dynamically scheduled single-instruction, multiple-data (SIMD) processing element (PE) array. In the network ALUs, intermediate values are transported among ALUs using local, dedicated paths rather than global bypass buses. The reduction in operand transport latency results in a 35% instruction per cycle (IPC) speedup over a conventional ILP processor [36]. The SIMD PE array supports data-parallel processing dynamically. It also exposes opportunities to lower operand transport latency by converting global communication into local transport and by removing unnecessary communication. The resulting latency reduction combined with increased parallelism of additional FUs produces an IPC speedup of 2.59x over a 16-way, four-clustered microarchitecture [37].

1.3 Overview of Content

The architectural community is responding to the interconnect problem with a variety of approaches, including new microarchitectures and instruction sets, better compilation techniques, and improved run-time mechanisms. This research addresses the interconnect problem using current ISAs and compilers, minimizing the transition cost for new processor designs. A major contribution of this dissertation is the development of efficient operand transport mechanisms, especially focusing on multimedia applications. It includes a study of operand characteristics, an evaluation of bypass network

architectures, and the development of dynamic execution techniques to efficiently control the operand movement within a datapath.

This dissertation is organized as follows. Chapter 2 characterizes the operand usage and transport properties for multimedia applications. Chapter 3 presents a traffic-driven operand bypass network for dynamically scheduled architectures to reduce the wire delay latency. Chapter 4 introduces the concepts of a dynamic instruction clustering mechanism and operand transport pattern recognition technique. Two implementations are presented as examples of efficient cluster execution units: network ALUs for standard multimedia applications and a SIMD PE array targeting for image processing applications. Finally, Chapter 5 presents a summary of results and suggestions for future research.

CHAPTER 2

CHARACTERIZATION AND MODELING OF OPERAND USAGE AND TRANSPORT

2.1 Introduction

Technology advances in the past decade have created opportunities for processors to support higher degree of parallelism inherent in the applications. With smaller and faster transistors, computer designers can integrate a large number of FUs and a high volume of storage to meet the required performance. This is particularly true for multimedia architectures since the multimedia applications are typically computation-intensive, require high throughput, and contain abundant parallelism. However, increasing wire delay make achieving higher parallelism difficult. An efficient operand transport mechanism is a critical challenge in processor design, which is optimized for the required operand communication.

Knowledge of data communication is the key to designing and making efficient use of communication structures. Toward this end, this research analyzes multimedia application workloads to understand the communication needs that occur in the execution of standard multimedia benchmarks. This involves modeling the usage and transport properties of the operands. The purpose of the analysis to characterize how operands move around, how often and where they are used, and what accounts for the majority of communication needs between FUs and storage during execution of application programs. Particularly, two aspects of operand locality properties are addresses: temporal locality and spatial locality.

Architectural techniques that exploit these operand transport characteristics are implemented and their effectiveness in reducing operand traffic is evaluated. These techniques include transport network configuration, storage organization, lifetime detection, and instruction steering strategy. Results that (i) 25% of operand reads are accessed through the shortest paths with eight-entry local storage and bypass paths; (ii) 81% of operand writes to global storage are eliminated by applying dynamic register operand lifetime detection; and (iii) 50% of operands are read directly from local storage by adding dedicated bypass paths between heavily trafficked resources and by applying a novel instruction mapping scheme based on operand consumer information [34].

The rest of this chapter is organized as follows. Section 2.2 introduces the empirical study by defining terms and metrics. It also presents our research methodology to analyze the operand usage and transport properties. Data on the operand locality properties and results of an empirical study of the operand traffic appear in Section 2.3. Section 2.4 summarizes conclusions.

2.2 Methodology

This section describes the methodology used in the empirical study of operand usage and transport properties in multimedia applications. Figure 2 shows the simulation environment based on the *Simplescalar* simulator [4] with the PISA (Portable Instruction Set Architecture) – a MIPS-like instruction set. The *sim-safe* implementation of the Simplescalar is extended to handle operand-based operations instead of traditional register-based operations, i.e., a unique operand identifier is allocated to each new instance of a register name and a memory location. All data are measured from a trace-driven simulation of a dynamic instruction stream.

The MediaBench [39] suite with default inputs is used as our set of benchmarks. Each benchmark program is simulated until completion, but the dynamic instruction window is limited to 100,000 instructions to complete the simulation in a reasonable amount of time. Table 1 briefly describes the applications in our test suite and lists the characteristics of the programs, including the total number of instructions executed in millions, the operand production rate (N_{PROD}), and the operand consumption rate (N_{CONS}) – the number of operands produced/consumed (respectively) per instruction.

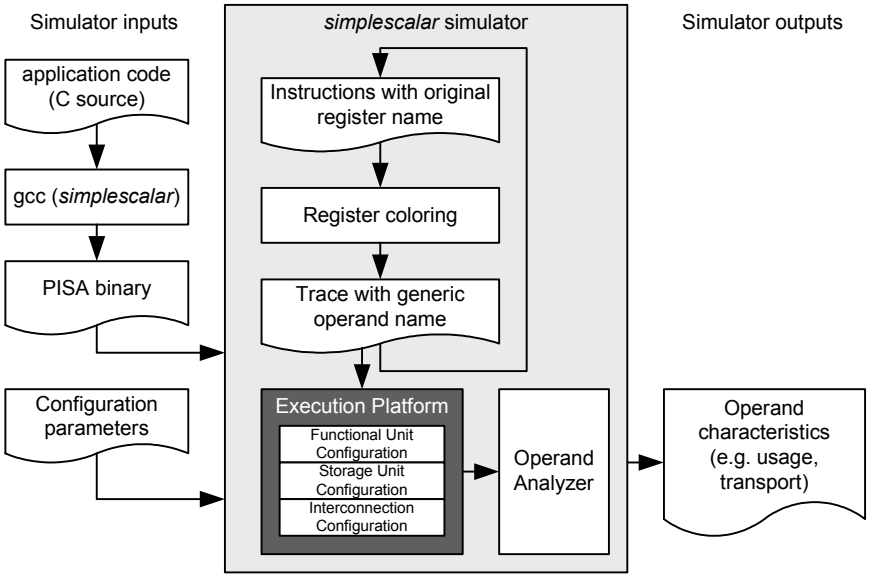


Figure 2: Simulation environment using Simplescalar toolset.

In this research, an operand is defined as a value in a register or a memory location. An operand is created or produced when an instruction generates a new value or an instruction accesses a value in a memory location for the first time. In addition, an operand is consumed or used when an instruction accesses a value from ISA-visible registers or memory locations. The operand production and consumption depend on the given ISA and the distribution of the executed instructions. When a typical two-input, one-output RISC instruction set is assumed, the operand production rate is just below one

and the operand consumption rate is between one and two as shown in Table 1. These rates give a measure of the total amount of operand traffic in the execution model.

Table 1: MediaBench application programs.

Name	Description	Total executed inst.	N_{PROD}	N_{CONS}
rawcaudio	adaptive differential pulse code modulation of audio coding/decoding	6.6	1.22	0.67
rawdaudio		5.4	1.18	0.63
epic	an experimental image compression/ decompression utility based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Hoffman entropy coder	52.7	1.49	0.85
epicun		6.7	1.34	0.65
g721decode	reference implementations of the CCITT G.721 voice compression/decompression	274.8	1.21	0.70
g721encode		267.6	1.21	0.69
gsmencode	European GSM 06.10 provisional standard for full-rate speech transcoding (encoding/decoding)	234.7	1.42	0.87
gsmdecode		75.8	1.22	0.60
cjpeg	a standardized compression/decompression method for full-color and gray-scale images	15.5	1.33	0.64
djpeg		4.6	1.53	0.80
mpeg2encode	a standard for high-quality digital video transmission (encoding/decoding)	1134.2	1.56	0.84
mpeg2encode		171.2	1.55	0.89

To understand the nature of the operand traffic that takes place in a benchmark program, two kinds of characteristics are analyzed: (i) which instructions consume or use operands after they are produced (*operand temporal locality property*), and (ii) from/to which FU are operands moved in the execution model (*operand spatial locality property*).

- **Metrics for temporal locality properties**

Temporal locality metrics defined by Franklin and Sohi [26] are adopted, but they are applied to memory as well as register operands. The temporal locality of each operand is determined by three metrics: *degree of use*, *operand lifetime* and *operand age*.

The degree of use indicates the number of times an operand is consumed. The operand lifetime is the distance in number of instructions between an operand creation and its last consumption. It determines how long the operands should be held in some form of storage, such as local register, global register, or memory location. The operand age is the distance in number of instructions between an operand production and its first consumption, which determines the minimum amount of time that the operand has to be kept in storage.

- **Metrics for spatial locality properties**

The following metrics have been defined to determine the spatial locality of operands: *degree of functionality*, *operand read transport rate (T_{rd})*, and *operand write transport rate (T_{wr})*. The degree of functionality is the number of FU types that use an operand; the higher degree of functionality an operand has, the more it needs to be communicated among FUs. The operand read transport rate and operand write transport rate are defined as follows.

$$T_{rd} = \frac{\sum_{trace} transported\ read}{\sum_{trace} operand\ read}, \quad T_{wr} = \frac{\sum_{trace} transported\ write}{\sum_{trace} operand\ write} \tag{1}$$

Note that the operand read transport rates are evaluated on each interconnect where the operands pass through, while the operand write transport rates are measured on each storage component where the operands reside.

While the temporal property metrics are determined only by the instruction sequences in the trace, the spatial property metrics depend not only on the instruction sequence, but also on the configurations of the execution model. The configurations may include the functionalities of each FU, the number of FUs, storage models, and transport

network models. In addition, architectural techniques, such as instruction mapping strategy and operand write scheme, also affect the transport rates. For example, when two levels of operand storage hierarchy are assumed (global buffers and local buffers attached to each FU), the most efficient place to read (write) an operand is the nearest buffer to the FU that consumes (produces) an operand – its own local buffer. An operand read transport occurs when an operand required by a FU is read from the local buffer of another FU or from global storage. A transported operand write occurs when the producer's local buffer is full and an operand needs to be written back to the global storage.

2.3 Empirical Analysis of Operand Usage and Transport

Our empirical analysis studies operand usage and communication patterns in the execution of standard multimedia application programs (e.g., MediaBench). The observed characteristics are then exploited to devise architectural techniques that localize operand communication. After the execution model is built based on FUs for operand computation, storage elements for buffering, and a communication network for operand transport, we explore the impact of several architectural techniques on operand transport. Our empirical study reveals how much local storage and what kind of additional information is needed to improve operand transport.

2.3.1 Operand Locality Characteristics

Figure 3 shows data on the observed temporal locality characteristics for the MediaBench application programs. Each graph represents the percentage distribution of the degree of use (Figure 3(a)), the operand age (Figure 3(b)), and the operand lifetime

(Figure 3(c)) described in Section 2.2. The x-axis denotes the benchmark programs and the right most bar represents the average over all MediaBench programs.

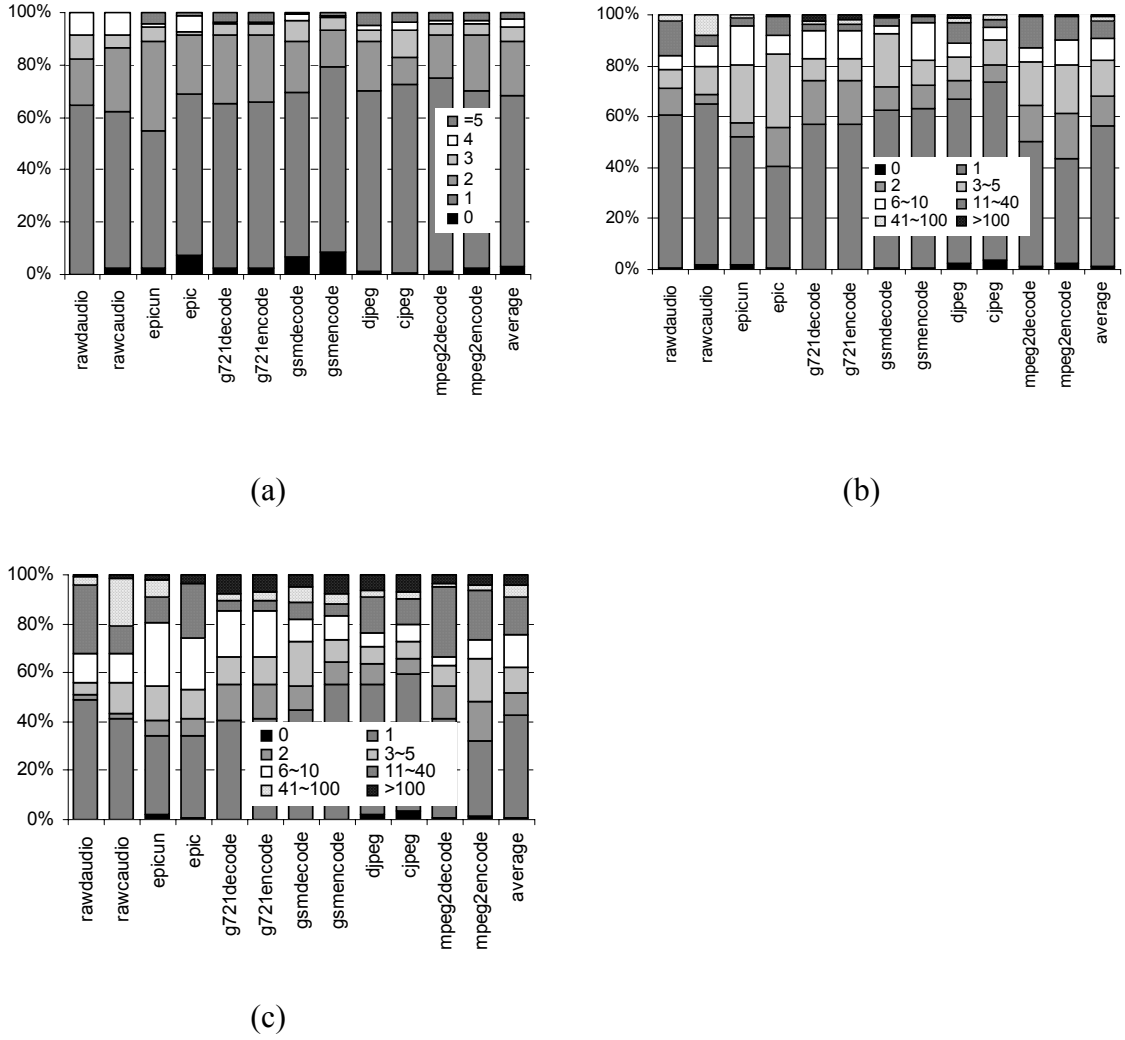


Figure 3: Observed operand temporal locality characteristics: (a) degree of use, (b) operand age, and (c) operand lifetime distribution.

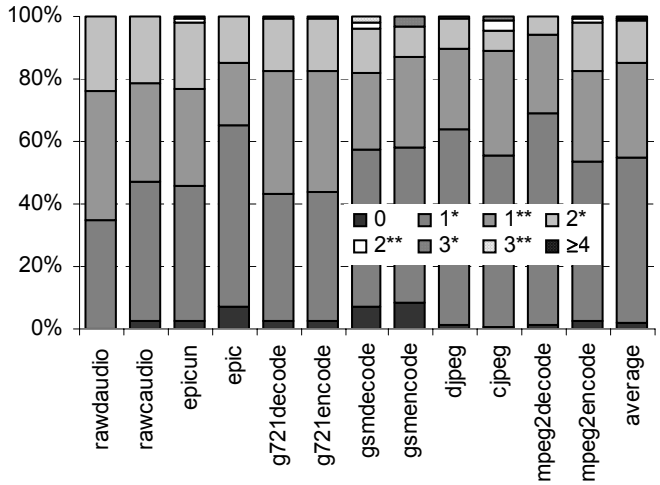
From Figure 3, we can observe that (i) operands tend to be used only a small number of times – on average 94.7% of operands are used at most three times, (ii) most operands are first consumed just after they are produced – 82.5% of operands are consumed within five dynamic instructions, and (iii) most operands have short lifetimes – 75.8% of operands are dead within a dynamic instruction window of size ten.

Interestingly, although the lifetimes of most operands are very short, average lifetime is extremely long – 451 dynamic instructions, which means a very small number of operands, typically memory operands, are long-lived. In fact, 68.7% of long-lived operands defined as operands living longer than 100 dynamic instructions are classified as memory operands.

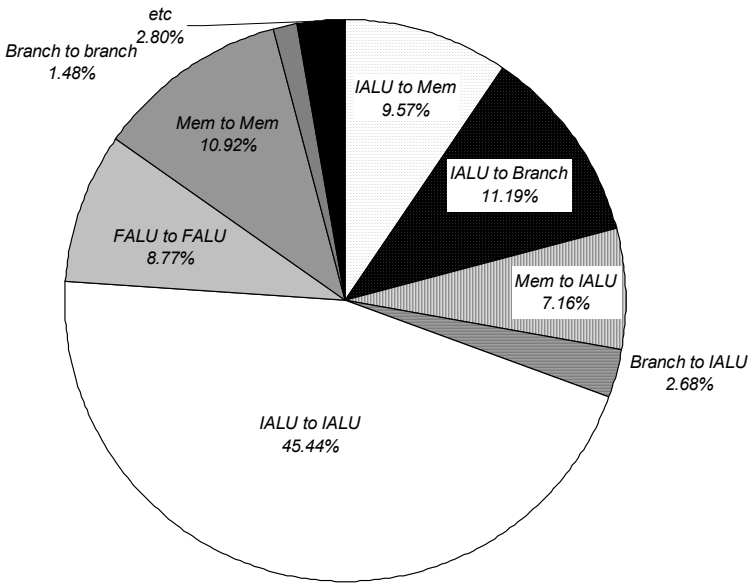
The locality properties are evaluated in the space domain as well as in the time domain. Figure 4 depicts data on the observed spatial locality properties. Each bar in Figure 4(a) represents percentage distribution of the degree of functionality. The number in the index indicates the number of FU types that consume an operand. In this analysis five FU types are assumed – memory unit (*Mem*), branch unit (*Branch*), integer ALU (*IALU*), integer multiplier (*IMUL*), and floating-point unit (*FALU*). In the graph, N^{*} denotes that the number of consumer FU types are N in which the producer type is included and N^{**} denotes N FU types excluding the producer type. The inclusion of the producer FU type must be differentiated since each operand transport can be optimized based on the different strategies. For example, if an operand is consumed by the same FU type as the producer, both instructions can be mapped to the same resource, removing the communication. However, when it is used by different FU types, a transport from the producer to its consumer cannot be avoided. In this case, special care should be taken for the transport such as assigning both instructions on the nearest resources connected by a local, dedicated path.

Looking at the data presented in Figure 4(a), a large number of operands in all benchmark programs are used by only one FU type. For example, on average, 52.3% of

operands are used by only the same FU type as the producer and 30.5% of operands are consumed by only one different FU type from the producer.



(a)



(b)

Figure 4: Observed operand spatial locality characteristics: (a) degree of functionality and (b) transport pattern distribution.

Figure 4(b) depicts percentage distribution of operand transport between FUs. The data represent the average distribution over all MediaBench programs. The result in

Figure 4(b) shows that the transport pattern is not evenly distributed spatially – certain paths are more heavily trafficked than others. For example, about 45% of operands are communicated between integer ALUs since a significant amount of integer ALU operations and data dependences between integer ALU instructions are found in the MediaBench program sequences. Note that a considerable amount of operand traffic occurs from integer ALU to branch unit caused by predicate value manipulations; and between integer ALU and memory unit caused by memory reference and spilling.

We can infer from the observations in Figure 3 and Figure 4 that most operands exhibit high degrees of temporal locality and spatial locality. Based on these properties, the complexity of operand transport can be reduced by shortening the transport distance. The key strategies are (i) to hold instruction's result in local storage attached to FUs, (ii) to directly forward them to their consumers without broadcasting or passing through the global storage, and (iii) to allocate the consumer instructions to a FU nearest to the producer. The next section explores and evaluates the architectural techniques that optimize operand transport in detail.

2.3.2 Evaluating Impact of Architectural Techniques on Operand Transport

The operand temporal locality properties imply that the local storage in the execution unit, which buffers the results of the last instructions, reduces the operand traffic. Operand reads can be reduced from the short operand age property and operand writes can be suppressed given the short operand lifetime property. Distributed register files [5][11] or reservation stations in modern processors are the examples of local storage. The bypass network [9], originally introduced to eliminate pipeline data hazards, also helps operand communication by forwarding the results to their targets, bypassing

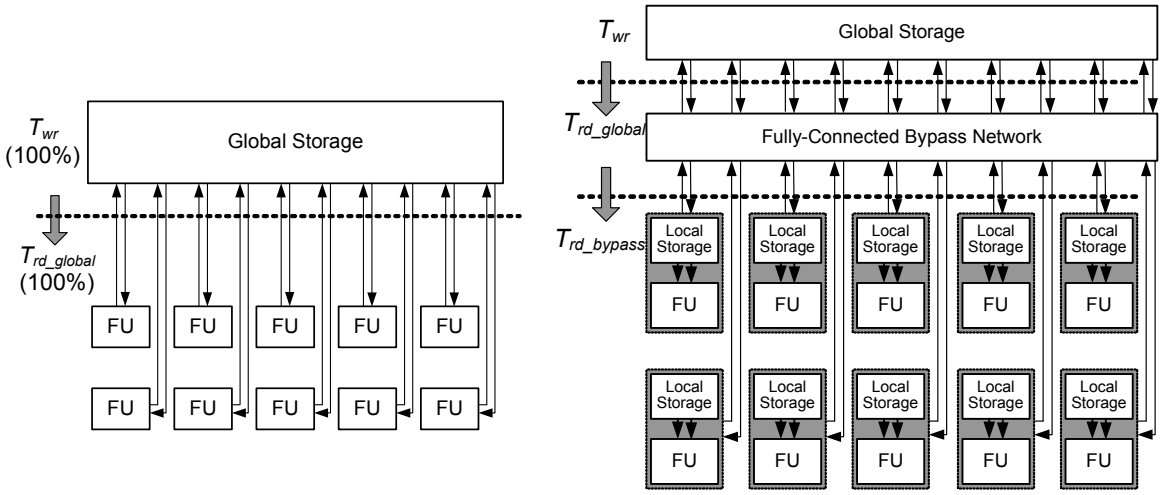
the global storage when multiple FUs are assumed. Though infinite local storage is ideal, it is too expensive and requires long access latency. In this analysis, we attempt to quantitatively evaluate the impact of the size of local storage on the operand transport rates and to determine what kind of additional structure and information are needed.

To measure the impact of each architectural technique, instruction execution models are built. Details of the execution model are listed in Table 2. A two-level storage hierarchy is assumed: a global storage and local storage attached to each FU. The local storage holds input and output operands of the recently executed instructions. Simulations are run with variable sized local buffers. The global storage serves as infinitely sized repository where all operands can reside.

Table 2: Execution model details for operand transport analysis.

Pipeline stage	Description
Front-end	Assume perfect branch prediction
Dispatch	Round-robin FU scheduling for multiple, homogeneous FUs
Read Operand	*if (the operand is in the consumer's local buffer) {* *read the consumer's local buffer;* *} else if (the operand is in the other local buffer) {* *read the other local buffer;* *copy the operand to the consumer's local buffer;* *read the consumer's local buffer;* *} else {* *read the global buffer;* *copy the operand to the consumer's local buffer;* *read the consumer's local buffer;* *}*
Execution	FU Configuration: 1 memory unit, 1 branch unit, 4 integer ALUs, 2 integer multipliers, 2 floating point units
Write back	*if (local buffer is not full) {* *write the result in the producer's local buffer;* *} else {* *write back the oldest entry to the global buffer;* *// replacement algorithm = LRU* *write the result in the producer's local buffer;* *}*

Figure 5 depicts block diagrams of the execution models. As shown in Figure 5(a), we assume a baseline model in which all operands are only communicated through the global storage, for comparison. The operand read and operand write transport rates of the baseline model are set to one as a basis. Recall that the read transport rate is measured on the paths between FUs and the global storage while the write transport rate is calculated only in the global storage. Figure 5(b) shows an execution model equipped with local storage and a fully-connected bypass network that links all resources. In this model, the operand read transport rate is measured at two points: (i) a path between the global storage and the bypass network (T_{rd_global}), and (ii) a path between the bypass network and the local storage (T_{rd_bypass}).



(a) (b)

Figure 5: Execution model block diagrams: (a) baseline model and (b) baseline plus local storage and fully-connected bypass network.

In the execution models, the operands are read from the nearest buffer, i.e., it attempts to access the operand first from its own local storage, then from one of other local storage buffers through the bypass network, and finally from the global storage.

Similarly, an operand is written to producer's local storage buffer by default. If the local storage is full, the oldest entry in the storage is written back to the global storage, incurring an operand write transport. The transportation cost is the highest when the global storage is accessed since it is the farthest location away from the producer and contention for its limited multiple read/write ports may cause long access times.

An important consideration in reducing operand communication caused by operand writes is operand lifetime detection. Even with the local buffers and the bypass network, the write to the global storage cannot be avoided because of the limited size of the local storage buffers. After the local buffers are filled up, an instruction that produces a value always has to spill out a value from the local to the global storage. However, most of the operand writes are useless since they are unnecessarily written after the operand's lifetime is expired. Thus, if the lifetime of each operand is known, the write transport rate can be reduced significantly.

Figure 6(a) and Figure 6(b) present the operand read and write transport rates for the execution model shown in Figure 5(b). Figure 6(c) depicts a bypassed transport rate that is calculated by the operand read from the bypass network over the total transported read. The data represent the average over the MediaBench application programs. Total read transport rate, i.e., the summation of T_{rd_global} and T_{rd_bypass}, is depicted in Figure 6(a). The x-axis denotes the number of local buffer entries assigned to each FU and y-axis denotes the percentage of the transported operands. Note the offset of the y-axis in Figure 6(a).

As shown in Figure 6(a), the read transport rate decreases as the size of the local storage increases since the required operands are more likely to be found in the local

storage. For example, the average read transport rate with an eight-entry local buffer is about 80% of the baseline. However, the read transport rates are saturated even if the size of local storage goes to infinite – 64.5% of the baseline. Figure 6(c) illustrates that with a small amount of local storage, a significant number of operands are read from global storage because of the limited local buffer size. However, as the size of local storage increases, a large number of operands are accessed through the bypass network instead of from global storage. For example, with a 64-entry local buffer, less than 5% of operands are read from global storage. The read transport rates are dominated by the bypassed transport when a sufficient number of local storage buffers is provided.

Though the local storage and the bypass network can reduce the operand read traffic, the traffic caused by the operand writes remains high even though the size of the local storage increases. Even with a 256-entry local buffer, over 99% of operands are written back to the global storage. This indicates that local storage alone cannot reduce the write traffic. If the lifetime of each operand is known at compile-time, the write transport rate can be reduced drastically as shown in Figure 6(b). The compiler marks the last instance of each operand. Operand write is suppressed if the operand is kept only in the local buffer – not written back to the global storage – after the marked instruction is issued. In this case, write demands are decreased suddenly with small local buffers. For example, with a 256-entry local buffer, less than 1% of operands need to be written. This corresponds with the short lifetime property discussed in Section 2.3.1. The results in Figure 6(b) demonstrate that lifetime information is critical for reducing the write transport rate.

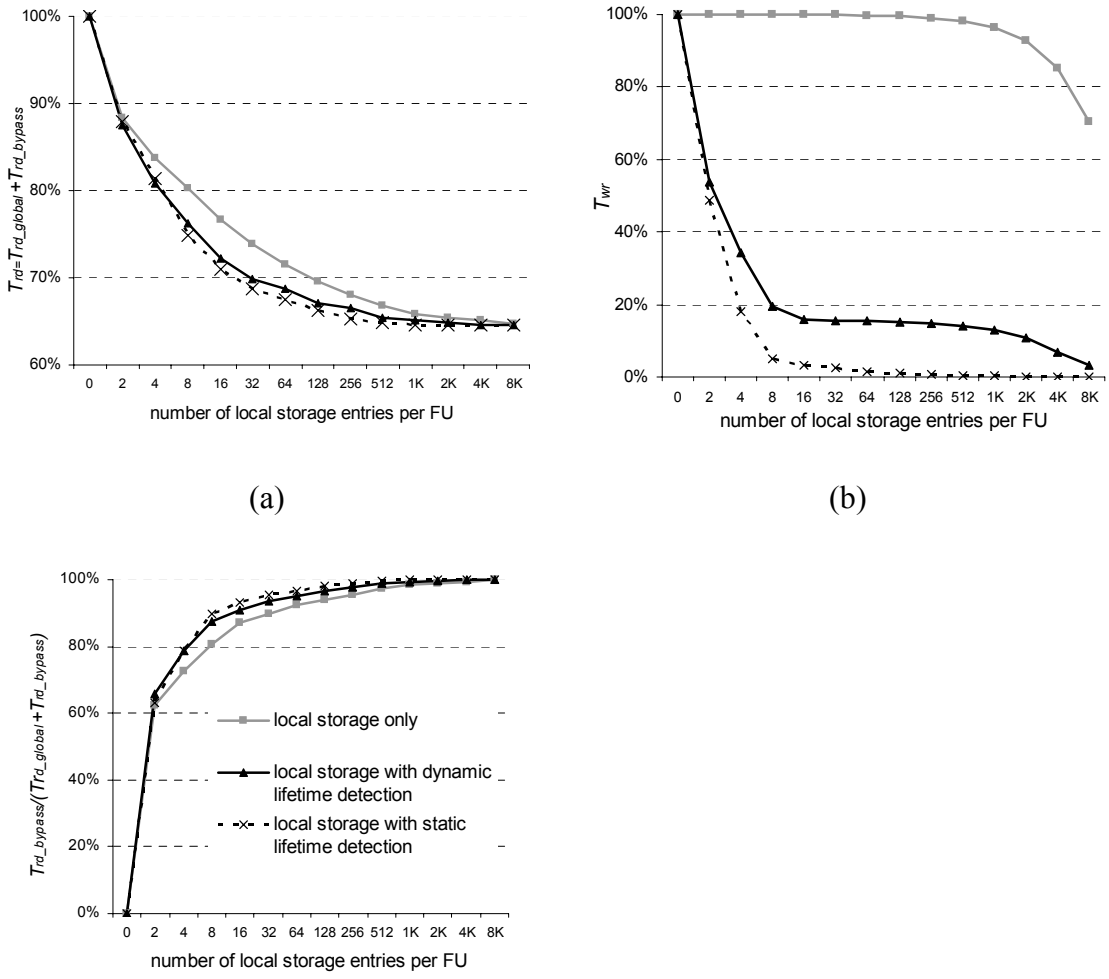


Figure 6: Impact of the local storage, bypass network, and lifetime detection on operand transport rates: (a) operand read transport rate, (b) operand write transport rate, and (c) bypassed read transport rate.

If compile-time lifetime detection is not available, the lifetime can be estimated at run-time. One way to approximate the operand lifetime at run-time is to detect a new instance for registers and memory locations. It is easy to find a new register instance since an instruction refers to registers by their names. Register renaming techniques in modern dynamically scheduled processors, originally developed to eliminate write-after-write hazards, can be used for run-time lifetime detection of register-based operands.

However, it is hard to detect a new memory instance since memory addresses are computed at run-time and there are too many locations for memory operands.

An operand write is suppressed if an instruction that creates a new instance for the same register is issued before the operand is written-back to global storage. Even though this technique has its limitations – it can be only applied to the *register* operand and the new instance could occur far after the operand's real lifetime - it works well, as shown in Figure 6(b). For example, with only eight-entry buffers per FU, about 80% of operand writes can be removed. The gap between the ideal compile-time lifetime detection and the run-time register lifetime detection arises mainly from the memory operands. Fortunately, our preliminary analysis results in Section 2.3.1 show that the lifetimes of most memory operands are extremely long and detecting their lifetimes does not significantly reduce the total write transport rate. The lifetime detection techniques also slightly reduce the read transport rates, since more free entries are available after local buffer entries are released on lifetime expiration.

The results in Figure 6(a) indicate that the read transport cannot be reduced only by attaching local storage and by linking them through the bypass network. Even with an infinite amount of local storage, we can attain at most 35.5% reduction in operand reads. Most read transports are caused by data movement between FUs. This type of transport can be reduced by exploiting the common operand transport patterns and by directly forwarding the values from the producer to the consumer based on the extracted patterns.

Our preliminary results on the transport patterns shown in Figure 4(b) suggest that the direct forwarding paths between integer ALUs; an integer ALU and a memory unit; an integer ALU and a branch unit; and floating-point units would be effective. Figure 7

depicts a new execution model chaining pairs of FUs together based on the common transport patterns. In this model, each FU writes the result to the local buffer of the chained FUs instead of its own local buffer. It is assumed that the consumer of each operand is known at compile-time and instructions are issued to the designated FUs based on the extracted consumer information.

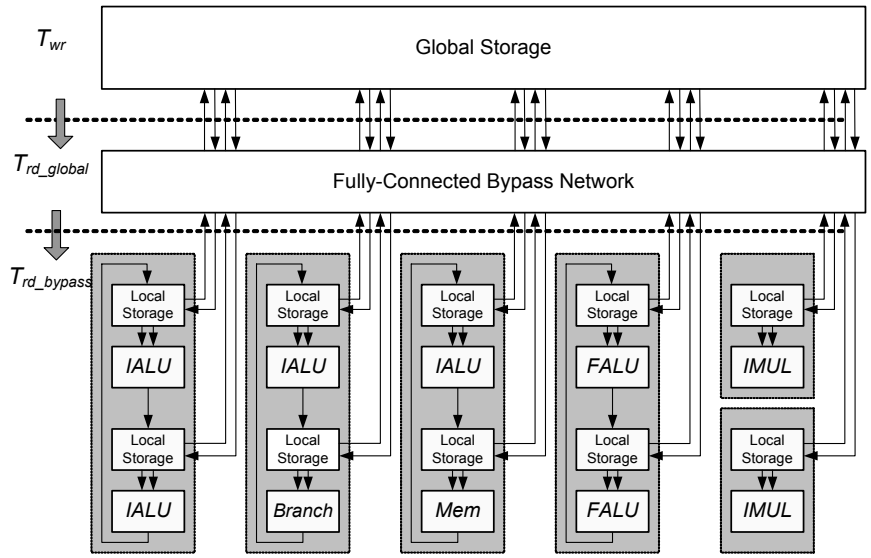
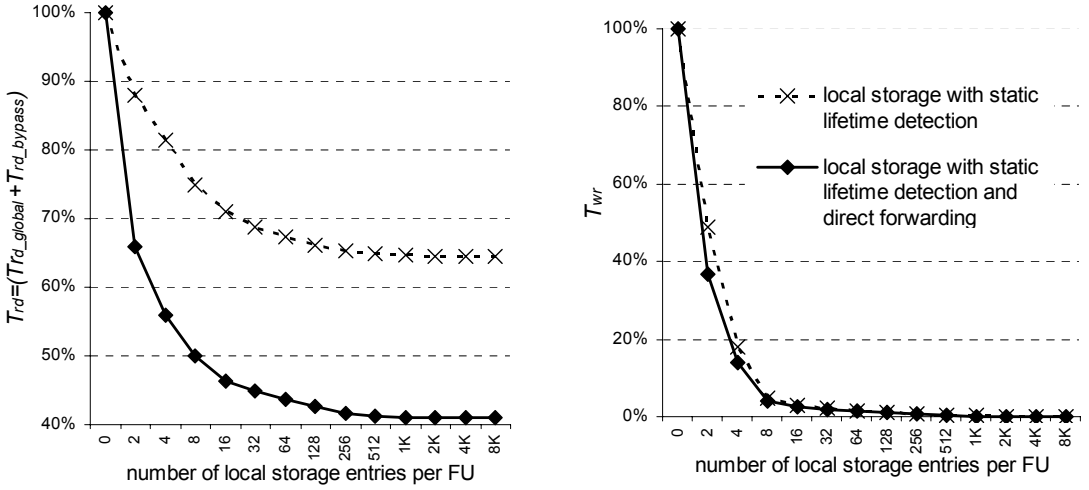


Figure 7: New execution model equipped with selective direct data forwarding paths.

Figure 8 presents the effect of the chained FUs connected by direct operand forwarding paths on the operand read and write transport. Compared to the model in Figure 5(b), the new model replaces a significant amount of bypass network traffic with accesses of the nearest local buffer, as shown in Figure 8(a). For example, with an eight-entry local storage, direct forwarding paths converts 28.5% of the read transport to the nearest local buffer accesses. As expected, the execution model in Figure 7 shows a slightly less write transport rate than the model without dedicated paths when the number

of local storage is small. This is because the results are directly moved to the target instead of residing in the producer's local buffer.



(a) (b)

Figure 8: Impact of the direct operand forwarding on operand transport rates: (a) operand read transport rate and (b) operand write transport rate.

2.4 Conclusion

Recognizing and understanding the nature of operand communication is valuable in the development of alternate low cost, low latency operand transport mechanisms that efficiently control the operand traffic. This research analyzes the operand usage and transport characteristics during execution of multimedia application programs, focusing on the operand temporal and spatial localities.

Our empirical analysis shows that most operands exhibit a high degree of locality; 95% of operands are used at most three times, 83% of operands are consumed within five dynamic instructions, 76% of operands are dead within a ten dynamic instruction window, and 83% of operands are used by only one FU type.

Architectural techniques that exploit these locality properties are implemented and their effectiveness in reducing operand transport is evaluated. Our results show that (i) 25% of operands are read through the nearest local path with eight-entry local storage and a fully-connected bypass network; (ii) 81% of operand writes to global storage are eliminated by applying dynamic register operand lifetime detection; and (iii) 50% of operands are read directly from local storage by adding dedicated bypass paths between heavily trafficked resources and by applying a novel instruction mapping scheme based on operand consumer information. These results are used to devise novel communication mechanisms in the following chapters.

CHAPTER 3

TRAFFIC-DRIVEN OPERAND BYPASS NETWORK

3.1 Introduction

As computation-intensive multimedia workloads have become increasingly important in general-purpose computing, modern processors have evolved to deliver needed performance. The continuing trend in processor architecture design is to integrate more parallel computing resources and to increase clock frequency with the goal of achieving higher throughput. This trend is fueled by the ever increasing availability of fast transistor devices enabled by the growth in semiconductor technologies. However, as semiconductor feature size decreases, interconnect has become the limiting resource in processor implementations [42] and interconnect delay dominates processor cycle time. The operand bypass networks of ILP processors are a critical example. They employ poorly scaling broadcast buses to distribute operands and are particularly demanding of wiring resources. Forwarding path delays are increasing relative to execution unit delays, resulting in a negative impact on performance by reducing the clock speed or by introducing extra latencies [60].

This chapter explores a lower cost, more efficient operand bypass network than traditional bypass networks. It exploits common operand communication patterns in multimedia applications to reduce the latency, storage requirements, and interconnect demand of operand transport.

Our exploration is strongly tied to the anticipated VLSI implementations, necessitating an approach based on accurate technology modeling. Toward this, we

define a model of data transport and buffering based on technology models and we provide a workflow for predicting the transport cost, based on the operand movement and storage demands. Our approach combines technology modeling techniques for architectural evaluation with cycle accurate simulation to explore a range of microarchitectural configurations and to quantitatively predict their performance. The communication patterns in the execution of the application programs are also analyzed. Combining the technology-based modeling methodology and the operand characteristics from the workload analysis, we present and evaluate an improved operand bypass network. This is targeted specifically for multimedia applications, which have tremendous operand communication demands associated with processing high volume data streams.

Our approach consists of three phases. It places microarchitectural components to exploit the transport communication patterns between them (*traffic-based FU placement*), configures each bypass path based on the traffic rate under a given wiring budget (*selective point-to-point bypassing*), and maps inter-instruction communication on the local paths (*geometry-aware instruction steering*) [35]. Our technique improves the instruction throughput performance by increasing the clock rate and by reducing global communication. It also reduces the demand for interconnect resources. This technique produces a 26% instruction throughput gain over a conventional clustered architecture at 45nm technology while reducing 50% of the total bypass wire length.

The rest of this chapter is organized as follows. Section 3.2 overviews prior work. Section 3.3 provides a research methodology for predicting the transport cost using technology and architecture models. It also describes the approach to implement an

improved bypass network in detail. The empirical analysis of forwarded operand traffic and transport patterns is presented in Section 3.4. Details of the experimental setup and results are given in Section 3.5. Section 3.6 summarizes conclusions.

3.2 Related Research

An operand transport network is defined as a set of mechanisms that link the operands and operations to enact the computation specified by a program. These mechanisms include physical interconnection as well as an operation-operand matching system that coordinates values to a coherent computation [69]. An execution unit and storage form the simplest transport network, e.g., an arithmetic logic unit (ALU) and a register file. Though it is simple and straightforward (values are only communicated through a register file), it cannot avoid hazard-induced stalls in pipelined processors.

Bypassing, first introduced in the IBM Stretch [9], is a simple, powerful, and widely used method for eliminating certain data hazards in pipelined processors. With bypassing, additional datapaths and control logics are implemented so that an operation's result is available for subsequent operations before it is written to an architectural register. The number of bypass paths in a scalar processor increases linearly with the number of cycles between execution and the last stage of register file write-back.

The introduction of multiple ALUs creates an additional demand on the bypass network. As pointed out in [2], if IW is the issue width (or the number of ALUs), and if there are S pipeline stages after the first result producing stage, a fully-bypassed design would require ($2 \times IW^2 \times S$) bypass paths assuming two-input ALUs. The number of bypass paths grows quadratically with the issue width.

According to the Sankaralingam's model [57], a fully-connected bypass network is classified as a *broadcast* (the output of an ALU is sent to all ALUs), *single-hop* (an operand is sent directly from the output of an ALU to the input of another ALU) network. A broadcast, single-hop bypass network allows any ALU to read its inputs from any of the subsequent pipeline stages. However, as architectures get wider, the complexity of the bypass network, such as the number of bypass paths and the distance of the ALU-register execution core, also increases. This demands a significant amount of wiring resources and area. In addition, logic paths including bypassed data put pressure on the cycle time around an ALU because of the wire delay, multi-driver buses, and wide input multiplexers. For instance, the Alpha 21064 has 45 separate bypass paths [41]. The Itanium processor spends half of the execution cycle on ALU computation and half on bypassing [25]. This section summarizes architectural techniques developed to reduce the complexity of the operand bypass networks.

3.2.1 Variations of Operand Bypass Networks

Several techniques have been proposed to reduce the forwarding delay of the operand bypass networks. Many researchers have studied incomplete bypass networks that remove selective bypass paths. Ahuja *et al.* [2] show that certain bypass paths are rarely used and these buses can be removed without a great performance loss. They have attempted to exploit the bypass patterns in in-order pipelines of a scalar processor. Brown and Patt [10] have studied the effect of limited bypasses on pipelined functional units and multi-cycle register files. Their results demonstrate that one level of bypass paths in a multi-level bypass network can be removed with little loss in IPC (less than 3% compared to a fully-connected bypass network).

Other studies have focused on the bypass networks of VLIW processors. Cohn *et al.* [16] have studied a partial bypass configuration of the iWarp VLIW processor, concluding that the partial bypass helps reduce the cost with negligible reduction in performance. On the VIPER VLIW microprocessor [28], each FU has bypass paths to only itself and its closest FU to reduce the bypass network complexity. Fan *et al.* [23] have explored the utilizations of each bypass path. They have synthesized an application-specific VLIW processor that has a customized incomplete bypass network. However, these approaches require extensive compiler support for instruction scheduling and FU assignment.

An alternative technique is to add prioritized bypass paths between highly trafficked datapaths, based on the communication patterns of the application. Buss has proposed a pipelined clustered VLIW architecture with additional bypass interconnections between two datapaths in distinct clusters to reduce the number of copy operations using global copy buses [12]. Sassone and Wills [59] have studied transient operands, which are produced values that have only one consumer, and efficiently executes small groups instructions that are linked by transient operands (called *strands*). In this scheme, the execution targets are the normal ALUs with a self-bypassing mode using a closed-loop bypass. Dynamically detected strands are steered to the target, resulting in fast forwarding. These approaches expend additional wiring overhead for the prioritized paths.

Our approach is in some ways similar to the incomplete bypassing approach in that it attempts to reduce the interconnect burden based on common operand transport patterns. However, there are important differences. While the incomplete network for clustered VLIW architectures uses a compiler to pre-schedule instructions, our approach

applies dynamic scheduling to maintain binary compatibility. More importantly, we build a bypass network by placing communicating resources as near as possible and by reorganizing each of the bypass paths based on the traffic rate to efficiently utilize interconnect resources. On the contrary, others remove entire less trafficked paths maintaining the broadcasting nature of bypass wires.

3.2.2 Resource Partitioning: Clustered Architecture

Clustering, partitioning some of the critical components into simpler structures, is becoming widely recognized as an effective method for overcoming scaling and complexity problems. This technique is implemented commercially on the Alpha 21264 processor, which has two identical pipelines with distinct register files, bypass networks, and issue logic [33]. In the clustered microarchitecture, each cluster is formed by a set of FUs, a register file, and an intra-cluster data transfer network. The clusters are connected by an inter-cluster data transfer network. Intra-cluster signals are still propagated through fast and efficient interconnects while inter-cluster communication uses global wires that are long and slow. Therefore, a key issue for reducing operand transport complexity in the clustered mechanisms is to assign operations to the clusters, called *instruction steering*, to minimize the inter-cluster communication.

The assignment can be carried out statically by the compiler or assembly programmer (*static clustered architecture*) or it can be accomplished dynamically during run-time (*dynamic clustered architecture*). There have been many academic endeavors to propose and evaluate the static clustered architectures [24][63][68]. Sohi proposes the *Multiscalar* architecture [63] in which each cluster independently fetches the instructions assigned to it. The instruction distribution is based on information in the binary. The

Multicluster [24] architecture is similar to the Multiscalar architecture, except that the instruction distribution is based on the architectural registers named by each instruction and it shares a common instruction fetch stream. In general, these static approaches share a common need for good static scheduling by a smart compiler.

Dynamic clustering approaches initially experimented with a decoupled or heterogeneous cluster implementation. Many commercial processors, such as MIPS R10000 [29], SUN UltraSparc [30], and AMD K5 [65], have been developed using the decoupling structure. They comprise a common fetch unit and two subsystems: one set of units and registers for addressing and integer computation and the other set for floating-point computation. The drawback of the heterogeneous clustering is that when an integer program or integer-intensive portion of a floating-point program is executing, the floating-point resources are idle. Many dynamic instruction steering mechanisms have been investigated and attempts have been made to optimize the trade-off between inter-cluster communication penalty and workload balance [13][51].

Over the past few years, more general approaches that distribute the same resources to each cluster have been proposed. These uniform cluster configurations can eliminate the instruction steering restriction resulting from the structural hazards. In recent literature, the prevailing philosophy is to assign instructions to a cluster based on data dependence and workload balance [7][11][32][38][50]. The precise methodology varies according to the underlying architecture and execution cluster characteristics.

Instruction-level distributed processing (ILDP) [38] defines a new accumulator-based instruction set exposing dependences and local value communication patterns to the microarchitecture. It uses this information to steer chains of dependent instructions to

the same processing elements. Though new compilers or new ISAs can recognize the dependences and allocate a dependence chain to the same execution cluster, binary compatibility must be dealt with through virtual machine software or on-the-fly hardware translation.

Palacharla's dependence-based clustered architecture [50] and Kemp's parallel execution windows (PEWs) [32] replace the centralized issue window with smaller, distributed windows. The key idea is to exploit the natural dependences among instructions since dependent instructions cannot be executed in parallel. Depending on the availability of an instruction's operands, the instruction is steered to a new first-in first-out buffer (FIFO) when all the required operands are already residing in the resister file or in the FIFO where the source instruction(s) is residing. By allocating dependent instructions to the same windows dynamically during the dispatch (or issue) stage, communication localities are exploited, thereby minimizing global communication.

Bunchua and Wills [11] have proposed a fully distributed register file where broadcast transport is replaced by an explicit on-demand local bypass at the cost of longer inter-ALU latency. In general, the dispatch (or issue) time instruction assignment does not scale well since dependence analysis is an inherently serial process. To eliminate critical latency from the front end of the pipeline, the clustered trace cache processor (CTCP) [7] assigns instructions at the commit (retire) stage by physically reordering instructions within a trace cache line so that they are issued directly to the desired clusters. Table 3 summarizes important qualitative and quantitative characteristics of the dynamic clustered architectures described in this section. Note that Alpha 21264 decouples integer and floating units, and integer units are further separated into two uniform clusters.

Though clustering is an effective technique for reducing the impact of wire delays and the complexity of microarchitecture, it runs into the inter-cluster communication latency and wiring resource overhead problems as semiconductor feature sizes decrease. Our research approach achieves a similar effect as clustering, but reduces the wiring overhead while sustaining the performance at deep sub-micron technologies.

Table 3: Comparison of dynamic clustered architectures.

Architecture	Cluster Configuration	Number of Clusters	Instruction Allocation Stage	Inter-cluster Communication
MIPS R10000	Heterogeneous	2	Dispatch	Register move instruction
Sun UltraSparc	Heterogeneous	2	Dispatch	Register move instruction
Alpha 21264	Heterogeneous*	3	Execution	Dedicated path to register file
CTCP	Uniform	4	Retire	Point-to-point multi hop
Palacharla's model	Uniform	2	Dispatch/Execution	Bus-based
ILDP	Uniform	8	Dispatch	Bus-based
PEWs	Uniform	8	Dispatch	Point-to-point multi hop

* Two uniform integer cluster and a floating-point cluster

3.3 Methodology

This section describes a methodology for predicting the transport cost using technology and architectural models. In addition, architectural techniques are presented, which exploit the operand distribution patterns to reduce the transport latency, storage requirements, and interconnect demands of the operand bypass networks.

3.3.1 Technology Modeling and Transport Cost Prediction

The cost and performance of a processing system is a product of architecture and implementation technology. While the Semiconductor Industry Association's

International Technology Roadmap for Semiconductors [62] provides detailed expectations for future CMOS technology, feature-based scaling of an existing design is often inaccurate as detailed constraints within the technology are taken into account. The Generic System Simulator (GENESYS) is an analytical modeling tool developed by the gigascale integration group at Georgia Tech [22] (Figure 9). GENESYS integrates a hierarchical set of models that captures key limits (fundamental, material, device, circuit, and system), introduced in [43]. It accepts early design parameters from an architectural block and combines model results from across this hierarchy to predict parameters, such as area, cycle time, wire delay, dynamic energy, and static power for a specified technology.

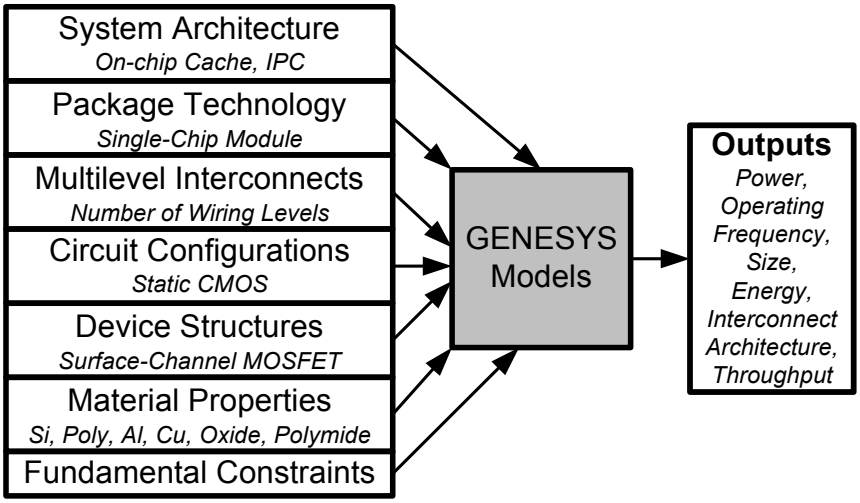


Figure 9: GENESYS system hierarchy.

GENESYS is less accurate than circuit simulators, such as HSPICE, where design variations that affect performance and efficiency are captured (e.g., circuit design style, clocking strategies, and layout techniques). However, GENESYS requires a far more flexible analysis tool, requiring less developed design specifications. In this research, the

primary outputs are module area, gate delay, and interconnect delay predictions based on architectural configurations. To access the accuracy of these predictions, GENESYS has been used to predict similar qualities of commercial microprocessors for which actual implementation details are known [15].

Figure 10 shows the workflow for system analysis, combining application simulation and technology modeling to predict interconnect and buffering demand. Architectural parameters from the architectural configuration file are combined with FU and storage models in the configuration builder to generate a hierarchical input file for GENESYS. A FU is defined by a gate count, gate depth, Rent's parameters, and bus connections. GENESYS estimates unit speed, area, and transfer latency. It also assembles the units in a user-defined floorplan. The delay builder computes expected delays for operand transport and supplies them to a modified version of Simplescalar.

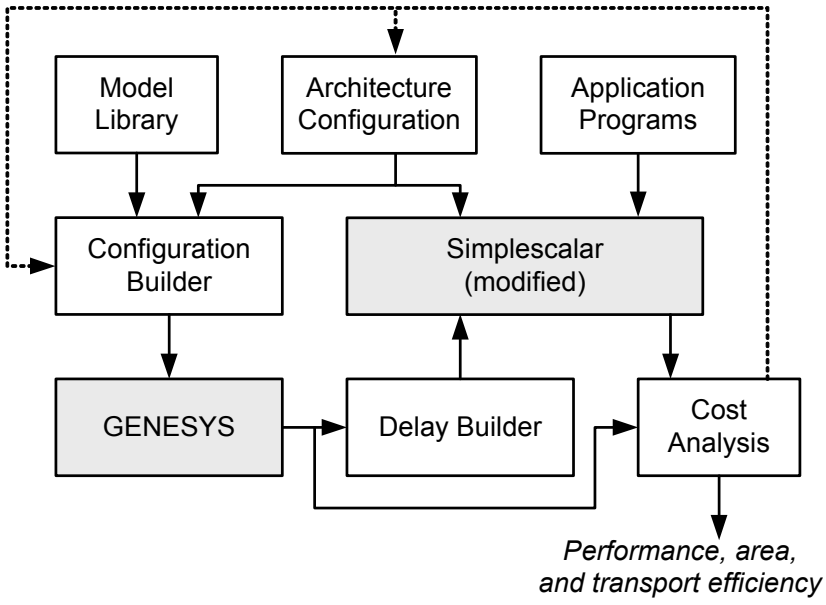


Figure 10: Workflow for system analysis.

The application suite is then simulated and execution statistics are passed to the cost analysis module, where interconnect and storage cost model are generated based on the physical distance that operands must travel and the buffer time required before operands are used. As shown in Figure 11, the overall cost of transport is determined by two parameters: transport distance (D_{OP}) and buffer time (T_{BUF}), defined below.

$$D_{OP} = \sum_{operand} \{\max(P_P, P_{C1}, \ldots, P_{Cn}) - \min(P_P, P_{C1}, \ldots, P_{Cn})\}$$
$$\text{where } P_P \text{ is the physical position of operand producer and}$$
$$P_{Ci} \text{ is the physical position of operand consumers} \quad (2)$$

$$T_{BUF} = \sum_{operand} (T_{issue} - T_{written})$$
$$\text{where } T_{issue} = \text{time when the instruction is issued and}$$
$$T_{written} = \text{time when the operand is written to a reservation station} \quad (3)$$

Using results of GENESYS and Simplescalar, estimates of resources required for operand transport and storage are predicted. Once a model is constructed, parameters of the execution configuration are adjusted to improve the execution performance, cost and/or efficiency. The cost analysis module provides the feedback (shown with dashed line in Figure 10) to the architecture configuration and configuration builder that can enhance execution.

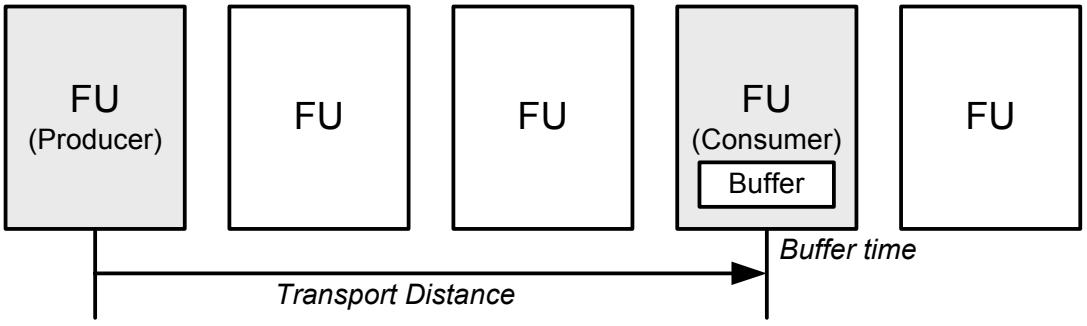


Figure 11: An operand transport model captures both the distance traveled and buffer time required.

3.3.2 Design Customization Process for Operand Bypass Networks

This section describes a design exploration methodology for a low cost, high performance operand bypass network in wide-issue architectures. It is driven by the technology model-based evaluation methodology described in Section 3.3.1. Central to this work is a set of architectural techniques aimed at reducing operand transport cost.

The operand bypass network is implemented along three phases. Initially, conventional broadcasting result buses are assumed to be connected to all FU outputs. First, a microarchitectural placement based on transport pattern distribution between components is applied (*traffic-based FU placement*) [35]. For local bypassing, a few carefully chosen paths between highly trafficked datapaths are replaced with point-to-point bypass paths to reduce the transport latency. The remaining paths are converted to low-cost shared buses to reduce the interconnect cost measured in the total length of bypass wires (*selective point-to-point bypassing*). After the bypass network is implemented, a new FU assignment algorithm is applied, which tries to map inter-instruction dependences on the local bypass path (*geometry-aware instruction steering*). These approaches mainly benefit from shortening the transport distance by exploiting common operand communication patterns. They are discussed in the following sections in more detail.

3.3.2.1 Traffic-based FU placement and Selective Point-to-Point Bypassing

A cycle accurate simulation is performed to measure the amount of traffic between FUs for given application programs. The area of each FU is also estimated using GENESYS, based on a processor model. After the traffic and area information is collected, each FU is assigned in a sub-block which is defined as a collection of FUs

placed together and the position of each FU is determined at the current iteration. The cost of each placement is calculated by following equation:

$$\sum_{(i,j)\in E} \alpha_{ij}(|x_i - x_j| + |y_i - y_j|) \tag{4}$$

In this equation, E denotes a set of directed edges, where (i,j) represents a edge from FU_i (producer) to FU_j (consumer). The parameter a_{ij} is the statistical traffic rate on edge (i,j) (the summation of a_{ij} is equal to one). Finally, (x_i,y_i) denotes the position of the center of FU_i in two-dimensional space. Thus, the cost is defined as the traffic-weighted sum of edges between FUs. All permutations of FUs sequences are explored, from which the minimum cost are determined.

Interconnect wires in the operand bypass networks are classified along two axes: the ownership and the range of distribution. The ownership indicates whether a wire is driven by an *exclusive (dedicated)* source, or whether it is *shared* by multi-sources. The range of distribution indicates whether an operand is to be *broadcast* by default to all possible targets; it is propagated only to the subset of the resources (*multicast*); or it is sent *point-to-point*. According to these classifications, the conventional bypass network of ILP processors is made up of a set of *exclusive-broadcast* wires.

To minimize the forwarding wire delay latency that is taking a dominant fraction of the total transport delay, the length of the bypass path should be kept as short as possible. At the same time, the total length of the bypass wires should be held within a given wiring budget. After the traffic-based FU placement, selective point-to-point bypassing is performed to reorganize each of the bypass paths between FUs based on the traffic rate. It assigns dedicated, point-to-point (p2p) or multicast wires to heavily trafficked paths (the first-level network) while operand transport between low trafficked

resources are accomplished through shared-broadcast wires (the second-level network). The detailed wiring resource assignment process is described in Figure 12.

```
P_OP =operand transport pattern distribution in descending order;
TotalWireLength = 0.0;
N = number of shared-broadcast buses;

// Put the 2nd level shared buses
add (shared-broadcast, N);
TotalWireLength += N*length_of(shared-broadcast bus);

// Put the 1st level p2p or multicast paths
for_all CurPat(FU_i,FU_j,α_ij) ← P_OP
   if (TotalWireLength < TotalWireBudget)
         if (∃ dedicated-broadcast from FU_i ) remove(dedicated-broadcast, FU_i);
         if (∃ dedicated-p2p from FU_i)
            add(dedicated-p2p, FU_i, FU_j);
            for_all (FU_i,FU_k ) ← ∀dedicated-p2p from FU_i
                  remove(dedicated-p2p, FU_i, FU_k);
                  add(dedicated-multicast, FU_i, FU_k);
                  TotalWireLength -= length_of(FU_i, FU_k);
            end for_all
            TotalWireLength += length_of(max(FU_i, FU_k1 ,..,FU_kn)-min(FU_i, FU_k1 ,..,FU_kn));
         else
            add(dedicated-p2p, FU_i, FU_j);
            TotalWireLength += length_of(FU_i, FU_j);
         end if
   end if
end for_all
```

Figure 12: Selective point-to-point path assignment algorithm.

In this process, P_{OP} denotes a set of the operand transport patterns, in which each entry consists of a producer (FU_i), a receiver (FU_j), and the transport rate between them (α_{ij}); and the entries are arranged in descending order. The second-level bypass network is comprised of shared-broadcast buses which can be accessed by all resources. The number of shared-broadcast buses and total wiring budget are adjusted as architectural parameters at the architecture configuration module in Figure 10. After the second-level bypass network is established, the first-level network is added on the edge (FU_i, FU_j)

until the given wiring budget is reached. The edges linked by p2p wires are marked in the path tables to be consulted by the instruction steering logic. By allocating short, dedicated wires only on high trafficked paths, the traffic-based FU placement can be fully exploited and the transport wire delay can be kept short without increasing the total wiring overhead.

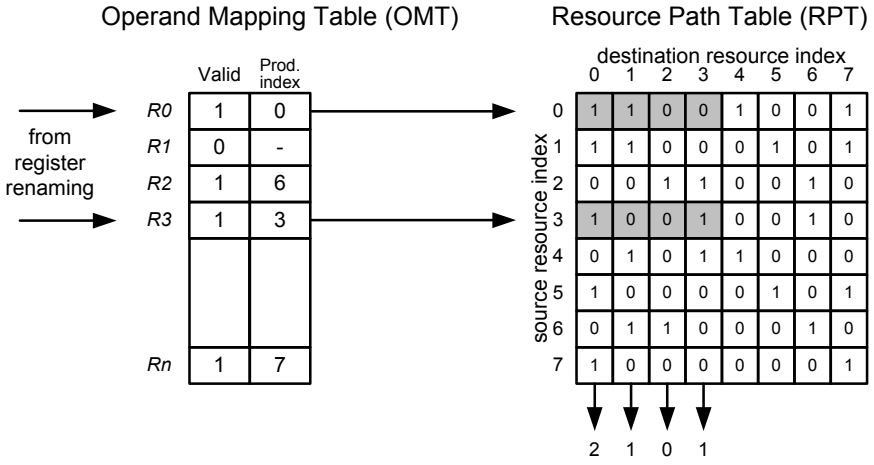
3.3.2.2 Geometry-aware instruction steering

To benefit from the underlying bypass network configuration, the most important decision is to determine the FUs where the instructions are executed. We are interested in the dynamic steering that is performed during the instruction dispatch. Several dynamic steering heuristics have been explored to assign instructions for the clustered architectures [5][13], and the dependence-based method [50] is known to be efficient to reduce the communication-induced stalls. It uses natural dependences between instructions and attempts to assign a given instruction to a cluster that possesses most of the required operands.

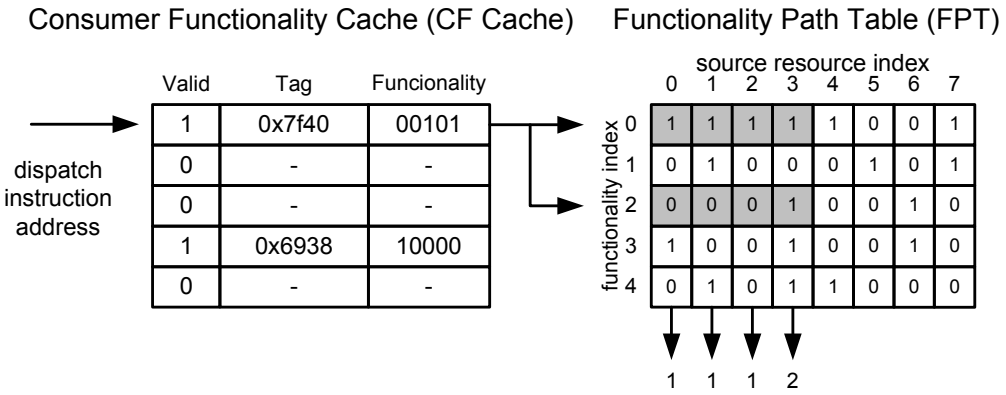
We apply the dependence-based method on a fully decentralized dispatch window in which each FU has its own dispatch queue, but extend it to exploit the underlying transport network configuration, called *geometry-aware instruction steering*. Two types of dependence information, the input and the output dependence, are used in conjunction with resource connectivity information.

The input dependence detection is achieved with a small structure called the *operand mapping table* (OMT) and the *resource path table* (RPT), as shown in Figure 13(a). The OMT is indexed using physical register designators after register renaming. This structure has one entry per physical register, containing the valid bit and the

producer index. The valid bit indicates whether the register operand is accessed from the register file (0) or directly from the producer through the bypass network (1). The RPT is a predefined two-dimensional table where each entry indicates the connectivity of the first-level bypass network between resources. The row and column denotes the indexes of the source and the destination resources of a path, respectively. If the valid bits of the input operands are set, the row of the RPT that is designated by the producer index field is activated. Then, each column adds the activated rows (gray boxes) to calculate the number of available operand which can be transported through the first-level paths.



(a)



(b)

Figure 13: Dependence detection mechanisms: (a) input dependences and (b) output dependences.

Figure 13(a) shows the example entries in the OMT and the RPT. In this example, two source operands are produced or will be produced by the instructions dispatched to FU_0 and FU_3, respectively. FU_0 is selected as a candidate resource since it has local paths both from FU_0 and FU_3. Note that only resources that can execute the current instruction type are considered (from FU_0 to FU_3 in this example).

As shown in Figure 13(b), the output dependence detection mechanism is similar to the input dependence detection in that it refers to a path table and calculates the number of available local paths wired to a given resource. However, there are important differences. Since the output dependences are determined in instruction sequences after the current instruction, they can be checked when the instruction committed. If a sufficiently large instruction window is provided, most output dependences can be observed during the instruction retirement from the operand locality properties [34].

The output dependence detection logic checks the potential consumer's functionalities of the current instruction and stores them in the *consumer functionality cache* (CF cache). During instruction dispatch, the CF cache is accessed by the instruction address. Then, the rows of the *functionality path table* (FPT), in which the corresponding functionality bit in the CF cache is set, are activated. For example, row 0 and row 2 are activated since the bit position 0 and 2 are set as shown in Figure 13(b). It is noted that FPT is indexed by the functionality since the consumer(s) are not assigned to the specific resources yet.

The geometry-aware instruction steering algorithm begins with a check for a free entry of the instruction queue in each resource. If several resources have free entries, the one that has most of the input operands wired through the dedicated paths (from the input

dependence detection) is chosen. If multiple resources have the same maximum number of connections, a resource that has most of the potential consumers possibly transported by the first-level bypass network (from the output dependence detection) is selected. Note that the input dependence has priority over the output one since the former is deterministic while the later may be speculative. The final tie-breaking rule is to select a resource with the lightest load (the resource that has minimum number of the occupied queue) to reduce the potential issue stalls.

3.4 Analysis of Bypass Traffic in ILP Processors

This section explores the operand communication patterns in the execution of standard multimedia application benchmarks. It focuses on the usage of the operand bypass network and the operand traffic between components in dynamically scheduled ILP processors. In this analysis, the same architectural configuration parameters are used as Table 4 in Section 3.5.

Empirical analysis [34] of operand usage and communication properties for MediaBench programs has revealed that operands tend to be used only a small number of times (about 95% of all operands are used at most three times), are usually consumed shortly after they are produced (on average 83% of operands are consumed within five dynamic instructions), and a large number of operands are used by only one consumer type (52% of operands are used only by the same type of FU as producer and 31% of operands are used only by one different type from producer).

The temporal locality properties imply that most operands are transported through the bypass network in the current ILP architectural model when a sufficiently large instruction window is provided. Figure 14 shows the prevalence of bypassed operands

with 128-entry instruction window during the execution of MediaBench application programs. In this graph, the height of each bar indicates the average number of produced operands per instruction, which gives a measure of the total amount of operand traffic. It also presents the distribution of transport media types through which operands are communicated. If a consumer instruction is dispatched into the instruction queue before the required operand is written to the register file, it is passed through the bypass network. If not, it is communicated through the register file.

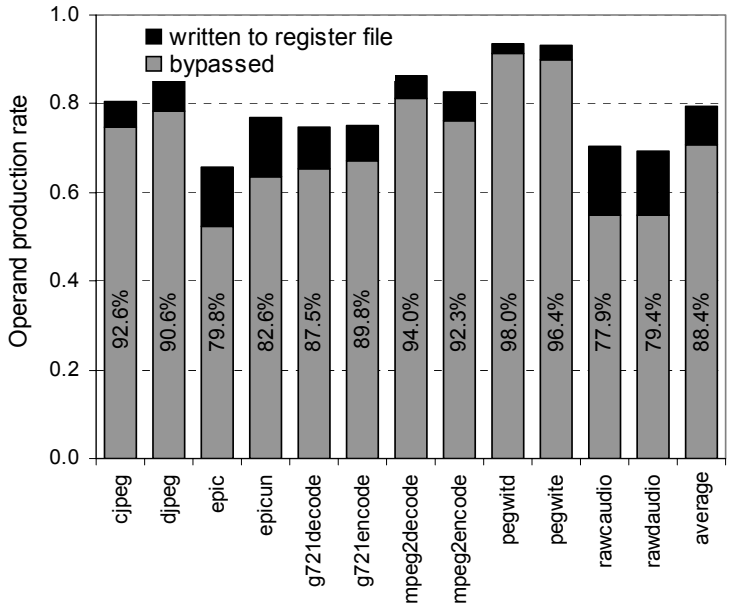


Figure 14: Average number of produced operand per instruction which were broken down by transport media.

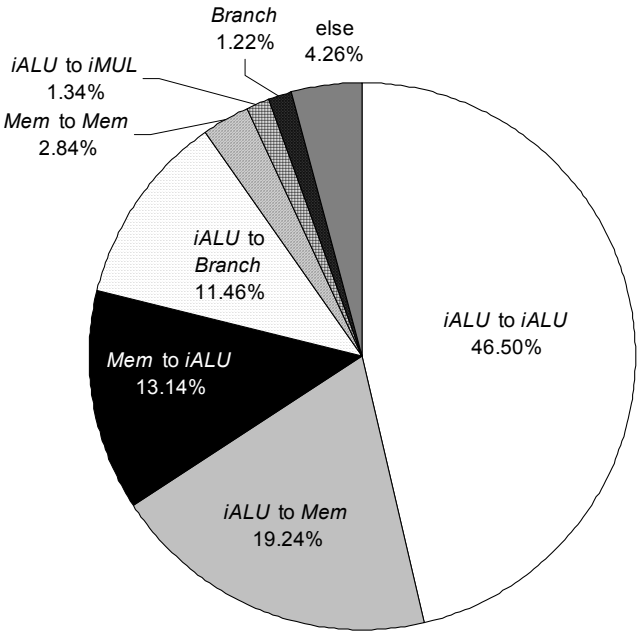
Across all applications, about 80% of dynamic instructions produce operands and 88.4% of the produced operands are transported through the bypass network. Only the remaining 11.6% of operands are sent directly to the register file for future uses. The results imply that the majority of inter-instruction communication needs are resolved by

the bypass network and that its transport performance and efficiency will be a key issue in future processor design.

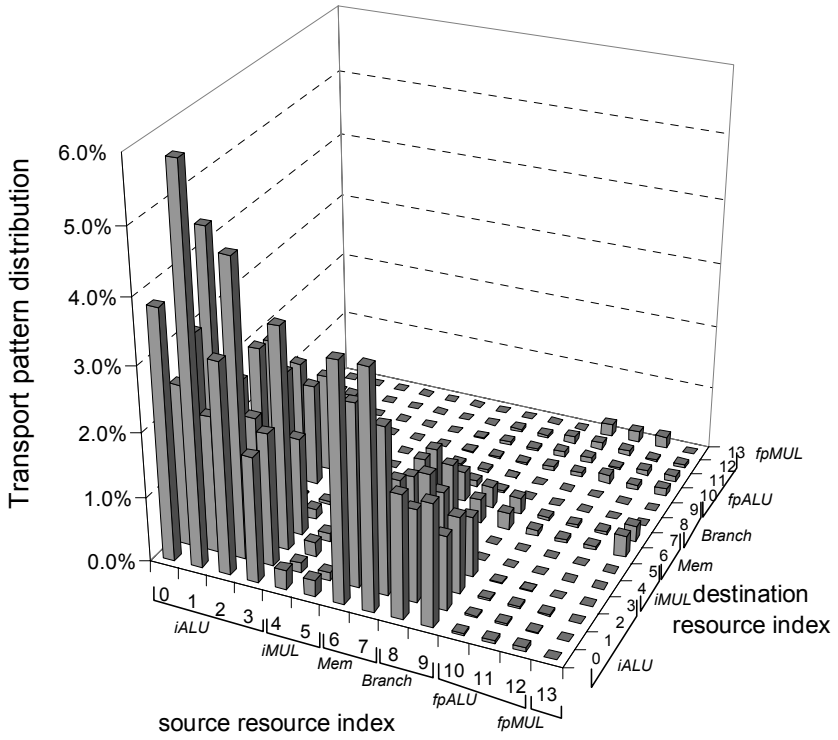
Figure 15 shows the percentage distribution of the operand transport pattern based on the true data dependences between instructions. The transport patterns are classified according to the functionality of producer-consumer pairs (Figure 15(a)) and according to the particular FU pairs (Figure 15(b)), respectively.

The data represent the average across MediaBench application programs. As expected from the operand spatial locality properties, the results in Figure 15 show that the traffic is not evenly distributed; certain paths are more heavily trafficked than others. For example, about 47% of operands are transported between integer ALUs (*iALU*), while almost no operands are passed through some paths, such as paths from floating-point ALUs (*fpALU*) to integer ALUs. Note that the distribution within the same type of FUs depends on the instruction distribution heuristics that determine the resource where an instruction is executed. The distribution patterns between specific FUs can be exploited to reorganize the bypass network as described in the previous section.

Though the operand traffic is not even, all operands are treated alike in current architectural models; they contribute to the same amount of traffic congestion on the fully connected broadcast bypass buses. The results in Figure 14 and Figure 15 highlight the traffic bottleneck through the bypass network and reveal a huge potential for alleviating the communication burden by exploiting the temporal locality and spatial locality characteristics.



(a)



(b)

Figure 15: Percentage distribution of dynamic operand transport patterns: (a) between functional unit types, and (b) between functional units.

3.5 Experimental Results

Benchmarks from MediaBench [39] are simulated using the Simplescalar simulator [4] with the PISA instruction set. The default MediaBench inputs are enlarged to lengthen their execution. For each simulation, 500 million committed instructions are executed. The first 100 million instructions, consisting mainly of common initialization code, are skipped. Table 4 enumerates the parameters common to all architectural models evaluated in this section. It also summarizes the results of the delay and area estimations from GENESYS.

Table 4: Common parameters and GENESYS results.

Architectural configuration parameters			
Fetch/decode/issue/commit width	8		
Total number of FUs	14		
iALU/iMUL/Mem/Branch/fpALU/fpMUL	4/2/2/2/3/1		
Issue queue size (per FU)	4 entries		
Reorder buffer size	128 entries		
Load/store queue size	32 entries		
Branch predictor	Combined bimodal/gshare, 4K-entry BHT, 4-way 2K-entry BTB, 10 cycle branch penalty		
Cache system	64K 2-way IL1, 64K 2-way DL1, 1024 16-way unified L2		
Main memory	Infinite size		
Technology configuration parameters from GENESYS results			
Feature size [nm]	100	65	45
Total execution engine area [mm^2]	1.317	0.5566	0.2663
FU width [um]	339.0	220.3	152.5
Execution gate delay [ns]	0.2953	0.1523	0.1054

To evaluate the effect of technology, three different technology models, i.e. 100, 65, and 45nm, are used by GENESYS. The area of the FUs and the total execution engine are estimated based on an R10000 processor model [73] for each technology level. It is

assumed that the microarchitectural implementation has the FU heights reported in [50]. A two-dimensional layout geometry for the execution engine is also assumed.

The bypass network configurations of simulation models are shown in Table 5. An eight-way ILP processor with a broadcast bypass network is modeled as a baseline (*base*). Two different versions of typical clustered configurations are implemented for comparison: (i) a decoupled implementation which divides FUs into two sub-blocks, one for integer FUs and the other for floating-point FUs (*CL1*) and (ii) a homogeneous clustered microarchitecture which further partitions the integer sub-block into two identical clusters (*CL2*). In the clustered models, the inter-cluster data transport network is assumed to be formed by a set of shared-broadcast buses to reduce the wiring burden. To minimize the inter-cluster communication, the dependence-based instruction steering heuristic is applied to the *CL2* model.

Table 5: Simulation model configurations.

Model name	Bypass path configuration	Instruction steering heuristic	Microarchitectural placing
base	*exclusive-broadcast*	*first-fit*	N/A
CL1	integer and floating-point decoupled *exclusive-multicast* (intra), *shared-broadcast* (inter)	*first-fit*	N/A
CL2	two-clustered *exclusive-multicast* (intra), *shared-broadcast* (inter)	*dependence-based*	N/A
TM0	*exclusive-p2p*	*dependence-based*	*max-place*
TM1	*exclusive-p2p*	*dependence-based*	*min-place*
TM2	*exclusive-p2p/multicast* (selected paths), *shared-broadcast* (other paths)	*dependence-based*	*min-place*
TM3	*exclusive-p2p/multicast* (selected paths), *shared-broadcast* (other paths)	*geometry-aware*	*min-place*

To evaluate the impact of the bypass network optimization techniques on the operand transport cost, fully-connected point-to-point bypass networks are initially

assumed. Two physical FU placements (*max-place* and *min-place*) are studied. All permutations of FU sequences are explored, from which the minimum and the maximum cost placements are determined from Equation (4) in Section 3.3.2.1. They are referred as *TM0* and *TM1*, respectively. By default, the dependence-based instruction steering [5] is assumed. The initial minimum cost model is modified to be equipped with selective point-to-point wires and shared-broadcast buses according to the traffic rate (*TM2*). The detailed reorganization process is described earlier in Section 3.3.2.1. The total wiring budget is set to 0.25, which means the total length of the bypass wires must be within 25% of the baseline. Finally, the geometry-aware instruction steering is applied, replacing the dependence-based steering (*TM3*). Note that multi-level bypass paths are assumed for *CL1*, *CL2*, *TM2*, and *TM3* models: the first-level paths for fast, local transport and the second-level paths for global transport.

The gate delay of the execution unit and the delays of the bypass wires for each simulation model are shown in Figure 16. The delays are measured at three process generations by GENESYS. For multi-level bypass networks, the wire delay of the longest first-level path is presented. Actual cycle times of the models are estimated by adding the gate delay and the longest bypass wire delay since the execute/bypass stage(s) determine the machine clock frequency assuming that the other pipeline stages can be pipelined [60]. The delays of the second-level paths are estimated from the delay of *base* and they are converted to extra cycles according to the cycle time of the simulation models.

As expected, it is observed that the interconnect wire delay does not scale across process generations compared to the significant reduction in the gate delay. This trend demonstrates that the bypassing delay dominates the cycle time when broadcast

bypassing is applied. For example, *base* spends about 68% of the cycle time in bypassing at 45nm while it consumes about 43% at 100nm. The multicast wires implemented in the clustered microarchitectures shorten the length of the bypass wires, resulting in significant reduction of wire delays. The wire delay fraction of the cycle time in *CL1* and *CL2* is reduced to 31.1% and 7.6% at 45nm, respectively. Also expected is that the exclusive-p2p paths achieve shorter wire delays – 25.3% and 2.6% of total cycle time in the fully-connected models (*TM0* and *TM1*) and in the partially-connected models (*TM2* and *TM3*) at 45nm, respectively.

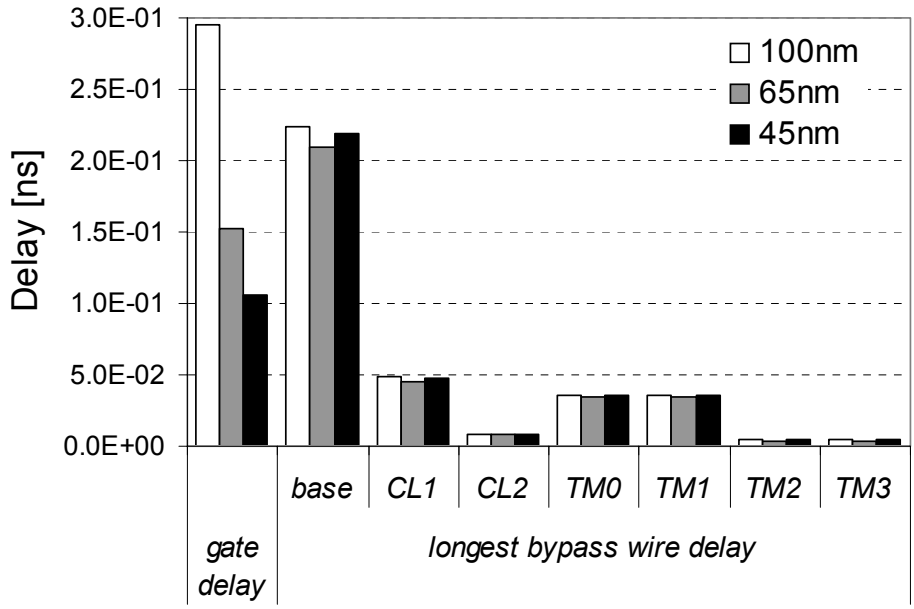


Figure 16: Estimated the execution gate and the longest bypass wire delays for the simulation models.

It is important to note that the partial connections shorten the cycle time with the expense of the extra forwarding latency caused by broadcasting the operands. This occurs when the operands cannot be delivered by the first-level network. For instance, if the operands need to be communicated among different clusters in *CL2*, they must be

delivered through the inter-cluster broadcasting buses. The extra cycles directly translate into additional buffer times and IPC drops. Thus, the amount of traffic on the broadcasting buses is the key issue for the multi-level bypass networks. The fully-connected point-to-point bypass networks can send the operands to all possible targets directly without the extra cycle penalties. However, the additional connections increase the interconnect demand. This wiring overhead may have a negative impact on the cycle time by increasing the fan-out gate delay or by increasing the physical wiring distance caused by routing constraints [57].

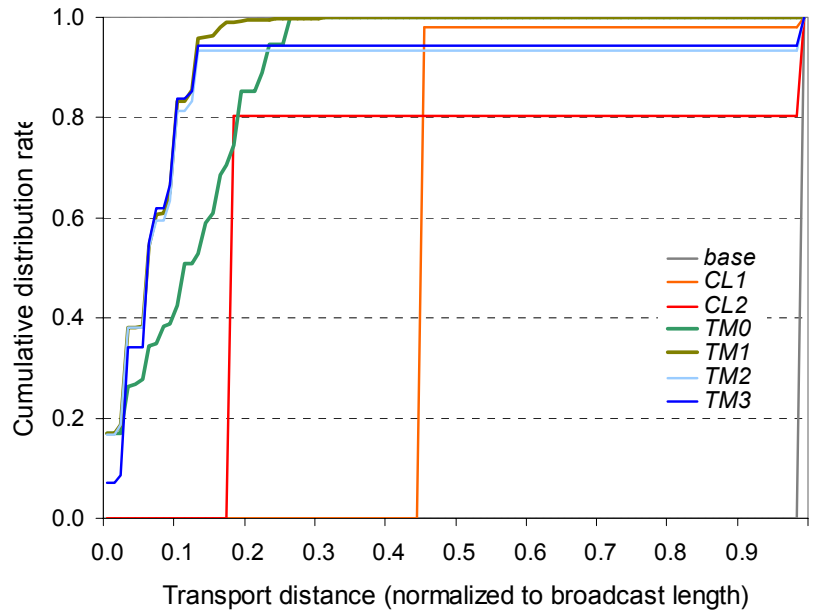
3.5.1 Operand Transport Cost Results

The operand transport costs are measured in terms of the transport distance and buffer time as defined in Section 3.3.1. The transport distance results for the simulation models are shown in Figure 17. The plots in Figure 17(a) represent cumulative distribution of operand transport distance and the height of each bar in Figure 17(b) indicates the average transport distance per operand. The results are averaged across all evaluated MediaBench programs. Note that the x-axis in Figure 17(a) denotes the transport distance normalized to the length of a broadcast bus of *base*. The step shapes in Figure 17(a) indicate that the remaining operands are routed on the second-level broadcast buses. With decreased feature size, all simulation models experience the same level of reduction in the average transport distance as shown in Figure 17(b).

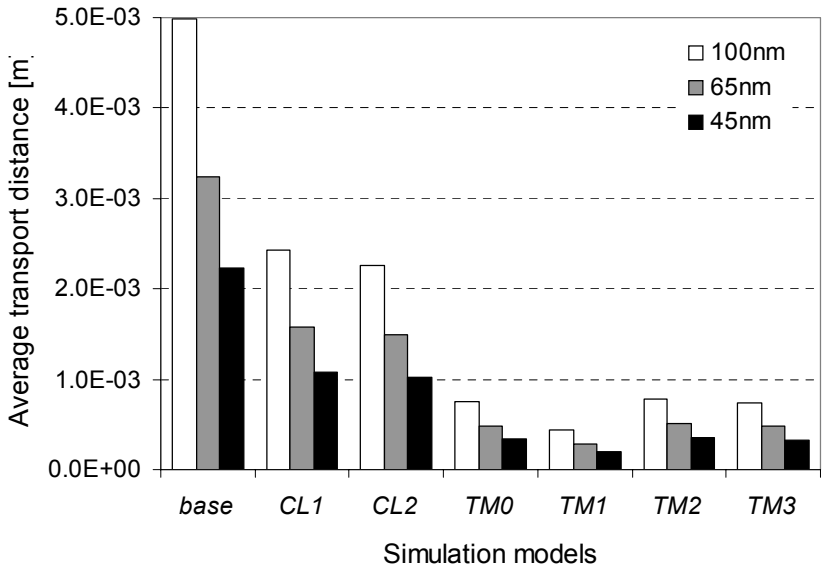
The decoupling (*CL1*) can effectively reduce the distance with only 2% of the second-level transport – the average transport distance is about 49% of *base*. This is mainly because (i) most MediaBench programs do not have floating-point operations - only *epic encode/decode* and *mpeg2 encode/decode* have some floating-point codes, and

(ii) the interaction between integer and floating-point codes is very small even if there exist floating-point operations. The average transport distance is further reduced to 46% of *base* as the resources are divided into the clusters (*CL2*). Interestingly, the impact of the clustering on the distance is offset by the significant amount of inter-cluster communication (about 20% of the total communication), even with the dependence-based cluster assignment.

Ideally, the transport distance can be minimized by the traffic-based FU placement technique and fully-connected dedicated paths (*TM1*). In this model, the shortest paths are always taken to deliver the operands and the average distance reduces to 8.8% of *base*. Across all models, it exhibits the lowest transport distance. Considering partially-connected paths by removing the lightest trafficked paths and by adding shared result buses (*TM2*), the average distance is slightly increased to 15.8% of *base*, incurred by 6.5% of the second-level bypassing. As seen in Figure 17(a), it keeps track of the transport distance of the *TM1* until the second-level bypassing is used. When the geometry-aware instruction steering is applied, the transport distance is slightly decreased to 14.9% of *base*. An interesting observation is that the *TM3* model exhibits a slightly higher cumulative rate that uses the first-level paths though the initial transport distance rate of the *TM3*, i.e. zero distance, is lower than that of the *TM2*. This result demonstrates that the geometry-aware instruction steering is effective in reducing global communication.



(a)



(b)

Figure 17: Operand transport distance of the simulation models: (a) Accumulative distribution of the transport distance and (b) average transport distance.

Table 6 presents the buffer time sensitivity as the feature size decreases. In general, the average buffer time increases as the technology shrinks due to the extra

bypass cycle, except in the *base*, *TM0*, and *TM1* models. In these models, all data forwarding is accomplished during the instruction execution stage (no explicit bypass stages) so the buffer time does not change. As seen in Table 6, the amount of inter-cluster communication through the shared-broadcast buses translates into the buffer time increases since the extra transport latency incurs additional waiting of the operand that is already available in storage. The buffer times continue to grow as the gap between FU gate delay and wire delay increases though the amount of inter-cluster communication remains the same. For instance, the *CL2* takes an average buffer time of 1.27x and 1.52x of *base* at 100nm and 45nm, respectively.

Table 6: Operand buffer time of the simulation models [cycle/operand].

Model name	Technology node		
	100nm	65nm	45nm
base	1.1914	1.1914	1.1914
CL1	1.2537	1.2537	1.3406
CL2	1.5109	1.8149	1.8149
TM0	1.1914	1.1914	1.1914
TM1	1.1914	1.1914	1.1914
TM2	1.4783	1.5964	1.5964
TM3	1.2855	1.4073	1.4073

On the other hand, the *TM3* model can minimize the amount of the global communication by exploiting operand transport characteristics. It suppresses the increases of the buffer time within 7.9% of *base* at 100nm. Furthermore, it can be held to only 18.1% until the feature sizes reach 45nm. The data in Table 6 also show that the *TM3* can achieve shorter buffer time than the conventional clustered implementation (*CL2*) – 15% and 22% reduction in buffer time at 100nm and 45nm, respectively. These

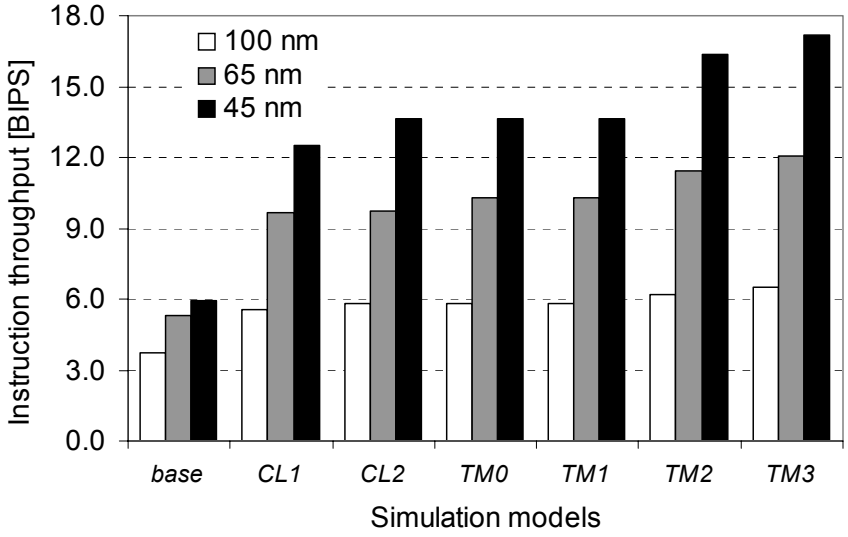
results demonstrate that our approach can be an ideal implementation candidate for highly parallel ILP processing due to efficient operand transport.

3.5.2 Performance and Cost Results

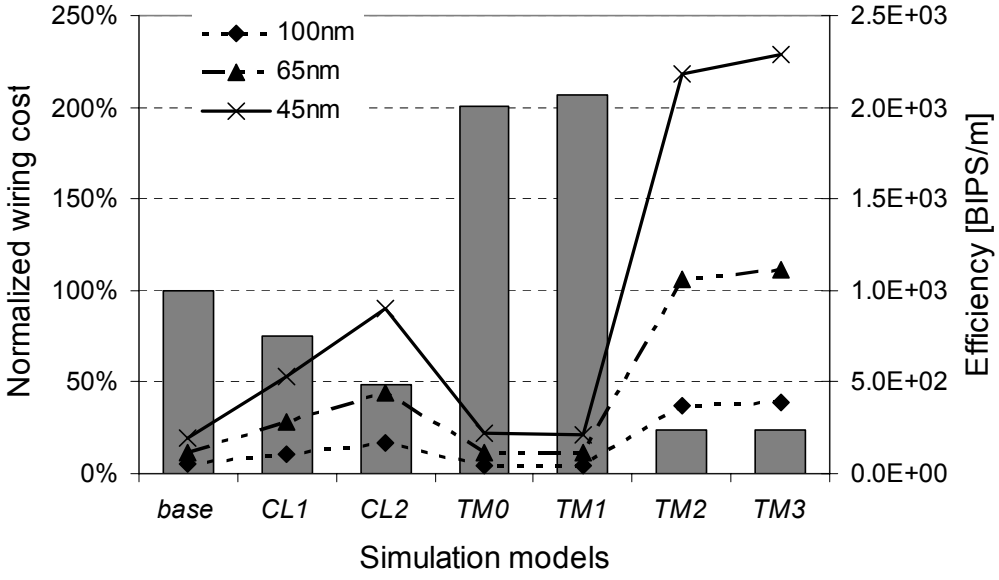
This section presents the impact of the architectural techniques on the execution performance for the benchmarks. In addition, the implementation cost of each bypass network configuration is also presented. Figure 18 shows execution performance and wiring cost for the simulation models. Instruction throughput, measured by dividing IPC by the cycle time, determines the execution rate performance as shown in Figure 18(a). The figure also indicates the sensitivity of the performance to technology migration. Total wire length of the bypass networks normalized to *base* model determines implementation cost as shown in Figure 18(b). It also depicts the wiring efficiency, which is calculated by dividing the simulated instruction throughput by the total wire length.

The results in Figure 18(a) show that total execution throughput increases significantly with technology advances, except in the *base* model. The performance bottleneck of *base* is the wire delay latency of the broadcast buses as previously identified in Figure 16. The traffic-driven transport network without the geometry-aware instruction steering (*TM2*) outperforms *base* by 1.67x, 2.16x, and 2.76x for each technology point, respectively. It also outperforms the typical clustered implementation (*CL2*) by 1.07x, 1.18x, and 1.20x, respectively. When the geometry-aware instruction steering is factored in, the *TM3* model produces the highest instruction throughput. Moreover, the performance gap increases as the technology shrinks. It achieves higher performance than the dependence-based instruction assignment – 5.1% instruction

throughput gain over the *TM2* at 45nm. This result demonstrates the effectiveness of the geometry-aware instruction steering in the performance of the operand transport network.



(a)



(b)

Figure 18: Estimated perforamnce and cost for the simulation models: (a) Instruction throughput and (b) normalized wiring cost and efficiency.

An important consideration when designing an operand transport network is interconnect cost since the primary goal of this study is to reduce the complexity of the operand bypass network. Though actual wiring cost incorporates the physical VLSI design constraints of wire routing which is in turn, a function of several factors, such as routing strategy, this is beyond the scope of this research. Instead, we simply measures the wiring cost in terms of the total wire length of the bypass network that is calculated by adding the lengths of all bypass paths. The results in Figure 18(b) indicate that partitioning broadcast result buses into a set of multicast buses reduces the wiring costs. For example, the *CL1* and *CL2* models consume only 75.2% and 48.5% of the wiring cost of *base*, respectively. Similar trends are observed when partially-connected, dedicated networks (*TM2* and *TM3*) are implemented by removing low traffic paths from fully-connected, dedicated networks (*TM0* and *TM1*). Interestingly, though the *TM0* and *TM1* achieve similar instruction throughput performance as the *CL2*, they demand about four times the interconnect. Therefore, the wiring cost efficiency should be measured to form the basis of comparison.

The lines in Figure 18(b) represent the calculated wiring cost efficiency. Higher efficiency means better utilization of the interconnect resource which is projected to be the limiting resource in future semiconductor fabrication. As shown in the graphs, both dividing the broadcast buses into multiple multicast buses and introducing the incomplete bypass networks increase the cost efficiency considerably. In addition, the technology advances also improves efficiency since the total wire lengths are shorten and the instruction throughput performance increases. By using the traffic-driven bypass network configuration, a 2.41x cost efficiency is achieved over the typical clustering at 45nm.

When the geometry-aware instruction steering is applied in addition, the efficiency gap is increased to 2.54x. These results demonstrate our techniques are efficiently utilizing the given wiring resources.

3.6 Conclusion

Operand transport is becoming a bottleneck in improving the performance of modern processors. Current operand transport design, which mainly utilizes an operand bypass network, demands a high volume of long interconnects. This wire-dominated structure creates a critical wire delay problem in processor design in future technology. To address this problem, the operand bypass networks are explored, especially for multimedia applications.

Technology modeling techniques for architectural evaluation are combined with cycle accurate simulation to measure the operand transport cost in transport distance and buffer time. Based on the technology-based methodology and operand characteristics from workload analysis, we present and evaluate a lower cost, higher performance bypass network than traditional bypass networks.

Our systematic design customization process for the bypass networks is a set of three techniques aimed at reducing the operand transport cost and maintaining performance scalability. It first determines the physical position of the resources based on the transport distribution pattern of the application programs. Then, each of the bypass paths are configured according to the traffic rate. Finally, dependent instructions are assigned to adjacent computing resources connected by the local paths.

Our results show that the overall instruction throughput gain over the conventional broadcast bypass network is 2.9x for a wide range of multimedia

applications in 45nm technology. The total length of wires can be kept within 24% of the broadcast bypass network. In addition, the instruction throughput gain over typical clustered architecture is 1.3x only with 50% of the bypass wire length. Most performance benefits come from increasing the clock speed and reducing the amount of the global transport.

CHAPTER 4

DYNAMIC INSTRUCTION CLUSTERING

4.1 Introduction

Multimedia applications contain abundant data-level (DLP) and instruction-level parallelism (ILP), demanding tremendous computational throughput. In pursuit of higher performance, modern processors that employ dynamic techniques to exploit ILP are integrating an ever greater number of computing resources and larger instruction windows. However, this approach is nearing its limit due to interconnect delay. Inter-instruction communication latency is a critical example [50]. This chapter explores a more efficient operand bypass network along with an instruction scheduling mechanism to reduce the communication latency.

Alternatively, the plentiful DLP inherent in multimedia applications has motivated the development of multimedia extensions on general-purpose processors [21][49][53][54]. While significant performance improvements have been demonstrated, the primary stumbling issue is software compatibility; these extensions rely on software support, such as compiler and retargeting techniques [6][66]. Communication between multimedia FUs and ILP units is also problematic [14]. To address these issues, this chapter presents an execution mechanism that dynamically forms and executes data-parallel operations while maintaining binary compatibility.

Our technique exposes opportunities to lower operand transport latency within a specialized execution unit. First, it converts global communication into local transport. Second, it removes unnecessary communication. It also detects opportunities for data-

parallel execution based on the identification of regular data access patterns. This dynamic execution technique groups data-dependent instructions into clusters during instruction execution, detects the operand lifetime, streamlines intra- and inter-cluster operand transport patterns, and maps the clustered instructions to an efficient cluster execution unit.

Two cluster execution unit implementations are presented and evaluated: network ALUs and a dynamically-scheduled SIMD PE array. In the network ALUs, intermediate values within the inner loops of multimedia applications are propagated among ALUs without distribution through global bypass buses. The reduction in operand transport latency results in a 35% IPC speedup over a conventional ILP processor [36].

The dynamically-scheduled SIMD PE array supports DLP processing of the innermost loops in image processing applications. Data-parallel operation on SIMD PEs can be achieved by predicting stride values between loop iterations. In addition, operand communication is localized from the observed operand characteristics (e.g., the range of distribution). These techniques produce an IPC speedup of 2.59x over a 16-way, four-clustered microarchitecture [37].

The rest of this chapter is organized as follows. Section 4.2 summarizes background information. Section 4.3 introduces an instruction clustering and operand transport pattern recognition mechanisms. Two possible implementations of the cluster execution units along with their operand transport networks are given in Section 4.4 and Section 4.5. Finally, conclusions are drawn in Section 4.6.

4.2 Related Research

This section summarizes previous approaches to enhance the performance of multimedia applications and architectural techniques to reduce operand transport complexity.

4.2.1 Solutions for Reducing Operand Transport Complexity

The architectural community is responding to the operand transport problem with a variety of execution approaches, including new microarchitectures, new ISAs, better compilers, and improved run-time mechanisms.

With the growing concern in wire delay caused by operand communication, many researchers have proposed new architectures focusing on communication-aware execution. The RAW architecture [75] and grid processor architecture (GPA) [46] propose network-connected tiles of distributed processing elements running new ISAs that expose the underlying parallel hardware organization.

The GPA uses a grid of ALUs to remove the global transport path, though ALU assignment is done statically by a two-dimensional VLIW compiler. It maps blocks of statically-scheduled instructions to the ALU array and executes them dynamically in dataflow order. The strategy is to localize inter-instruction communication by forwarding temporary values generated inside a code block directly from the producers to their consumers. The key advantage of GPA is that instructions can be executed without broadcasting results. Communication can take place along short, point-to-point wires.

The general philosophy of the RAW processor is to build an architecture by replicating a simple tile, each with its own instruction stream. Each tile contains a simple RISC-like pipeline and is connected with other tiles over a pipelined, point-to-point

network. The RAW processor also integrates a statically scheduled router, which eliminates the need for dynamic arbitration for the shared router and wire resources. Unlike a current superscalar processor, it does not bind register-renaming or dynamic instruction issue logic into hardware. The RAW processor simplifies instruction scheduling hardware and exposes it to the compiler with a new ISA. While RAW implements a static transport and the GPA uses a dynamic transport, both perform compile-time optimizations for instruction scheduling and localize the communication through direct dedicated forwarding paths.

Corporaal and Arnold [19] propose a novel *transport-triggered* architecture, called MOVE, which is programmed by explicitly specifying data transport. It directly forwards operands between FUs and reduces the latency of the bypass network by eliminating the associative hardware. All bypassing is done in software under compiler control. Pattern detection techniques have been developed [3] to synthesize new ISAs but they are software-based. In general, these static approaches require extensive compiler support.

An alternative approach is to reduce operand traffic dynamically. Hardware-based instruction optimization is a recent research area, moving some of the compiler's burden on-chip. The idea of reforming instructions in hardware and caching them has been introduced in [44] using a fill unit to achieve high bandwidth instruction delivery. Some methods use the fill unit to dynamically retarget a scalar instruction stream into pre-scheduled instruction groups [8][47][74]. The idea is to do as much work as possible on a small number of trace instructions, mainly focusing on alleviating the burden of instruction scheduling. The fill unit approach is extended by Friendly [27] to arrange

instructions within the trace segment (hyperblock) to minimize the impact of the operand transport latencies in a clustered microarchitecture. RePLay [52] forms hyperblock regions (called frames) in a similar fashion, but guarantees atomicity in its frames. Though it improves the performance through aggressive intra- and inter-block optimization, such as dead code elimination, common sub-expression removal, and reassociation, a huge, long latency (the authors assume between 100 and 10,000 cycles) embedded hardware optimization engine is required.

While previous dynamic mechanisms target general applications, such as SPEC benchmarks, the empirical analysis results [34] in our research show that similar benefits can be achieved with simpler techniques, which exploit specific characteristics of multimedia applications. The inner loops of multimedia applications can be covered by a small number of deterministic, computation-intensive dataflow graphs (e.g., the DCT routine in MediaBench's JPEG encoder contains 151 RISC-type instructions). Additionally, a large number of operands exhibit temporal locality and spatial locality. These properties make our mechanism far less complex to implement.

Other researchers have studied dynamic collapsing on a multi-input execution unit. This technique combines dependences among multiple instructions into a single entry and maps the entry to special hardware that can efficiently execute it [61]. This technique increases the efficiency of the issue queue and reorder buffer. It also removes operand transport within the collapsed dependence strings. Collapsing for specific instances of floating-point operations, with the addition of new instructions such as multiply-add, has been implemented in a number of processors [45]. A general collapsing scheme involving fixed point arithmetic and logical operations has been proposed in [40], and a subset of

this proposal is implemented in a commercial processor POWER 2 [77]. Sazeides explores the potential of instruction dependence collapsing on 3-1 and 4-1 (three and four inputs respectively, with one output) ALUs [61]. Our approach is similar to dependence collapsing in that both group dependent instructions and focus on transporting the intermediate values through the fast paths. However, there are important differences: (i) our mechanism maintains the original instruction atomicity while the dependence collapsing replaces a set of instructions into an atomic macro instruction, and (ii) unlike our general instruction grouping, the dependence collapsing can only group restricted instruction combinations supported by the execution unit.

Sassone's dynamic strand [59] similarly groups dependence chains of at most three integer ALU instructions joined by transient operand, with no fan out, and steers them to the self-bypassing ALUs. It works well over a variety of applications since the linear form of a dependence graph (e.g., pointer manipulation for memory accesses and branch predicate calculation) is popular. For multimedia applications, we present a technique to form larger and more general clusters of instructions (e.g., by lifting the restriction on fanout) to accommodate a broader class of operand distribution patterns inherent in these applications.

4.2.2 Solutions for Multimedia Processing

To address the demand for data-parallel processing, general-purpose architectures employing SIMD functionality have been developed. Examples include Intel's SSE [54], AMD's 3DNow! [49], and Motorola's AltiVec [20]. Digital signal processors (DSPs) and media processors, such as TigerShark [70] and Trimedia [72], have followed the trend. They incorporate SIMD functionality typically at subword level, i.e. they operate

concurrently on multiple narrow data types, e.g., eight-bit or 16-bit in a 64-bit register. While significant speedups have been demonstrated for multimedia kernels and applications, they have created a need for software support to develop and port applications. Examples include compiler/automatic retargeting technology and hand optimization using in-line assembly code, intrinsic functions, or library routines. Another challenge is scalability. An option to exploit more parallelism is to add more multimedia FUs next to ILP units and to increase issue width. However, this incurs critical communication problems between computing resources [14].

An alternative approach is to implement scalable processors and to take advantage of available FUs. The most commonly suggested method is clustering [24][48] – dividing a processor's resources into logical groups. Recently, there has been interest in modulo scheduling for the clustered architectures, which overlaps successive iterations of a loop and uses the same schedule to optimize resource utilization [56]. Previously described tile architectures, such as the TRIPS architecture based on grid processor cores [58] and the RAW architecture [68], can be configured to support data-parallelism. Network-connected tiles are filled with unrolled innermost loops of streaming applications. They achieve high performance by leveraging the technology scalable PEs connected by fast operand communication networks in a static environment.

Our approach complements the large body of previous research that focuses on detecting *independent* computations that can be performed in parallel through the definition of new ISAs and additional software support. In contrast, we focus on improving the performance of regular patterns of *dependent* instructions, which are inherently sequential, in the dynamic execution environment.

4.3 Methodology

This section describes the instruction clustering mechanism in detail, addressing efficient operand traffic control and data-parallelism detection.

4.3.1 Basic Instruction Clustering Concept

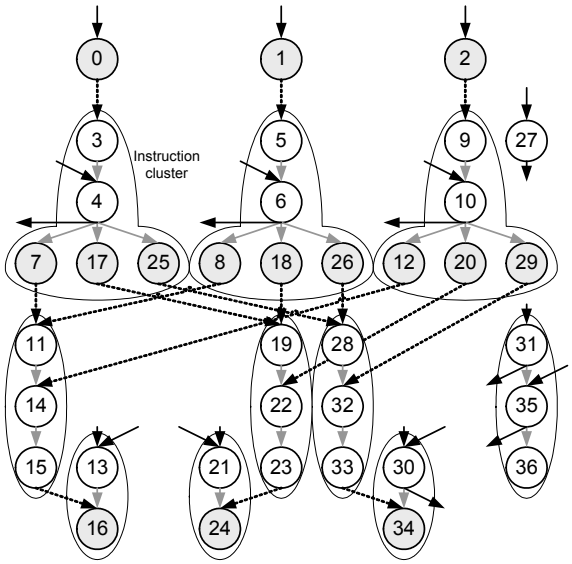
Our empirical analysis [34] of operand usage and communication properties for multimedia programs has revealed that operands tend to be used only a small number of times (95% of all operands are used at most three times), are usually consumed shortly after they are produced (83% of operands are consumed within five dynamic instructions), and have short lifetimes (76% are dead within ten dynamic instructions). Yet, in current architectural models, all operands are treated alike; these intermediate, short-lived operands consume the same storage as long-lived operands and contribute greatly to traffic congestion among the FUs, register file, and broadcast buses. *Local operands*, which are values produced within a code block and consumed by an instruction within the same block, form the building blocks of our instruction clusters. These values often connect critical dependent instructions but may not be committed to the architectural state of the machine.

Figure 19 illustrates the basic concept of instruction clustering on the data flow graph generated from the color conversion basic block in the MediaBench JPEG encode program. Each node represents an instruction (gray nodes denote memory instructions, e.g., load or store, and white nodes denote ALU instructions) and each edge represents a true data dependence.

During the instruction clustering, dependence edges are classified according to the producer-consumer relationship: (i) *external* (solid line): an operand which is produced

by previous blocks or may be consumed by subsequent blocks; (ii) *memory* (dotted line): an operand which is produced by a load as data read or consumed by a store instruction as data to be written; and (iii) *local* (gray line): an operand which is produced and consumed within the current block by instructions that perform integer computations. Note that some operands are local, memory, and/or external at the same time since they may be consumed in the current block as well as by instructions in subsequent blocks. These are indicated by multiple edges fanning out of an output port, for example, gray lines to local consumers and a solid black line pointing out of the dataflow graph.

```
0: lbu r4, 0(r9)
1: lbu r5, 1(r9)
2: lbu r6, 2(r9)
3: sll r4, r4, 0x2
4: addu r4, r4, r10
5: sll r5, r5, 0x2
6: addu r5, r5, r10
7: lw r2, 0(r4)
8: lw r3, 1024(r5)
9: sll r6, r6, r10
10: addu r6, r6, r10
11: addu r2, r2, r3
12: lw r3, 2048(r6)
13: addu r7, r25, r8
14: addu r2, r2, r3
15: sra r2, r2, 0x10
16: sb r2, 0(r7)
17: lw r2, 3072(r4)
18: lw r3, 4096(r5)
19: addu r2, r2, r3
20: lw r3, 5120(r6)
21: addu r7, r15, r8
22: addu r2, r2, r3
23: sra r2, r2, 0x10
24: sb r2, 0(r7)
25: lw r2, 5120(r4)
26: lw r3, 6144(r5)
27: addiu r9, r9, 3
28: addu r2, r2, r3
29: lw r3, 7168(r6)
30: addu r7, r12, r8
31: addiu r8, r8, 1
32: addu r2, r2, r3
33: sra r2, r2, 0x10
34: sb r2, 0(r7)
35: sltu r2, r8, r16
36: bne r2, r0, 0x412188
```



(a) (b)

Figure 19: Basic instruction clustering example based on data flow graph of a basic block from MediaBench JPEG encode: (a) assembly source code and (b) dataflow graph and instruction clustering.

An *instruction cluster* is defined as a connected subgraph of instructions that are joined by local operands. The input fringe (top) of a cluster consists of instructions for which no register sources are local; the output fringe (bottom) consists of instructions that generate no output or whose outputs are the sources of only memory or external edges.

The most important aspect of the cluster formation is the assignment of the dependence edges to local and global communication paths. Local operands are allowed to be delivered through local communication paths and to be safely discarded, while external operands should be assigned to a global path for future use. This separation guarantees fast transport of local operands [36].

4.3.2 Extended Instruction Clustering for Loop-Oriented Applications

Most multimedia applications, especially image processing applications, are characterized by predictable loop-based control flow with large iteration counts [67]. Moreover, empirical analysis has revealed that most operands have good locality properties as described earlier. This section introduces an extended instruction clustering mechanism, which targets data-parallelism detection as well as efficient operand traffic control based on the application characteristics.

Typically, image processing algorithms involve heavy usage of multiply nested loops (commonly "*for*" loops in C source code). We focus on innermost loops since they are the elementary blocks of the multi-level loops and dominate overall processing time. Figure 20 illustrates the concept of the extended instruction clustering mechanism on the dataflow graph generated from the innermost loop of the image convolution code in the Texas Instruments (TI) IMGLIB [71] suite.

It forms the instruction clusters in the same manner as Figure 19 (In the graph, IC_i represents the instruction cluster i). The key differences are the region where the instruction clustering is performed and the additional classification of the external operands. While the former clustering mechanism focuses on the deterministic (non-speculative) producer-consumer relationship within a basic block, the later one applies it

to an innermost loop body that may consist of multiple basic blocks. This is because the innermost loops exhibit the most primitive level of data-parallelism in general [37].

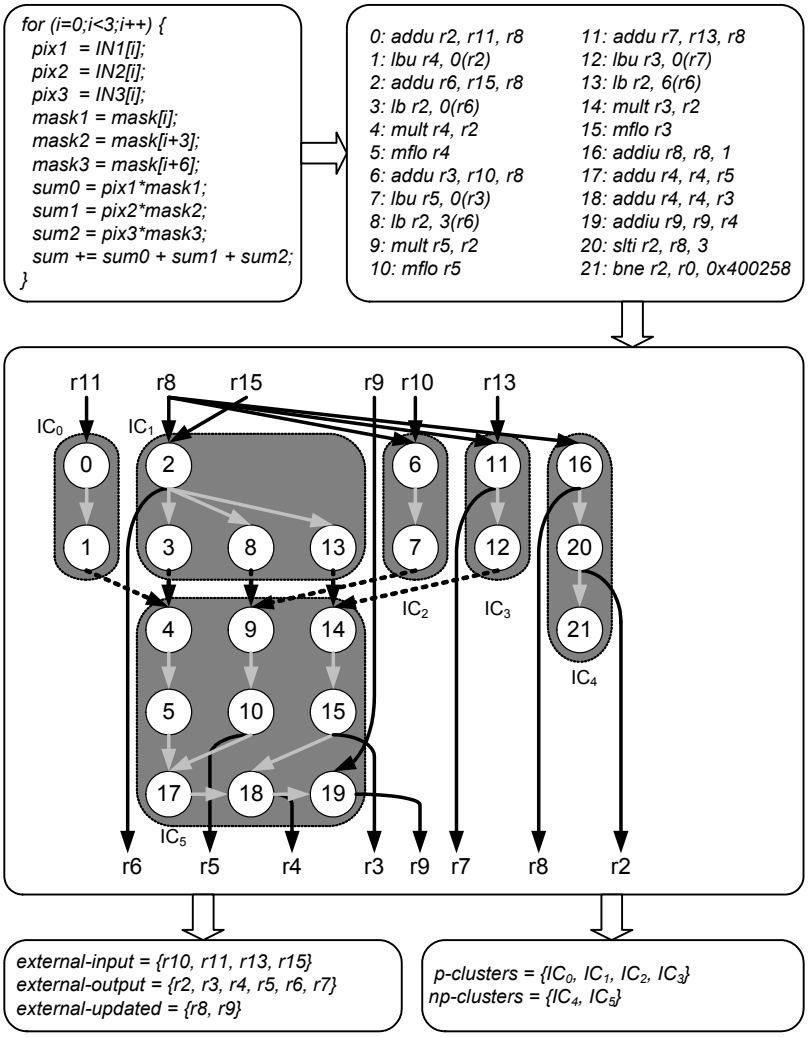


Figure 20: Extended instruction clustering example based on the dataflow graph of an innermost loop from IMGLIB convolution code.

To determine the scope of the external operands, they are further classified according to the input-output relationship of the loop body: (i) *external-input*: an operand which only serves as input to a loop iteration; (ii) *external-output*: an operand which only serves as output; and (iii) *external-updated*: an operand which serves as both input and output. The bottom box in Figure 20 shows an example of this classification. This allows

the operands to move only to the targets that require them. The local operands and the input edges of the external-updated operands are only used during the current iteration while the output edges of the external-updated operands are only consumed by the next iteration. The external-inputs are consumed by all iterations. Finally, all external-outputs except the last iteration have no consumers. Given range of distribution, operand traffic can be bounded to the pre-defined targets and unnecessary communication can be removed.

Another key feature is the detection of data-parallelism based on sequential dataflow representations. The instruction clusters that produce the external-updated operands are analyzed since they form critical loop-carried dependences. Once the operand transport patterns (from edges) and specific computations (from nodes) are recognized and identified, the stride values between loop iterations are predicted. Typically, they are chains of a small number of instructions connected by a simple operand transport pattern. From the example in Figure 20, the instruction 16 in IC_4 and instruction 19 in IC_5 produce the external-updated operand r8 and r9 respectively. The constant stride value comes from the immediate source of instruction 16 for r8; that is, r8 values for subsequent loop iterations can be easily computed by accumulating the identified stride. However, the stride for r9 cannot be predicted. The next r9 value is computed by a complex combination of operations and unknown memory operands in IC_5. The attribute of stride predictability is assigned to each external-updated output and propagated to its counterpart (external-updated input). It determines parallelism of the instruction clusters. A detailed explanation is presented in Section 4.5.1.

4.4 Operand Traffic Control for ILP Processing

This section describes the overall microarchitectural support for the instruction clustering mechanism presented in Section 4.3.1. The detailed experimental results of our mechanism on dynamically scheduled ILP processors are also presented.

4.4.1 Microarchitectural Support for the Instruction Clustering Mechanism

Figure 21 shows the basic organization of our instruction clustering mechanism and its corresponding pipeline stages. There are three new major components (gray boxes): cluster formation logic and cache; cluster queue and scheduling logic; and cluster execution unit. Each component is discussed in turn. Note that compatibility with the existing code is maintained because no changes are needed at the instruction set architecture level.

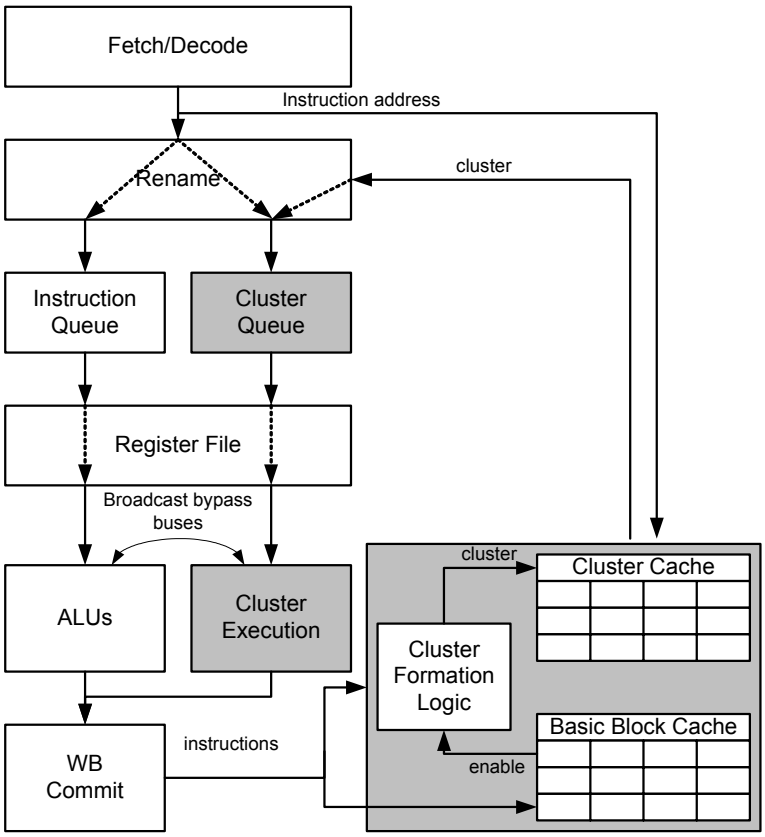


Figure 21: Basic organization of clustering mechanism and its pipeline stages.

Cluster formation logic and cache: As instructions are retired from the pipeline, they are collected by the cluster formation unit and combined into traces similar to a trace cache fill unit [27]. Each trace is segmented at basic block boundaries and then explicitly annotated with dependence information. By default, all output ports of non-memory instructions are initially considered to be sources of external edges. However, they are removed when the associated registers are overwritten by subsequent instructions within the trace, since the operand lifetime is expired. Each instruction is then checked to see if it connects to an existing instruction cluster through a local operand. If it does, the cluster formation unit appends the instruction to that cluster; otherwise, a new cluster is begun.

After the clusters are formed, each instruction in the cluster is assigned a dependence depth that is the number of instructions in the longest dependence chain from the input fringe of the cluster to the input of the instruction. This attribute information is used to steer the instruction to a specific ALU.

The basic block cache keeps track of basic block statistics, such as the number of times the blocks have been seen, and is indexed by the start address of each block. When a basic block is committed a second time, the cluster formation is activated and the abstracted cluster information is stored in the cluster cache. This guarantees infrequent cluster formation since many multimedia applications have a high degree of code locality. Each cluster cache entry holds instruction addresses and attribute bits such as source/destination operand locality bits and dependence depth bits.

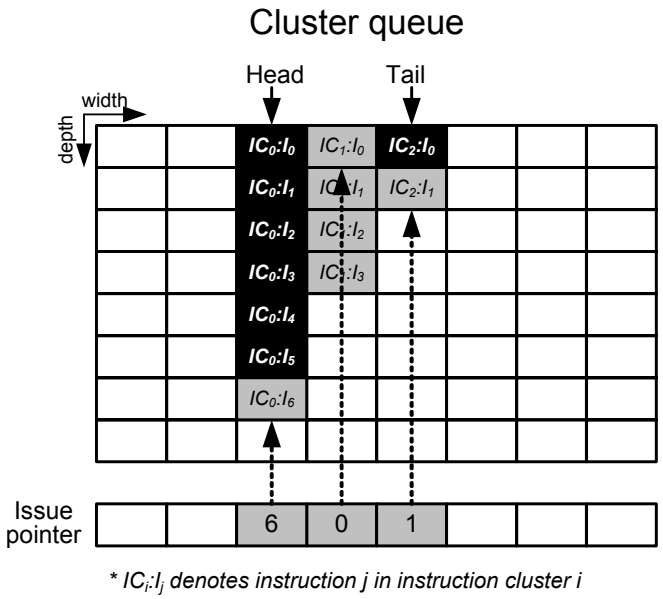
Cluster queue and scheduling logic: The dispatch/renaming logic is responsible for checking the address of the instruction stream, locating the matching cluster to the cluster queue, and removing the individual instructions from the stream if the instruction address

finds a matching entry in the cluster cache. Figure 22(a) illustrates the cluster queue. The gray and black boxes indicate the occupied entries, i.e., the black boxes represent issued instructions and the gray boxes to-be-issued instructions. The contents of the cluster queue are similar to the conventional instruction queue (e.g., register update unit (RUU) [64]) except for the following: (i) multiple dependent instructions (an instruction cluster) reside within a single entry, shown as a column in Figure 22(a); (ii) the ready flags of local operands are automatically set to one, which means they are ready when the instruction is dispatched; and (iii) each queue has a pointer to the instruction to be issued next.

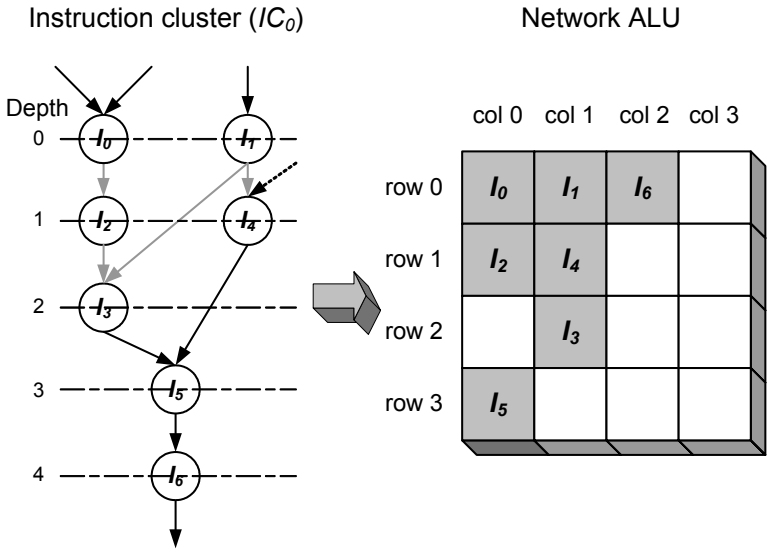
Each instruction in the cluster, once source operands are ready, is issued to one of the network ALUs in the cluster execution unit. The outputs of the ALU array in a row are connected to all inputs of the ALU array in the next row. Figure 22(b) illustrates how the clustered instructions (IC_0 in Figure 22(a)) are mapped to the network ALUs. The depth bits are used to determine the row of the ALU. They guide the dependent instructions to back-to-back rows and guarantee that inter-instruction communication is resolved through the local inter-ALU paths. Note that the depth wraps around when it is greater than or equal to the number of rows (e.g., I_6 is steered to row 0).

Cluster execution unit: The network ALUs are the core of the cluster execution unit. As shown in Figure 23, this unit consists of a set of ALUs that are arranged in two-dimensional space, with wire connections between rows of ALUs and between the input/output ports and the ALU within each column. Some of the individual ALUs in the execution engine are converted to network ALUs which handle the clustered instructions

(we use four, which is the maximum limit of fully-connected bypassing with no penalties [36]).



(a)



(b)

Figure 22: The function of the cluster queue and cluster scheduling logic: (a) organization of the cluster queue and (b) instruction issue and mapping.

The operands can move along three paths: (i) a local path (fully-connected dedicated wires between consecutive rows of ALUs) when dependent instructions are

safely mapped to consecutive rows; (ii) an input/pass-through path (shared buses connecting an input port and the ALUs in the same column) when the required operands come from the register file or one of the conventional ALUs through a global broadcast bus, or if operands needs to be passed-through two or more rows in the cluster execution unit (for example, I_1 to I_3 in Figure 22(b)); and (iii) an output path (shared buses connecting an output port and the ALUs in the same column) to transport memory/external operands to the other parts of the processor.

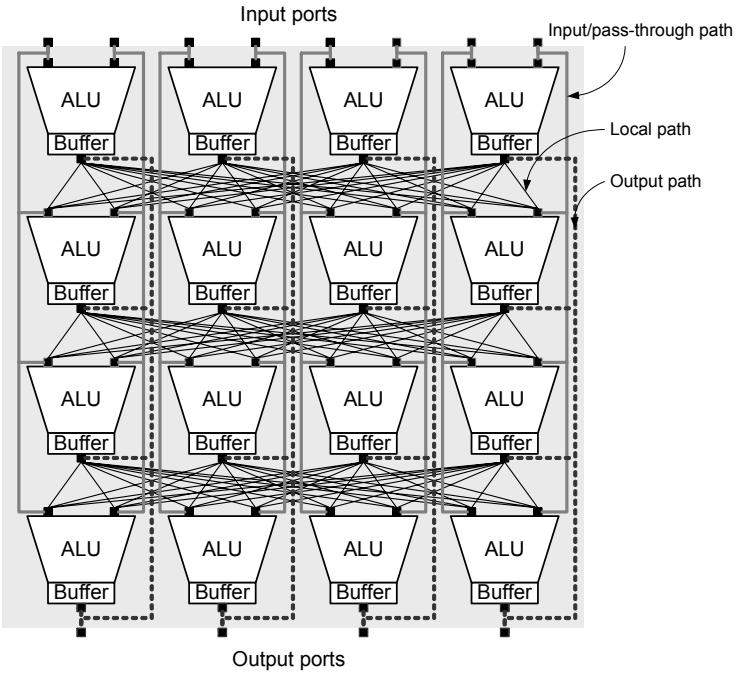


Figure 23: Cluster execution unit example: network ALUs.

Input/output ports of the cluster execution unit are connected to the global broadcast buses. Note that the wire delay latency of broadcast bypass remains constant when the cluster execution unit replaces the conventional ALUs. The total number of input and output ports of the execution unit does not change though the number of available FUs increases.

4.4.2 Experimental Results

The effectiveness of our instruction clustering mechanism is measured in an ILP processing environment. The simulation models are implemented based on the detailed out-of-order processor model provided with Simplescalar (*sim-outorder*) [4]. Both eight- and 16-way machine configurations are simulated with parameters as shown in Table 7. Benchmarks from MediaBench [39] are used for analysis. The default MediaBench inputs are enlarged to lengthen their execution. For each simulation, we execute 500 million committed instructions after skipping the first 100 million instructions (initialization routines).

Table 7: Simulation model configurations.

	8-way	16-way
Queues	24 instruction queue, 8 cluster queue, 16 load/store queue	48 instruction queue, 16 cluster queue, 32 load/store queue
Computing resources	4 integer ALUs, 1 (4x4) network ALU	8 integer ALUs, 2 (4x4) network ALUs
	2 integer MULs, 2 floating ALUs, 1 floating MUL, 2 memory ports	
Operand transport network (latency)	local (0), path-through (1), global (up to 1)	local (0), path-through (1), global (up to 3)
Memory system (latency)	64K 2-way IL1(3), 64K 2-way DL1(3), 1024K 16-way unified L2(8), main memory (160)	
Branch	Combined bimodal/gshare, 4K-entry BHT, 4-way 2K-entry BTB, 10 cycle branch penalty	

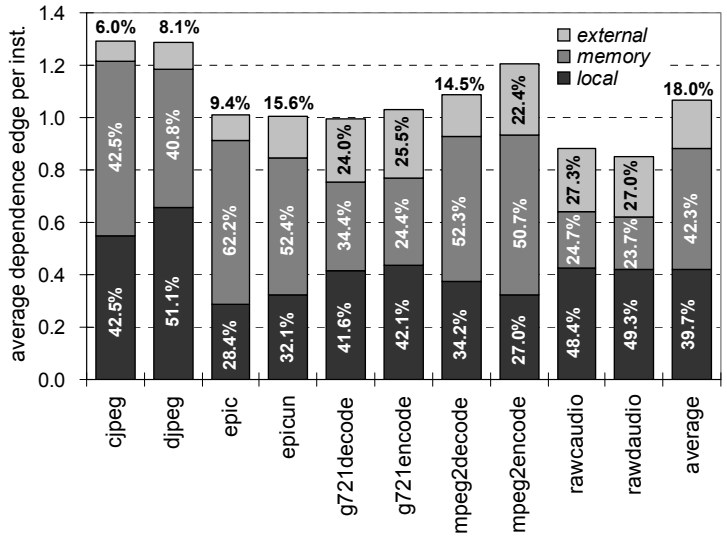
In Table 7, the operand transport latencies are estimated from the wire delays calculated by GENESYS [43]. The delays are measured at the 100nm technology point and the areas of the computing resources are estimated based on the R10000 processor model [73]. The global wire latency varies according to physical transport distance from

the operand producer to its consumer. This emulates the resource partitioning (hardware clustering) technique. The baseline models are configured by setting the size of instruction queue to the number of instruction queue plus the number of cluster queue and by replacing network ALUs to their normal counterparts (four individual ALUs).

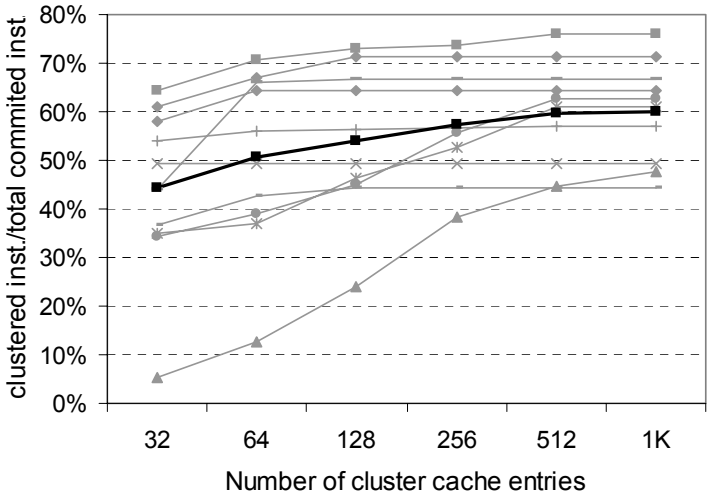
To determine the scope for cluster formation, we initially characterize the type of dependence edges in the instruction window. Figure 24(a) shows the prevalence of dependence edges with a 64-entry instruction window. It also presents the distribution of dependence edge types. In the graph, each bar denotes the average number of edges into instructions in the window, which gives a measure of the total amount of operand traffic currently passing through the expensive global bypass mechanism. On average, about 40% of dependence edges are classified as local, exposing a huge potential for exploitation. Interestingly, about 42% of edges are memory type – effective address transport from ALU to load/store queue, and data to be stored to memory or to be loaded from memory. The other 18% of edges pass through the control boundary. We preclude the external edges since they may be incorrect when instructions are grouped across a mispredicted branch.

Figure 24(b) shows the percentage of dynamic instructions that are grouped into clusters with various cluster cache sizes. Each line represents a benchmark program. The more instructions are executed as a cluster, the more operands can be transported through the local paths. But the coverage itself may not be directly proportional to the performance benefit since the criticality of instructions varies. Also the coverage itself cannot be directly correlated to the rate of local edges as shown in Figure 24(a). In general, the number of local edges depends on the application program itself. Across all

benchmarks, 45~76% of total instructions can be clustered using a 1024-entry cache. The saturation points occur at very different points for each benchmark. The target cache size must be carefully chosen with several factors in mind, such as the target coverage rate, cache access time and area cost, the size of cluster queue, and dimension of network ALUs. The rest of this section assumes a 256-entry cache.



(a)



(b)

Figure 24: Dependence edge type distribution and dynamic instruction coverage: (a) average number of dependence and (b) instruction coverage.

Figure 25 shows the percentage distribution of the actual operand transport paths. Note that the percentage of operands transported within the cluster execution unit (local plus pass-through transport) is slightly higher than that of local edges in Figure 24(a). Some inter-cluster communications, which are classified as external in Figure 24(a), can be done within the network ALUs when multiple clusters are mapped and executed simultaneously. On average, about 30% and 32% of total dependence edges are mapped on the fastest local path for eight-way and 16-way respectively. The pass-through transports occur due to the program structure (as shown in Figure 22(b)) or instruction mapping failure caused by contention at the target row. For example, high instruction coverage and highly prevalent dependence edges in *cjpeg* and *djpeg* cause a considerable amount of pass-through transport – 26% and 30% of the total transport for eight-way respectively.

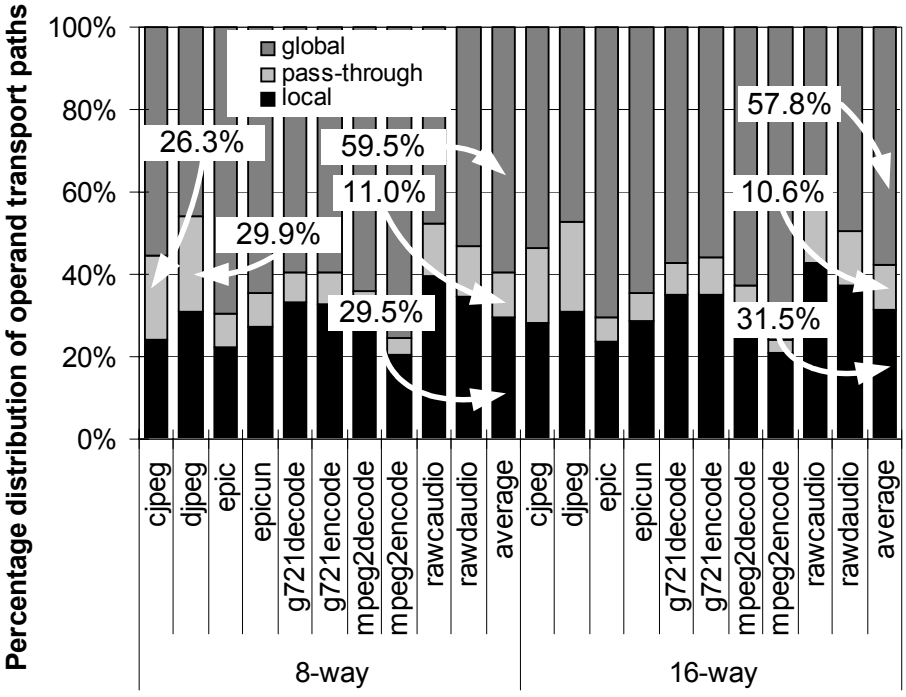


Figure 25: Percentage distribution of operand transport paths.

Figure 26 presents the IPC speedup of our instruction clustering mechanism compared to the baseline models. By being able to deliver the required operands to the target ALUs immediately, the dependent instructions on the critical path can be executed in consecutive cycles. In addition, the enhanced ILP mechanism, supported by a virtually larger window of pre-scheduled in-flight instructions and more parallel computing resources, offers more opportunities to discover independent instructions to be issued simultaneously. On average, the resulting speedups are 16% and 35% for eight-way and 16-way configurations, respectively. The higher speedups for 16-way configuration compared to those of the eight-way machine demonstrate that our mechanism is efficient for wide architectures.

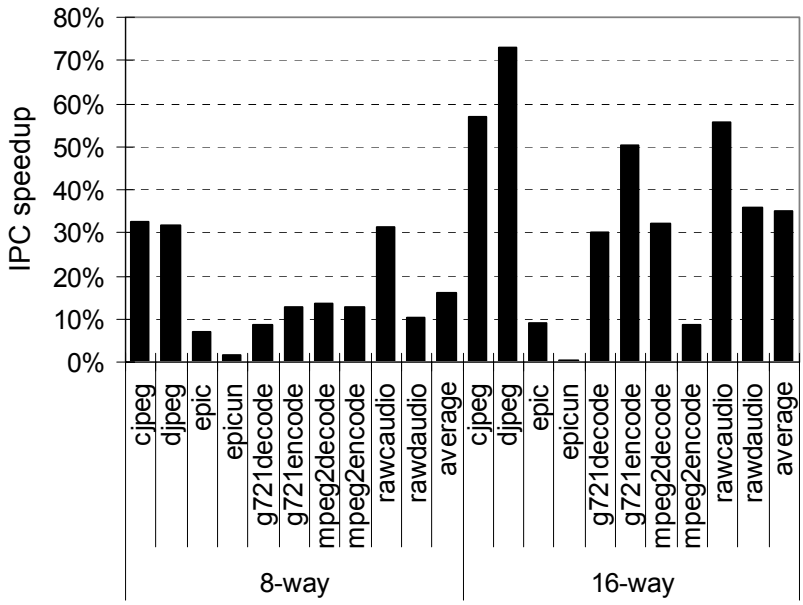


Figure 26: Instruction clustering performance result (IPC speedup over the baseline model).

Several programs, such as *cjpeg* and *djpeg*, show significant speedup since many of their dependence edges are converted to local transport. Interestingly, a few programs,

such as *epic*, *epicun*, and *mpeg2encode* show little speedup. In the first two cases, the bypass latency has little effect on the performance. Even with free bypassing (zero latency), the speedups are only 15~17% for the eight-way. The *mpeg2encode* benchmark has a small amount of local transport as shown in Figure 25. Note that in the superscalar execution, several run-time conditions can contribute to slack in the normal execution pipeline, such as mispredicted branches and cache misses. This slack can hide the effect of our mechanism and reduce the amount of speedup our mechanism achieves.

4.5 Operand Traffic Control for DLP Processing

This section provides the details for a dynamic execution mechanism for loop-oriented applications, called *dynamic SIMDization*. We describe the microarchitecture and present the experimental results.

4.5.1 Microarchitectural Support for Dynamic SIMDization

The microarchitecture for the dynamic SIMDization technique is shown in Figure 27. It is based on a dynamic optimization mechanism using trace-cache techniques [31]. Three new hardware components (darkly shaded blocks in Figure 27) are introduced next to the existing superscalar out-of-order pipeline: loop analysis logic and cache, SIMD instruction queue, and SIMD PE array. An overview of the additional hardware is discussed in the following.

Loop analysis logic and cache: The loop analysis logic observes the instructions being committed and updates the loop cache. It finds the innermost loop regions, analyzes operand transport patterns in the loop trace, and caches them for future use. The loop detection is achieved with a small structure, called a *loop register*, which is composed of

a loop start address, a loop end address, and a loop counter. It checks every direct conditional branch which has a backward target address (a candidate loop). When the same branch occurs consecutively, it marks the branch as an innermost loop. The contents of the loop register are cached in the loop cache to determine the boundary of the loop and its iteration count.

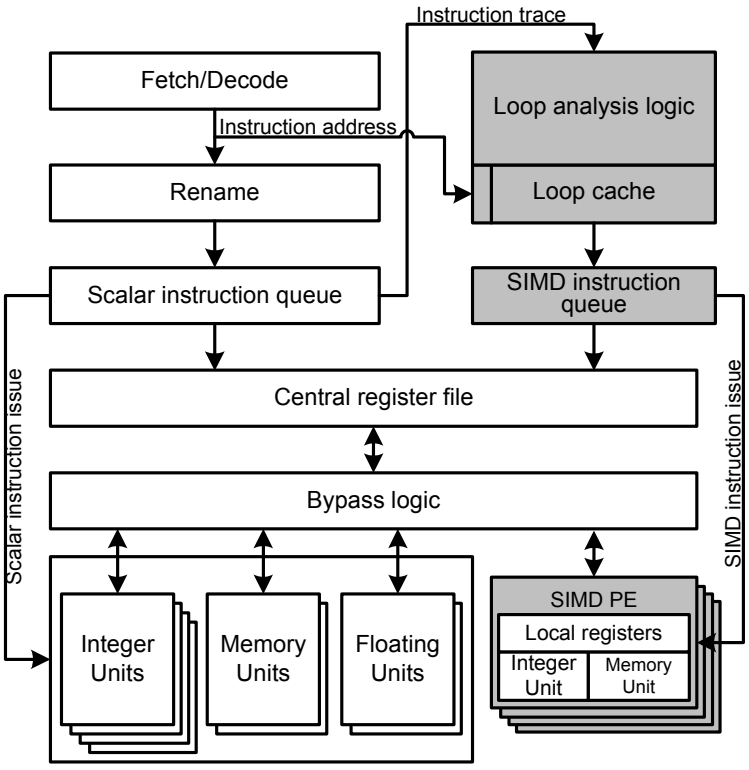


Figure 27: Block diagram of the dynamic SIMD architecture.

The loop analysis logic watches the address of committed instructions. Once the address of the committed instruction matches the loop start address in the cache, the cluster formation is activated and the subsequent instructions are collected in a trace until the corresponding loop end address is seen. It forms the instruction clusters in the same manner as the cluster formation unit in Figure 21: the dependence edge classification and the cluster number assignment. Then, the external input register set (E_{in}) and output

register set (E_{out}) are collected to further classify the external edges as *input*, *output*, and *updated*. They are identified by computing E_{in}–E_{out}, E_{out}–E_{in}, and $E_{in} \cap E_{out}$, respectively. These processes are simply implemented by mask operations.

The cluster analysis logic keeps track of operations and operand connections in the clusters to look for value prediction opportunities. If a stride of an external-updated operand is identified as predictable, the predicted stride values are computed and stored in the stride predictor tables in the PEs. For instance, the stride value for r8 in Figure 20 is predicted as one. Each PE stores zero, one, two, and three for r8, respectively, when four PEs are assumed. In this way, all instructions in PEs which require r8 as a source can receive the value upon the arrival of r8 for the first PE (other PEs use predicted r8 values).

Based on the predictability of the external-updated operands, our mechanism separates the parallelizable and non-parallelizable regions in the loop body. For example, the $IC_0 \sim IC_3$ in Figure 20 are marked as *p-cluster*. A p-cluster is defined as an instruction cluster that produces no external-updated output and that has no unpredictable external-updated inputs. They can be issued and executed in the PE array in SIMD fashion. The others are declared as *np-clusters* and handled using conventional ILP processing mechanisms.

SIMD instruction queue: Each instruction, after being decoded, is sent to the dispatch engine. The dispatch logic checks the address of the instruction stream and maps the instruction clusters to the SIMD instruction queue when the instruction address matches an entry in the loop cache. An entry in the SIMD instruction queue represents instructions of multiple loop iterations (equal to the number of PEs). It has a single instruction address and opcode, but keeps track of multiple versions of flags such as ready and instruction

status. The cluster information and transport type for input and output operands are also appended to control data communication.

The instructions in the p-cluster and np-cluster are scheduled in different ways. The arrival of the last required operand for the first PE triggers the issue of a p-cluster instruction to all PEs. The stride prediction mechanism makes it possible for PEs to execute the instructions with predicted values. On the other hand, the np-cluster instructions are always issued one-by-one when corresponding operands are ready. Figure 28 depicts a possible scheduling of a p-cluster (IC_1) and an np-cluster (IC_4) in the example of Figure 20. In this example, it is assumed that SIMD array contains four PEs and no structural hazard occurs during execution. The x-axis represents relative timing. The number in the box represents the instruction id and a subscript is attached to identify the target PE.

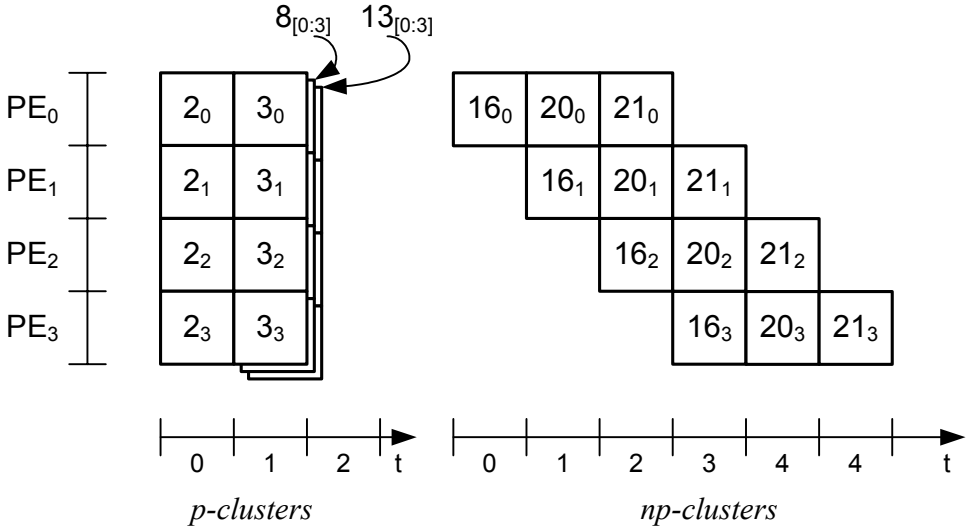


Figure 28: Cluster scheduling example.

It is important to note that the instructions in the p-clusters (except those in PE_0) are executed speculatively; that is, a recovery process is required when the prediction

fails. Furthermore, loop-carried dependences caused by memory operands may exist. In both cases, the local results in a PE that has a mispredicted value or a memory dependence are discarded and the correct processor state is recovered.

SIMD PE array: The execution target for the dynamic SIMDization is a SIMD PE array that connects nearest neighbors in a one-dimensional mesh. A PE consists of a small number of fine-grained FUs and a local register file to store temporary values. The operand transport network facilitates communication within a PE, between PEs, and between a PE and scalar resources. It provides alternative routes for each communication. Figure 29 shows the basic organization of a single PE having two FUs.

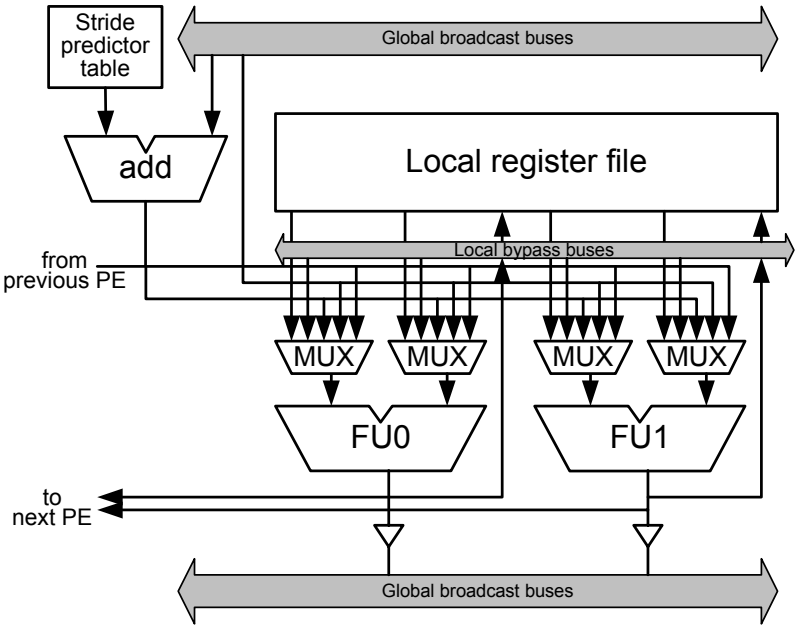


Figure 29: Basic organization of single PE in SIMD array.

By default, an operand produced within a PE is written to its local register file. It is also forwarded to all FUs within the PE through the fully-connected local bypass buses. The external-input operands directly come from the global broadcast buses. Typically, the external-updated operands are passed through the dedicated neighboring

network. However, special hardware support is provided for the predictable external-updated values. A dedicated adder along with the stride predictor table computes predicted external-updated values for each PE. Among the results of the last PE, those marked as external are passed to the global broadcast buses.

4.5.2 Experimental Results

To evaluate the effects of the dynamic SIMDization mechanism, we implemented our structures and algorithms on the cycle-accurate Simplescalar simulator. As baselines for comparison, dynamically-scheduled ILP processors are simulated at three different superscalar widths. Simulation model configurations are shown in Table 8.

Table 8: Simulation model configurations.

Feature	baseline	ILP increase		SIMD extension	
	base4	*base8*	*base16*	*base4+ SIMD4*	*base4+ SIMD8*
Fetch/decode/ issue width	4	8	16	4	4
Scalar resources (integer)	4 ALUs, 1 Mult	8 ALUs, 2 Mults	16 ALUs, 4 Mults	4 ALUs, 1 Mult	4 ALUs, 1 Mult
Scalar resources (floating)	2 floating ALUs and 1 floating Mult/Div/Sqrt				
Vector resources (SIMD)	-	-	-	4 ALUs	8 ALUs
Memory ports	2	4	8	4	8
Reorder buffer size (slots)	64	128	128	128	128
Memory system (latency)	64K 2-way IL1(3), 64K 2-way DL1(3), 1024K 16-way unified L2(8), and main memory(160)				
Branch prediction	Combined bimodal/gshare, 4K-entry BHT, 4-way 2K-entry BTB, 10 cycle branch penalty				

Our test suites consist of a number of image processing applications taken from the TI IMGLIB library [71]. Table 9 lists the application programs in our test suite. Each

benchmark is compiled using gcc 2.95.3 with O2 optimizations. A quarter common intermediate format (QCIF) input image is assumed. We focus on measuring two benefits of our work: the IPC gain from the data-parallel execution of the instruction clusters and the reduction of IPC penalty caused by the global operand transport.

Table 9: IMGLIB test programs.

Benchmark	Description	Applications
conv_3x3	Convolution of image with a 3x3 filter mask	Noise removal, image smoothing
equalizer	Histogram equalization to improve contrast of image	Image quality enhancement
fdct_8x8	8x8 forward discrete cosine transform	Image compression
mad_8x8	8x8 minimum absolute distance	Video compression
pix_sat	All pixels above threshold value set to maximum value	Image dilation/erosion
quantize	Quantize pixel values to a smaller range of discrete values	Image compression
sobel	Object edge detection	Object detection/recognition
wave_ver	Computing vertical wavelet transform	Image compression (JPEG2000)
wave_hor	Computing horizontal wavelet transform	Image compression (JPEG2000)
color	Color space conversion from YCrCb to RGB	Display of digital video data

Figure 30 shows the percentage of dynamic instructions that are recognized as the retargetable innermost loop region. The bars also show how many of these instructions are analyzed to be p-cluster (gray bars) and np-cluster instructions (black bars). On average over all benchmarks, 88% of dynamic instructions are covered by our loop detection mechanism. Specifically, about 79% of them can be replaced to SIMD operations (p-clusters). The remaining 9% of instructions are observed in the non-data-parallel region (np-clusters) though they are still handled in the SIMD PE array. The

results demonstrate that most image processing applications exhibit a high degree of DLP at the innermost loop level and they can be easily detected dynamically. This reveals a huge potential for enhancing performance from our dynamic SIMDization mechanism.

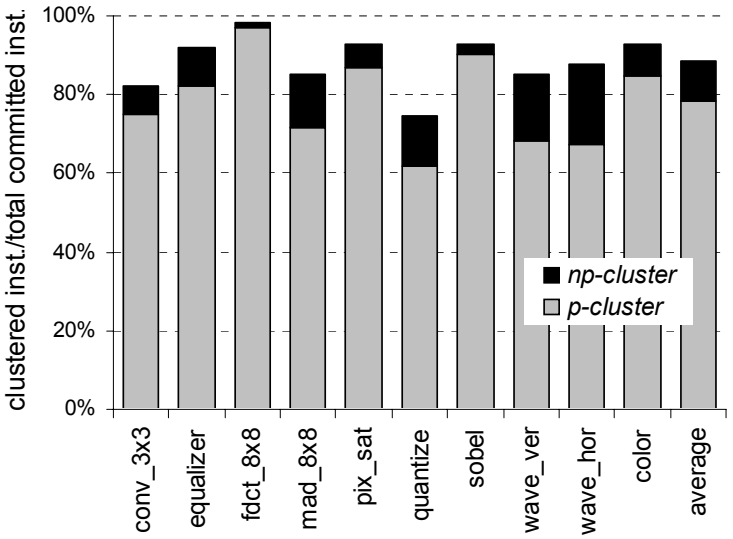


Figure 30: Percentage of dynamic instructions covered by the instruction clustering mechanism for dynamic SIMDization.

Figure 31 presents the IPC speedup obtained when adding our dynamic SIMDization mechanism to a four-wide ILP processor (*base4*). The results are compared to eight-wide and 16-wide ILP architectures. The right-most bars represent the harmonic means of the speedups across all benchmarks. As shown in the left bars, the performance of ILP processors increases moderately with wider pipelines. Additional resources raise the potential to detect independent instructions and to issue them together. In some programs, such as *conv_3x3*, *quantize*, and *wave_ver*, the conventional ILP mechanism benefits from the program properties. In these programs, single control flow of the innermost loop spans tens and sometimes hundreds of instructions and they are repeated a large number of times. Most of the speedup is from aggressive ILP execution exploiting the long deterministic control flow. However, if the innermost loops were made up of

multiple basic blocks (e.g., "*if*" statements in the loops), a mispredicted branch affects the subsequent iterations of the loop. This incurs a pipeline flush even if the subsequent iterations are independent of the result of the mispredicted branch. The innermost loops of *mad_8x8*, *pix_sat*, *sobel*, and *color* programs consist of three or more basic blocks and the results show that the ILP mechanism alone does not improve performance.

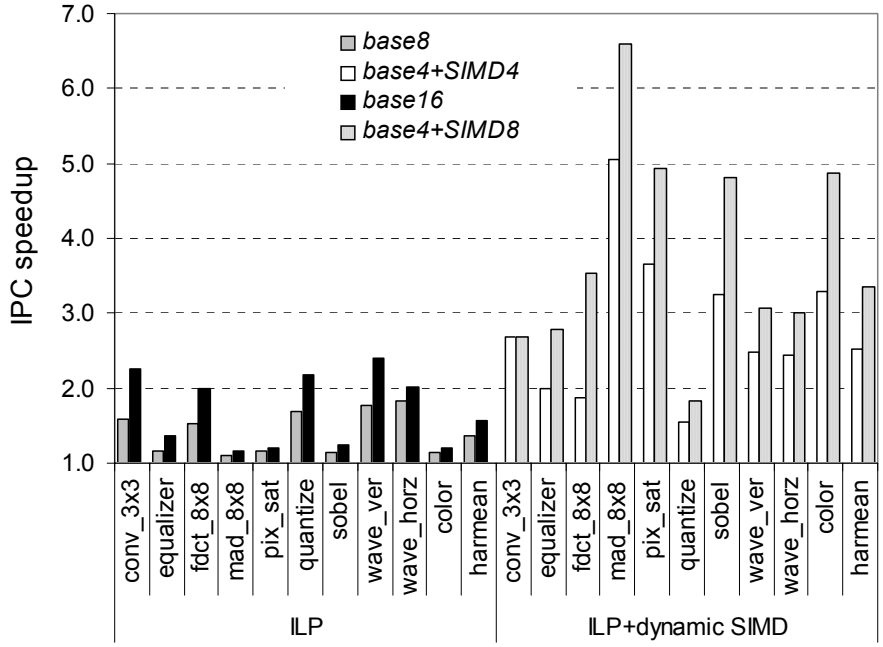


Figure 31: Performance results of ILP increase and SIMD extension over the baseline.

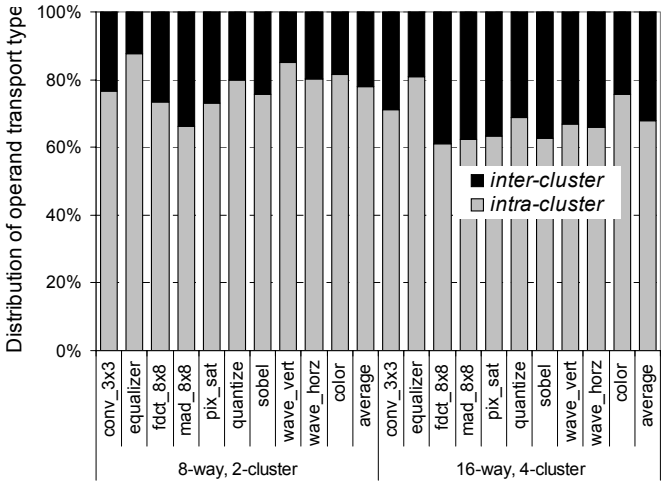
The bars to the right in Figure 31 represent the performance of our dynamic SIMDization mechanism. In most benchmarks, it outperforms conventional ILP architectures. Most of this speedup comes from exploiting data parallelism as well as instruction parallelism. For example, the average speedup of *base4+SIMD4* over *base4* is 2.53 while that of *base8* is 1.36. Interestingly, the performance of *base4+SIMD4* shows less speedup than *base8* in *quantize* where the loop count is extremely high (in our simulation, it is 1200). In this benchmark, the benefit from wider pipeline structures,

including fetch, decode, issue, and commit width, is more dominant than that from data-parallel execution.

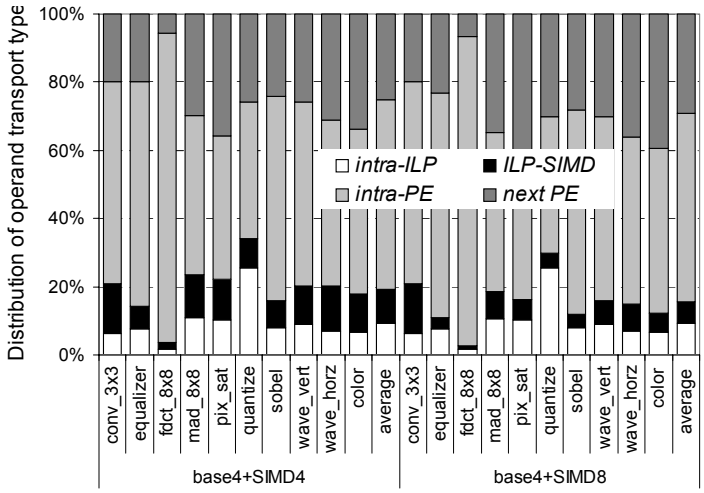
The performance scales well as the size of SIMD PE array increases, which directly translates to more data parallelism. On the other hand, the performance of the ILP architecture is saturated due to the limited instruction parallelism. For example, while the ILP extension from *base8* to *base16* improves the average performance about 16%, the increase in the number of PEs from four (*base4+SIMD4*) to eight (*base4+SIMD8*) yields 32% speedup. An exception is *conv3x3* which shows no IPC gain. Its loop iteration count is three which is less than the number of PEs so the additional resources have no effect on the performance.

An important consideration when architectures become wider is the operand transport complexity. Wide architectures complicate operand communication due to interconnect wire delays. To evaluate the impact of operand transport on the performance, we analyzed the operand traffic as shown in Figure 32. The resource partitioning technique (e.g., the clustered microarchitecture) is applied to the ILP models. The dependence-based instruction steering heuristic is also used to minimize the amount of expensive inter-cluster communication.

Figure 32(a) presents the percentage distribution of operand transport types for the clustered architecture. As expected, it is observed that the amount of *inter-cluster* communication increases as the cluster count increases. On average, the *two-cluster* model configured from *base8* incurs about 22% of inter-cluster communications. The amount is increased to 32% in the *four-cluster* model configured from *base16*.



(a)



(b)

Figure 32: Percentage distribution of dynamic operand transport: (a) clustered ILP and (b) dynamic SIMD architecture.

The operand transports in the dynamic SIMDization mechanism are divided into four groups based on producer-consumer relationship: *intra-ILP* (produced by a scalar resource and only consumed within the scalar resources), *ILP-SIMD* (produced by a scalar resource and consumed in PEs and vice versa), *intra-PE* (produced by a PE and only consumed by the same PE), and *next PE* (produced by a PE and only consumed by

the next PE). The percentage distribution results are shown in Figure 32(b). All except ILP-SIMD can be communicated with no extra penalty since they are local transports. For *base4+SIMD4* model, only 10% of dynamic operands are transported between ILP processor and SIMD array on average. Of interest is that this drops to about 6% when the width of the SIMD PE array increases to eight. Some ILP-SIMD transports are converted into the next PE transports. This implies that our mechanism can minimize the impact of operand movement, reducing performance degradation.

Figure 33 presents the results of the IPC speedup, including operand transport latencies. The exact delay based on the technology model is beyond the scope of this research. We simply apply one-cycle penalty for the global transports, such as inter-cluster transports and ILP-SIMD transports. The operand transport latency directly translates to IPC drops as shown in Figure 33 compared to Figure 31. The average IPC drops of the resource partitioning are 13.8% and 18.3% for two-cluster and four-cluster models respectively. Much lower IPC drops are observed for our dynamic SIMD architectures: 3.4% and 0.9% for base4+SIMD4 and base+SIMD8 models, respectively. The results demonstrate the efficiency of our operand transport mechanism.

4.6 Conclusion

For multimedia processing, the performance of modern ILP processors is restricted due to the limited instruction parallelism existing in applications [76][78] and the operand communication latency. To address these problems, a dynamic execution mechanism is presented and evaluated. It improves performance by performing data-parallel operations and by reducing IPC penalties caused by global communication. Our mechanism effectively groups dependent instructions into a cluster, recognizes the

operand range of distribution, detects data-parallelism from the analysis of intra- and inter-cluster operand transport patterns, and maps the clustered instructions to an efficient cluster execution unit. Two implementations are explored as examples of efficient cluster execution units: the network ALUs and SIMD PE array.

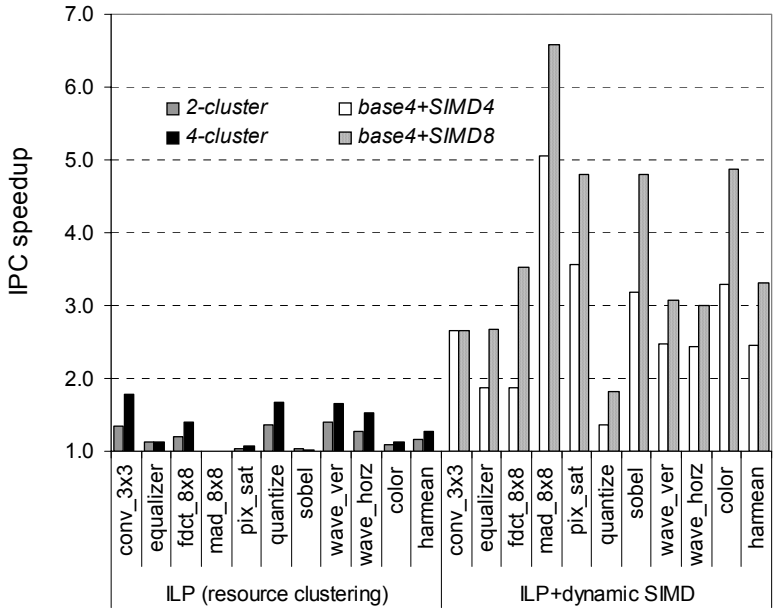


Figure 33: Performance results of ILP increase and SIMD extension including consideration of the operand transport latency.

The network ALUs are shown to improve the performance over a wide range of multimedia applications (MediaBench) – the average IPC speedups over the conventional ILP mechanism are 16% for eight-way and 35% for 16-way configurations respectively. This mainly benefits from significant reduction in global communication – 28% and 30% of inter-instruction communication residing in the instruction queue can be converted to local transport on eight- and 16-way configurations. The effectiveness of our mechanism is more obvious in the SIMD PE array that aims at loop-oriented applications. For image processing applications (IMGLIB), the results show that the overall performance gains

over the conventional ILP processors are 2.1x for eight-way and 2.6x for 16-way respectively. Most of the speedup comes from exploiting more parallelism (shown as IPC increases) and reduction in global communication (shown as reduction in IPC penalty).

CHAPTER 5

CONCLUSION AND FUTURE WORK

This dissertation has addressed architectural issues combining with technology-level issues to provide efficient operand transport mechanisms in executing multimedia application programs. In particular, this dissertation focused on reducing inter-instruction communication latencies caused by wire delays of operand transport networks. As the number of parallel computing resources increases to meet the required performance, a conventional design requires a complex, long-latency operand transport network since all operands are delivered to all operations ready to be executed through poorly scaling broadcasting wires.

To reduce latency associated with operand movement, this research first explored data access patterns and data movement occurring during execution of multimedia applications. Based on the recognized operand characteristics and properties, two architectural enhancements that efficiently control operand traffic at run-time have been developed and evaluated: (i) a traffic-driven operand bypass network and (ii) a dynamic instruction clustering technique for multimedia architectures. Unlike typical broadcasting bypass networks used in conventional processors, the traffic-driven operand bypass network significantly reduces the physical distance required to deliver data while maintaining the wiring cost of the transport network low. Additionally, the dynamic instruction clustering technique supports efficient operand traffic control by exploiting the recognized operand distribution patterns and locality properties. It has been explored

in the context of dynamically scheduled ILP and SIMD execution mechanisms. The following sections conclude our investigations and suggest future research directions.

5.1 Summary of Results

In this section, experimental results obtained from the topics studied in this dissertation are summarized.

5.1.1 Characterization and Modeling of Operand Usage and Transport

This research analyzes the operand usage and transport characteristics in executing multimedia programs. Exploiting common operand distribution patterns and prevalent locality properties observed from the empirical analysis is essential in the design of alternate low cost and low latency operand transport mechanisms.

Our empirical analysis shows that most operands in multimedia applications exhibit a high degree of locality temporally as well as spatially. In the time domain, we observed that 95% of operands are used at most three times, 83% of operands are first consumed within five dynamic instructions, and 76% of operands are dead within a dynamic instruction window of size ten. It is also shown that there are regular patterns of operand transport in the spatial domain. Some are simple passing of intermediate, transient values from an FU to another, usually of the same type (52% of operands). Some are common traffic patterns among certain pairs of FUs (31% of operands).

This research also explores the impact of architectural techniques, which are aiming at localizing the operand communication, on the operand transport. Our results show that (i) 25% of operands are read through the nearest local path with eight-entry local storage and a fully-connected bypass network; (ii) 81% of operand writes to global storage are eliminated by applying dynamic register operand lifetime detection; and (iii)

50% of operands are read directly from local storage by adding dedicated bypass paths between heavily trafficked resources and by applying a novel instruction mapping scheme based on operand consumer information. The effectiveness of these techniques is the key to designing efficient operand transport mechanisms.

5.1.2 Traffic-driven Operand Bypass Network

This research presents a technology-based methodology to evaluate the operand transport designs by predicting transport cost. Based on the methodology, a lower cost, high performance bypass network, called traffic-driven operand bypass network, is presented and evaluated, especially for multimedia applications. Unlike a conventional fully-connected broadcast bypass network, the traffic-driven operand bypass network reorganizes each of bypass paths based on the operand transport patterns. It achieves substantial instruction throughput performance by reducing the operand transport latency and by increasing the clock speed. It also reduces the complexity of the bypass network by reducing the interconnect demand.

Our results show that the overall instruction throughput gain over a conventional broadcast bypass network is 1.76x, 2.27x, 2.89x for a wide range of multimedia applications at 100, 65, 45nm technology, respectively. The interconnect demand, which is measured in the total length of the bypass wires, can be kept within 24% of the broadcast network. The traffic-driven bypass network also achieves average instruction throughput gains of 1.12x, 1.24x, 1.26x over a typical clustered mechanism at 100, 65, 45nm technology, respectively. The interconnect demand is only 50% of the clustered mechanism. These results demonstrate that the traffic-driven operand bypass network is

an lower cost, higher performance candidate for multimedia application than traditional operand bypass mechanisms in future technology.

5.1.3 Dynamic Instruction Clustering Mechanism

This research present a dynamic optimization mechanism, called dynamic instruction clustering. It improves the performance of multimedia applications by performing data-parallel operations and by reducing the operand transport latency. Our instruction clustering mechanism is evaluated on two execution platforms: network ALUs and a dynamically-scheduled SIMD PE array. They share some common hardware structures such as cluster formation logic and cluster queues, yet detail implementations are tuned to the specific targets.

The network ALUs are shown to improve the performance over a wide range of multimedia applications. The average IPC gains over conventional ILP mechanism are 16% for eight-way and 35% for 16-way respectively. The key to their substantial performance gains lies in significant reduction in global communication. 28% and 30% of inter-instruction communication residing in the instruction queue are bypassed through the local path instead of long-latency global buses. The dynamically-scheduled SIMD PE array supports data-parallel processing of the innermost loops in image processing applications. The results shows that the overall performance gains over the conventional ILP processors are 1.87x for eight-way and 2.14x for 16-way by exploiting more parallelism. It also benefits from lowering operand transport latency. It converts global communication into local transport and removes unnecessary communication. When operand transport latencies are factored in, it produces 2.09x for eight-way and 2.59x for 16-way, compared to the clustered microarchitectures.

5.2 Future Research Directions

The research presented in this dissertation is the first attempts to explore and evaluate operand transport for multimedia with dynamic execution techniques. In this section, a number of future research directions are outlined.

Design of Operand Transport Networks

- Enhance dynamic instruction assignment and scheduling algorithm with accurate implementation cost and operating frequency predictions. This will involve developing instruction pre-scheduling mechanisms to map a group of instructions on the computing resources instead of individual instruction.
- Analyze the operand traffic from/to the storage elements and provide an efficient storage configurations as well as the operand transport network.

Dynamic Instruction Clustering for ILP Processing

- Extend the instruction clustering idea to support for larger blocks (e.g., a hyperblock which is a set of basic blocks stitched together) based on the branch prediction and speculative execution.
- Evaluate various types of operand storage configurations (e.g., distributed register files and FU's local storage) that permit efficient operand transport.
- Develop methodology to evaluate the optimum inter-ALU network configurations (FU composition, local storage size, dimension of ALUs, the number of input/output ports) for standard and specific multimedia application domains.

Dynamic Instruction Clustering for DLP Processing

- Extend the instruction clustering concept to support for the innermost loops that contains multiple control flows. This will be performed with dynamically determining the conditionally executed instructions and efficiently controlling their resulting operands.
- Perform an in-depth analysis of multimedia applications and adaptively apply the optimum instruction clustering technique according to the characteristics of specific code regions. Given the information about primary types of parallelism for certain code sections, specific optimization techniques can be selected and applied during run-time.

REFERENCES

[1] T. Agerwala and S. Chatterjee, "Computer architecture: challenges and opportunities for the next decade," *IEEE Micro*, vol. 25, no. 3, pp. 58-69, May 2005.

[2] P. Ahuja, D. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines," *Proceedings of the 28th International Symposium on Microarchitectures*, pp. 36-45, November 1995.

[3] M. Arnold and H. Corporaal, "Automatic detection of recurring operation Patterns," *Proceedings of the 7th International Workshop on Hardware/Software Co-Design*, pp. 22-26, May 1999.

[4] T. Austin, et al., SimpleScalar: "An infrastructure for computer system modeling," *IEEE Transactions on Computers*, vol. 35, no. 2, pp. 259-67, February 2002.

[5] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad-cluster, dynamically scheduled, superscalar processors," *Proceedings of the 33rd International Symposium on Microarchitectures*, December 2000.

[6] L. Baumstark and L. Wills, "Retargeting sequential image-processing programs for data-parallel execution," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 116-136, February 2005.

[7] R. Bhargava and L. John, "Improving dynamic cluster assignment for clustered trace cache processors," *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[8] B. Bishop, R. Owens, and M. Irwin, "Aggressive dynamic execution of multimedia kernel traces," *Proceeding of the International Symposium on Parallel and Distributed Processing*, pp. 640-646, April 1998.

[9] E. Bloch, "The engineering design of the stretch computer," *Proceedings of the Eastern Joint Computer Conference*, pp. 48-59, 1959.

[10] M. Brown and Y. Patt, "Using internal redundant representations and limited bypass to support pipelined adders and register files," *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 289-298, February 2002.

[11] S. Bunchua, D. Wills, and L. Wills, "Reducing operand transport complexity of superscalar processors using distributed register files," *Proceedings of the 21st International Conference on Computer Design*, pp. 532-535, October 2003.

[12] M. Buss, R. Azevedo, P. Centoducatte, and G. Araujo, "Tailoring pipeline bypassing and functional unit mapping to application in clustered VLIW architectures," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 141-148, 2001.

[13] A. Canal, J. Parcerisa, and A. Gonzalez, "Dynamic cluster assignment mechanisms," *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pp. 132-142, January 2000.

[14] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff, "The CSI multimedia architecture," *IEEE Transactions on VLSI Systems*, vol. 13, no. 1, pp. 1-13, January 2005.

[15] L. Codrescu, S. P. Nugent, J. D. Meindl, and D. S. Wills, "Modeling technology impact on cluster microprocessor performance," *IEEE Transactions on VLSI Systems*, vol. 11, no. 5, pp. 909-920, October 2003.

[16] R. Cohn, T. Gross, M. Lam, and P. Tseng, "Architecture and compiler tradeoffs for a long instruction word processor," *Proceedings of the 3rd International Conference on Architecture Support for Programming Languages and Operation Systems*, pp. 2-14, April 1989.

[17] T. Conte, P. Dubey, M. Jennings, R. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, A. Wolfe, "Challenges to combining general-purpose and multimedia processing," *IEEE Computer*, vol. 30, no. 12, pp. 33-37, December 1997.

[18] J. Cong, "An interconnect-centric design flow for nanometer technologies," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 505-528, April 2001.

[19] H. Corporaal, "TTAs: Missing the ILP complexity wall," *Journal of System Architectures*, vol. 45, no. 12, pp. 949-973, 1999.

[20] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85-95, March/April 2000.

[21] K. Diefendorff and P. Dubey, "How multimedia workloads will change processor design," *IEEE Transactions on Computers*, vol. 30, no. 9, pp. 43-45, September 1997.

[22] J. Eble, V. De, D. Wills, and J. Meindl, "A Generic System Simulator (GENESYS) for ASIC technology and architecture beyond 2001," *Proceedings of the 9th Annual International ASIC conference*, pp. 193-196, September 1996.

[23] K. Fan, N. Clark, M. Chu, K. Manjunath, R. Rajiv, M. Smelyanskiy, and S. Mahlke, "Systematic register bypass customization for application-specific processors,"

Proceedings of the Application-Specific Systems, Architectures, and Processors, pp. 64-74, June 2003.

[24] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The Multicluster architecture: reducing cycle time through partitioning," *Proceedings of the 30th International Symposium on Microarchitectures*, pp. 149-159, December 1997.

[25] E. Fetzer, et al., "A fully-bypassed 6-issue integer datapath and register file in Itanium-2 microprocessor," *IEEE Journal of Solid State Circuits*, vol. 37, no. 11, pp. 1433-1440, November 2002.

[26] M. Franklin and G. Sohi, "Register traffic analysis for streaming inter-operation communication on find-grain parallel processors," *Proceedings of the 25th International Symposium on Microarchitectures*, pp. 236-245, December 1992.

[27] D. Friendly, S. Patel, and Y. Patt, "Putting the fill unit to work: Dynamic optimization for trace cache microprocessors," *Proceedings of the 31st International Symposium on Microarchitectures*, pp. 173-181, November 1998.

[28] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A VLIW integer microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 12, pp. 1377-1383, December 1993.

[29] L. Gwennap, "MIPS R10000 uses decoupled architecture," *Microprocessor Report*, vol. 8, no. 14, October 1994.

[30] L. Gwennap, "Ultrasparc unleashes SPARC performance," *Microprocessor Report*, vol. 8, no. 13, October 1994.

[31] Q. Jacobson and J. Smith, "Instruction pre-processing in trace processors," *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 125-129, January 1999.

[32] G. Kemp and M. Franklin, "PEWs: A decentralized dynamic scheduler for ILP processing," *Proceedings of the International Conference on Parallel Processing*, pp. 239-246, August 1996.

[33] R. Kessler, "The Alpha 21264 microprocessor," *IEEE MICRO*, vol. 19, no. 2, pp. 24-36, March 1999.

[34] H. Kim, D. Wills, and L. Wills, "Empirical analysis of operand usage and transport in multimedia applications," *Proceedings of the International Workshop on System-on-Chip for Real-Time Applications*, pp. 168-171, July 2004.

[35] H. Kim, D. Wills, and L. Wills, "Technology-based architectural analysis of operand bypass network for efficient operand transport," *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2005.

[36] H. Kim, D. Wills, and L. Wills, "Reducing operand communication overhead using instruction clustering for multimedia applications," *Proceedings of the International Symposium on Multimedia*, pp. 345-352, December 2005.

[37] H. Kim, D. Wills, and L. Wills, "Optimizing operand transport using dynamic SIMDization in multimedia systems," *Proceedings of the International Workshop on Multimedia Signal Processing*, October 2006.

[38] H. Kim and J. Smith, "An instruction set and microarchitecture for instruction level distributed processing," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 71-81, May 2002.

[39] C. Lee, et al., "MediaBench: A tool for evaluating multimedia and communications systems," *Proceedings of the 30th International Symposium on Microarchitectures*, pp. 40-51, December 1997.

[40] N. Malik, "Interlock collapsing ALU for increased instruction-level parallelism," *Proceedings of the 25th International Symposium on Microarchitectures*, pp. 149-157, September 1992.

[41] E. McLellan, "The Alpha AXP architecture and 21064 processor," *IEEE MICRO* vol. 13, no. 3, pp. 36-47, June 1993.

[42] J.D. Meindl, "Interconnect opportunities for gigascale integration," *IEEE MICRO*, vol. 23, no. 4, pp.28-35, May/June 2003.

[43] J. D. Meindl, "Low power microelectronics: Retrospect and prospect," *Proceedings of IEEE*, vol. 84, no. 4, pp. 619-635, April 1995.

[44] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," *Proceedings of the 21st International Symposium on Microarchitecture*, pp. 60-63, December 1988.

[45] R. Montoye, E. Hokenek, and S. Runyon, "Design of the IBM RISC system/6000 floating-point execution unit," *Proceedings IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59-70, January 1990.

[46] R. Nagarajan, K. Sankaralingam, D. Burger, S. Keckler, "A design space evaluation of GRID processor architectures," *Proceedings of the 34th International Symposium on Microarchitectures*, pp. 40-51, December 2001.

[47] R. Nair and M. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 13-25, 1997.

[48] E. Nystrom and A. Eichenberger, "Effective cluster assignment for modulo scheduling," *Proceeding of the 31st International Symposium on Microarchitecture*, pp. 103-114, December 1998.

[49] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: Architecture and implementations," *IEEE Micro*, vol. 19, no. 2, pp. 37-48, March/April 1999.

[50] S. Palacharla, "Complexity-effective superscalar processors," PhD dissertation, University of Wisconsin-Madison, 1997.

[51] S. Palacharla and J. Smith, "Decoupling integer execution in superscalar processors," *Proceedings of the 28th International Symposium on Microarchitectures*, pp. 285-290, November 1995.

[52] S. Patel and S. Lumetta., "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 300-318, June 2001.

[53] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 51-59, August 1996.

[54] S. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *IEEE Micro*, vol. 20, no. 4, pp. 47-57, July/August 2000.

[55] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 124-135, May 1999.

[56] B. Rau and C. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *Proceeding of the 14th Workshop on Microprogramming*, pp. 183-198, October 1981.

[57] K. Sankaralingam, V. Singh, S. Keckler, and D. Burger, "Routed inter-ALU networks for ILP scalability and performance," *Proceedings of the 21st International Conference on Computer Design*, pp. 170-177, October 2003.

[58] K. Sankaralingam, et al., "Exploiting ILP, TLP, and DLP with polymorphous TRIPS architecture," *Proceeding of the 30th International Symposium on Computer Architecture*, pp. 422-433, June 2003.

[59] P. Sassone and D. Wills, "Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication," *Proceedings of the 37th International Symposium on Microarchitecture*, December 2004.

[60] P. Sassone and D. Wills, "Multicycle broadcast bypass: Too readily overlooked," *Proceedings of the International Workshop on Complexity Effective Design*, June 2004.

[61] Y. Sazeides, S. Vassiliadis, and J. Smith, "The performance potential of data dependence speculation and collapsing," *Proceedings of the 29th International Symposium on Microarchitectures*, pp. 238-247, December 1996.

[62] Semiconductor Industry Association, "The international technology roadmap for semiconductors," 2003, available at http://public.itrs.net, October 2006.

[63] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp.414-425, June 1995.

[64] G. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance, interruptible, multi functional unit pipelined computers," *IEEE Transactions on Computers*, vol. 39, no. 3, pp.349-359, March 1990.

[65] M. Slater, "AMD's K5 designed to outrun Pentium," *Microprocessor Report*, vol. 8, no. 14, October 1994.

[66] N. Sreraman, R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363-400, August 2000.

[67] D. Talla, L. John, and D. Burger, "Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements," *IEEE Transactions on Computers*, vol. 52, no. 8, pp. 1015-1031, August 2003.

[68] M. Taylor, et al., "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 2-14, June 2004.

[69] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: On-chip interconnect for ILP in partitioned architectures," *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pp. 341-353, February 2003.

[70] Tigershark, available at http://www.analog.com/processors/tigersharc/index.html, September 2006.

[71] *TMS320C62x image/video processing library programmer's reference*, Texas Instruments Literature Number SPRU400, March 2000.

[72] Trimedia tm-1300, available at http://www.tm1300.com/trimedia, October, 2006.

[73] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad, "200-MHz superscalar RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1675-1685, November 1996.

[74] S. Vajapeyam and T. Mitral, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 1-12, June 1997.

[75] E. Waingold, et al., "Baring it all to software: Raw machine," *IEEE Transactions on Computers*, vol. 30, no. 9, pp. 86-93, September 1997.

[76] D. Wall, "Limits of instruction-level parallelism," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, April 1991.

[77] S. Weiss and J. Smith, *Inside IBM Power and PowerPC*, Morgan Kaufmann Publishers Inc., San Mateo, CA, 1994.

[78] L. Wills, T. Taha, L. Baumstark, and S. Wills, "Estimating potential parallelism for platform retargeting," *Proceedings of the 9th International Working Conference on Reverse Engineering*, pp. 55-64, October 2002.