

Technology-based Architectural Analysis of Operand Bypass Networks for Efficient Operand Transport

Hongkyu Kim, D. Scott Wills, and Linda M. Wills
Microelectronics Research Center
School of Electrical and Computer Engineering
Georgia Institute of Technology
{hongkim, scott.wills, linda.wills}@ece.gatech.edu

Abstract

As semiconductor feature sizes decrease, interconnect delay is becoming a dominant component of processor cycle times. This creates a critical need to shift microarchitectural design focus from operation computation to operand transport. Operand bypass networks of out-of-order superscalar processors are particularly demanding of wiring resources. Forwarding path delay has become a limiting factor of processor performance. This paper proposes a novel technology-based methodology to evaluate bypass network configurations by predicting operand transport cost. It combines technology modeling techniques with cycle-accurate simulation of benchmark applications to characterize operand movement and storage requirements. Our analysis shows that the operand transport cost heavily depends on the physical location of functional units (FUs) and instruction steering strategy. We propose a traffic-based placement which places FUs based on the transport distribution pattern; and a geometry-driven instruction steering which tries to assign each pair of dependent instructions to adjacent computing resources. Performance is evaluated on an aggressive eight-way, 16 functional unit processor operating at 1.9 GHz in 100 nm technology. Combining these two techniques, the IPC penalties resulting from wire delay latency can be kept within 6.8% of the ideal zero bypass delay processor for Spec2000Int and within 5.5% for MediaBench.

1. Introduction

Historically, computer builders have focused primarily on the design of functional units (FUs) and storage, since these elements consumed the majority of implementation resources (typically transistors). Today, interconnect is becoming the limiting resource in processor implementation [12]. Semiconductor industry projections indicate an increasing disparity between wire delay and gate delay [17]. At the architectural level, the interconnect problem is exacerbated by execution mechanisms (e.g., operand bypassing, register renaming) that enhance parallelism using difficult to scale broadcast buses to

distribute operands. Forwarding path delays are increasing relative to execution unit delay and modern processors already spend as much time in bypassing the result as computing it [6].

To reduce latency associated with operand movement within a datapath, we shift the microarchitectural focus from operation computation to operand transport, addressing the buffering and delivery of operands to FUs that require them. The objective of this paper is to systematically evaluate a range of microarchitectural configurations and new dynamic execution techniques that recognize and exploit operand communication patterns to reduce the latency, storage requirements, and interconnect demand of operand transport. This is strongly coupled to the anticipated VLSI implementation, so accurate technology modeling must be employed.

Toward this goal, a model of data transport and buffering is defined based on technology models. A workflow for predicting the transport cost, based on the operand movement and storage demands, is presented. Technology modeling techniques for architectural evaluation are combined with cycle accurate simulation (using *Simplescalar* [2]) to explore microarchitectural configurations and to quantitatively predict their performance on benchmark applications. Then, the communication patterns in the execution of standard application benchmarks (Spec2000Int and MediaBench [11]) are analyzed to understand the communication needs of a bypass network. Combining the technology-based modeling methodology and the operand characteristics extracted from the analysis, we propose a microarchitectural placing which exploits the transport communication patterns between components (*traffic-based FU placement*) and a new instruction steering strategy using geometry information when assigning dependent instruction pairs to the computing resources (*geometry-driven instruction steering*).

The rest of this paper is organized as follows. Section 2 overviews prior work. Section 3 provides a background summary of technology modeling for architectural analysis and details our methodology for predicting the transport cost using technology and architectural models. The empirical analysis of forwarded operand traffic and

transport patterns in standard benchmarks is presented in Section 4. These empirical results and the technology model-based exploration point to new *traffic-based FU placement* and *geometry-driven instruction steering* strategies, which are described in Section 5. Details of the experimental setup, transport cost results, and performance results of these strategies are given in Section 5. Finally, Section 6 summarizes conclusions.

2. Related Work

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 [18] and Grid Processor Architecture (GPA) [14] propose network-connected tiles of distributed processing elements running new ISAs that expose underlying parallel hardware organization. While RAW implements a static-transport and GPA uses a dynamic-transport, both RAW and GPA perform compile-time optimizations for instruction scheduling. Corporaal [4] propose a *transport-triggered* architecture, called MOVE, which is programmed by explicitly specifying data transports. It directly forwards operands between FUs and reduces the latency of the bypass network by eliminating the associative hardware. Pattern detection techniques have been developed [1] to synthesize new ISAs and all bypassing is done statically. In general, these approaches require extensive compiler support and are not binary-compatible.

An alternative approach, Instruction-Level Distributed Processing (ILDLP) [9], supports distributed processing elements using dynamic transport network and dependence-based dynamic instruction assignment. Though it defines an accumulator-based new ISA, it can realize the binary-compatibility through run-time binary translation. The PEWs (Parallel Execution Windows) approach [8] also uses dependence-based run-time instruction steering but uses versioning for both registers and memory. Both recognize dependency chains among the instructions and reduce the operand communication costs by allocating them to the same execution cluster at run-time.

As shown, the architectural community is responding to the operand transport problem with a variety of execution approaches including new microarchitectures, better compilers, and improved run-time mechanisms. This paper employs existing compiler and ISAs to complement this work. Its primary contribution is a methodology to evaluate the operand transport efficiency of target execution platforms quantitatively, based on the physical distance among FUs and storage requirements, which leads to new FU placement and instruction steering strategies. Our work resembles [5] in placing components guided by traffic profiling among them, but we focus on

the local wires used for operand bypassing instead of global wires. Our work is also related to run-time instruction scheduling [8][15], but extends it by exploiting physical geometry information.

3. 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* [17] 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 integrating a hierarchical set of models that captures key limits (fundamental, material, device, circuit, and system), introduced in [13] (Figure 1). 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.

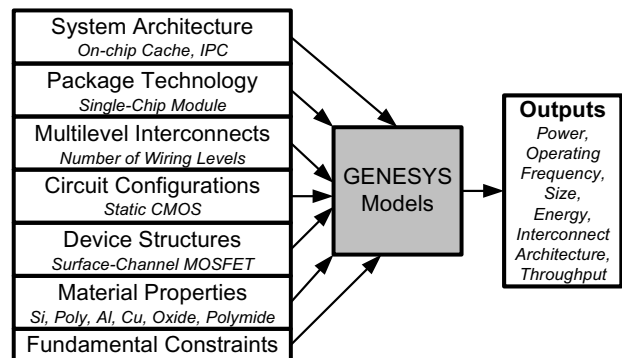


Figure 1. GENESYS system hierarchy.

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 model 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 [3].

Figure 2 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 connection parameters. GENESYS estimates unit speed, area, energy, and transfer latency. It also assembles the units in a user-defined floorplan or it can autoplacement them. The delay builder computes expected delays for operand transport and supplies them to a modified version of SimpleScalar.

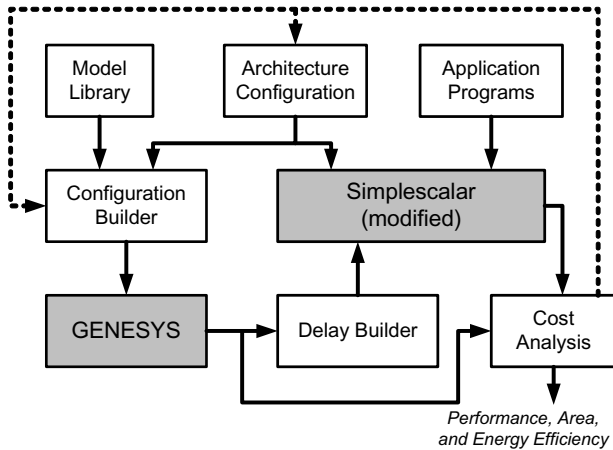


Figure 2. Workflow for System Analysis.

The application suite is then simulated and execution statistics are passed to the cost analysis module, where an interconnect and storage cost model is generated based on the physical distance that operands must travel and the buffer time required before operands are used. As shown in Figure 3, the overall cost of transport is determined by two parameters: transport distance (D_{OP}) and buffer time (T_{BUF}), defined below.

$$D_{OP} = \sum_{operand} \{ \max(P_p, P_{C_1}, \dots, P_{C_n}) - \min(P_p, P_{C_1}, \dots, P_{C_n}) \}$$

where P_p is the physical position of operand producer and P_{C_i} is the physical position of operand consumers

$$T_{BUF} = \sum_{operand} (T_{issue} - T_{written})$$

where T_{issue} = time when the instruction is issued and $T_{written}$ = time when the operand is written to a reservation station

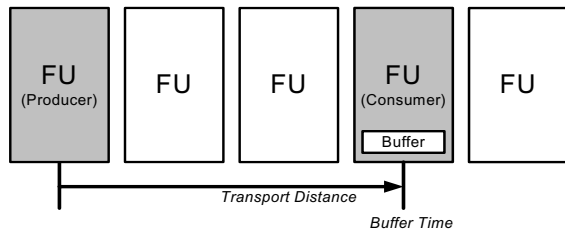


Figure 3. An Operand Transport Model.

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 feedback (shown with dashed lines) to the architecture configuration and configuration builder that can enhance execution.

4. Empirical Analysis of Bypass Traffic

In this section, we explore the operand communication patterns in the execution of standard application benchmarks when the operands are bypassed. Specifically, we focus on the usage of the operand bypass network, operand traffic between components, and finally identifying bottlenecks in conventional broadcasting-style bypassing. The goal is to characterize how operands move and what accounts for the majority of communication needs in executing standard programs. In addition, cost metrics based on interconnection technology models are applied to the resulting operand transport model to determine how to make efficient use of storage and wiring resources when mapping to particular execution platforms.

The operand bypass network in an out-of-order execution ILP processor is examined since this structure limits cycle-time in future processor and the increasing impact of wire delay will be most problematic in future processor designs [15]. The bypass network is responsible for transporting operands from instructions that have completed execution, but have not yet written their results to the register file, to subsequent dependent instructions residing in reservation stations.

To understand the nature of the operand traffic that takes place in a given benchmark program, two kinds of operand characteristics are analyzed: which instructions consume operands after they are produced (temporal locality), and from/to which FU are operands moved during execution (spatial locality). The same architectural configuration parameters are used as Table 1 in Section 5.

Figure 4 shows the prevalence of transient operands in Spec200Int and MediaBench applications. The height of each bar indicates the percentage of dynamic operands which are transported through the bypass network. Since most operands are used a small number of times and they are short-lived [7][10], on average 75% of dynamic operands are transported only through the bypass network without being written to the register file, given a large enough instruction window. (In our analysis, four-entry issue queue per each FU is assumed.). If operands that are forwarded through the bypass network and also written back to the register file are included, 92% of dynamic operands are communicated using the bypass network.

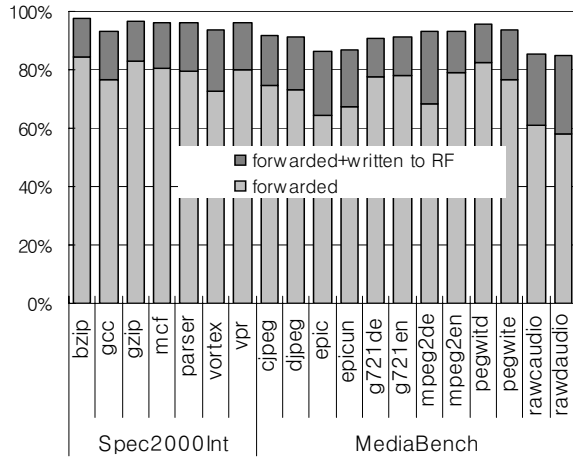
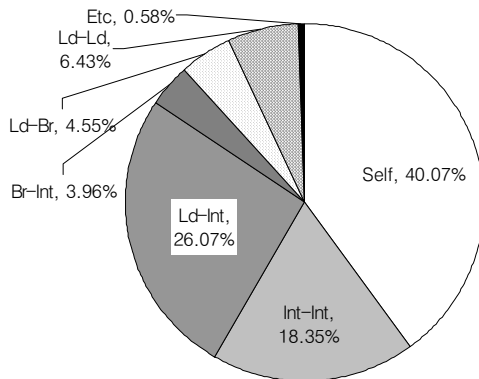
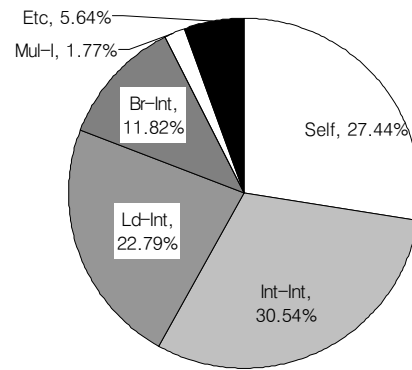


Figure 4. Percentage of dynamic operands which are transported through bypass network.

Figure 5 shows the percentage distribution of the operand transport pattern for benchmark programs: Figure 5 (a) and (b) represent distributions for the average of Spec2000Int benchmark programs and of MediaBench programs, respectively. The results in Figure 5 show that certain paths between FUs are more heavily trafficked than others and two benchmark suites exhibit different operand transport patterns. For example, Spec2000Int programs have more self-bypassing – i.e., the operand consumer instruction is mapped to the same FU as the operand producer instruction – than MediaBench programs. Spec2000Int has a considerable amount of traffic from/to the load/store unit and the branch unit because of its more complex control and memory structures. MediaBench has considerable amount of traffic from/to integer ALUs and integer multiplier since most programs in MediaBench are integer computation intensive and highly regular. The empirical results in Figures 4 and 5 show that most operands tend to have high degree of temporal and spatial locality and reveal a



(a) Spec2000Int



(b) MediaBench

Self: self-bypassing, *Int*: integer ALU, *Ld*: load/store unit, *Br*: branch unit, and *Mul*: integer multiplier

Figure 5. Percentage distribution of dynamic operand transport pattern.

huge potential for alleviating the communication burden by exploiting the locality characteristics.

To identify bottlenecks in the bypass network, the operand transport cost of a conventional bypass network which is implemented by broadcasting result buses is measured. The *baseline* cost and performance, which is the cost/performance when the operating frequency is low enough to compute the result and forward it at the same cycle, is given in the *ref* model column of Table 2 in Section 5. Figure 6 represents the sensitivity of the buffer time cost and performance in IPC. The x-axis denotes the operating frequency which varies from the gate delay plus the longest wire delay to the gate delay only.

From the data in Figure 6, we can see that the forwarding path delay consumes extra cycles, which result in extra buffer time and an IPC drop as the operating frequency increases. For example, the buffer times at 1.9 GHz are 3.67 and 3.47 times longer than those at the baseline frequency in Spec2000Int and MediaBench, respectively. These buffer time increases result in IPC drops to 30% of the IPC at the baseline frequency. The next section provides mechanisms that can minimize the transport cost and gives results of experimenting with them.

5. Impact of Architectural Configurations on Operand Transport

The results of Section 4 reveal that the traditional broadcasting-style bypassing does not scale since the wire delay latency increases as the issue width and the number of FUs increases. Therefore we will focus on point-to-point fully-connected bypass networks instead of broadcasting result bus. Though a point-to-point network consumes greater wiring resources than a broadcast bus, it provides better operand transport performance and greater communication capacity. Next, we introduce two techniques to minimize the operand transport cost: FU

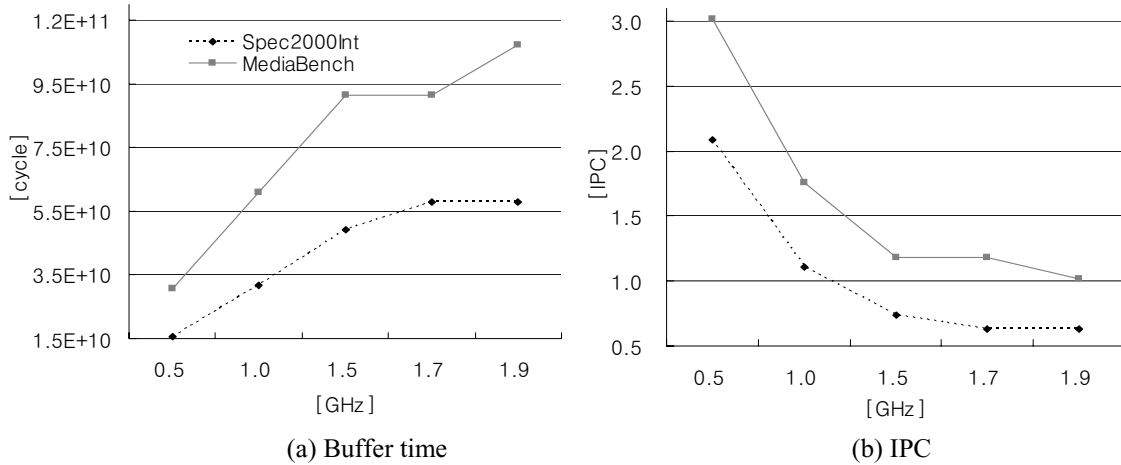


Figure 6. Sensitivity of operand transport cost and performance to operation frequency.

placement based on transport pattern distribution; and instruction steering that exploits the dependency between the instructions and their physical placement.

Traffic-based FU placement

As shown in Figure 5, a cycle-accurate simulation is performed to measure the amount of traffic between FUs for given benchmark programs. Then, the area of each FU is estimated using GENESYS, based on an R10000 processor model [19]. It is assumed that the microarchitecture implementation has: i) the FU heights reported in [16], ii) a two dimensional layout geometry, and iii) two subblocks, one for integer FUs and the other for floating point FUs. After the traffic and area information is collected, each FU is assigned in the subblock and the location of all FUs is determined at the current iteration. The cost of each placement is calculated by the following equation:

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

In this equation, E denotes the set undirected edges, where (i,j) represents a edge between FU_i and FU_j . The parameter α_{ij} is the statistical traffic rate on edge (i,j) (the summation of α_{ij} is equal to one). Finally, (x_i, y_i) denotes the location of the center of FU_i in 2-D space. Thus, the cost is defined as the traffic-weighted sum of edge between FUs. All permutations of FU sequences are explored, from which the minimum cost (*min-place*) and the maximum cost placements (*max-place*) for the benchmark program are determined.

Geometry-driven instruction steering

To minimize the wire delay latency caused by forwarding an operand, the length of the bypass path should be kept as short as possible. Traffic-based FU placement reduces the average transport length by shortening heavily-trafficked paths. After the physical

placement is fixed, it can be further reduced by assigning each pair of dependent instructions to the nearest pair of FUs. We are interested in the dynamic steering and assume a decentralized dispatch window in which each FU has its own dispatch queue. The main function of steering is to assign an instruction to a FU instruction queue. If multiple homogeneous FUs are used, the steering logic should select one of the FUs. The steering of instructions is performed at run-time during the dispatch stage. Two instruction steering methods are investigated as described in Figure 7.

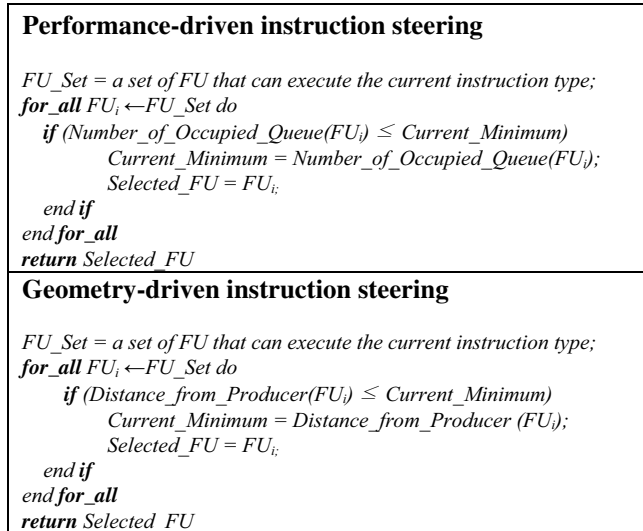


Figure 7. Instruction steering algorithms in dispatch.

The idea behind *performance-driven* instruction steering is to reduce instruction dispatch stalls which occur when a queue is full. It ensures that an instruction can be scheduled to the queue which has the minimum number of active entries and guarantees an even load balance among the queues. Thus, it decreases the

possibility of dispatch stall and tends to reduce the wait time from instruction dispatch to issue.

Unlike *performance-driven* steering, *geometry-driven* steering exploits the physical placement information to map an instruction to a particular queue. Dependence information between instructions is maintained in a table which is indexed by register name and identifies the queue which contains the producer instruction. After the producer is identified, the steering logic tries to map the consumer instruction to an appropriate FU that is nearest to the producer. If an instruction has two register sources, the more recently generated value is used to steer it. Though the *geometry-driven* method may incur more dispatch stall because of its biased utilization of queues, it can shorten the length of the bypass path and keep the forwarding latency low as issue widths and the number of execution resources increases.

Experimental Setup and Results

Benchmarks from Spec2000Int and MediaBench are simulated using SimpleScalar cycle-accurate simulator [2] with the PISA instruction set. The default MediaBench inputs were enlarged to lengthen their execution. For each simulation, we execute 500 million committed instructions, after skipping the first 100 million instructions which is initialization code that is common to all the benchmarks. Table 1 enumerates the parameters common to all configurations evaluated in this section, and summarizes the results of the delay and area calculations from GENESYS. Provisions are made to focus only on the wire delay latency during operand forwarding. The simulator assumes ideal cache access and branch prediction since cache miss and misprediction penalties overlap with and hide wire delay latency, making it difficult to accurately measure the wire delay.

Table 1. Common parameters and GENESYS Result.

Architectural configuration parameters	
Fetch/Decode/Commit Width	8
Total Number of FU	16
Int/Mul/Ld/Br/FpALU/FpMUL	5/1/3/3/1
Issue queue (per FU)	4 entries
Load/store queue	16 entries
I-cache and D-cache	Ideal
Branch predictor	Ideal
Technology configuration parameters	
Feature size [nm]	100
FU width [μm]	339
Performance and area results (GENESYS)	
Total execution engine area [mm^2]	1.369
Execution gate delay [ns]	0.5129

To evaluate the impact on the operand transport cost across different architectural configurations, the two instruction steering algorithms (*performance-driven* and *geometry-driven*) and two physical FU placements (*min-place* and *max-place*) are studied. The detailed configurations are listed in Table 2. The *ref* column denotes the reference model which is equipped with the broadcasting result bus used by conventional processors. The other models (*cfg1* to *cfg4*) use a point-to-point bypass network and use the four different combinations of placement and steering strategies. The table also presents the transport cost and performance at the baseline operating frequency. The ratio means the ratio of values to those of the reference model. The values are averages over all Spec2000Int and MediaBench programs respectively.

As expected, across all configurations, the point-to-point bypass network exhibits lower transport distance than the reference model. For example, the operands in *cfg1* travels only 17% of *ref*. Note that the distance further decreases if *min-place* and/or *geometry-driven* steering are adopted. The buffer time and IPC at the baseline frequency are only a function of the instruction steering algorithm since the operand bypassing is done concurrent with execution. But these values change as the operating frequency increases and an extra bypass cycle is imposed. As expected, the *performance-driven* steering shows less buffer time and better performance than *geometry-driven* because of fewer dispatch stalls and shorter wait time. But the performance degradation of *geometry-driven* steering due to the longer buffer time is negligibly small – on average, it is within 2% of the *performance-driven* steering.

Figure 8 shows the buffer time sensitivity when the operating frequency increases and extra cycles are required to forward operands. In general, the buffer time increases as the frequency become high due to the extra bypass cycle. As the results show, *min-place* FU placement and *geometry-driven* instruction steering do not actually incur a significant buffer time increase compared to *max-place* and *performance-driven* steering, even at very high frequency. For instance, if FU placement is only taken into account, the buffer time of *cfg2* at 1.9 GHz is 85% of that of *cfg1* in Spec2000Int. For the most optimal configuration (*cfg4*), it remains 78% of *cfg1*. Interestingly, the buffer times with *performance-driven* (*cfg1* and *cfg2*) are initially less than those with *geometry-driven* (*cfg3* and *cfg4*), but they cross near the middle of the graph, implying that *geometry-driven* steering can better maintain low bypassing latency.

The buffer time increases directly translate to IPC drop as shown in Figure 9. The IPC drops of the *ref* model at 1.9 GHz are 70% and 67% of the IPC at the baseline frequency in Spec2000Int and MediaBench, respectively (refer to Figure 6 (b)). By only using the

point-to-point bypass network, we can reduce the drops to 25% and 18%. Furthermore, when the *traffic-based* FU placement and *geometry-driven* steering are applied, the IPC drops can be kept within 6.8% and 5.5% of the IPC at the baseline frequency until the maximum operating frequency. As shown, the advantage of placing FUs based on the traffic profiling information between functional units and mapping dependent instructions as near as possible stands out when the clock speed increases. As technology feature size decreases and integration increases, our approach can maintain better scalability of operand transport performance.

6. Conclusion

Operand transport demand is becoming a limiting factor in improving the performance of modern

microprocessors due to wire delays. To minimize this latency, operand transport should be optimized by reducing the distance that operands travel between FUs.

Toward this end, a workflow for computing the total operand transport cost, including operand movement and buffering is proposed. We analyzed the impact of the architectural configurations, including FU geometry, instruction steering, and bypass network topology, on the efficiency of operand transport of bypass network. We find that the physical location of FUs has a significant impact on operand transport latency. The total transport distances of the best placement (placing each FU to their optimal position based on the operand transport distribution pattern) are 59.2% of the worst placement in Spec2000Int and 60.0% in MediaBench. Instruction steering strategy also affects the transport and a new

Table 2. Microarchitecture configurations and experimental results.

	ref	cfg 1	cfg 2	cfg 3	cfg 4
Bypass Network	Broadcasting	Point-to-Point			
FU placement	<i>Min-place</i>	<i>Max-place</i>	<i>Min-place</i>	<i>Max-place</i>	<i>Min-place</i>
Instruction steering	<i>Performance-driven</i>	<i>Performance-driven</i>	<i>Performance-driven</i>	<i>Geometry-driven</i>	<i>Geometry-driven</i>
D_{OP} [mm]*	1,225,273	197,719	117,188	102,119	67,832
	1,277,673	218,813	131,255	137,562	87,949
D_{OP} Ratio*	1.000	0.161	0.096	0.083	0.055
	1.000	0.171	0.103	0.108	0.069
T_{BUF_BASE} [cycle]*	15,788 M	15,788 M	15,788 M	16,014 M	16,000 M
	30,864 M	30,864 M	30,864 M	32,089 M	31,859 M
T_{BUF_BASE} Ratio*	1.000	1.000	1.000	1.014	1.013
	1.000	1.000	1.000	1.040	1.032
IPC_{BASE} *	2.086	2.086	2.086	2.068	2.068
	3.018	3.018	3.018	2.961	2.978
IPC_{BASE} Ratio*	1.000	1.000	1.000	0.991	0.991
	1.000	1.000	1.000	0.981	0.987

* Data in 1st row represent Spec2000Int and 2nd row MediaBench

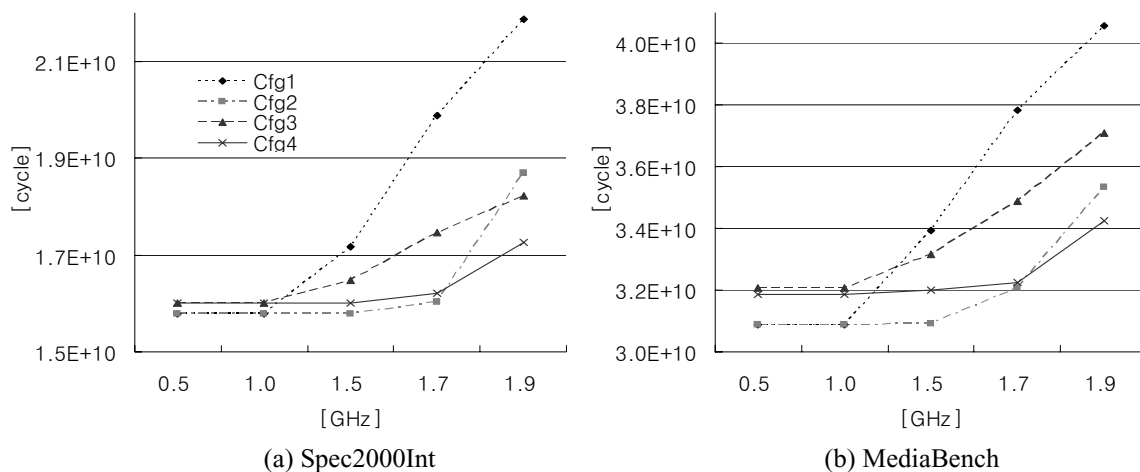


Figure 8. Sensitivity of buffer time to operation frequency.

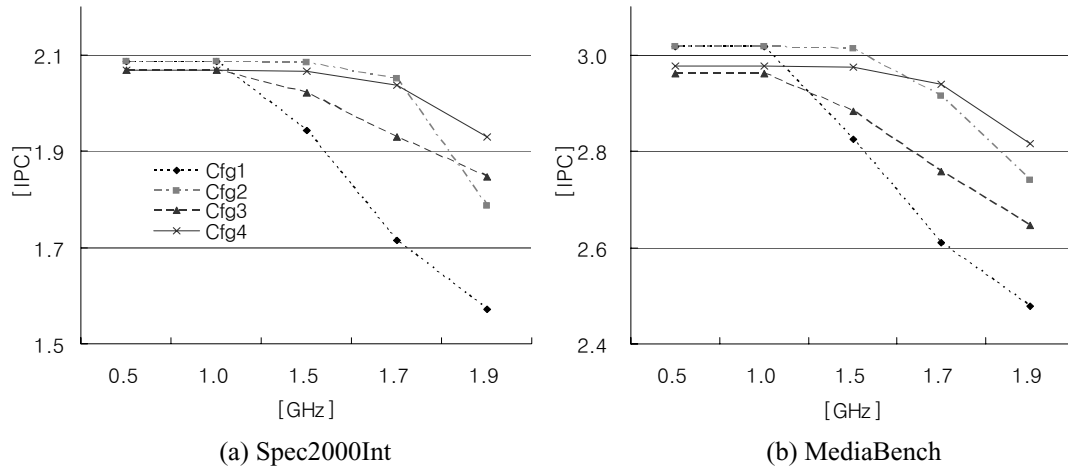


Figure 9. Sensitivity of performance to operation frequency.

steering algorithm which maps pairs of dependent instructions to the nearest FUs (geometry-driven steering) is proposed. The geometry-driven steering can further reduce the transport distance by 34.3% of conventional performance-driven steering in Spec2000Int and 40.2% in MediaBench. As a result, the proposed approach of combining two techniques (traffic-based placement and geometry-driven steering) with point-to-point bypass networks keeps the IPC penalty at 1.9 GHz less than 6.8% of IPC at baseline frequency at 100 nm feature size.

References

- [1] M. Arnold and H. Corporaal, "Automatic detection of recurring operation patterns," *7th Intl. Workshop on Hardware/Software Co-Design*, 1999.
- [2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, pp. 59-67, 2002.
- [3] L. Codrescu, et al., "Modeling technology impact on cluster microprocessor performance," *IEEE Transactions on VLSI systems*, 11(5), pp. 909-920, October, 2003.
- [4] H. Corporaal, "TTAs: Missing the ILP complexity wall," *Journal of System Architectures*, vol. 45, no. 12, pp. 949-973, 1999.
- [5] M. Ekpanyapong, et al., "Profile-guided microarchitectural floorplanning for deep submicron processor design," *Proceedings of Design Automation Conference 2004*.
- [6] E. Fetzer and J. Orton, "A fully bypassed 6-issue integer datapath and register file on an Itanium-2 microprocessor," *Proceedings of Intl. Solid State Circuits Conference*, 2002.
- [7] M. Franklin and G. Sohi, "Register traffic analysis for streaming inter-operation communication in fine-grain parallel processors," *Proceedings of Intl. Symp. on Microarchitecture*, 1992, pp. 236-245.
- [8] G. Kemp and M. Franklin, "PEWs: A decentralized dynamic scheduler for ILP processing," *Proceedings of Intl. Conference on Parallel Processing* 1996.
- [9] H. Kim and J. Smith, "An Instruction Set and Micro-architecture for Instruction Level Distributed Processing," *Proceedings of Intl. Symp. on Computer Architecture* 2002.
- [10] H. Kim, S. Wills, and L. Wills, "Empirical analysis of operand usage and transport in multimedia applications," *Proceedings of Intl. Workshop on System-on-Chip for Real-Time Applications*, July 2004, pp. 168-171.
- [11] C. Lee, et al., "Mediabench: a Tool for evaluating multimedia and communications systems," *Proceedings of the 30th Annual Intl. Symp. on Microarchitecture*, December 1997, pp. 40-51.
- [12] J.D. Meindl, "Interconnect opportunities for gigascale integration," *IEEE Micro*, 23:(3), pp. 28-35, May/June 2003.
- [13] J.D. Meindl, "Low power microelectronics: Retrospect and prospect," *Proceedings of IEEE*, 84:(4), pp. 619-635, April, 1995.
- [14] R. Nagarajan, et al, "A design space evaluation of Grid Processor Architectures," *Proceedings of Intl.Symp. on Microarchitecture*, December 2001, pp. 40-51.
- [15] S. Palacharla, et al., "Complexity-effective superscalar processors," *Proceedings of Intl. Symp. on Computer Architecture*, 1997, pp. 206-218.
- [16] S. Palacharla, et al., "Quantifying the complexity of superscalar processors," Technical Report CS-TR-96-1328, University of Wisconsin-Madison, November 1996.
- [17] Semiconductor Industry Association, The International Technology Roadmap for Semi-conductors, 2003, <http://public.itrs.net>.
- [18] M. Taylor, et al., "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," *Proceedings of Intl. Symp. on Computer Architecture* 2004.
- [19] N. Vasseghi, et al., "200-MHz superscalar RISC microprocessor," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 11, November 1996, pp. 1675-1685.