Reducing Operand Communication Overhead using Instruction Clustering for Multimedia Applications

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

Abstract

As technology trends yield shorter cycle times and larger, wider datapaths in architectures for multimedia systems, global broadcast networks for operand communication are becoming a major bottleneck in processor performance. New low-latency operand transport techniques are needed. This paper proposes and evaluates lower cost mechanisms than traditional bypass networks, exploiting regular operand distribution patterns in multimedia applications. To reduce latency associated with operand movement within a datapath, our mechanism, called dynamic instruction clustering, groups chains of dependent instructions within a basic block at run-time, identifies intermediate value transportation, and schedules it on networked ALUs which are connected by a local dedicated network. By converting global communication into local, the transport latency can be minimized and the critical path of the application code can be executed in consecutive, shortened cycles, resulting in improved performance. We demonstrated that 28% and 30% of total dependence edges residing in the instruction window can be localized on 8- and 16-way machines, respectively. Our results show that the overall performance gains over a wide range of multimedia applications are 16% for 8-way and 35% for 16-way on average.

1. Introduction

Though many multimedia applications contain significant inherent parallelism (both data-level (DLP) and instruction-level (ILP)), general-purpose processors are nearing the limit of what can be achieved with instruction parallel mechanisms such as superscalar execution [25][26]. To fully exploit inherent parallelism in multimedia applications, new processor designs require wider instruction windows and more functional units (FUs) connected by a complex operand transport network. Unfortunately, this requires execution mechanisms (e.g., operand by-passing, register renaming, and instruction scheduling) that employ poorly scaling broadcast buses to distribute operands. The area, delay [17], and power required by these busses are becoming limiting factors in high performance processor designs.

To reduce latency associated with operand movement within a datapath, especially for multimedia applications where the movement is highly regular, we shift the microarchitectural focus from FUs to operand transport, which addresses the buffering and delivery of operands to FUs that require them. In this paper, we propose and evaluate a dynamic execution mechanism that exploits regular operand distribution patterns in multimedia applications to reduce the latency. We also develop a lower cost operand transport network than traditional bypass networks which converts the global communication to local transport, exposing and exploiting opportunities to lower latency.

In previous work [10], we have empirically analyzed the operand communication patterns between dependent instructions that occur in the execution of standard multimedia application benchmarks from MediaBench [12] with the goal of understanding operand movement so that it can be efficiently controlled. Based on this, we have developed dynamic microarchitectural mechanisms, which we present in this paper, that efficiently detect and streamline repeated operand transport patterns. Specifically, we propose reducing operand traffic by dynamically grouping data-dependent instructions into clusters and mapping the clustered instructions to an efficient cluster execution unit in which intermediate values are propagated among ALUs without distribution through global bypass busses. An additional queue, which stores clusters to be issued, provides the clustered instructions to the execution unit based on the

Proceedings of the Seventh IEEE International Symposium on Multimedia (ISM'05)
0-7695-2489-3/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

dependence information collected during previous execution. The latency reduction, resulting from shortening the transport distance, can directly translate to performance improvement for wide-issue machines.

Our technique focuses on the intermediate values which are produced and only consumed within the same basic block since our empirical analysis reveals a large number of such short-lived, or "transient," operands within the inner loops of most multimedia applications, which typically span tens and sometimes hundreds of instructions. The typical producer-consumer relationship is deterministic (non-speculative), so their communication can be localized within the basic blocks. Our analysis also shows that a relatively small number of clusters are needed to cover a typical basic block, which simplifies the clustering hardware resources (e.g., cluster cache size and detection logic) required.

Our approach complements the large body of previous research on improving performance of multimedia applications that focuses on detecting *independent* computations that can be performed in parallel (e.g., multimedia instruction set extensions [13][19] have been developed to harness the data parallelism inherent in these applications; and sophisticated compiler and retargeting techniques [2][21] are emerging to automatically parallelize these computations). To complement this work, we are focusing on improving the performance of regular patterns of *dependent* instructions, which are inherently sequential, by optimizing the delivery of short lived "transient" operands connecting these instructions.

The rest of this paper is organized as follows. Section 2 overviews prior work. Section 3 introduces the concept of instruction clustering and summarizes our empirical results. The overall microarchitectural support for our mechanism is described in Section 4. Section 5 quantifies the bypass latency in wide issue processors. Section 6 details the experimental setup and results of our mechanism on wide-issue superscalar out-of-order processors. Finally, Section 7 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 [22] and Grid Processor Architecture (GPA) [15] 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 perform compile-time optimizations for instruction scheduling and localize the communication through dedicated forwarding paths between producer and consumers. Corporaal [5] proposes a transport-triggered architecture, called MOVE, which is programmed by explicitly specifying data transports. It directly forwards operands between FUs and reduces the bypass latency by eliminating the associative hardware. Through synthesized new ISAs, all bypassing is done statically. In general, these static approaches require extensive compiler support and are not binary-compatible.

The most commonly suggested method of communication-aware execution is clustering or resource partitioning. This technique is implemented commercially on the Alpha 21264 processors, which have two identical pipelines with distinct register files, bypass networks, and issue logic [9]. A key issue to reduce the operand transport complexity in clustered mechanisms is to assign operations to clusters – instruction steering to minimize the inter-cluster communication. Implementations with more clever steering techniques can be found in academic research such as Multicluster [6], Palacharla et al. [17], Parallel Execution Windows (PEWs) [8], and Clustered Trace Cache Processor (CTCP) [4]. Bunchua and Wills [3] proposed a fully distributed register file where broadcast transport is replaced by explicit local bypass on-demand at the cost of longer inter-ALU latency. 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 problem as issue width and cluster count increase. Our technique achieves a similar effect as clustering without increasing global communication latency by moving the steering burden off the critical path.

Recently, many researchers have proposed using the trace cache fill unit for dynamic optimizations [16][24]. They used the fill unit to dynamically retarget a scalar instruction stream into pre-scheduled instruction groups to minimize the impact of latency through the operand transport network with a clustered backend. RePLay [18] forms hyperblock regions (called frames) in a similar fashion and executes aggressive code optimization at the micro-operation granularity. While previous research targets general applications, our empirical analyses show that similar benefits can be achieved with our simpler techniques which exploit characteristics of multimedia applications. In particular, the inner loops in most multimedia application span tens or even hundreds of

IEEE COMPUTER SOCIETY

instructions (e.g., the DCT routine in MediaBench's JPEG encoder contains 151 RISC-type instructions) and a large fraction of operands are short-lived and local to these basic blocks. Furthermore, a small number of clusters are able to cover the dependent instructions within these basic blocks, reducing the need for complex implementations. The mechanism we propose is far less complex since we are able to focus only on checking the true dependences between instructions within the same block, and grouping and mapping them to be executed on a local inter-ALU network.

Sassone's Dynamic Strands [20] similarly group dependence chains of at most three integer ALU instructions joined by transient, with no fan out, and steer them to the self-bypassing ALUs. For multimedia applications, we propose forming larger and more general clusters of instructions (e.g., by lifting the restriction on fanout) to accommodate a broader class of regular operand distribution patterns that are inherent in these programs.

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 for binary compatibility, complementing previous work.

3. Instruction Clustering Concept

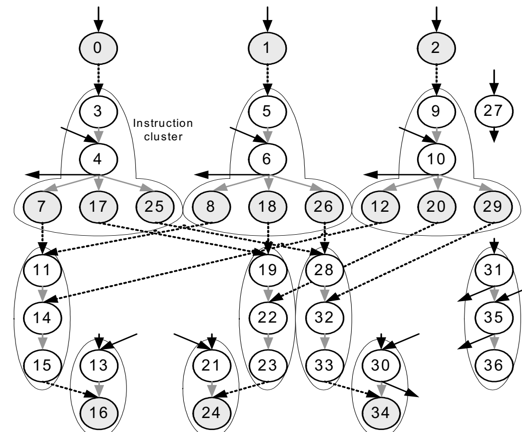
Empirical analysis [10] of operand usage and communication properties for MediaBench programs has revealed that operands tend to be used only a small number of times (over 80% of all operands are used at most three times; in fact, 42% have only one consumer), are usually consumed shortly after they are produced (on average 80% of operands are consumed within five dynamic instructions) and have short lifetimes (about 70% 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 and contribute greatly to traffic congestion among the FUs and broadcast buses. *Local operands*, which are values produced within a basic 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 1 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 (a gray nodes denote memory instructions, e.g., load or store, and a white nodes denote ALU instructions) and each edge represents a true data dependence.

Dependence edges are classified according to the producer-consumer relationship: i) *external* (solid line): an operand which is produced by previous basic blocks or may be consumed by subsequent basic blocks; ii) *internal* (dotted line): an operand which is produced by a load as data read, consumed by a store instructions as data to be written, or produced/consumed by a floating-point instruction in the current block; and iii) *local* (gray line): an operand which is produced and consumed within the current block by instructions that perform ALU computations, which include integer ALU instructions, branch instructions, and effective address calculations for memory instructions. Note that some operands are local, internal, 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.

```
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) Assembly source code



(b) Dataflow graph and instruction clustering

Figure 1: Instruction clustering example based on data flow graph of a basic block from JPEG encode.

0-7695-2489-3/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

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 internal or external dependence edges.

To determine the scope for cluster formation, we characterize the type of dependence edge between instructions in the instruction window. Figure 2 shows the prevalence of dependence edges with a 64-entry instruction window during dynamic execution of MediaBench application programs. It also presents the distribution of dependence edge types. In the graph, each bar denotes the average number of edges into nodes (instructions) in the instruction window, which gives a measure of the total amount of operand traffic currently passing through the expensive global bypass mechanism.

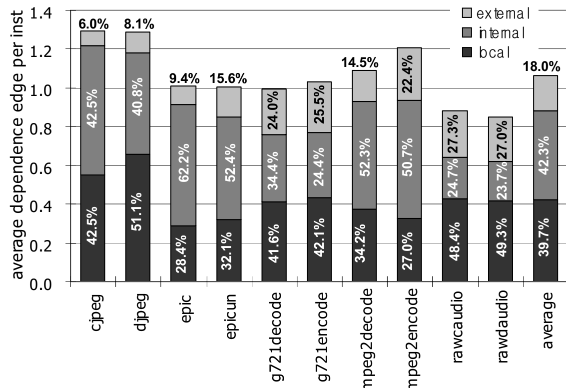


Figure 2. Average number of dependence edge per instruction which were broken down by edge type.

On average, about 40% of dependence edges are local, showing a large potential for exploitation by converting them to more efficient local transport. Interestingly, about 42% of edges are internal, mainly due to memory instructions – effective address transport from ALU to load/store queue, and data to be stored to memory or loaded from memory. The other 18% of edges pass through the control boundary and we preclude them since they can be incorrect if instructions are grouped across a mispredicted branch.

4. Instruction Clustering Logic

Instruction clustering benefits from shortening dependence edge latency by grouping dependent instructions in a basic block and scheduling them to the special hardware in which local operands are forwarded through dedicated bypass paths instead of broadcasting. Figure 3 shows the basic organization of our clustering mechanism and its corresponding pipeline stages. This section describes the details of its three major components: cluster formation and cache; cluster scheduling and queue; and cluster execution.

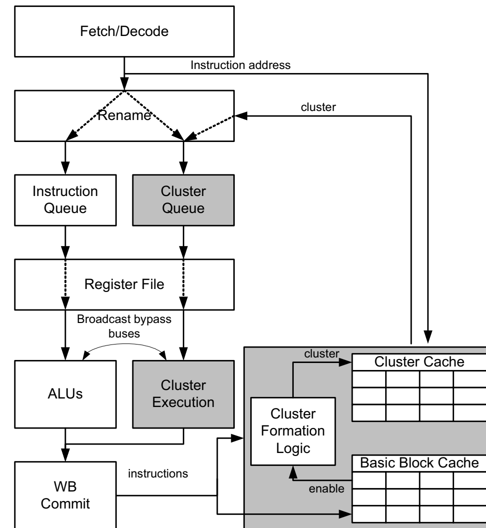
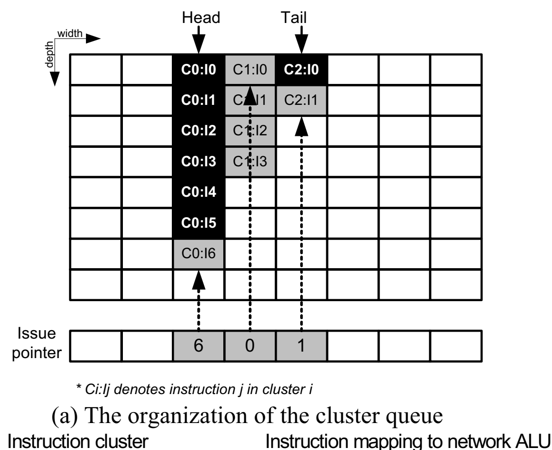


Figure 3. Basic organization of clustering mechanism and its pipeline stages.

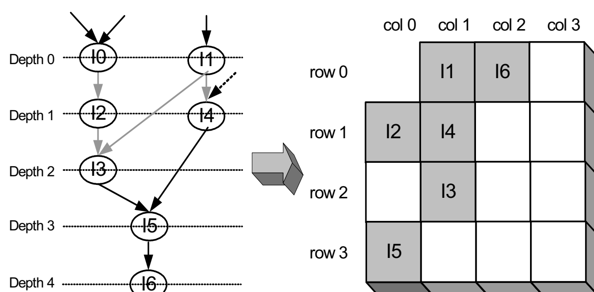
Cluster formation unit: groups instructions that are connected by true data dependences into instruction clusters. It also classifies the dependence edges as local, internal, or external, based on the data dependence information in the basic block. By default, all output ports of non-memory instructions are initially considered to be sources of external edges. However, these external edges are removed if the associated register is overwritten by subsequent instructions within the same basic block (indicating that the operand's lifetime is limited and does not need to be passed out to the broadcast buses). After a cluster is formed, each instruction is assigned a dependence depth which is the number of instructions in the longest dependence chain from the input fringe of the cluster to the input of the instruction. This 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 exhibit a high degree of code locality. Each cluster cache entry holds instruction addresses, source/destination operand locality bits classifying dependence edges, and dependence depth information of each instruction in the cluster.

0-7695-2489-3/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

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 4a presents the concept of the cluster queue. The gray and black boxes indicate the occupied queue. The contents of the cluster queue is similar to the conventional instruction queue except for the following: i) multiple dependent instructions (a cluster) reside within a single entry, shown as a column in Figure 4a, ii) the ready flags of local operands are always 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. In Figure 4a, the black boxes indicate issued instructions and the gray boxes to-be-issued instructions.



(a) The organization of the cluster queue



(b) Instruction issuing and mapping for cluster execution

Figure 4. The function of the cluster queue and cluster scheduling logic.

Each instruction in the cluster, once source operands are ready, is issued to one of the networked ALUs in the cluster execution unit, which are equipped with dedicated forwarding paths between consecutive rows. Figure 4b illustrates how the clustered instructions (cluster 0 in Figure 4a) are mapped to the networked ALUs. The depth bits are used to determine the row of the ALU, guaranteeing that the dependent instructions are located back-to-back, taking the shortest path to transport the operand. The depth wraps around if it is greater than or equal to the number of rows (e.g., I6 is steered to the row 0).

Cluster execution unit: The networked ALUs are the core of the cluster execution unit. As shown in Figure 5, it consists of a set of ALUs and wires connecting them. Parts of traditional ALUs are converted to the network ones which handle the clustered instructions (we use four, which is the maximum limit of fully-connected bypass without additional penalties). The operands can be transported through three paths: i) a local path when dependent instructions are mapped to consecutive rows of the networked ALUs, ii) an input/pass-through path 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, I1 to I3 in Figure 4b), and iii) an output path to transport internal /external operands to the other parts of the processor. Input/output ports of the execution unit are connected to the broadcast buses.

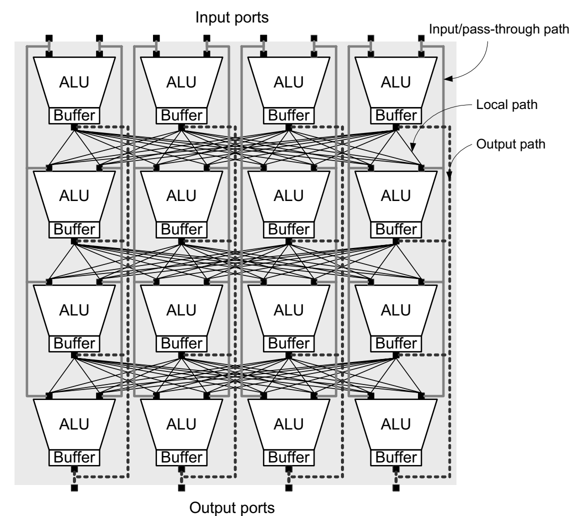


Figure 5. The cluster execution unit hardware.

The operands which are communicated through a local path can be transported without additional bypass cycle latency. Though the number of ALUs increases, the latency of broadcast bypass remains constant since the number of inputs and outputs of the FUs does not change.

5. Analysis of Bypass Delay

0-7695-2489-3/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

In most modern processors, the forwarding path is implemented as a fully-connected broadcast bypass – all FU outputs are connected to all FU inputs – and this is becoming the most critical bottleneck in pipelined processors. The complexity and delay of bypass paths is increasing with issue width and technologies. According to the Palacharla's model, the bypass delay grows quadratically with the number of FUs if 2-inputs per FU are assumed [17].

We computed the bypass wire delay for hypothetical machines using GENESYS [14], which is an analytical modeling tool that integrates a hierarchical set of models: fundamental, material, device, circuit, and system. GENESYS 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 specific technology.

The delays computed by GENESYS are presented in Figure 6. The delays are calculated at 100 nm technology. The area of ALU is estimated based on the R10000 processor model [23]. It is assumed that the micro-architecture implementation has the ALU heights reported in [11] and a two-dimensional layout geometry. Though the results from GENESYS are less accurate than circuit simulators, such as HSPICE, the gate delay and inter-connect delay can be approximated based on the architectural configuration. The results in Figure 6 shows that bypass delay indeed scales quadratically with the issue width and it is more and more dominant than ALU gate delay as the issue width grows. For example, though the bypass delay is negligible at four-wide FU, it becomes comparable to the gate delay at eight-wide and explodes beyond eight. Thus, the bypass can no longer fit in a single execution cycle, requiring pipelined bypassing.

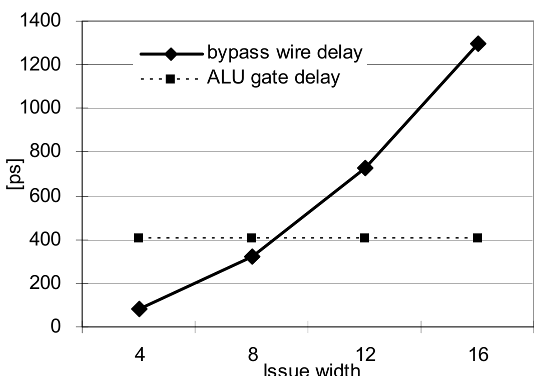


Figure 6. ALU and bypass wire delays with various issue width at 100nm technology.

6. Experimental Results

To measure the effectiveness of our instruction clustering and local bypassing, the cycle-accurate simulator derived from the SimpleScalar 3.0 tool set [1] with PISA instruction set was extended to implement our hardware and algorithm. Benchmarks from MediaBench are used for all results presented in this paper. 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.

We modeled an out-of-order pipelined machine with 8-and 16-instruction width because the wider the width becomes, the longer the bypass latency. Table 1 shows the detailed configurations of each machine model. The global bypass latencies are determined based on the results in Section 5. The baseline models of each machine can be configured by changing the size of instruction queue to the instruction queue plus cluster queue, and replacing each network ALU to four conventional ALUs.

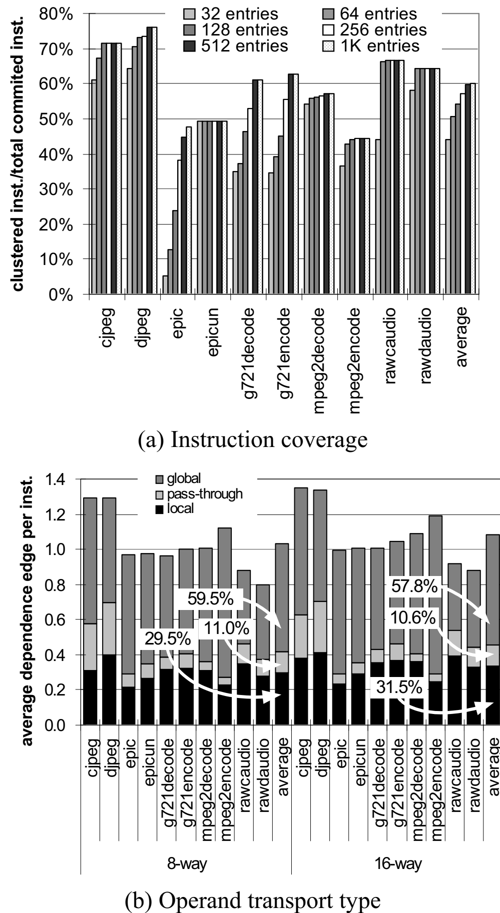
Table 1. Processor model configurations.

	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
FU 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 bypass (latency)	Local (0), pass-through (1), Global (1)	Local (0), pass-through (1), Global (3)
Memory system	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	

Figure 7a shows the percentage of dynamic instructions that were grouped by clusters with various cluster cache sizes. The more instructions are executed in the cluster execution unit, the more operands can be potentially transported through the shortest path. 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 7b. 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 ALU. The rest of this section assumes a 256-entry cache.

Figure 7b shows the average number of dynamic dependence edges (per instruction) converted to local transport. The data in this graph is plotted in the same way as Figure 2, except for types of edges: data in the graph represent actual transport types. The local transport edge and pass-through edge represent dependence edges which are clustered and transported through the local and pass-through pass in the cluster execution unit, respectively. The global edge means an operand transported through the global bypass network.



(a) Instruction coverage

(b) Operand transport type

Figure 7. Percentage of dynamic instructions covered by the instruction clustering and operands transported by local bypass network.

Note that for 16-way machine, the average numbers of dependence edges are slightly higher than those of Figure 2 because the instruction windows became virtually wider due to the cluster queue. The numbers of dependence edges transported within the cluster execution unit (local transport edges plus pass-through edges) are also higher than the number of local edges in Figure 2. Actually, some inter-cluster operands, which are classified as external in Figure 2, can be transported within the network ALU when multiple clusters are being executed. On average, about 30% for 8-way and 32% for 16-way of total dependence edges can be mapped onto the local path which has the shortest latency. The pass-through edges occur due to the program structure (as shown in Figure 5b) or instruction mapping failure, caused by the limited number of ALUs at the target row. For example, high instruction coverage and dependence edges in *cjpeg* and *djpeg* cause a considerable amount of pass-through transport as shown in the graph.

Figure 8 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 ALU immediately, the dependent instructions on the critical path can be executed in the consecutive cycle. The average speedups are about 16% and 35% for 8-way and 16-way machine configuration respectively. As expected, the speedups of 16-way machine are higher than those of 8-way machine since the former suffers from longer global bypass latency and has more local transport. Several programs, such as *cjpeg* and *djpeg*, show significant speedup since many of their dependence edges convert to local ones.

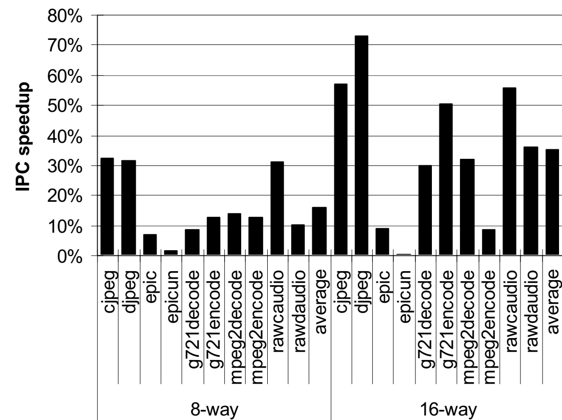


Figure 8. Instruction clustering performance result under 256-entry cluster cache.

Interestingly, a few programs, such as *epic*, *epicun*, and *mpeg2encode* show little speedup. In the former two cases, the bypass latency has little effect on the performance. Even with ideal bypassing (zero latency), the speedups are only 15~17% for the 8-way case. In the third case, it has a small number of local transport edges compared to the total dependence edges as shown in Figure 8b. Note that in the superscalar

IEEE COMPUTER SOCIETY

execution several run-time conditions can contribute to slack in the normal execution pipeline, such as mispredicted branches and cache misses. In our clustered execution, the limited number of input/output ports also contributes. This slack can hide the effect of our mechanism and reduces the amount of speedup our mechanism achieves.

7. Conclusion

Operand transport complexity through broadcast bypass networks is becoming a limiting factor in improving the performance of modern processors due to wire delays. The operand transport should be optimized to minimize this latency. Towards this end, we developed a technique for detecting regular data dependence patterns that are prevalent in multimedia applications and that can be executed efficiently as clusters whose instruction dependencies are converted to local communication. In particular, we first introduced the idea of instruction clustering to group dependent instructions within a basic block. Second, we characterized the operands connecting the dependent instructions and identified the intermediate values which need not to be broadcasted. Third, we presented an instruction clustering mechanism that dynamically converts multiple instructions into a single cluster entity, schedules them to the dedicated cluster execution unit, and bypasses the results through the local bypass paths. Our dynamic communication-aware mechanism transports the local values through the shortest path and minimizes the latency.

We find that about 28% and 30% of total dependence edges residing in the instruction queue can be converted to local communication on 8- and 16-way configurations. Our results show that the overall performance gains over a wide range of applications are 16% for 8-way and 35% for 16-way on average. Although our proposed architecture does not benefit across all benchmark programs, many multimedia applications result in considerable performance improvement over a traditional architecture.

Acknowledgements
This work was supported in part by the U.S. National Science Foundation under NSF grant CCR-0092552.

References

[1] T. Austin, E. Larson, and J. Cook, SimpleScalar: An infrastructure for computer system modeling, IEEE Computer, Vol. 35, pp. 59-67, 2002.

[2] L. Baumstark and L. Wills, Retargeting Sequential Image-Processing Programs for Data-Parallel Execution, IEEE Transaction on Software Engineering, Vol. 31, No. 2, pp. 116-136, February 2005.

[3] S. Bunchua, S. Wills, and L. Wills, "Reducing Operand Transport Complexity of Superscalar Processors Using Distributed Register Files," *Proc. of the 21st Int. Conf. on Computer Design*, pp. 532-535, Oct. 2003.

[4] R. Bhargava and L. John, "Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors," *Proc. of the 30th Int. Symp. on Computer Architecture*, June 2003.

[5] H. Corporaal, TTAs: Missing the ILP Complexity Wall, Journal of System Architectures, Vol. 45, No. 12, 1999.

[6] K. Farkas, et al., "The Multicluster Architecture: Reducing Cycle Time through Partitioning," *Proc. of the 30th Int. Symp. on Microarchitectures*, pp. 149-159, December 1997.

[7] Q. Jacobson and J. Smith, "Instruction Pre-Processing in Trace Processors," *Proc. of the 5th Int. Symp. on High Performance Computer Architecture*, January 1999.

[8] G. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing," *Proc. of the Int. Conf. on Parallel Processing*, pp. 239-246, August 1996.

[9] R. Kessler, The Alpha 21264 Microprocessor, IEEE MICRO, Vol. 19, No. 2, pp. 24-36, March 1999.

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

[11] H. Kim, S. Wills, and L. Wills, "Technology-based architectural analysis of operand bypass network for efficient operand transport," *Proc. of the Int. Parallel and Distributed Processing Symp.*, April 2005.

[12] C. Lee, et al., "MediaBench: a Tool for Evaluating Multimedia and Communications Systems," *Proc. of the 30th Int. Symp. on Microarchitectures*, pp. 40-51, Dec 1997.

[13] A. Peleg and U. Weiser, "MMX Technology Extensions to the Intel Architecture," *IEEE Micro*, Vol. 16, No. 4 , pp. 42-50, August 1996.

[14] J.D. Meindl, "Low Power Microelectronics: Retrospect and Prospect," *Proc. of IEEE*, Vol. 84, No. 4, pp. 619-635, 1995.

[15] R. Nagarajan, et al., "A Design Space Evaluation of GRID Processor Architectures," *Proc. of the 34th Int. Symp. on Microarchitectures*, pp. 40-51, December 2001.

[16] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," *Proc. of the 24th Int. Symp. on Computer Architecture*, pp. 13-25, 1997.

[17] S. Palacharla, "Complexity-Effective Superscalar Processors," PhD Thesis, U. Wisconsin-Madison, 1997.

[18] S. Patel and S. Lumetta., "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Computer* Vol. 50, No. 6, pp. 300-318, June 2001.

[19] 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.

[20] P. Sassone and S. Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication," *Proc. of the 37th Int. Symp. on Microarchitecture*, December 2004.

[21] N. Sreraman and R. Govindarajan, "A Vectorizing Compiler for Multimedia Extensions," Int. Journal of Parallel Programming, Vol. 28, No. 4, pp. 363-400, August 2000.

[22] M. Taylor, et al., "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Proc. of the 31st Int. Symp. on Computer Architecture*, pp. 2-14, June 2004.

[23] N. Vasseghi, et al., "200-MHz superscalar microprocessor," IEEE Journal of Solid-State Circuits, Vol. 31, No. 11, pp. 1675-1685, November 1996.

[24] S. Vassiliadis and T. Mitral, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proc of the 24th Int. Symp. on Computer Architecture*, pp. 1-12, 1997.

[25] D. Wall, "Limits of Instruction-Level Parallelism," *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, April 1991.

[26] L. Wills, T. Taha, L. Baumstark, and S. Wills, "Estimating Potential Parallelism for Platform Retargeting," *Proc. of the 9th Int. Working Conference on Reverse Engineering*, pp. 55-64, October 2002.

IEEE COMPUTER SOCIETY