

Optimizing Operand Transport using Dynamic SIMDization in Multimedia Systems

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

Abstract—For multimedia processing, modern instruction-level parallel (ILP) processors are faced with limited parallelism and inevitable communication issues, as silicon feature size decreases and pipelines become wider. In this paper, we propose and evaluate a dynamic optimization mechanism that exploits regular operand distribution patterns in multimedia applications. The proposed methodology i) improves performance by recognizing and extracting more parallelism than conventional ILP processors; ii) lowers the burden of operand transport by localizing the communication between instructions based on the operand characteristics; and iii) maintains binary compatibility by applying run-time optimization rather than requiring changes to the ISA, recompilation with a vectorizing compiler, or manual retargeting and optimization.

Topic area—multimedia systems design and implementation.

I. INTRODUCTION

Many multimedia applications have an abundance of inherent data-level parallelism (DLP) as well as ILP. However, modern general-purpose and embedded processors provide parallel execution mechanisms typically targeting only ILP, such as superscalar execution, and they are nearing the limit of what can be achieved with them [22]. Additionally, increasing the number of functional units (FUs) to take advantage of existing ILP introduces a wire-dominated operand transport network that employs poorly scaling broadcast buses to distribute operands. Today, interconnect is becoming the limiting resource in integrated circuit fabrication [10] and architectural focus is shifting from operand computations to *operand communications*.

To harness inherent DLP, manufacturers of general-purpose processors extended their instruction set architecture (ISA) with instructions targeting multimedia applications [4]. Examples include Intel's SSE [14], AMD's 3DNow! [12], and Motorola's AltiVec [3]. In the application-specific signal processing domain, many wide issue statically scheduled very long instruction word (VLIW) processors with multimedia extensions [6][21] have been proposed as well. They incorporate single-instruction, multiple-data (SIMD) functionality typically at subword level. Although simple SIMD execution units would be able to achieve significant speedups with relatively simple architectural support [15], developing and porting applications for these instruction sets have proven challenging. Typically, loop kernels employing

multimedia instructions must be supported by sophisticated compiler/automatic retargeting technology, manual assembly coding and/or hand optimization using in-line assembly code, intrinsic functions, or library routines. Furthermore, adding multimedia FUs implemented next to scalar resources and increasing issue width require a substantial amount of hardware and incur the same operand transport problem as ILP processors [13][2].

An alternative approach for multimedia processing is to implement scaleable processors and to take advantage of the available FUs. With a growing concern in wire delay caused by operand communication, many researchers have proposed new architectures focusing on communication-aware execution. The most commonly suggested method is clustering [5][11] – dividing a processor's resources into logical groups and steering the instructions between them based on dependencies. Recently, there has been interest in modulo scheduling for clustered architectures which overlaps successive iterations of a loop and uses the same schedule for each iteration to optimize resource utilization [16]. New architectures focusing on communication-aware execution have also been proposed. The TRIPS architecture based on Grid Processor cores [17] and the RAW architecture [19] propose network-connected tiles of distributed processing elements running new ISAs that expose underlying parallel hardware organization. Innermost loops of streaming applications are unrolled to fill the tiles. They exploit the abundant parallelism and regular communication patterns in stream programs, and leverage the technology scalable processing elements (PEs) and fast operand communication network. In general, these static approaches require extensive compiler support and are not binary-compatible. This paper employs existing compilers and ISAs for binary compatibility, complementing previous work.

We propose an execution mechanism that recognizes regular operand transport patterns and optimizes the operand movement in the dynamic execution environment. This exploits the data parallelism without increasing the communication overhead associated with operand movement within a datapath, especially for multimedia applications where the movement is highly regular. This paper has two primary contributions. The first contribution is the characterization of operand movement in media workloads from the perspective of support required for SIMD processing. Our study focuses on recognition of data access patterns

between dependent instructions. We show that data-parallel operation can be achieved by detecting and predicting stride values based on the recognized operand movement patterns. The second contribution of the paper is to optimize operand traffic by reducing needless communication within the datapath based on the operand characteristics. We focus on the “intermediate” operands which are produced and only consumed within the same iteration of the loop or within the next iteration of the loop. Our empirical analysis reveals a large number of such short-lived, transient operands within the innermost loops of most multimedia applications and their communication can be localized. The major focus of the proposed mechanism is on dynamic optimization to complement the large body of previous research requiring new ISAs and/or compiler support.

This paper is organized as follows. Section II introduces our instruction clustering mechanism and operand transport pattern recognition technique. An implementation is described in Section III. Section IV details the experimental setup and results for our mechanism. Finally, conclusions are drawn in Section V.

II. METHODOLOGY

Most multimedia applications, especially image processing applications, are characterized by predictable loop-based control flow with large iteration counts [18]. Moreover, empirical analysis [8] of operand usage and communication properties for multimedia programs has revealed that most operands have good locality properties (e.g., small number of consumption and short lifetime). However, in current ILP architecture models, all operands are treated alike; all operands consume the same storage and contribute greatly to traffic congestion among the FUs and broadcast bypass buses. In this section, we introduce a dynamically scheduled SIMD mechanism that targets data-parallel execution and efficient control of operand traffic for multimedia applications.

Source code for multimedia algorithms involves heavy usage of multiply nested loops (commonly “for” loops in C). Among them, we attempt to focus on the innermost loops since they are the elementary blocks of the multi-level loops and dominate overall processing time. Fig. 1 illustrates the basic concept of our instruction clustering mechanism on the dataflow graph generated from the innermost loop of the image convolution code in the Texas Instruments (TI) IMGLIB [20] suite. Each node represents an instruction 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 a previous iteration of the loop or loop initialization, or may be consumed in a later iteration; ii) *memory* (dotted line): an operand which is produced by a load as data read or consumed by a store as data to be written; and iii) *local* (gray line): an operand which is produced and consumed within the current iteration. An *instruction cluster* is defined as a connected subgraph of instructions that are joined by local operands (IC_i in the graph). Initial results of our instruction clustering mechanism to reduce the operand transport complexity were presented in [9].

To determine the scope of an external operand, they are further classified according to the input-output relationship of the loop body: i) *external-input*: an operand which only serves

as input; ii) *external-output*: an operand serving only as output; and iii) *external-updated*: an operand which serves as both input and output. The bottom box in Fig. 1 shows an example of this classification. Identification of locality allows the operands to be moved only to the required target. For instance, the local operands and external-updated input operands are only needed during the current iteration; that is, the external-updated outputs are only consumed by the next iteration. The external-inputs are required by all the iterations. All external-outputs except the last iteration have no consumers. As operands are transported only to the pre-defined target, operand traffic can be reduced by removing needless communication.

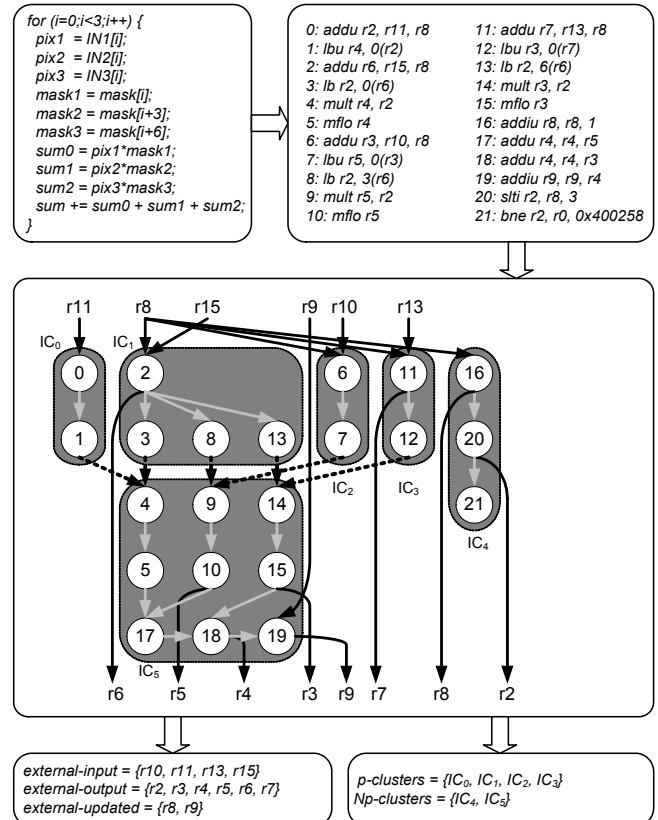


Figure 1. Instruction clustering example based on the dataflow graph of an innermost loop from image convolution code.

To perform data parallel execution on as many instructions as possible, the data-parallel region of the loop must be detected. The key feature of this work is the recognition of regular data access patterns through the external-updated edges since they form loop-carried dependences. We focus on the instruction clusters which produce the external-updated edges and attempt to identify operand transport patterns and specific computations in the clusters. Once they are recognized and identified, the stride values are easily computed and predicted, resulting in breaking the loop-carried dependences. Typically, these are chains of a small number of instructions with a simple operand transport pattern. From the example in Fig. 1, the instruction 16 in IC₄ and instruction 19 in IC₅ generate the external-updated operand r8 and r9 respectively. The constant stride value comes from the immediate field of the instruction 16 for r8; that is, values for

each loop iteration can be easily predicted by accumulating the identified stride. On the other hand, the stride for the external-updated value r9 is not identified since IC₅ have memory operands which are not predicted in general. The predicted and unpredicted external-updated operands are handled in a different way. A detailed explanation is presented in Section III.

III. HARDWARE ORGANIZATION

Our dynamic SIMD architecture is based on a dynamic optimization mechanism using trace-cache techniques [7]. Fig. 2 illustrates the block diagram of the proposed architecture. To perform the SIMD operations dynamically, we introduce new hardware units (darkly shaded blocks in Fig. 2) next to the existing superscalar out-of-order pipeline. This section describes the details of three major components: loop analysis logic, SIMD instruction queue, and SIMD processing elements.

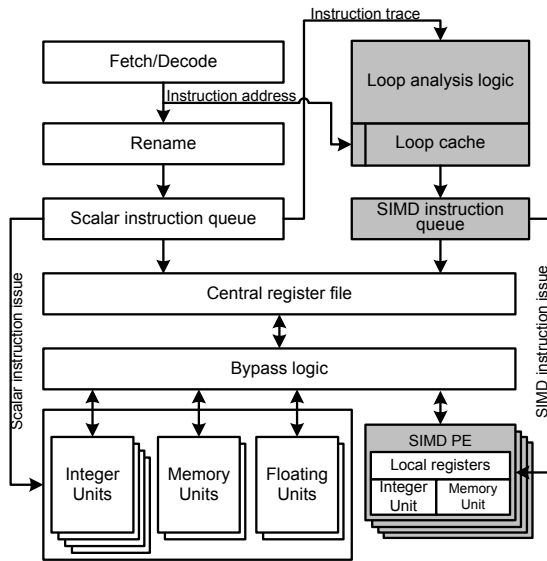


Figure 2. Block diagram of the proposed microarchitecture.

A. 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 detected loop, and caches them for future use. The loop detection is achieved with a small structure called the *loop register* which is composed of a loop start address, loop end address, and loop counter. It checks every direct conditional branch which has a backward target address (candidate loops) and marks a branch as an innermost loop when the same branch occurs consecutively. The contents of the loop register are cached in the loop cache to determine the innermost loop region and loop iteration count.

When an innermost loop is committed next time, the cluster formation is activated and the abstracted cluster information is stored in the loop cache. First it groups instructions that are connected by true data dependences into an instruction cluster and classifies the dependence edges as *local*, *memory*, and *external*. By default, all input/output ports of the instructions are considered to be sources of external

edges. Data dependence changes the external input edge of the consumer to local, and adds a local edge to the output ports of the producer. The same cluster number is assigned on both instructions. The external edges are removed when the associated register is overwritten by subsequent instructions. Then, the external input and output register set are collected to further classify the external edges as *input*, *output*, and *updated*.

The detected edge and cluster information are used to analyze the operand transport patterns along the instruction clusters which produce external-updated edges. The cluster analysis logic keeps track of operations and operand connections in the clusters and attempts to compute stride values to predict the external-updated values for each SIMD PE array. Based on the predictability of them, our mechanism evaluates the parallelizable and non-parallelizable regions in the loop separately. For example, the IC₀ ~ IC₃ in Fig. 1 are marked as *p-cluster* which is defined as a instruction cluster that produces no external-updated output and does not have unpredictable external-updated inputs. The p-clusters can be issued and executed by the PE array in lockstep operations similar to a typical SIMD machine. The others are declared as *np-clusters* and handled using conventional ILP processing.

B. SIMD instruction queue

The dispatch logic is responsible for checking the address of the instruction stream and locating the matched loop to the SIMD instruction queue when the instruction address finds a matching entry in the loop cache. An entry in the SIMD instruction queue represents instructions of multiple loop iteration. It has a single instruction address and opcode, but keeps track of multiple versions of flags such as ready and instruction status flags. Additionally, the cluster information and transport type for input and output operands are appended to control the data communication.

The instructions in the p-cluster and np-cluster are processed in different ways. For p-cluster instructions, the arrival of the last operand for the first PE triggers issue to all PEs. Our stride value prediction mechanism makes it possible for PEs to execute the instruction with predicted values. On the other hand, the np-cluster instructions are always issued one-by-one when corresponding operands are ready. Fig. 3 depicts a possible scheduling of a p-cluster (IC₁) and an np-cluster (IC₄) in the example of Fig. 1. In this example, it is assumed that SIMD array is composed of 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.

It is important to note that the instructions in the p-cluster (except those in PE₀) are executed speculatively with the predicted stride values; that is, a recovery process may be required when the prediction fails. Furthermore, loop-carried dependences caused by memory operands may exist. In both cases, the local results produced by a PE which has a mispredicted value or memory dependence are discarded and the correct processor state is recovered.

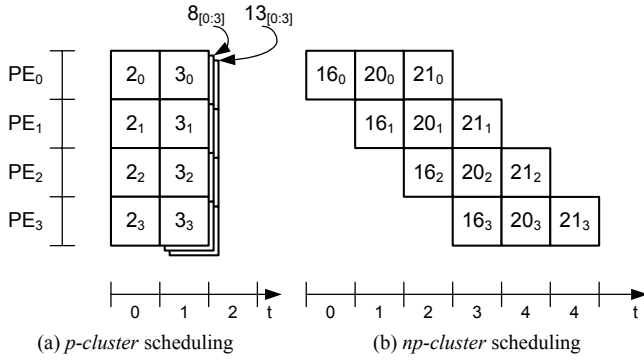


Figure 3. A cluster scheduling example.

C. SIMD PE array

The execution target for our instruction cluster is a SIMD PE array that connects nearest neighbors in a 1D mesh. A PE consists of a small number of fine-grained FUs and a small local register file to store temporary values. The operand transport network facilitates communication within a PE, between PEs, and between PEs and scalar resources by providing alternative routes. Fig. 4 shows the basic organization of a single PE having two FUs.

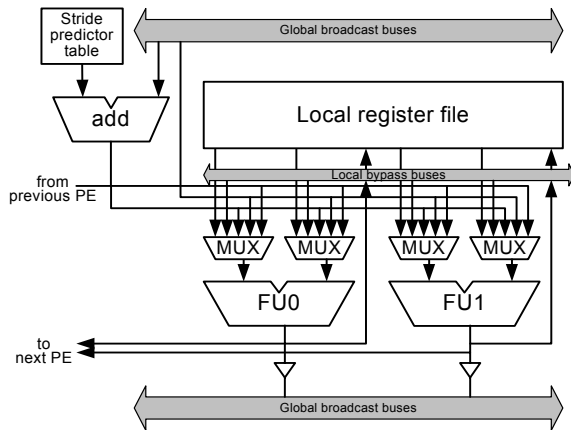


Figure 4. Basic organization of single PE in SIMD array

By default, an operand produced in a PE is transported to the local register file and to all FUs in the PE through the local bypass network. The external-input operand directly comes from global broadcast buses. Typically, the external-updated operand is passed through the dedicated neighboring network. However, special hardware support for dynamic SIMDization is provided by the stride predictor table which stores pre-computed offset values for each external-updated operand when a loop is dispatched to the queue. The stride predictor table with a dedicated adder computes external-updated values for each PE and parallelizes the instructions in p-clusters. Among the results of the last PE, those marked as external are passed to the global broadcast buses.

IV. EXPERIMENTAL RESULTS

To evaluate the effects of our dynamic SIMDization using instruction clustering, we implemented our structures and algorithms on the cycle-accurate SimpleScalar simulator with

the PISA instruction set [1]. The proposed microarchitecture is built on a conventional superscalar processor extended by a dynamically-scheduled SIMD array. As baselines for comparison, large window dynamic ILP processors are simulated at three different superscalar widths. Simulation model configurations are shown in Table 1. Our test suites consist of a number of image processing applications taken from the TI IMGLIB library [20]. Each benchmark is compiled using gcc 2.95.3 with O2 optimizations and an input image of QCIF format is assumed.

TABLE I. SIMULATION MODEL CONFIGURATIONS

Feature	baseline	ILP increase			SIMD extension	
	<i>base4</i>	<i>base8</i>	<i>base16</i>	<i>base4+</i> <i>SIMD4</i>	<i>base4+</i> <i>SIMD8</i>	
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					

Fig. 5 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 a p-cluster (gray bars) and an np-cluster instruction (black bars). On average over all benchmarks, 88% of dynamic instructions are covered by our innermost loop detection mechanism. In particular, about 79% of them can be replaced by SIMD operations and about 9% are observed in the non-data-parallel region though they are still handled in the SIMD array. The results in Fig. 5 show that most image processing applications exhibit a high degree of DLP at the innermost loop region which can be easily detected dynamically with our mechanism. It is noted that a large fraction of code in the detected loops can be parallelized by our stride prediction method, and this reveals a huge potential for enhancing performance with our dynamically scheduled SIMD mechanism.

Fig. 6 presents the IPC speedup obtained when adding our dynamic SIMD mechanism to a four-wide ILP processor. The results are compared to eight-wide and 16-wide ILP architectures. The right-most bars represent the harmonic means of speedup across all benchmarks. As shown in the left bars in Fig. 6, the performance of ILP processors increases moderately with wider pipelines. Typically, additional resources raise the potential to detect independent instructions and issue them together. In some programs, such as *conv_3x3*, *quantize*, and *wave_ver*, ILP mechanism benefits from single control flow of the innermost loop spanning tens and sometimes hundreds of instructions, and/or repeating a large number of times; that is, a deterministic control flow with a wide instruction window. However, if the innermost loops were made up of multiple basic blocks (e.g., “if” statements in

the loops), a mispredicted branch effects the next iterations of the loop; it incurs a pipeline flush even if the next iterations are independent of the current iteration. The innermost loops of *mad_8x8*, *pix_sat*, *sobel*, and *color* benchmark programs consist of three or more basic blocks and the results show that the ILP mechanism alone does not improve performance.

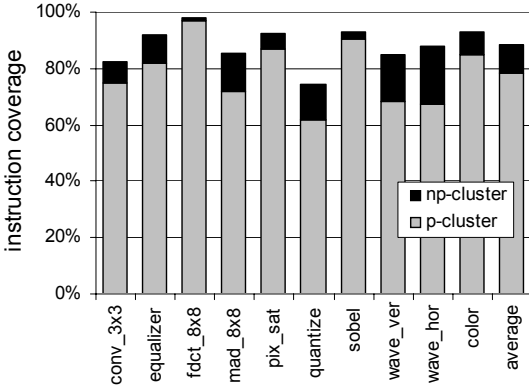


Figure 5. Percentage of dynamic instructions covered by our dynamic SIMDization mechanism.

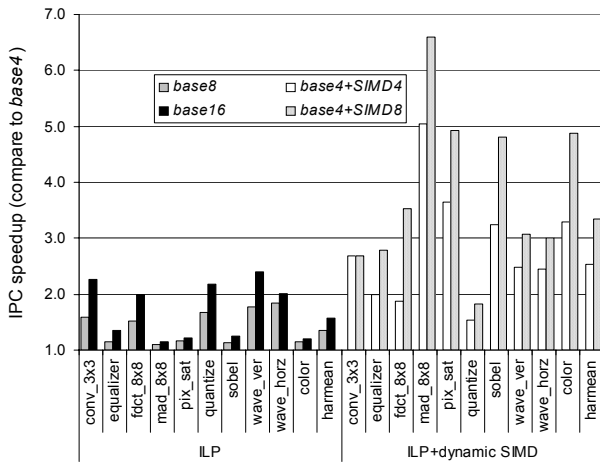
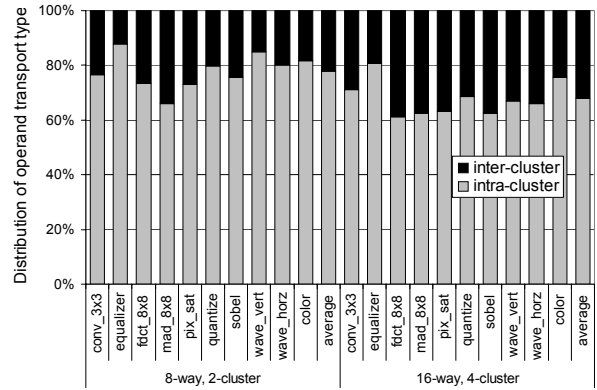


Figure 6. Performance results of ILP increase and SIMD extension compared to 4-way ILP processor.

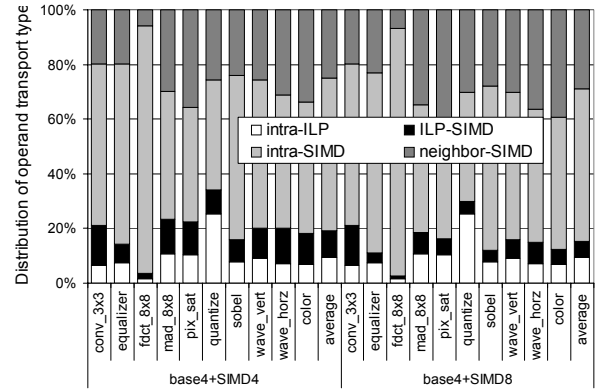
The right bars in Fig. 6 represent the performance of the proposed architecture. In most benchmarks, our dynamic SIMD outperforms conventional ILP architectures. Most of this speedup is from exploiting data parallelism as well as instruction parallelism. For example, the average speedup of *base4+SIMD4* 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* in which the loop count is extremely high (in our simulation, it is 1200) and it benefits from wide pipeline structures, including fetch, decode, issue, and commit width. It is noted that, the performance scales well as the size of the SIMD PE array increases since it can directly translate to more data parallelism. On the other hand, the performance of the ILP architecture is saturated due to the limited instruction parallelism existing in the applications. For example, while the change from *base8* to *base16* in ILP processors improves the average performance about 16%, the increase in the number of PEs from four to eight yields about

32% speedup. (An exception is *conv3x3* which shows no IPC gain due to data parallelism since the loop 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 Fig 7. For efficient operand transport, the resource clustering technique is applied to the ILP models.



(a) clustering ILP architectures



(b) dynamic SIMD architectures

Figure 7. Performance distribution of dynamic operand transport.

Fig. 7a presents the percentage distribution of operand transport types for the clustering 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 of the *base8* configuration incurs about 22% of inter-cluster communications and it is increased to 32% for the *four-cluster* model of the *base16* configuration. On the other hand, the operand transport type in our dynamic SIMD models is divided into four groups based on producer-consumer relationship: *intra-ILP*, *ILP-SIMD*, *intra-SIMD*, and *neighbor-SIMD*. The percentage distribution results are shown in Fig. 7b. It is noted that all except *ILP-SIMD* transport can be communicated without extra latency since the communication is localized. For the *base4+SIMD4* model, only 10% of dynamic operands are transported between ILP processor and SIMD array on average. Of interest is that it drops to about

6% when the width of the SIMD PE array increases from four to eight because ILP-SIMD transport changes into the neighbor-SIMD type. This implies that our mechanism can minimize the impact of operand movement, minimizing performance degradation.

Fig. 8 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 paper and we assumed one-cycle latency for long transport, such as inter-cluster and ILP-SIMD transport. The operand transport latency directly translates to IPC drops as shown in Fig. 8 compared to Fig. 6. The average IPC drops of resource clustering are 13.8% and 18.3% for two-cluster and four-cluster models respectively. On the other hand, much lower IPC drops are observed for our dynamic SIMD architecture: 3.4% and 0.9% for SIMD4 and SIMD8 configurations, respectively. These simulation results show the efficiency of the operand transport of our dynamic SIMD architecture.

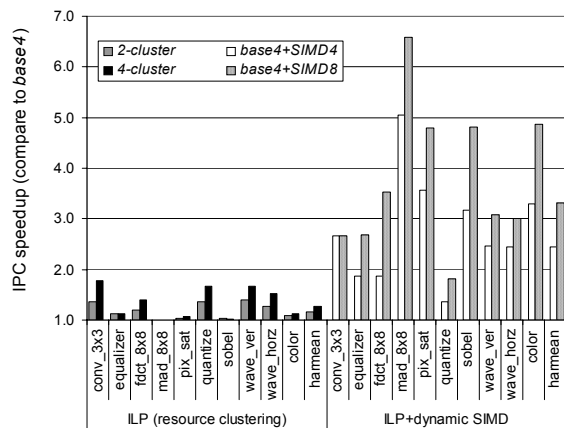


Figure 8. Performance results of ILP increase and SIMD extension considering the operand transport complexity.

V. CONCLUSION

The performance of modern ILP processors is approaching their limits due to the limited instruction parallelism and complex operand communication mechanism needed to enhance parallelism. By analyzing the characteristics of the operand transport during the execution, we have shown that inherent data parallelism can be detected dynamically by exploiting regular operand transport patterns. Our dynamic mechanism can also control the operand traffic based on the dependence between instructions and loop iterations, resulting in minimizing the latency associated with operand movement. In particular, we first introduced the idea of instruction clustering which groups dependent instructions for the innermost loop. Second, we characterized the operands connecting the instructions and loop iterations. Third, we presented a dynamic SIMD mechanism which separates the data parallel and non-parallel region from the stride prediction scheme, schedules them to the dedicated SIMD unit, and bypasses the results of instructions through the dedicated paths.

Our results show that the overall performance gains over the conventional ILP processors are 87% for 8-way and 114%

for 16-way on average. When wire delay latency is factored in, the performance gap increases to 109% for 8-way and 159% for 16-way respectively. Most of the speedup comes from exploiting more parallelism and localizing most operand communication.

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, no. 2, pp. 59-67, February 2002.
- [2] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff, The CSI multimedia architecture, *IEEE Transactions on Very Large Scale Integration Systems*, vol. 13, no. 1, pp. 1-13, January 2005.
- [3] 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.
- [4] K. Diefendorff and P. Dubey, How multimedia workloads will change processor design, *IEEE Computer*, vol. 30, no. 9, pp. 43-45, Sept. 1999.
- [5] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing cycle time through partitioning," *Proc. of the 30th Int. Symp. on Microarchitectures*, pp. 149-159, December 1997.
- [6] Tigershark, 1999. <http://www.analog.com/new/ads/html/-SHARC2>.
- [7] Q. Jacobson and J. Smith, "Instruction pre-processing in trace processors," *Proceedings of the 5th Int. Symp. on High Performance Computer Architecture*, pp. 125-129, January 1999.
- [8] 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.
- [9] H. Kim, S. Wills, and L. Wills, "Reducing operand communication overhead using instruction clustering for multimedia applications," *Proc. of the 7th Int. Symp. on Multimedia*, pp. 345-352, December 2005.
- [10] J. Meindl, "Low Power Microelectronics: Retrospect and Prospect," *Proc. of the IEEE*, vol. 84, no. 4, pp. 619-635, April 1995.
- [11] E. Nystrom and A. Eichenberger, "Effective cluster assignment for modulo scheduling," *Proc. of the 31st Int. Symp. on Microarchitecture*, pp. 103-114, December 1998.
- [12] 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.
- [13] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," *Proc. of the 31st Int. Symp. on Computer Architecture*, pp. 376-386, June 2004.
- [14] 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.
- [15] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," *Proc. of the 26th Int. Symp. on Computer Architecture*, pp. 124-135, May 1999.
- [16] B. Rau and C. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *Proc. of the 14th Workshop on Microprogramming*, pp. 183-198, October 1981.
- [17] K. Sankaralingam, et al., "Exploiting ILP, TLP, and DLP with polymorphous TRIPS architecture," *Proc. of the 30th Int. Symp. on Computer Architecture*, pp. 422-433, June 2003.
- [18] 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, August 2003.
- [19] M. Taylor, et al., "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," *Proceedings of the 31st Int. Symp. on Computer Architecture*, pp. 2-13, June 2004.
- [20] *TMS320C62x image/video processing library programmer's reference*, Texas Instruments Literature Number SPRU400, March 2000.
- [21] Trimedia tm-1300. <http://www-us3.semiconductors.com/>.
- [22] 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.