Functional Unit Chaining: A Runtime Adaptive Architecture for Reducing Bypass Delays

Lih Wen Koh and Oliver Diessel

School of Computer Science & Engineering,
The University of New South Wales, Sydney, Australia.

Embedded, Real-Time, and Operating Systems (ERTOS) Program,
National ICT Australia.*

Abstract. Bypass delays are expected to grow beyond 1ns as technology scales. These delays necessitate pipelining of bypass paths at processor frequencies above 1GHz and thus affect the performance of sequential code sequences. We propose dealing with these delays through a dynamic functional unit chaining approach. We study the performance benefits of a superscalar, out-of-order processor augmented with a two-by-two array of ALUs interconnected by a fast, partial bypass network. An online profiler guides the automatic configuration of the network to accelerate specific patterns of dependent instructions. A detailed study of benchmark simulations demonstrates these first steps towards mapping binaries to a small coarse-grained array at runtime can improve instruction throughput by over 18% and 25% when the microarchitecure includes bypass delays of one cycle and two cycles, respectively.

1 Introduction

The datapath of a microprocessor includes *bypass* (also known as forwarding) paths that route computed results among the register file, data cache and execution units. These bypass paths are typically routed in higher-level metal layers [10] with resistance and capacitance delays that increase with the scaling of feature size [2]. Thus, under continuous scaling of feature size and processor frequency, the performance of future processors is increasingly limited by the wire delays associated with bypassing for data-dependent sequences [9]. Several approaches have been suggested over the past decade to cope with this problem.

The best known approach focuses on reducing bypass latency through bypass hierarchy, as seen in clustered architectures [8, 9] where each cluster contains a small number of functional units interconnected via a fast local bypass network. Data-dependent sequences are ideally steered into the same cluster to make use of the faster intra-cluster bypass.

Self-forwarding arithmetic & logic units (ALUs) with closed loop bypass were introduced in NetBurst [4] and Sassone's work in [12] to efficiently execute linear

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

dependent chains. Sassone uses the self-forwarding ALUs to compute results for *transient* chains of sequences, where the intermediate results in the chain are only ever consumed once by the immediate successor instruction. Intermediate results are not forwarded after use but are simply discarded. The applicability of this approach is limited to situations where the transient rule is known to hold.

As an alternative approach, hardware/software partitioning speeds up execution by collapsing a sequence of operations into an atomic operation. This approach shortens the critical path of sequential operations and absorbs result bypasses into custom circuits. Typically, this is achieved by Application Specific Integrated Circuits (ASIC) co-processors in the embedded domain, whereas fine-grained units in the form of Field Programmable Gate Arrays (FPGA) are employed in reconfigurable microprocessors as configurable functional units in the datapath or as coprocessors attached to the memory or system bus. Traditionally, sequences that can be collapsed are identified at compile time. Runtime analysis has recently been introduced by Stitt *et. al.* [13] and Yehia *et. al.* [14] to support a more dynamic system. Nevertheless, FPGAs come with high synthesis and runtime reconfiguration costs.

In contrast to previous approaches, we propose the acceleration of program binaries through the mapping of data-dependent sequences to a small array of coarse-grained structures at runtime. We add to a superscalar, out-of-order processor an execution unit called the *chained* integer ALU (CIALU) which consists of a two-by-two array of closely-packed ALU cells (Fig. 1).

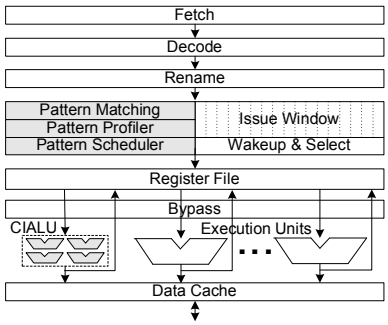

Fig. 1. A MIPS-like architecture (adapted from [9]) enhanced with a CIALU and modules (shown as grey blocks) to support runtime analysis of dataflow patterns.

By reorganizing the floorplan and reorienting the ALU cells, we show that a fast, partial internal bypass network quickly routes results among the cells. Runtime analysis is a natural choice to make efficient use of the CIALU. An online profiler tracks the relative frequency of specific dataflow patterns over time to guide the CIALU configuration. Significant shifts in the profile history lead to reassessment of the configuration choice, providing a CIALU that adapts to various dataflow patterns over successive periods of execution.

Our results demonstrate that a partial internal bypass network is sufficient to handle the small set of data-dependent patterns commonly seen at runtime. In comparison to the full local bypass network in clustered architectures, the smaller size of our partial network results in lower capacitive loads and faster operations. Overheads associated with collapsing sequences and mapping them to FPGAs do not apply in our design, but some overheads incurred by runtime analysis remain. The runtime nature of our design achieves binary compatibility for pre-compiled code and offers performance benefits without exposing application developers to additional design complexity.

In Section 2, we provide an architectural overview of our design and a normalized model of typical integer ALUs that forms the basis of our timing characterization of the CIALU. We describe in Sect. 3 our experimental framework based on the SimpleScalar toolset [3]. Our results are presented in Sect. 4. We conclude with an assessment of our results and our plans for future work in Sect. 5.

2 Proposed Architecture

2.1 Supported Dataflow Patterns

Our CIALU is a two-by-two array of integer ALU cells tightly interconnected via an internal bypass network. Assuming a full internal bypass network, the CIALU structure supports execution for a variety of dataflow patterns. Figure 2 depicts some of these patterns as nodes and edges. Each node in the pattern may correspond to an instruction in the processor's issue queue and each directed edge represents dataflow between two nodes. A CIALU with a full internal bypass network can support indefinitely long patterns, where up to three outgoing edges for results forwarding are allowed for each node in the pattern. We use the term *branching pattern* to refer to a pattern with more than one incoming or outgoing edge for any of its nodes, such as patterns 2b, 2c and 3b. We note that pattern 1a simply represents up to four parallel operations with no data dependencies. In our experiments, we consider a small subset that covers all possible data-dependent patterns of up to four nodes. Our analysis of the benefits of accelerating patterns 2a, 2b, 2c, 2d, 2e, 3a, 3b and 4a is reported in Sect. 4.

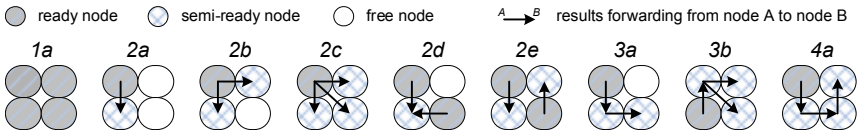

Fig. 2. An array of four integer ALUs with low-latency internal bypasses can support a variety of data-dependent patterns.

2.2 Architecture Overview

The CIALU is an execution unit in the integer datapath of a superscalar, out-of-order processor. Figure 1 illustrates our model based on the MIPS architecture. We enhance the baseline model with a CIALU and three modules that support runtime analysis of dataflow patterns: the *pattern matching* circuit, the *pattern profiler* and the *pattern scheduler*.

Initially, the processor executes a binary just as normal. The pattern profiler monitors the integer issue queue entries for instances of the supported dataflow patterns. A pattern instance is detected when the pattern matching circuit is able to map one or more ready-to-execute instructions to some of the nodes of a particular pattern, and all other nodes of the pattern are semi-ready, i.e. waiting only on the results that will be forwarded by the ready instructions. A dataflow pattern is merely characterized by the relationship of data dependencies. Thus the actual operation (e.g. addition, subtraction or logical operation) of each pattern node may vary.

The profiler updates a count history to reflect the relative frequencies of each dataflow pattern over a period of execution. Based on the count history, the profiler selects a CIALU configuration that appears most beneficial for the next period. A CIALU configuration involves setting multiplexer select signals to internally route results among the ALU cells according to the interconnection required by the selected pattern mapping.

When the next instance of the selected mapping is matched, the pattern scheduler takes over the normal integer scheduler. Each instruction in the pattern instance is scheduled to the CIALU in the order enforced by data dependencies. If the CIALU is requested but not ready to accept new operations, scheduling is handed back to the normal integer scheduler. Execution of ready instructions then falls back onto the processor's fixed ALUs.

In our architecture model, the CIALU is able to replace some of the processors' fixed integer ALUs. Therefore, the number of register file ports need not increase. Currently, we are studying the integration of the runtime analysis modules into the processor's issue logic. The performance gains of our design may be partially offset by the overheads of these modules. In this paper, we assume negligible overheads for the runtime analysis modules and we measure the performance benefits of the CIALU to save bypass cycles for dataflow patterns with dependencies. A detailed characterization of the runtime overheads will be part of our on-going work.

2.3 Normalized Model of Integer ALUs

It is difficult to compare performance of architecturally diverse processors such as MIPS, Alpha, NetBurst and our proposal. We therefore propose an execution performance model in which delays of various architectures are normalized against clock ticks. Execution times therefore need to be compared on the basis of number of clock cycles and clock frequency.

Our normalized model of an integer ALU has the following characteristics: *computation latency*, C; *bypass latency*, B; and *issue latency* (or initiation rate), R. In the case of the single-issue processor in Fig. 3(a), the bypass path feeds the ALU output back to its own input as well as routing the ALU output to the register file for result writeback. Normally, execution and forwarding (C + B) fits into a single processor clock cycle. As processor frequency increases, the ALU logic and/or the bypass path have to be pipelined to meet the requirement of the clock frequency. We expect the pipelined ALU will continue to have an initiation rate of one cycle, i.e. R = 1.

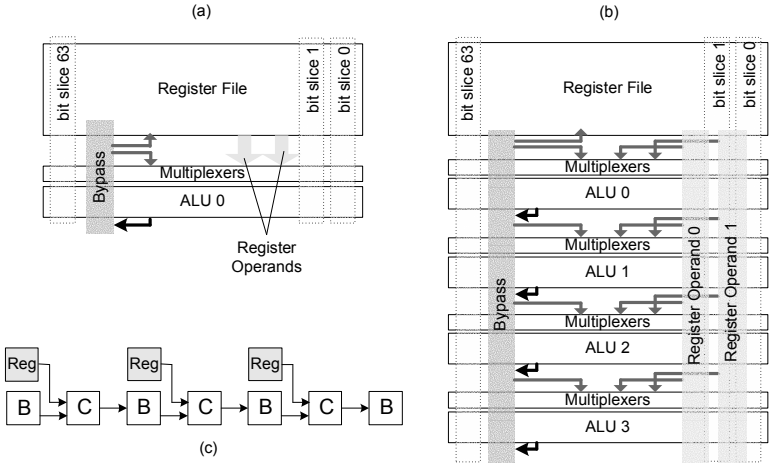

Fig. 3. Layout of the integer datapath for (a) a single-issue processor and (b) a multiple-issue processor. (c) shows the timeline of pattern 3a executed on a group of four normalized ALUs with parameters C=1, B=1 and R=1.

Figure 3(b) shows a typical bit-sliced, linear layout of a multiple-issue processor datapath with four integer ALUs. For a multiple-issue processor, the bypass paths span all ALUs. Similar to the single-issue case, (C + B) fits into a processor clock cycle at relatively low clock frequencies. However, as processor frequency increases, the ALU logic and/or the bypass path have to be pipelined.

While logic and local wires scale according to feature size, wires routed in higher metal levels grow slowly in speed relative to logic [5]. Authors have different opinions on whether the bypass paths for multi-issue ALUs are local wires routed in low-level metal layers [5] or more global wires routed in higher-level metal layers [10]. The assumption that bypass paths are local wires implies that C = B regardless of the scaling of feature size. On the other hand, although the bypass wires are unlikely to be routed in the topmost metal level as the global clock signal does, we lean towards the assumption that the bypass paths are routed in higher metal levels such that the values of B will grow relatively more quickly with respect to C, i.e. C < B. Nevertheless, we derive ALU timing models with values of C and B as shown in Table 1 to cover both design views. We

also list in Table 1 the number of cycles needed for register access, execution and writeback/forwarding, for each of the dataflow patterns from Fig. 2, scheduled to a group of four ALUs. For example, Fig. 3(c) illustrates the execution timeline of pattern 3a on a group of four normalized ALUs with C=1, B=1 and R=1. The motivation behind our design is to eliminate the bypass delays found between the dependent computations.

Table 1. Clock cycles needed by a group of four normalized ALUs to compute and bypass results for the dataflow patterns of Fig. 2.

Delay assumption	Data-dependent patterns								
	1a	2a	2b	2c	2d	2e	3a	3b	4a
C + B=1, R=1	2	3	3	3	3	3	4	4	5
C=1, B=1, R=1	3	5	5	5	5	5	7	7	9
C=1, B=2, R=1	5	8	8	8	8	8	11	11	14
C=2, B=2, R=1	6	10	10	10	10	10	14	14	18

As shown in Fig. 3(b), the operands from the register file are routed vertically to the ALUs, like the bypass paths. Thus we expect the delays for routing operands from the register file to the ALUs to scale linearly with the bypass delays. Figure 3(c) shows that these delays are incurred only for the first computation in a sequence, where subsequent routing of operands from the register file can be hidden by results bypassing. This is accordingly reflected in the timing calculations for Tables 1 and 2.

2.4 CIALU Model

Our CIALU model comprises a two-by-two array of cells, organized to optimize the internal bypass paths. The CIALU has the following characteristics: *computation latency* of each cell, C; *global bypass latency*, B; *internal bypass latency*, I and *issue latency* (or initiation rate), R. Each of the cells in the CIALU is a fully-fledged ALU, with similar latency scaling trends to the normalized ALU model described in Sect. 2.3. The *global bypass* paths of the CIALU are the usual paths that feed the cell results back to the register file, the cache and other functional units in the processor's datapath. We use the term *global bypass* here to differentiate from the *internal bypass* paths, which are shorter wires in lower level metals that route results between the cells in the CIALU.

The CIALU structure accelerates computation sequences where global bypass paths are normally in use. The tight internal bypass interconnect within the CIALU quickly feeds the cells with pending operands. The number of execution cycles for each dataflow pattern scheduled to a CIALU is listed in Table 2. The bracketed values show the savings in global bypass cycles over a typical multi-issue processor with four normalized ALUs of equivalent parameters, as assessed in Table 1. We note that an aggressive, four-cell CIALU has the same performance as a group of four normalized ALUs when executing instances of

pattern 1a because there is no need to wait for results to be forwarded via the global bypass paths.

Table 2. Timing model for a CIALU with an initiation rate of one clock cycle.

Delay assumption	Data-dependent patterns							
	2a	2b	2c	2d	2e	3a	3b	4a
C=1, B=1, I=0	4 (1)	4 (1)	4 (1)	4 (1)	4 (1)	5 (2)	5 (2)	6 (3)
C=1, B=2, I=0	6 (2)	6 (2)	6 (2)	6 (2)	6 (2)	7 (4)	7 (4)	8 (6)
C=1, B=2, I=1	7 (1)	7 (1)	7 (1)	7 (1)	7 (1)	9 (2)	9 (2)	10 (3)
C=2, B=2, I=0	8 (2)	8 (2)	8 (2)	8 (2)	8 (2)	10 (4)	10 (4)	12 (6)
C=2, B=2, I=1	9 (1)	9 (1)	9 (1)	9 (1)	9 (1)	12 (2)	12 (2)	15 (3)

The internal and global bypass paths essentially form a bypass hierarchy, where I < B. For processors with B > 1, we consider two models: an aggressive CIALU model where the internal bypass delays are absorbed into the computation latency such that I=0; and a conservative model with I=1. Figures 4(a) and 4(b) illustrate the execution timelines of pattern 3a on each of these models.

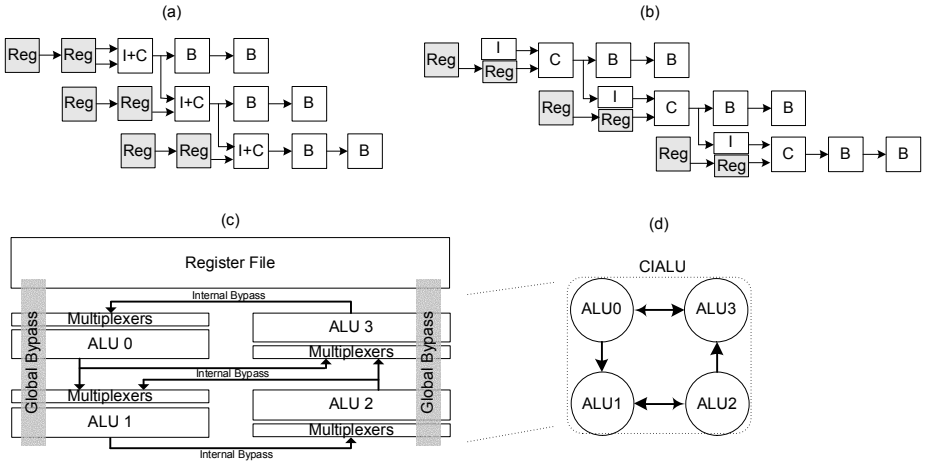

Fig. 4. Timeline for pattern 3a executed on (a) an aggressive CIALU model with C=1, B=2, I=0, R=1 and (b) on a conservative model with C=1, B=2, I=1, R=1. (c) shows the floorplan of the CIALU and (d) a high-level view of its internal bypass network.

Figure 4(c) shows the layout of our final CIALU design. The dataflow orientation of each ALU cell is chosen to minimize the internal bypass routing between the cells for the most prevalent patterns (2a, 2b, 2e, 3a and 4a) as indicated

by our results in Sect. 4. For example, pattern 2e may be mapped as ALU0 → ALU1 and ALU2 → ALU3, whereas pattern 3a may be mapped as ALU0 → ALU1 and ALU1 → ALU2.

Our design essentially restricts the full bypass network to a partial interconnection network as shown in Fig. 4(d). This partial interconnection network allows the mapping of indefinitely long patterns, where up to two outgoing edges are allowed for each of the pattern nodes. As a result, patterns 2c and 3b cannot be efficiently executed on the CIALU. However, both are relatively infrequent and are simply extensions to pattern 2b. They can thus be scheduled without loss of performance as an instance of pattern 2b and an additional operation.

The set of supported dataflow patterns do not utilize all four of the CIALU cells simultaneously. Thus some of the cells in the CIALU may be idle in a particular clock cycle. Each of these free cells functions just like an integer ALU, and therefore may accept ready-to-execute integer operations. Furthermore, the CIALU's issue latency of one cycle allows the scheduling of a new pattern instance, if available, on a per cycle basis. Scheduling constraints may occur when there is a conflict of cell mappings for two pattern instances. For example, pattern 2e cannot be scheduled to the CIALU on the third execution cycle of pattern 3a due to a cell mapping conflict on ALU2. In this case, the scheduling of pattern 2e is delayed by a further cycle.

A two-dimensional organization of the ALU cells implies that a completely bit-sliced design is no longer possible. Horizontal wires must be added within the register file to connect the two sets of global bypass paths to the ALU cells. The internal bypass paths between the cells will also occupy physical space. In spite of these considerations, we do not expect the total height of the modified datapath to be any greater than that of the linear layout shown in Fig. 3(b).

While our normalized ALU model fits the 64-bit implementation of MIPS-like processors, it does not quite model the fast ALUs in NetBurst [4]. These fast ALUs and address generation units are also organized as a two-by-two array. In a 32-bit implementation, each of the fast ALUs consists of two 16-bit slices with closed-loop bypass. A staggered add mechanism allows a 16-bit addition to complete in half a clock cycle, and a full 32-bit addition to complete in one clock cycle. The pipelined nature of the fast ALU allows an initiation rate of half a clock cycle. ALU results are sent to the register file and other execution units via a multiple cycle global bypass network. We note that the aggressive design of NetBurst allows a clock frequency that is at least three times that of a 1GHz MIPS implementation. However, for a cycle-by-cycle comparison, a 64-bit version of the fast ALU should be normalized to C=2 (assuming staggered add with two 32-bit slices), an optimistic value of B=2, I=0 and R=1. The normalized fast ALUs are able to execute indefinitely long linear chains of dependent operations such as patterns 2a, 2e, 3a and 4a in the same time as the CIALU model listed in the fourth row of Table 2. However, the timing of the fourth row of Table 1 applies for the other branching patterns.

3 Experimental Setup

We use the cycle-accurate SimpleScalar [3] *out-of-order* simulator to evaluate the impact of varying processor configurations on the performance of our design.

Global bypass delays, B, are added as a parameter to the *out-of-order* simulator. The CIALU is also added as an integer functional unit, with a default issue latency of one cycle (R=1), a parametrizable computation latency, C, and a parametrizable internal bypass delay, I. We considered four-way and eight-way baseline and enhanced processor configurations in which the total number of ALUs corresponds to the issue width of the processor. Table 3 lists the non-default parameter settings of interest.

The mechanism of the pattern matching circuit, pattern profiler and scheduler are as described in Sect. 2.2. The modified simulator implements an architecture where the runtime analysis modules do not incur any overheads and the size of the profile window is set to one instruction. This setting allows us to measure the performance gains contributed by the ability of our CIALU to save bypass cycles for dependent patterns.

Table 3. Non-default processor configurations for *sim-outorder*.

Parameters	4-way Baseline	4-way Enhanced	8-way Baseline	8-way Enhanced
Register Update Unit size	16	16	32	32
Load/Store Queue size	8	8	16	16
Number of ALUs	4	0	8	4
Number of CIALUs	0	1	0	1
Computation latency of an ALU	1/2	1/2	1/2	1/2
Computation latency of a CIALU	1/2	1/2	1/2	1/2
Global bypass delay	0/1/2	1/2	0/1/2	1/2
Internal bypass delay of a CIALU	n/a	0/1	n/a	0/1

We performed simulations for benchmarks selected from the SPEC2000 [1] and MediaBench [7] suites. We included program binaries we were able to compile for SimpleScalar's Portable Instruction Set Architecture (PISA) model. The input sets packaged with the MediaBench benchmarks were used to run the programs to completion. The SPEC2000 benchmarks were run using the MinneSPEC [6] inputs, favoured for their smaller sizes. All performance figures reported, unless otherwise stated, are averages of the results obtained over the applications in each benchmark set.

4 Results and Analysis

4.1 Prevalence of Dataflow Patterns at Runtime

In order to determine the prevalence of the dataflow patterns of interest, we disabled the runtime profiler and fixed the CIALU configuration for a partic-

ular pattern for the entire execution of a program. Pattern instances matched at runtime are scheduled to the CIALU. The processors' fixed ALUs (if any) and idle CIALU cells were used for the execution of parallel integer operations. For pattern 1a, our choice of parameters yields the same settings for both the baseline and enhanced processors. Thus, no performance gains are expected from pattern 1a.

From Table 4, we observe that linear chains of dependent sequences (patterns 2a, 2e, 3a and 4a) are most prevalent at runtime. For four-way processors, up to 45.92% of integer operations are scheduled as pattern 2a, achieving an Instructions Per Cycle (IPC) gain of 13.05%. Patterns 2b and 3b which involve results bypassing to two pending operations are less frequent, with a frequency of up to 7.62% and an IPC gain of 2.23%. Patterns 2c and 2d account for approximately 1% of integer instructions for four-way processors. Clearly, patterns 2c, 2d and even 3b do not contribute much to the overall performance benefits of the CIALU. While similar trends are observed for eight-way processors, the larger processor bandwidth allows more patterns to be mapped at runtime, achieving a slightly better performance speedup.

Table 4. Percentage of integer operations scheduled to the CIALU (*op2cialu*) and Instructions Per Cycle (IPC) gain (%) achieved by processors enhanced with a CIALU (C=1, B=1, I=0 and R=1) fixed to accelerate the given pattern for the entire execution of a program.

Data-dependent patterns	MediaBench 4-way *op2cialu*	MediaBench 4-way Gain	MediaBench 8-way *op2cialu*	MediaBench 8-way Gain	SPEC2000 4-way *op2cialu*	SPEC2000 4-way Gain	SPEC2000 8-way *op2cialu*	SPEC2000 8-way Gain
2a	45.92	13.05	55.18	14.11	35.64	8.71	41.24	9.21
2b	7.62	2.23	13.41	3.31	6.22	1.16	9.82	1.62
2c	0.92	0.23	2.37	0.23	1.09	0.24	4.54	0.59
2d	0.87	0.13	1.43	0.19	0.72	0.09	0.97	0.13
2e	14.67	3.07	32.56	6.80	12.86	3.78	21.88	5.16
3a	24.22	10.86	33.16	13.39	21.73	10.33	25.73	11.92
3b	3.69	1.62	9.15	3.90	2.68	0.74	5.25	2.01
4a	11.91	6.49	16.79	14.70	6.45	2.97	9.59	4.67

4.2 A Runtime Adaptive CIALU

Here we attempted to gain a sense of the benefit of a CIALU which is able to adapt its internal bypass paths to match dataflow patterns at runtime. Due to the profile window size of one instruction, more than one dataflow pattern may be matched in a given clock cycle. Thus, a simple greedy priority scheme was used. Priority was given to ready instances of patterns that offered the greatest savings of bypass cycles and then to those that used the highest number of cells in the CIALU. Thus, ready instances of pattern 4a were given precedence over patterns 3a, 2e, 2d, 2b, and 2a, in that order. Our CIALU design in Fig. 4(c)

excludes patterns 2c and 3b due to the limitation of the partial internal bypass network. The results from Sect. 4.1 show that the low frequencies of patterns 2c and 3b allow us to reschedule both patterns as pattern 2b without loss of performance. The CIALU's issue latency of one cycle allows the scheduling of a new pattern instance on a per cycle basis, subject to the scheduling constraint discussed in Sect. 2.4.

Table 5 lists for each benchmark the IPC gains for a four-way enhanced processor. Similar to the CIALU with fixed configurations in Sect.4.1, instances of pattern 2a are most frequent in many of our benchmarks, followed by pattern 3a, 4a, 2e, 2b and 2d. However, slightly different trends are observed for the *epic* and *301.apsi* applications.

Table 5. IPC gains (%) and breakdown of pattern frequencies (%) for a four-way processor enhanced with a runtime adaptive CIALU (C=1, B=1, I=0, R=1).

Benchmarks	Gain	Data-dependent patterns					
		2a	2b	2d	2e	3a	4a
MediaBench							
adpcm.enc	21.45	16.57	0.31	1.56	12.21	12.42	15.56
adpcm.dec	16.73	14.54	0.47	0.00	12.12	22.00	5.27
epic.enc	2.74	20.65	0.15	0.02	2.50	4.62	0.25
epic.dec	5.68	13.35	1.81	0.02	7.74	8.59	0.67
g721.enc	18.80	24.27	2.84	0.14	9.87	10.65	7.62
g721.dec	16.53	22.55	3.29	0.28	10.13	9.58	7.59
jpeg.enc	24.00	14.42	1.07	0.76	9.67	19.56	7.52
jpeg.dec	29.04	11.69	0.52	0.02	4.39	19.51	18.31
mpeg2.enc	13.23	14.38	13.29	1.03	3.10	9.28	4.20
mpeg2.dec	15.23	14.98	0.87	0.01	1.64	14.84	13.74
pegwit.enc	27.01	10.47	3.58	1.11	5.02	17.87	16.43
pegwit.dec	26.32	11.64	3.11	0.37	4.83	18.46	15.22
average	18.06	15.79	2.61	0.44	6.93	13.95	9.36
SPEC2000							
171.swim	10.98	14.83	1.61	0.00	3.70	8.53	5.00
173.applu	18.42	17.70	1.29	0.05	3.38	16.76	15.95
176.gcc	7.18	12.29	2.84	1.24	3.46	7.91	3.85
181.mcf	5.12	13.73	3.02	0.01	1.62	5.31	9.62
188.ammp	3.14	17.47	4.13	0.13	3.48	3.96	1.91
197.parser	10.73	16.57	9.97	0.55	2.61	3.18	1.15
301.apsi	42.49	2.57	0.18	0.01	0.32	65.92	1.15
average	14.01	13.59	3.29	0.28	2.65	15.94	5.52

The *epic* benchmark recorded an unusually low percentage (<1%) of pattern 4a at runtime, in contrast to the average 11.92% for the other applications in MediaBench. Pattern 4a is the longest chain in the set of patterns we analyzed and instances of this pattern can potentially save the largest number of global bypass cycles. The low frequency of pattern 4a yields the low IPC gains of

2.74% and 5.68% for the *epic* encoder and decoder applications, respectively. As for *301.apsi*, 65.92% of operations scheduled to the CIALU belong to instances of pattern 3a, but less than 3% correspond to pattern 2a. The larger number of global bypass cycles saved by pattern 3a contributes to a large IPC gain of 42.49% for *301.apsi*.

4.3 Impact of Global Bypass Delays

Table 6 reports on the impact of global bypass delays on both baseline and enhanced processors. As we expect, IPC decreases as both register access delays and global bypass delays are increased from zero to two clock cycles. The decrease in IPC can be partly compensated for by the higher processor frequency possible with the pipelining of the ALUs and/or the global bypass paths. For ease of comparison, we report our results in terms of IPC.

We also observe that the additional instruction issue bandwidth provided by eight-way processors is insufficient to compensate for the loss of IPC caused by the gradual increase in the global bypass delays, due to the prevalence of dependent data patterns in the benchmark applications. For example, a baseline eight-way processor with ALUs of parameters C=1 and B=1 achieved an IPC of 1.1487, which is a slowdown of 30.9% compared to a baseline four-way processor with ALUs of parameters C=1 and B=0.

Table 6. IPC and IPC gains (%) for processors with different global bypass delays.

Delay assumption	MediaBench				SPEC2000			
	4-way		8-way		4-way		8-way	
	IPC	Gain	IPC	Gain	IPC	Gain	IPC	Gain
Baseline								
C=1, B=0	1.6634	n/a	2.1778	n/a	1.2720	n/a	1.5408	n/a
C=1, B=1	0.8715	n/a	1.1487	n/a	0.7175	n/a	0.8875	n/a
C=1, B=2	0.5810	n/a	0.7646	n/a	0.5002	n/a	0.6197	n/a
C=2, B=2	0.5125	n/a	0.6812	n/a	0.4526	n/a	0.5616	n/a
Enhanced								
C=1, B=1, I=0	1.0083	18.06	1.3612	20.43	0.8206	14.01	1.0205	15.25
C=1, B=2, I=0	0.7160	25.77	0.9789	29.67	0.6006	20.59	0.7506	22.45
C=2, B=2, I=0	0.6211	23.18	0.8394	25.83	0.5339	18.49	0.6700	20.29

4.4 Impact of Internal Bypass Delays of the CIALU

The timing model of the CIALU (Table 2) shows that the aggressive model saves twice as many global bypass cycles as the conservative model. This is reflected in the performance reported in Table 7 which indicates that the IPC gains for the aggressive model are roughly doubled that of the conservative model. The results also indicate that our design benefits both four-way and eight-way processors, achieving IPC gains of up to 25.77% and 29.67%, respectively.

Table 7. IPC gains (%) for the aggressive and conservative CIALU models.

Delay assumption	MediaBench		SPEC2000	
	4-way	8-way	4-way	8-way
C=1, B=2, I=0 *(aggressive)*	25.77	29.67	20.59	22.45
C=1, B=2, I=1 *(conservative)*	13.38	15.47	10.93	12.20
C=2, B=2, I=0 *(aggressive)*	23.18	25.83	18.49	20.29
C=2, B=2, I=1 *(conservative)*	12.12	13.91	10.03	11.05

5 Conclusion & Future Work

In this paper we studied the benefits of allowing the functional units of a modern microprocessor to reorganize themselves into connected structures to reduce delays in forwarding results to dependent operations. These delays are expected to increase to several clock cycles and substantially limit instruction throughput of superscalar architectures as process technology and clock periods continue to decrease. We proposed adding to a superscalar, dynamically scheduled processor a functional assembly we refer to as a chained functional unit (CIALU), a two-by-two array of fully-fledged integer ALUs with a fast, partial interconnection network. The network is configured to simultaneously bypass results between the ALUs with minimal delay. At high execution initiation rates, this structure allows long chains of linearly dependent operations and more complex branching dataflow patterns to be accelerated. The CIALU is dynamically configured for the dataflow pattern identified through runtime profiling of the executing binary. Over a defined number of subsequent cycles, instances of the configured pattern are sought out in the issue queue and mapped to the configured CIALU. During this period the dataflow patterns present in the queue are monitored to reassess the configuration choice and adapt to changes in the pattern distribution.

The diverse collection of existing architectures with disparate clock frequencies and computational output per cycle presented the problem of comparing the performance results of our proposal with these architectures. To overcome this problem, we proposed a normalized timing model that takes into account the number of clock cycles needed for register accesses, execution, results bypassing and writebacks. The model was then used to derive the relative execution performance of architectures such as MIPS, NetBurst and our proposed CIALU. We thereby laid the groundwork for a high fidelity SimpleScalar simulation of these architectures executing a variety of common benchmarks.

Our analysis of the results indicates that non-branching, linear chains of operations are by far the most prevalent dependent dataflow patterns found in the issue queue of pending instructions. These contributed most to speedups. We found that a four-way processor that has its four integer ALUs replaced by a CIALU with an internal forwarding delay of zero cycle and a global bypass delay of one cycle is capable of boosting the number of instructions executed per cycle by approximately 18% for MediaBench, and approximately 14% for

the MinneSPEC inputs for SPEC2000. The improvements on these benchmarks rise towards 30% when an eight-way processor is enhanced and as bypass delays increase to two cycles. Unfortunately, the contribution to performance improvement due to branching dataflow patterns is relatively small. They accounted for less than 10% of the dataflow patterns accelerated by our system.

Our on-going work will include profiling entire loop bodies, deriving the dataflow graphs and mapping them at runtime to larger coarse-grained arrays. We are also interested to investigate how our results will vary for in-order, soft-core processors targetted to FPGAs for embedded applications.

References

1. CPU SPEC2000 Benchmarks. http://www.spec.org
2. International Technology Roadmap for Semiconductors. http://www.itrs.net
3. The SimpleScalar Toolset. http://www.simplescalar.com
4. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. Intel Technology Journal Q1, 2001.
5. R. Ho, K. Mai, and M. Horowitz. The Future of Wires. In the Proceedings of the IEEE, April 2001, pp. 490-504.
6. A. J. KleinOsowski, and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. Computer Architecture Letters, Volume 1, June 2002.
7. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. International Symposium on Microarchitecture, 1997, pp. 330-335.
8. M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, M. Gowan, D. Priore, and K. Wilcox. Circuit Implementation of a 600MHz Superscalar RISC Microprocessor. International Conference on Computer Design, October 1998, pp. 104-110.
9. S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processor. In Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), 1997, pp. 206-218.
10. J. M. Rabaey, A. Chandrakasan, and B. Nikolic. Digital Integrated Circuits: A Design Perspective. Prentice-Hall, Second Edition, 2003.
11. P. G. Sassone, and D. S. Wills. Multi-cycle Broadcast Bypass: Too Readily Overlooked. In Proceedings of the Workshop on Complexity-Effective Design (WCED), May 2004.
12. P. G. Sassone, and D. S. Wills. Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. International Symposium on Microarchitecture (MICRO), 2004, pp. 717.
13. G. Stitt, R. Lysecky, and F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Design Automation Conference (DAC), 2003, pp. 250-255.
14. S. Yehia, and O. Temam. From Sequences of Dependent Instructions to Functions: A Complexity-Effective Approach for Improving Performance without ILP or Speculation. International Symposium on Computer Architecture (ISCA), 2004, pp. 238-249.