Realizing High-IPC Using Time-Tagged Resource-Flow Computing

November 17, 2001

Abstract

There have been numerous instruction-level parallelism studies that suggest that there exists a large of amount of parallelism in typical (e.g., SPECint) programs. In this work we present a new approach to exploiting this parallelism through the use of resource-flow computing. Resource-flow computing defines a machine where shared resources are allotted to the earliest, ready to execute, instruction. Many of the ideas embodied in our resource flow model come the Tomasulo Algorithm's reservation stations. We present the design of a machine that uses Active Stations, time tags and a registerless data path. We also discuss our use of disjoint path execution to provide a large window of instructions to execute. We evaluate our implementation using a trace-driven simulation environment, while also evaluating the scalability of our implementation using VHDL models of critical system elements. We call our implementation REFLOW.

While our current REFLOW implementation assumes a MIPS-like ISA, our ideas can be applied to any ISA (including superscalar or VLIW), and do not rely on any compiler support. We have evaluated a range of REFLOW design implementations, studying the effects of maximum issue width, disjoint path spawning, and value speculation. We present IPC numbers for SPECint95 and SPECint2000 programs, clearly showing that REFLOW can obtain IPC's greater than 10.

1 Introduction

A number of ILP studies have concluded that there exists a significant amount of parallelism in common applications [8, 12, 22, 25, 26]. So why haven't we been able to obtain these theoretical speedups? We believe there are two fundamental problems. The first is that the presence of true dependencies in programs can limit parallelism unless effective address and data value speculation is employed. A number of studies have already demonstrated the value of this idea [7, 8, 14]. The second problem is related to nondeterministic control flow. It is very difficult to generate a large window of instructions to supply wide issue processors [6].

Our resource flow model was developed through analyzing the results presented in prior ILP limit studies, and through our earlier work on on disjoint path execution [16]. Lam and Wilson showed us that if a machine could follow multiple flows of control, a speedup of 158 could be obtained on average [12]. Gonzalez and Gonzalez showed us that if we use aggressive data value speculation with an unlimited instruction window size, we could enjoy speedups of over 100 [8]. When the instruction window was limited (less than 512 instructions) speedups were in the single digits when compared to other ILP mechanisms.

Austin and Sohi [2] reported that that register lifetimes tend to be limited to one or two basic blocks. This suggests we only need to communicate register updates directly to a few, nearby, processing elements.

Research has already been reported that overcomes many control flow issues using limited multi-path execution [3, 25]. While we feel that the design proposed by Chen in [3] would be difficult to realize in hardware, this work has provided us with a peek of what might be possible if we could build *adaptive branch trees*. The paper also suggests that most of the benefits obtainied via a dynamic prediction scheme can be obtained using a static tree approach (e.g., Sindagi and Uht's disjoint eager execution model [25].

We have taken many of these ideas and incorporated them into our machine model. To support rampant speculation, while maintaining scalable hardware, we have introduced the use of a statically ordered machine that utilizes instruction time tags to allow for aggressive speculative execution. This paper describes our machine model and demonstrates how we obtain high IPC in an implementation that utilizes scalable hardware.

In Section 2, we review related work on high ILP. In Sections 3 and 4 we present some of the fundamental ideas embodied in the REFLOW machine. In Section 5 we present simulation results that substantiate our claims to both high IPC and scalability. In Section 6 we present the results of VHDL modeling of key REFLOW components. In Section 7 we discuss the relation of this work to past studies, and in Section 8 we summarize the contributions of this paper.

2 Related Work

Wall [26] presented an ILP study that assumed dynamic branch prediction in the presence of perfect memory disambiguation, perfect register renaming, unlimited instruction fetch bandwidth, and a large number of functional units. For this model, the range of speedups reports was between 4.1 and 7.4.

Lam and Wilson [12] conducted an extensive set of revealing simulations on some of the SPEC89 benchmarks assuming different control dependency resolution techniques. Their most advanced model employed Single-Path branch speculation (like a simple branch predictor) with reduced Control Dependencies (instructions after a forward branch's target are independent of the branch) and Multiple Flows of control (multiple program counters); it was called: SP-CD-MF. For this model, an average speedup 40X was achieved on the integer benchmarks; unlimited execution resources were assumed. For an Oracle predictor (a branch predictor that obtains 100% prediction accuracy), a speedup of 158 was obtained.

The Lam and Wilson study illustrated the need for providing a large instruction window to obtain large ILP. They concluded that a large amount of ILP exists in integer codes, but conceded that it was unlikely to be realized, particularly with high IPC, because of the machine limitations extant at the time. In particular, no commercial machine realized MF, and few realized CD. We view the Lam and Wilson results as a challenge to produce a machine model that will realize high IPC from the available high ILP. Our machine model goes beyond SP-CD-MF in both the data speculation and control speculation dimensions. Thus, we should be able to exploit ILP between 40 and 158, and possibly even beyond, since data speculation was not included in the Lam and Wilson study.

Gonzalez and Gonzalez [8] performed a study on the potential impact of address and data speculation on ILP. Using the SPEC95 benchmark suite, they evaluated the benefits of address and data value prediction. They assumed infinite execution resources and found that they could achieve a speedup of 42. While they assumed that multiple predictions could be in flight simultaneously, they did not consider following multiple paths of execution. Riseman and Foster [19], Chen [3] and Uht [25] found much ILP in general purpose code, but none of these studies made use of *data*

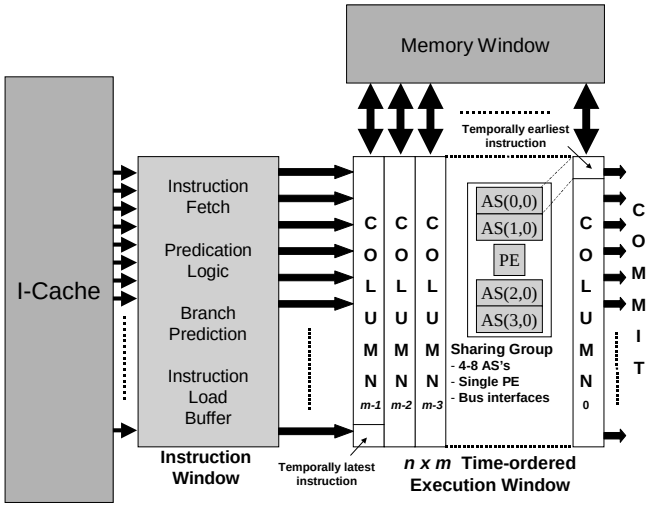

Figure 1: The REFLOW machine model.

speculation.

From our study of past work, and our experimentation through the REFLOW design space, we have recognized that high-IPC is possible, but only if we use aggressive address/data speculation and disjoint-path execution. REFLOW embodies a number of existing and new machine features, all in the same computing fabric. These features include:

- a registerless datapath,
- Active Stations,
- instruction, predicate and memory time tags, and
- disjoint path execution.

Next we describe these fundamental features of our proposed machine model.

3 REFLOW Machine Model

Figure 1 presents the overall model of REFLOW, which consists of 3 main components:

1. the Instruction Window,
2. the Execution Window, and
3. the Memory Window.

The Instruction Window fetches instructions from an instruction memory, performs dynamic branch prediction, and generates predicates. Instructions are fetched in the *static order* in which they appear in the program (similar to assuming all branches are not taken). This provides us with the ability to fetch a wide window of instructions. The only exceptions to this fetch policy are:

1. subroutines are inlined in the Instruction Window, and

2. in the case of conditional branches with *far* [1] targets, if the branch is strongly predicted taken (i.e., a value of 3 the Branch Target Buffer (BTB) 2-bit counter), we begin static fetching from its target.

In this paper, REFLOW utilizes a two-level gshare conditional BTB in a somewhat unconventional way. The predictor is used to guide both instruction fetch (as in case 2 above), as well as instruction issue. The branch predictor can also be interrogated when selecting which *disjoint paths* to spawn. Disjoint path execute will be discussed in Section 4.

REFLOW is an in-order issue, in-order completion machine. REFLOW supports rampant speculative *resource-flow-order* execution. The Execution Window is organized as columns of processing elements, arranged in a number of *Sharing Groups* (SGs) per column (a SG has a common processing element). Each sharing group contains a number of *Active Stations* (ASs); instructions are issued in-order to AS's in a column. Each issued instruction is assigned a *time tag*, based on its location in the column. Time tags play a critical role in the simplicity of REFLOW.

Our ASs are designed after Tomasulo's reservation stations [23]. There is one instruction per action station. REFLOW ASs are able to snoop and snarf data from buses with the help of the time tags. ASs are also used to evaluate predicates, and to squash redundant operand updates (again using REFLOW time tags).

Basic blocks are issued in-order (as they appear in the instruction window). A *column* in the Execution Window is completely filled with the sequence of instructions as they appear in the Instruction Window. During execution, hardware runtime predication is used for all forward branches with targets within the Execution Window. Backward branches are handled via dynamic loop unrolling [20, 24] and runtime conversion to forward branches.

The Memory Window in REFLOW also uses time tags to manage multiple versions of a single address in our memory system. We borrow the design of our Memory Window from the work of Gopal et al. of a Speculative Versioning Cache [9]. We exploit the presence of time tags to resolve coherency issues (another benefit of in-order execution). In our REFLOW machine we also employ data address and data value prediction within the Execution Window. The design and analysis of the Memory Window are described in [17].

3.1 Resource-flow Computing

REFLOW attempts to exploit as much speculation as possible; this is achieved via a *resource flow* computing model. A Sharing Group (SG) of ASs share a single PE, as is shown in Figure 2. Figure 3 shows the REFLOW AS in more detail, including forwarding and backwarding buses and predication logic.

Spanning buses are comprised of both forwarding and backwarding buses. Forwarding buses are used to broadcast register, storage and predicate values. If an AS needs an input value, it sends the request to earlier AS's via a backwarding bus. The requested data is returned via the standard forwarding bus protocol.

Referring to Figure 3, an Active Station connects to the spanning buses corresponding to the AS's position in its column. Each AS performs simple comparison operations on the time tags and addresses broadcast on the spanning buses to determine whether or not to snarf data or predicates.

[1] far implies that the branch target is outside of the range of the current Execution Window

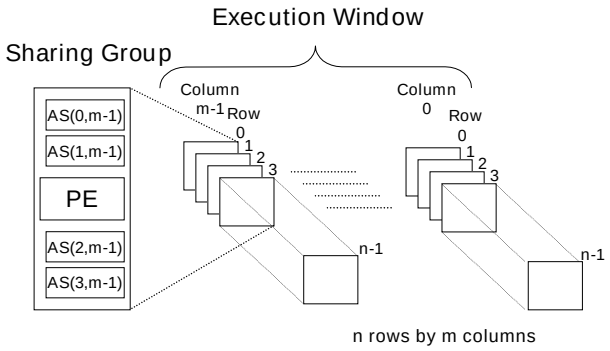

Figure 2: A REFLOW Sharing Group.

A *predicate* is assigned to AS's after a branch but before the branch's target (referred to as a branch's domain in [18]. A *canceling predicate* is assigned to instructions after the domain, to remove the enabling/disabling function of the predicate.

So far we have described a machine with AS's all connected together with some small number of spanning busses. In effect, so far there is little difference between a spanning bus and Tomasulo's Common Data Bus. This microarchitecture may reduce the number of cycles needed to execute a program via resource flow, but having the buses go everywhere will increase the cycle time unacceptably. Further, the machine is no where near being scalable.

The Multiscalar project demonstrated that register lifetimes are short, typically spanning only one or two basic blocks, 32 instructions at the high end [2, 5]. Based on this important observation, we partition the buses into short segments, limiting the number of ASs per sharing groups any spanning bus is connected to; this has been set to 32 AS's, or four sharing groups in our work presented here.

REFLOW assigns PEs to the highest priority instruction in a SG that has not been executed, independent of whether its inputs or operands are known to be correct (data flow independent), and regardless of whether this instruction is known to be on the actual (versus mispredicted) control path (control flow independent). The rest of the execution time is spent applying programmatic data flow (re-executions) and control flow constraints (squashes), so as to end up with a programmatically-correct execution of the program.

3.2 Registerless Datapath

In REFLOW there is no centralized register file, there are no central renaming buffers nor reorder buffer. REFLOW uses locally-consistent register values distributed throughout the execution window and among the PEs. A register's contents are likely to be globally inconsistent, but locally usable. A register's contents will eventually become consistent at instruction commit time. In REFLOW, PEs broadcast their results directly only to a small subset of the instructions in the

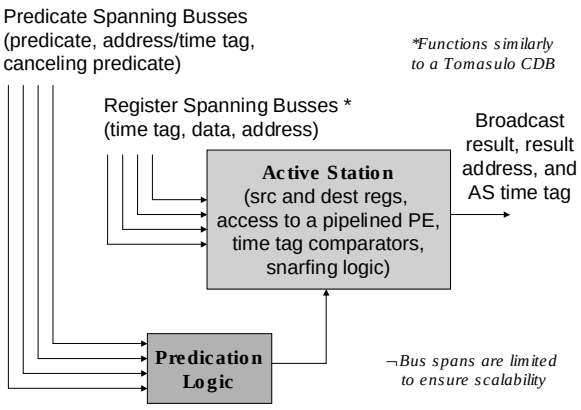

Figure 3: The REFLOW Active Station. AS's within a Sharing Group vie for the resources of the group, including one pipelined PE and the broadcast bus output(s) to the appropriate spanning bus(es). The spanning busses interconnect the Sharing Groups. Each SG sources one or more spanning busses. Each spanning bus is connected to temporally adjacent Sharing Groups among one or more columns. The spanning bus length is constant and does not change with the size of the Execution Window; this ensures scalability.

Execution Window, which includes the instructions within the same Sharing Group.

We connect each terminating bus to its adjacent originating bus with a Register Filter/Forwarding Unit (RFU). An RFU takes a register value from the preceding bus segment, stores the value in the RFU's local register file, then broadcasts it on the following bus, competing with later Sharing Groups for the bus. Thus, there is a one or more cycle delay for register values going from one bus segment to the next. This is why it is a critical observation that register lifetimes are typically short.

In certain cases it is necessary for AS's to request an operand from an earlier RFU. For example, an instruction that is just loaded into the Execution Window and has R1 as an operand must request R1's value from a previous RFU (there is no guarantee that R1 will be broadcast). To handle these requests, backwarding buses are added to connect Sharing Groups to the RFU's. These buses daisy chain the RFU's in the backwards direction, mirroring the RFU forwarding busses. The first RFU that detects a request on a backwarding bus that has a valid value for a register (R1 in our example) will satisfy the request by broadcasting the register (R1) forward. In this case, the backwarding request is terminated at the R1-sourcing RFU.

In previous work [18] we described a checking point protocol that allows for precise interrupt handling [21] in the presence of a registerless datapath. This is a critical issue if we intend to run real programs.

3.3 Time Tags

A key operational feature of REFLOW is to allow instruction execution whenever possible. This implies the need to temporally sequentialize data flow dependent instructions, or rather ensure that the end result is the same as program-ordered execution.

The hardware must have two abilities in order to realize a data flow execution of instructions without a priori or machine-wide dependency calculations. The first, commonly-known, ability is to compare register addresses between instruction operands and result data on one or more common buses. In REFLOW, this matching is done by comparators within the Active Stations. Secondly, time tags enforce a temporal order on operand snarfing, and hence instruction execution. Only the closest result (with an earlier time tag value) present on the forwarding buses is used as the final input operand value for an instruction. Each time tag corresponds to exactly one instruction, and hence exactly one Active Station.

Time tags are logically divided into two parts: a column tag concatenated with a row tag. The row time tag of an instruction never changes once the instruction is loaded into the Execution Window. However, the column time tags are all decremented by one whenever the oldest column (the column with time tag 0) commits and the Execution Window logically shifts right. That is, the Execution Window is logically a circularly shifting set of columns; however no shifting ever physically takes place, only the columns shift by decrementing their column time tags. The required range of time tag values is equal to the maximum number of instructions that can be present in the Execution Window (a processor with 8 AS per column and 8 columns requires a 6-bit time tag). One other benefit of the column renaming is the reduced design complexity resulting from not actually shifting the Execution Window.

4 Simulation Methodology

4.1 Simulation Results

To evaluate the performance of the ideas we have just described, we have developed a trace-driven simulation model of the REFLOW machine. The simulator takes as input a trace containing instructions and their associated operand values. We have traced 5 programs from the SPECint suite, (Bzip2, Gzip, Parser and Gap from SPECint2000 and Go from SPECint95). We utilize the SpecInt reference inputs for all simulations. Our current environment models a MIPS-1 ISA with some MIPS-2 and MIPS-3 instructions included which are used by the SGI compiler or are in SGI system libraries. While we have chosen a specific ISA in this work, our model can utilize any ISA.

For our Baseline system, we assume a machine that is bound by true dependencies in the program, and does no forwarding or backwarding of values. The machine follows a single path of execution (no disjoint paths are spawned).

The next system we consider includes both forwarding and backwarding (resource flow speculation), and also includes simple value prediction. This allows us to be more precise with our speculative execution. For value prediction, we load an AS with the value of input operands used the last time this operation was executed. This can be considered an ideal last value predictor in that the predictor uses an infinite amount of space (which is a function of the number of unique instructions executed in the program). We will refer to this model as the VP model.

Table 1 summarizes many of the machine parameters we use in the set of results presented. Note that we are not using the Memory Window design that we described earlier, but instead a more typical memory hiearchy. We also present results considering a perfect memory system. Table 2 shows the 5 different machine configurations studied.

Feature	Size	Comment
Fetch width	1-column each cycle	
I-Cache		100% hit
Branch predictor	2-level gshare 1024 PAg 4096 GPHT	multi-ported
D-Cache	32KB 2-way 32B line	4-way interleaved
D-hit time	1 cycle	
D-miss penalty	10 cycles	
L2 and Memory		100% hit
Forwarding unit delay	1 cycle	
Backwarding unit delay	1 cycle	
Bus delay	1 cycle	

Table 1: Common model simulation parameters

Figure 4 shows IPC results for our five benchmarks for the six machine configurations described. We include results for our Baseline system, with a D-cache model (Base-CM) and with an perfect

Machine Configuration	SGs per Column	ASs per SG	Columns
s4a4c4	4	4	4
s8a4c4	8	4	4
s8a4c8	8	4	8
s8a8c8	8	8	8
s16a8c4	16	8	4
s8a4c16	8	4	16

Table 2: REFLOW machine configurations.

data memory system (Base-PM). In our perfect data memory model, all accesses hit in the cache, and the cache hit time is modeled as one cycle.

We also include results for our resource flow model combined with value prediction. We use last values for initial data value predictions, and strides for all address accesses. We again use both a D-cache model (VP-CM) and a perfect memory system (VP-PM).

From inspecting the graphs, we see that we obtain significant gains enable forwarding and backwarding. We also see that an efficient memory design (our Memory Window) should provide us with a significant improvement in IPC. But one fact becomes clear from the set of graphs. REFLOW requires a wide window of instructions in order to obtain high IPC. Next we look at the effects of spawning disjoint paths in the Execution Window.

5 Disjoint Execution

Our resource flow Execution Window can only produce high IPC if it contains the stream of instructions that will be committed next. In an effort to insure that we can handle the ill effects of branch mispredictions, we have utilized *disjoint execution* to handle the cases where branch prediction is wrong.

In Figure 5 we show a modified Sharing Group containing both a mainline and disjoint set of ASs. The disjoint ASs will share the common PE with the mainline execution, though will receive a lower priority when attempting to execute an instruction.

Figure 6 shows an example showing where joint paths might be spawned. Remember that we fetch the static instruction stream into the Instruction Window. We do this to capture both the taken and not taken paths of a large percentage of conditional branches. The goal of disjoint path execution is to hedge on the impact of making a wrong prediction.

In Figure 6 we see 2 hammock branches [11], and one loop branch (the final branch in the loop). Since we have fetched the static instruction stream, we can then issue these disjoint paths. We provide parallel resources (i.e., ASs) for the disjoint path.

The disjoint path is used to hide potential latencies associated with branch mispredictions. The disjoint path is copied from a mainline path in a cycle when the instruction loading buses are free. The disjoint path will use a copy of the mainline path, though will start execution from a point after a branch instruction (if the branch was predicted taken), or at a branch target (if the branch was predicted as not taken) in the mainline execution. Figure 6 shows some examples of potential disjoint paths.

Disjoint paths are spawned based on a number of criteria. In Figure 6 we spawn disjoint paths

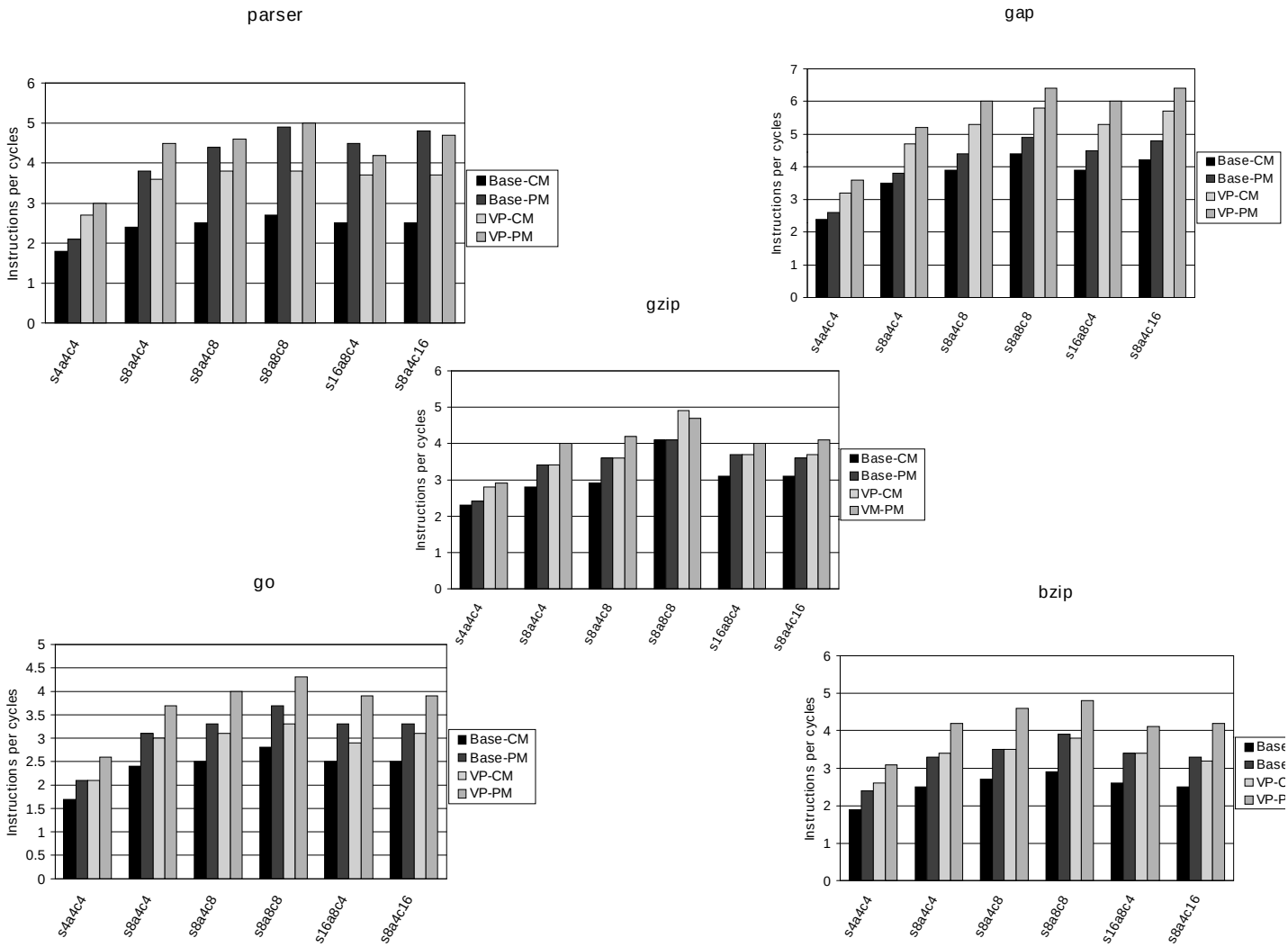

Figure 4: IPC results for our 5 benchmarks, comparing our Baseline model and a resource flow model that includes value prediction (VP). We provide results for both a D-cache memory system (CM) and a perfect memory system (PM).

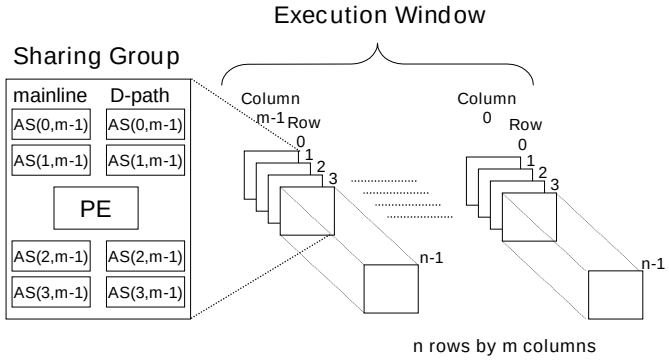

Figure 5: The REFLOW Sharing Group with disjoint path ASs.

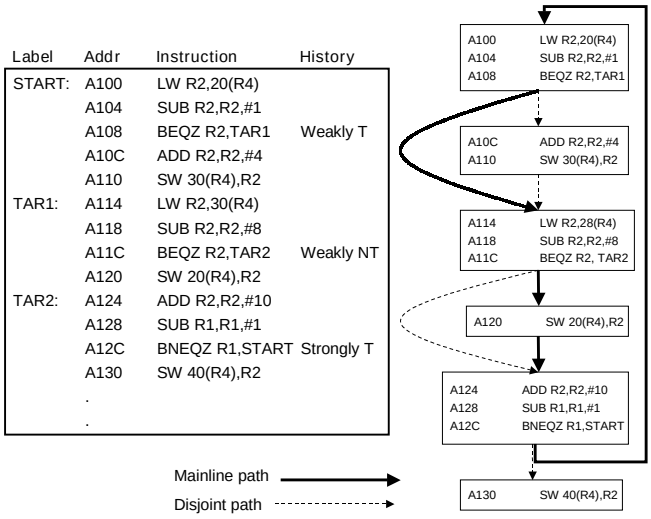

Figure 6: An example of disjoint path execution. Disjoint paths are denoted by the dashed line, while the mainline paths are denoted by a solid line.

at all conditional branches (the disjoint paths following the not-predicted path). For branches that exhibit a chaotic behavior (changing from taken to not-taken often), spawning disjoint paths should be highly beneficial. For more predictable branches (e.g., the loop ending branch in our example), we can even reap some benefit by executing down the loop exit path (depending on how often we exit the loop).

When should we spawn a path? The simplest heuristic is to look at some column in the window (let's say column d) and for every potential disjoint path found in column d, spawn a disjoint path. Spawning involves copying the instructions in column d to disjoint path ASs in another column (or potentially the same column, though using the same column has been shown to be a poor heuristic since we will compete directly with the associated mainline path). In the paper, we always start spawning from column 2 and spawn up to 5 paths (4 for 4 column configurations).

If we look at one of the D-columns, the code and state above the D-branch (the point at which we spawned a disjoint path) is the same as in the mainline path. The code is the same for the entire column (static order). The sign of the predicate of the D-branch is set to the not-predicted direction of the original branch. All other branch predications in the column follow those of the same branches in the mainline column.

Execution proceeds down mainline and disjoint paths according to available resources and the mainline path always gets priority over the disjoint path. When the D-branch resolves *correct prediction*, the disjoint path state is thrown away, and the D-column is reallocated to the next unresolved branch in the mainline. If the D-branch resolves *incorrect prediction*, the mainline state after the D-branch is thrown away, the D-column is renamed the mainline column, and all other D-columns' state (for different D-branches) is thrown away. Execution resumes with the new mainline state; new D spawning can begin again.

5.1 Disjoint path results

In Figure 7 we compare results from our value prediction configuration with the addition of D-paths. As we an see, we expose a large amount of ILP by pursuing disjoint paths. In some configurations we even double the IPC obtained in the VP-PM model. We also note that we obtain the best results for the (8,8,8) machine configuration.

6 Hardware scalability

To insure the scalability of REFLOW, we have developed a hardware description in VHDL of some of the key components of our machine. The current version of of our HD-REFLOW realizes and correctly simulates AS's (for registers and predication), Register Forwarding Units (RFUs), and register and predicate spanning busses. We have designed for a multi-column Execution Window. The synthesis of HD-REFLOW produced the gate-equivalent counts for the REFLOW sections shown in Table 3. While we target an Xilinx FPGA and a custom VLSI implementation would produce different counts, we feel the numbers serve as a guide.

We find that a 32 by 8 AS Execution Window that supports disjoint path execution but no value prediction, will require 14.5 million gate-equivalents, including flip-flops, or 58.6 million transistors (using Xilinx's 2-input NAND gate-equivalent (excluding PEs). Even doubling this for better performance and adding in the cache, PE and Instruction Window costs, this is certainly realizable with current custom VLSI technology.

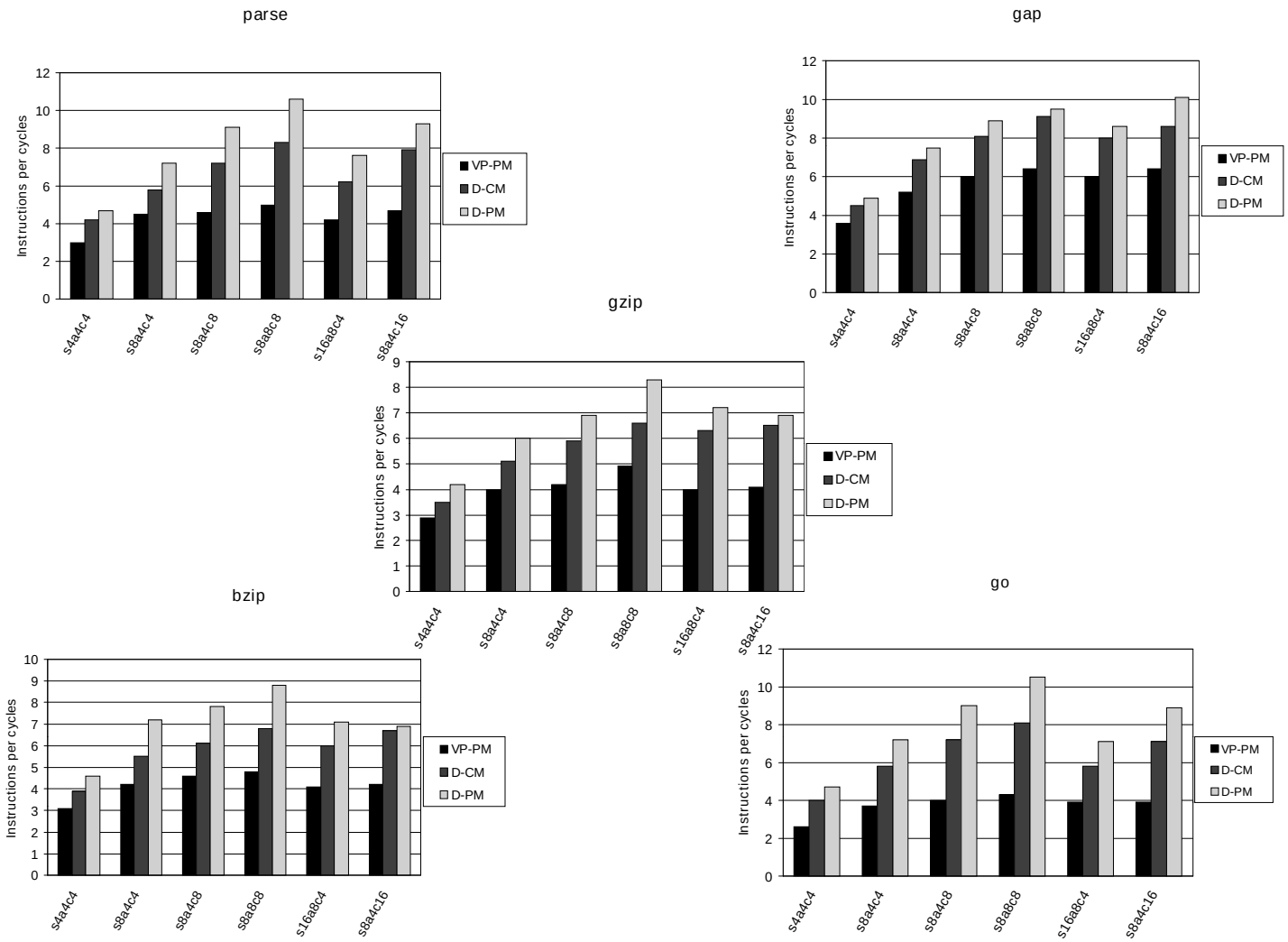

Figure 7: IPC comparison for VP with perfect memory, D-paths with D-cache memory, and D-paths with perfect memory

REFLOW Section	Gate equivalents	Transistors
Per AS	21,621	86,484
All ASs (512)	11,069,952	44,279,800
Average RFU (4/column)	135,247	540,988
Pred Forwarding Unit (4/column)	5,000 (est)	20,000 (est)
Mem Forwarding Unit (4/colum)	37,604	150,414
Total Execution Window (wo/PEs)	14,475,880	58,614,922

Table 3: Real and estimated hardware cost for a 32 by 8 Execution Window, computed in both 2-input NAND gate equivalents and transistors.

7 Discussion

Probably the most successful high-IPC machine to date is Lipasti and Shen's Superspeculative architecture [13], achieving an IPC of about 7 with realistic hardware assumptions. The Ultrascalar machine [10] achieves *asymptotic* scalability, but only realizes a small amount of IPC, due to its conservative execution model. The Warp Engine [4] uses time tags, like REFLOW, for a large amount of speculation; however their realization of time tags is cumbersome, utilizing floating point numbers and machine wide parameter updating.

Pure data flow machines [1] were once promising, but did not achieve their goals. Limited data flow, such as the Tomasulo algorithm [23] used in current superscalar microarchitectures is more successful, but is still not aggressive enough: predication is not handled, SP-CD-MF is not handled and one or two broadcast buses for the ALU results is quite limiting for high IPC.

Nagarajan et al. have proposed a *Grid Architecture* that builds an array of ALUs, each with limited control, connected by a operand network [15]. Their system achieves an IPC of 11 on SPEC2000 and Mediabench benchmarks. While this architecture presents many novel ideas in attempt to reap high IPC, it differs greatly in its interconnect strategy and register design. They also rely on a compiler to obtain this level of IPC, whereas REFLOW does not.

8 Summary

In this paper we have described the REFLOW machine model. We have illustrated the power of resource flow and especially d-path execution. We have been successful in obtaining IPC's above 10.

We still believe that there remains substantial ILP to be obtained. In Table 4 we compare the best D-path result with an *Oracle predictor* [12]. As we can see, there still remains a lot of IPC that can be obtained through improved control flow speculation. We plan to look at spawning dynamic paths (versus the static path described in this work). This should be motivation enough to pursue improvements in instruction delivery for REFLOW.

References

[1] Agerwal T. and Arvind. Data Flow Systems - Special Issue. *IEEE Computer Magazine*, 15(2), 1982.

Benchmark Benchmark	IPC wo cache	IPC w cache
go	13.2	19.0
bzip	12.4	16.7
gzip	9.9	13.0
parser	11.3	14.9
gap	9.3	10.3

Table 4: IPC for Oracle prediction for an 8,8,8 configuration with D-Cache and Perfect Memory

[2] Austin T.M and Sohi G.S. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th International Syposium on Computer Architecture*, pages 342–351, May 1992.

[3] Chen T.F. Supporting Highly Speculative Execution via Adapative Branch Trees. In *Proceedings of the 4th Annual International Symposium on High Performance Computer Architecture*, pages 185–194, January 1998.

[4] Cleary J.G, Pearson M.W and Kinawi H. The Architecture of an Optimistic CPU: The Warp Engine. In *Proceedings of the Hawaii Internationl Conference on System Science*, pages 163–172, January 1995.

[5] Franklin M. and G.S. Sohi. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–247, Dec 1992.

[6] Friendly D.H. and Patel S.J. and Patt Y.N.i. Alternative fetch and issue techniques from the trace cache fetch mechanism, 1997.

[7] Gonzalez J. and Gonzalez A. Speculative Execution via Address Predication and Data Prefetching. In *Proc. of the 11th Int. Conference on Supercomputing*, pages 196–203. ACM, July 1993.

[8] Gonzalez J. and Gonzalez A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, Department Architectura de Computadores, Universitat Po, 1997.

[9] Gopal S, T.N. Vijaykumar, Smith J.E. and Sohi G.S. Speculative Versioning Cache. Technical Report TR-1334, Unversity of Wisconsin, Madison, July 1997.

[10] Henry D.S and Kuszmaul B.C. and Loh G.H. and Sami R. Circuits for Wide-Window Superscalar Processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247. ACM, June 2000.

[11] Klauser A., Austin T., Grunwald D. and Calder B. Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures. In *Proceedings of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.

[12] Lam M.S. and Wilson R.P. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57. ACM, May 1992.

[13] Lipasti M.H and Shen J.P. Superarchitecture Microarchitecture for Beyond AD 200. *IEEE Computer Magazine*, 30(9), September 1997.

[14] Lipasti M.H, Wilkerson C.B. and Shen J.P. Value Locality and Load Value Prediction. In *Proceedings of the 7th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147. ACM, October 1996.

[15] Nagarajan R. and Sankaralingam K. and Burger D. and Keckler S. A Design Space Evaluation of Grid Processor Architectures, " To appear, 2001".

[16] Removed for anonymity.

[17] Removed for anonymity.

[18] Removed for anonymity.

[19] Riseman E.M. and Foster C.C. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, June 1972.

[20] Sherwood T. and Calder B. Loop Termination Predication. In *Proceedings of the International Symposium on High Performance Computer*, October 2000.

[21] Smith J.E. and Pleszkun A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, 1988.

[22] Smith M.D, Johnson M. and Horowitz M.A. Limits of Multiple Instruction-Level Issue. In *Proceedings of the 4th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302. ACM, April 1989.

[23] Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[24] Tubella J. and Gonzalez A. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, pages "14–23", January 1998.

[25] Uht, A. K. and Sindagi, V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 28th International Symposium on Microarchitecture, MICRO-28*, pages 313–325. ACM-IEEE, November/December 1995.

[26] Wall D.W. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188. ACM, April 1991.