

Realizing High IPC Using Time-Tagged Resource-Flow Computing

Augustus Uht¹, Alireza Khalafi², David Morano², Marcos de Alba², and David Kaeli²

¹ University of Rhode Island, Kingston, RI, USA,
uht@ele.uri.edu,

² Northeastern University, Boston, MA, USA,
akhalafi,dmorano,mdealba,kaeli@ece.neu.edu

Abstract. In this paper we present a new approach to exploiting Instruction-Level Parallelism through the use of *resource-flow computing*. This model begins executing instructions independent of data flow and control flow dependencies in a program. The rest of the execution time is spent applying programmatic data flow and control flow constraints to end up with a programmatically-correct execution. We present the design of a machine that uses *time tags* and *Active Stations*, realizing a registerless data path.

We focus our discussion on the Execution Window elements of our machine, present factors of speedup for SPECint95 and SPECint2000 programs, and discuss the scalability of our design to hundreds of processing elements.

1 Introduction

A number of ILP studies have concluded that there exists a significant amount of parallelism in common applications [9, 14, 18, 19]. So why haven't we been able to obtain these theoretical speedups? Part of the reason is that we have not been aggressive enough with our execution model.

Lam and Wilson showed us that that if a machine could follow multiple flows of control while utilizing a simple branch predictor and limited control dependencies (i.e., instructions after a forward branch's target are independent of the branch), a speedup of 40 could be obtained on average [9]. If an Oracle (i.e., perfect) branch predictor was used, speedups averaged 158. Research has already been reported that overcomes many control flow issues using limited multi-path execution [2, 18].

To support rampant speculation while maintaining scalable hardware, we introduce a statically ordered machine that utilizes instruction *time tags* and *Active Stations* in the Execution Window. We call our machine *Levo* [17]. Next we will briefly describe our machine model.

2 The Levo machine model

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

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

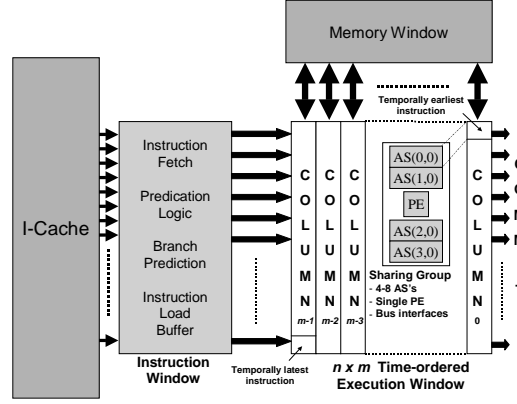


Fig. 1. The Levo machine model.

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 binary image (similar to assuming all conditional branches are not taken). By fetching down the not-taken path we will capture the taken and not taken paths of most branch hammocks [3, 8]. We exploit this opportunity by spawning disjoint execution paths to both paths (taken and not taken) for hard-to-predict hammocks. Static fetching also reduces the amount of fragmentation in the instruction memory hierarchy.

Some exceptions to our static fetch policy are:

1. unconditional jump paths are followed,
2. loops are unrolled dynamically in the execution window, and
3. in the case of conditional branches with *far targets*,³ if the branch is strongly predicted taken in the branch predictor, begin static fetching from its target.

In this paper we utilize a conventional two-level gshare predictor [12] in a somewhat unconventional way. The predictor is used to guide both instruction fetch (as in case 3 above), as well as instruction issue. Levo utilizes full run-time generated predicates, such that every branch the executes within the Execution Window (i.e., a branch domain⁴), is data and control independent of all other branches.

Levo is an in-order issue, in-order completion machine, though supports a high degree of speculative *resource-flow-order* execution. The Execution Window is organized as a grid; columns of *processing elements* (PEs) are arranged in a number of *Sharing Groups* (SGs) per column. A SG shares a common PE (see Figure 2).

Levo assigns PEs to the highest priority instruction in a SG that has not been executed, independent of whether the instruction's 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

³ far implies that the branch target is outside of the range of the current Execution Window

⁴ a branch domain includes the static instructions starting from the branch to its target, exclusive of the target and the branch itself [18]

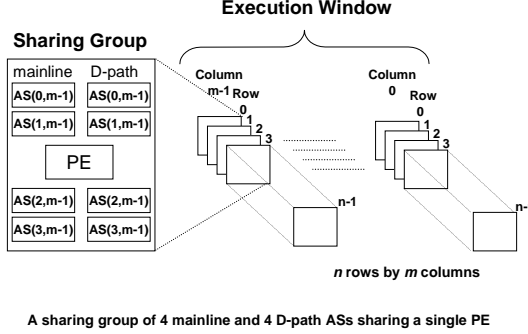


Fig. 2. A Levo Sharing Group.

data flow (re-executions) and control flow constraints (squashes), so as to end up with a programmatically-correct execution of the program.

Each sharing group contains a number of *Active Stations* (ASs); instructions are issued in static 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 Levo by labeling each instruction and operand in our Execution Window. This label is used during the maintenance/enforcement of program order in a highly speculative machine.

Our ASs are designed after Tomasulo's reservation stations [15]. There is one instruction per Active Station. Levo 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 time tags).

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 [13, 16] and runtime conversion to forward branches.

2.1 Levo Execution Window Datapath

Spanning buses play a similar role as Tomasulo's reservation stations Common Data Bus serve. 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 forwarding bus. Figure 3 shows the main busing structures that interconnect Active Stations.

An AS 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. Figure 4 shows the structure for this function of an AS.

2.2 Scalability

So far we have described a machine with ASs all connected together with some small number of spanning buses. 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) [1, 5]. Based on this important observation, we partition each bus into short segments, limiting the number of AS's connected to any segment; this has been set to 32 ASs in the results presented in this paper.

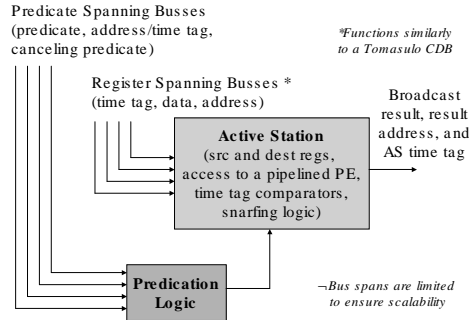


Fig. 3. The Levo Active Station. ASs within a Sharing Group vie for the resources of the group, including one pipelined PE and the broadcast bus output(s) to the appropriate spanning bus(es). The spanning busses interconnect the Sharing Groups. Each SG sources one or more spanning busses. Each spanning 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.

2.3 Registerless Datapath

In Levo there is no centralized register file, there are no central renaming buffers nor reorder buffer. Levo 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 Levo, PEs broadcast their results directly to only a small subset of the instructions in the 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.

2.4 Time Tags and Renaming

A time tag indicates the position of an instruction in the original sequential program order (i.e., in the order that instructions are issued). ASs are labeled with time tags starting from zero and incrementing up to one minus the total number of ASs in the microarchitecture. A time tag is a small integer that uniquely identifies a particular AS.

Similarly to the conventional reservation station, operand results are broadcast forward for use by waiting instructions. With ASs, all operands that are forwarded after the execution of an instruction are also tagged with the time tag value of the AS that generated the updated operand. This tag will be used by subsequent ASs to determine if the operand should be *snarfed*⁵ as an input operand that will trigger the execution of its loaded instruction. Essentially all values within the execution window are tagged with time tags. Since our microarchitecture can also allow for the concurrent execution of disjoint paths, we also introduce a path ID.

The microarchitecture that we have devised requires the forwarding of three types of operands. These are register operands, memory operands, and instruction predicate operands. These operands are tagged with time tags and path IDs that were associated with the ASs that produced them. The information broadcast from an AS to subsequent ASes in future program ordered time is referred to as a *transaction*, and consists of :

- a path ID
- the time tag of the originating AS
- the identifier of the architected operand
- the actual data value for this operand

Figure 4 shows the registers inside an active station for one of its input operands. The *time-tag*, *address*, and *value* registers are reloaded with new values on each snarf, while the *path* and *AS time-tag* are only loaded when the AS is issued an instruction.

This scheme effectively eliminates the need for rename registers or other speculative registers as part of the reorder buffer. The whole of the microarchitecture thus provides for the full renaming of all operands, thus avoiding all false dependencies. There is no need to limit instruction issue or to limit speculative instruction execution due to a limit on the number of non-architected registers for holding those temporary results. True flow dependencies are enforced through continuous snooping by each AS.

2.5 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 2 we showed a 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.

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

⁵ snarfing entails snooping address/data buses, and when the desired address value is detected, the associated data value is read

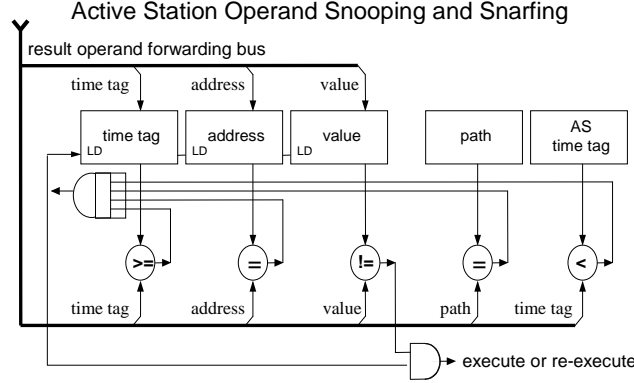


Fig. 4. *AS Operand Snooping and Snarfing.* The registers and snooping operation of one of several possible source operands is shown. Just one operand forwarding bus is shown being snooped, though typically several operand forwarding buses are snooped simultaneously.

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.

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., a loop ending branch), we can even reap some benefit by executing down the loop exit path.

In [7], we discuss how to select the best path to spawn disjoint paths. In this work, we always start spawning from the column prior to the current column being loaded, and spawn up to 5 paths (3 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.

If the D-branch resolves as a *correct prediction*, the disjoint path state is discarded, and the D-column is reallocated to the next unresolved branch in the mainline. If the D-branch resolves as an *incorrect prediction*, the mainline state after the D-branch is thrown away, the D-column is renamed as the mainline column, and all other D-column state (for different D-branches) is discarded. Execution resumes with the new mainline state; new D spawning can begin again.

3 Simulation Methodology

3.1 Simulation Results

To evaluate the performance of the ideas we have just described, we have developed a trace-driven simulation model of the Levo machine. The simulator takes as input a trace containing instructions and their associated operand values. We include results for 5 programs taken from the SPECint2000 and SPECint95 suites, using the reference inputs. 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 use a specific ISA in this work, Levo can utilize any ISA.

For our baseline system (BL), 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).

We compare the baseline to a variety of Levo systems that implement resource flow (RF). We include results for a machine that uses D-path spawning (D). We also study the effects of different memory systems, assuming either a conventional hierarchical memory system (CM) or a perfect cache memory (PM). All speedup results are relative to our baseline system with a conventional memory (BL-CM).

Table 1 summarizes many of the machine parameters we use in the set of results presented. The table includes the parameters for the conventional data memory system. Table 2 shows the 5 different machine configurations studied and presents our baseline IPC numbers which we will use to compare against.

Feature	Size	Comment
Fetch width	1-column each cycle	
L1 I-Cache		100% hit
Branch predictor	2-level gshare 1024 PAg 4096 GPHT	multi-ported
L1 D-Cache	32KB 2-way 32B line	4-way interleaved
L1 D-hit time	1 cycle	
L1 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 5 shows the percent speedup in IPC for our five benchmarks, For the six machine configurations described. All results are relative to our Baseline system with a conventional data cache memory hierarchy, as described in Table 2..

Machine Config	SGs per Column	ASs per SG	Columns	gzip BL-CM IPC	gap BL-CM IPC	parser BL-CM IPC	bzip BL-CM IPC	go BL-CM IPC
s4a4c4	4	4	4	2.3	2.4	1.8	1.9	1.7
s8a4c4	8	4	4	2.8	3.5	2.4	2.5	2.4
s8a4c8	8	4	8	2.9	3.9	2.5	2.7	2.5
s8a8c8	8	8	8	4.1	4.4	2.7	2.9	2.7
s16a8c4	16	8	4	3.1	3.9	2.5	2.6	2.5
s8a4c16	8	4	16	3.1	4.2	2.5	2.5	2.5

Table 2. Levo machine configurations and BL-CM IPC numbers for the 5 benchmarks.

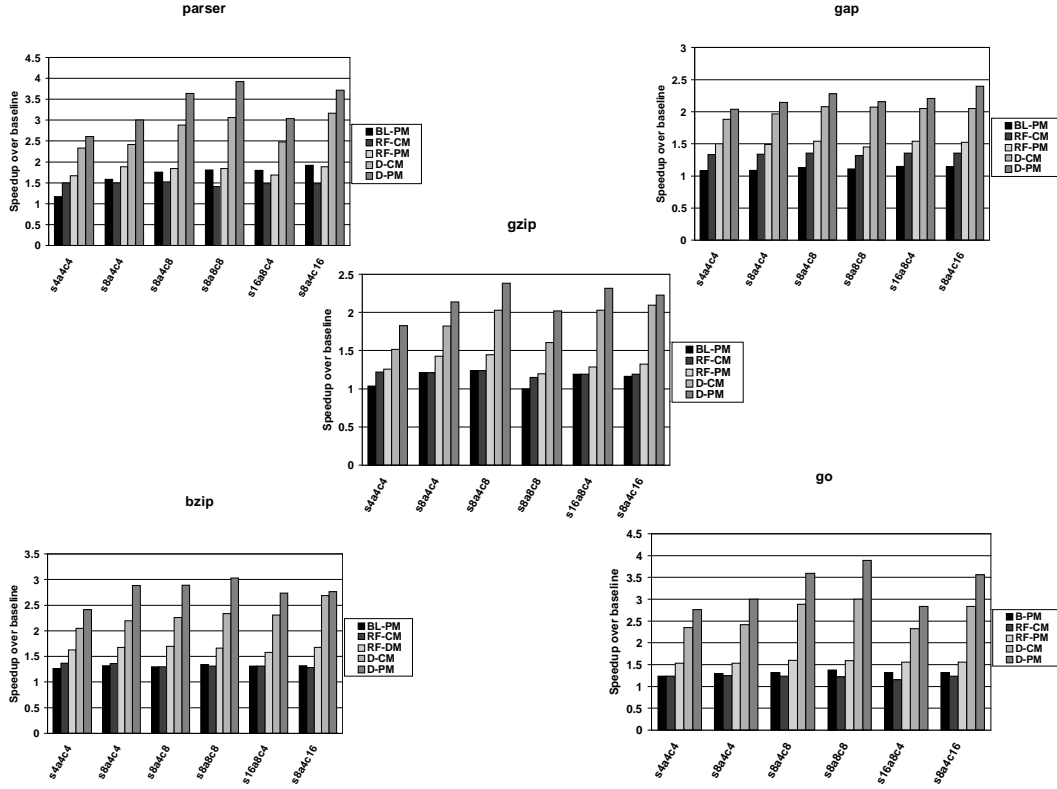


Fig. 5. IPC comparison for baseline perfect memory (BL-PM), resource flow conventional memory (RF-CM), resource flow perfect memory (RF-PM), D-paths conventional memory (D-CM) and D-paths perfect memory (D-PM). All speedups are versus our baseline assuming conventional memory (BL-CM).

Benchmark	IPC	IPC
Benchmark	D	Perfect Memory
go	1.6	1.8
bzip	2.1	2.4
gzip	1.5	1.6
parser	1.4	1.4
gap	1.0	1.1

Table 3. IPC Speedup for Oracle branch prediction for an 8,8,8 configuration with D-Cache and Perfect Memory. Speedup is relative to the D-CM and D-PM results in Figure 5.

4 Discussion

From the results in Figure 5 we can see that while resource flow provides us with some significant gains, we do not see the power of this model until we remedy a good portion of the branch mispredictions using D-paths.

Probably the most successful high-IPC machine to date is Lipasti and Shen’s Superspeculative architecture [10], achieving an IPC of about 7 with realistic hardware assumptions. The Ultrascalar machine [6] 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 Levo, for a large amount of speculation; however their realization of time tags is cumbersome, utilizing floating point numbers and machine wide parameter updating.

Nagarajan et al. have proposed a *Grid Architecture* that builds an array of ALUs, each with limited control, connected by a operand network [11]. 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 Levo does not.

5 Summary

In this paper we have described the Levo 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 3 we select the configuration that obtained the best D-path result (8,8,8) and show IPC speedup (relative to our D-path result in Figure 5) using an *Oracle predictor* [9]. 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 branch resolution for Levo.

References

1. Austin T.M and Sohi G.S. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 342–351, May 1992.
2. Chen T.F. Supporting Highly Speculative Execution via Adaptive Branch Trees. In *Proceedings of the 4th Annual International Symposium on High Performance Computer Architecture*, pages 185–194, January 1998.

3. Cher C.-Y. and Vijaykumar. Skipper: A Microarchitecture For Exploiting Control-Flow Independence, "December" "2001".
4. Cleary J.G, Pearson M.W and Kinawi H. The Architecture of an Optimistic CPU: The Warp Engine. In *Proceedings of the Hawaii International 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. 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.
7. Khalafi A. and Morano D. and Kaeli D. and Uht A. Multipath Execution on a Large-Scale Distributed Microarchitecture. In *Submitted to 16th International Symposium on Supercomputing*, February 2002.
8. 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.
9. 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.
10. Lipasti M.H and Shen J.P. Superarchitecture Microarchitecture for Beyond AD 200. *IEEE Computer Magazine*, 30(9), September 1997.
11. Nagarajan R. and Sankaralingam K. and Burger D. and Keckler S. A Design Space Evaluation of Grid Processor Architectures, "December" "2001".
12. M. S. Combining Branch Predictors. Technical Report DEC WRL TN-36, June 1993.
13. Sherwood T. and Calder B. Loop Termination Predication. In *Proceedings of the International Symposium on High Performance Computer*, October 2000.
14. 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.
15. Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.
16. 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.
17. Uht A. and Morano D. and Khalafi A. and Wenisch T. and Ashouei M. and Kaeli D. IPC in the 10's via Resource Flow Computing with Levo. Technical Report 092001-001, University of Rhode Island, Department of Electrical and Computer Engineering, Sept 2001.
18. 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.
19. 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.