Realizing High IPC Through A Scalable Memory-Latency Tolerant Multipath Microarchitecture

Alireza Khalafi
David Morano
David Kaeli
Northeastern University

Northeastern University
Computer Architecture Research

Augustus Uht
University of Rhode Island

Microarchitecture Research Institute
University of Rhode Island

presented at
Workshop on Memory Access Decoupled Architectures (MEDEA) 2002
September 22, 2002
in conjunction with PACT 2002

outline

- **introduction**
- **related work**
- **key ideas used**
- **the microarchitecture**
- **simple execution example**
- **experimental methodology & results**
  - IPC results
  - L1-I cache latency tolerance
  - L1-D cache latency tolerance
  - L2 cache latency tolerance
  - main-memory latency tolerance
  - L0 effect
- **summary**

introduction

- **want to explore a large microarchitecture (ISA independent) capable of extracting high ILP from sequential programs**
  - control & value speculation
  - multipath execution
  - massive out-of-order execution
- **large size really requires scalability of the microarchitecture**
- **proper program ordering is a big problem and needs to be handled creatively**
- **further, memory is slow ! -- how much is performance affected ?**
  - L2 is failing behind processor -- but DRAM is becoming a big problem !
- **L1 & L2 (L3) are good, but is something closer than the L1 cache ?**
  - L0 cache (distributed within the microarchitecture)

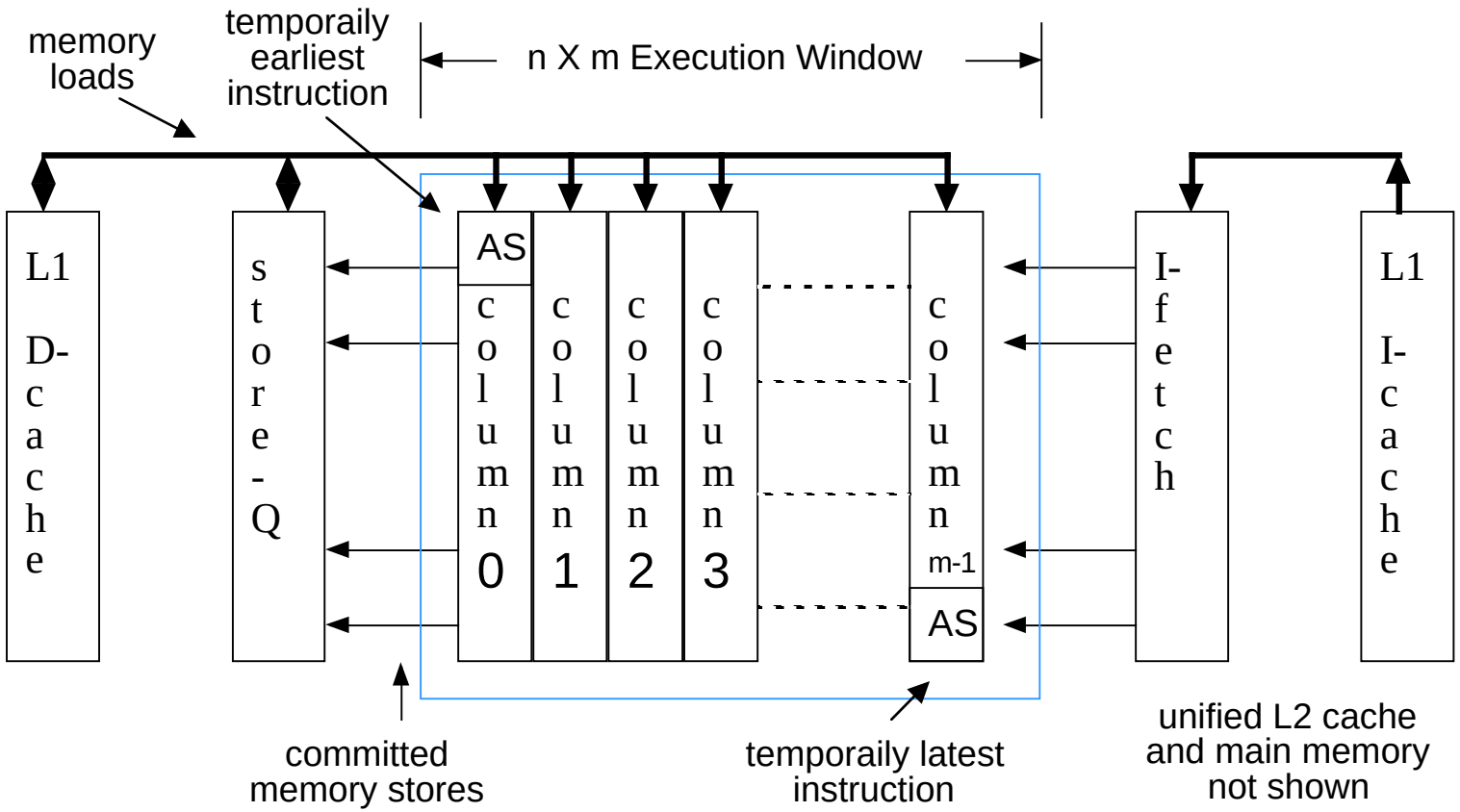
related work

- **Riseman & Foster (1972), Lam & Wilson (1992), others :**
  - explored unconstrained resource use
  - there is much ILP in general purpose sequential codes
  - > 100 (but general machines now only extract about 1-2 in IPC)
- **Sohi & Franklin (1992)**
  - register lifetimes are short, meaning that effective bypassing of the architected register file is possible with limited length buses -- key to spatial scalability of a machine
- **Cleary, et al -- Warp Engine (1995)**
  - used time-tags for program ordering, however they were cumbersome being represented in floating-point
- **newer work on large issue windows (ISCA 2002, others)**
  - but these do not scale and lack distributed execution units

keys ideas used in the machine

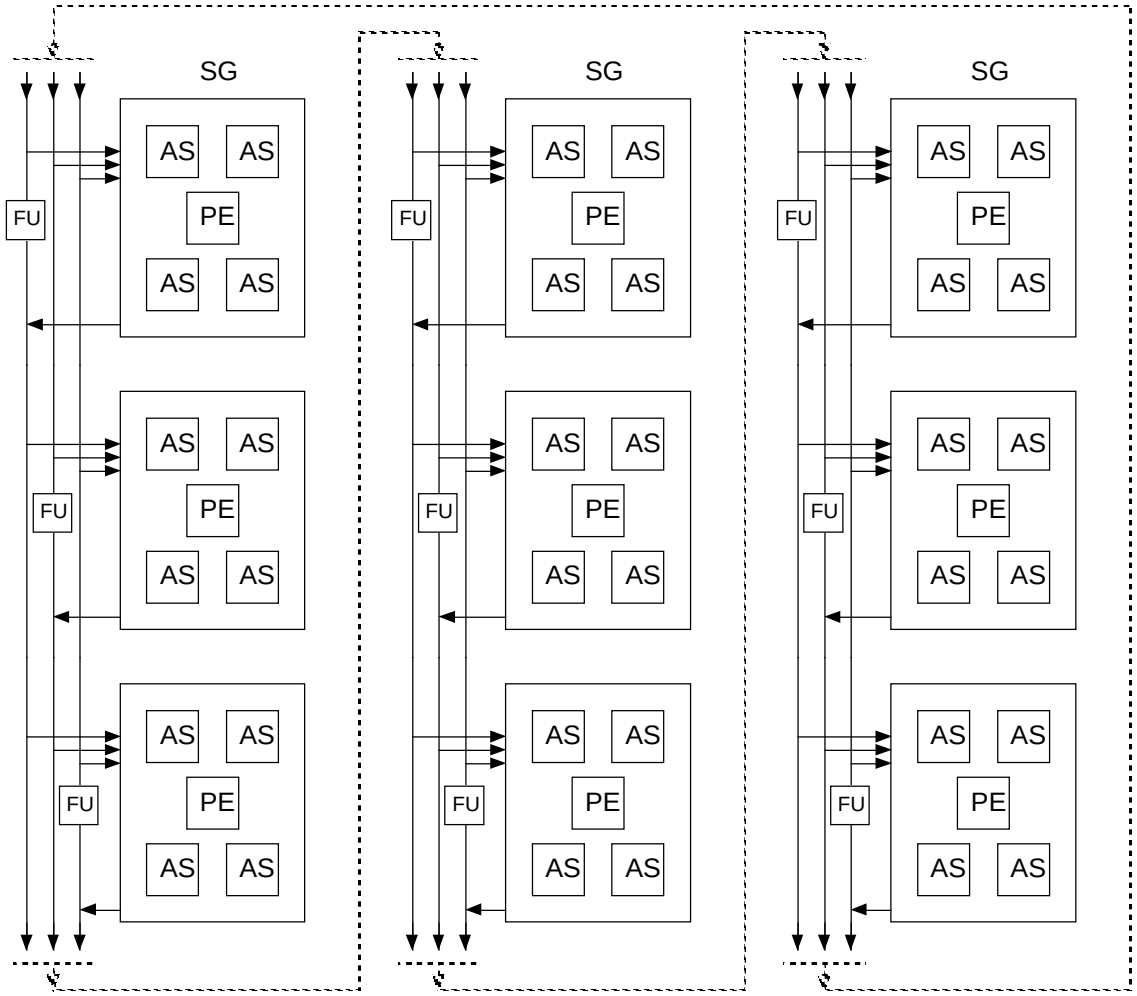
- **modified issue slot or reservation station -- termed "active station" (AS), for managing instruction execution and re-execution !**
- **time-tags for all operand and program ordering**
- **microarchitectural predication for all instructions**
  - facilitates out-of-order execution
  - allows for execution of control-flow independent instructions beyond conditional branches
- **disjoint spawning of alternative (not predicted) execution paths from conditional branches**
- **value speculation for input operands to instructions**
- **L0 caches (small cache close to execution units) distributed throughout an array of execution units**
- **out-of-order execution, only based on priority related to likelihood of instruction commitment, and availability of execution resources ("resource flow computing")**

microarchitecture overview



- **ASes arranged in columns forming the Execution Window**
- **processing elements (PEs) are distributed throughout e-window**
- **columns logically rotate as whole columns are loaded and committed**

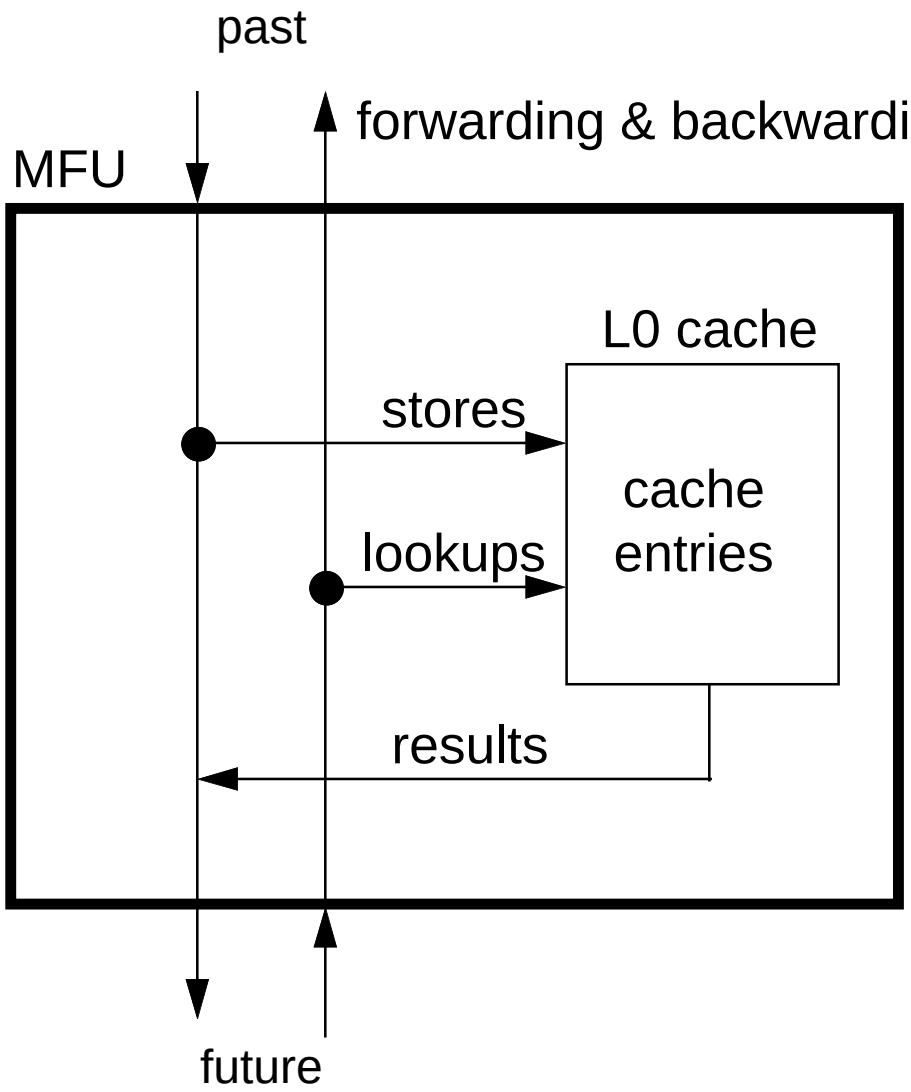
execution window



- **shown: 3 SG rows, 2 AS rows per SG, 3 SG columns**

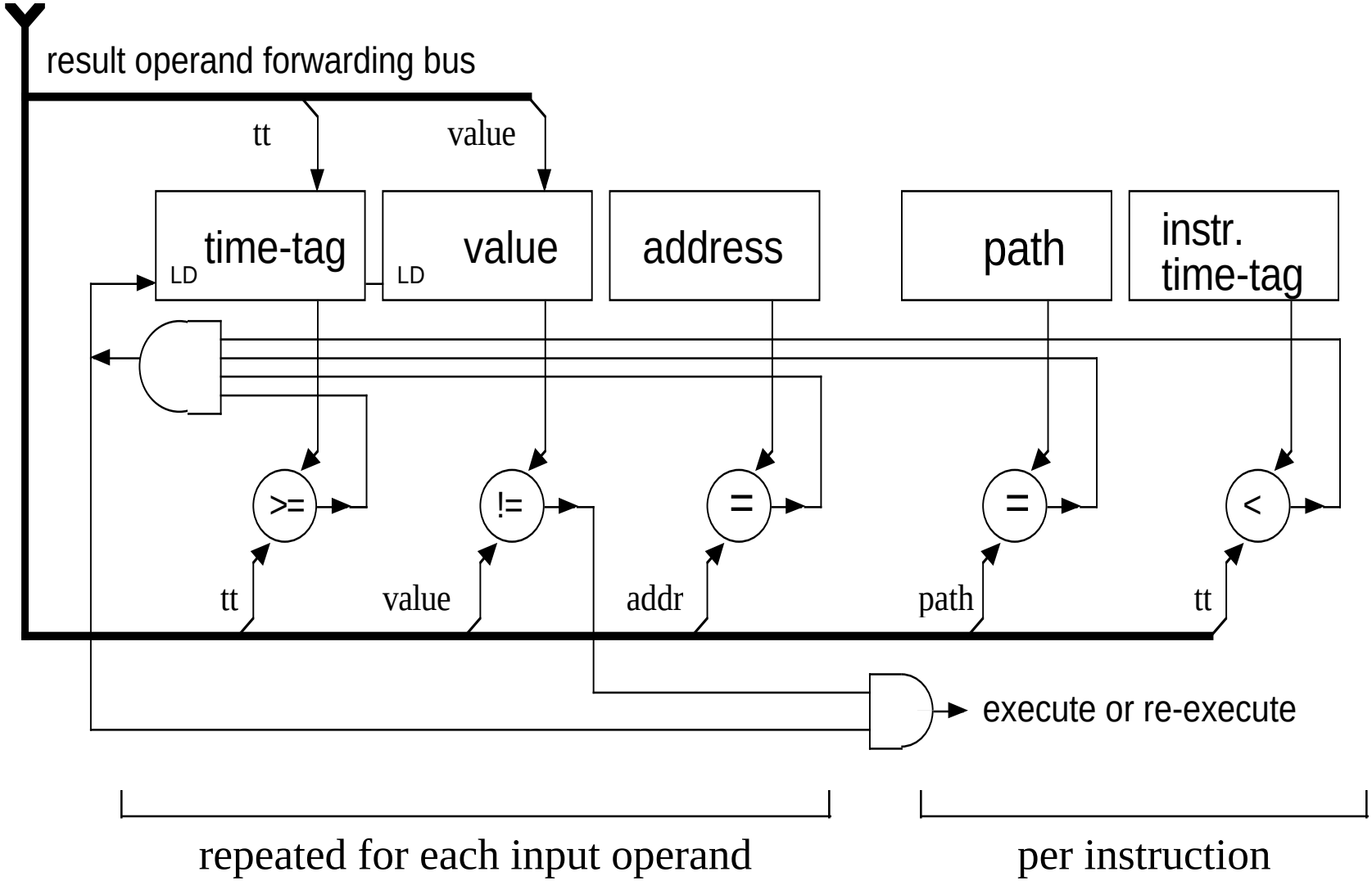
- **ASes and PEs are grouped into Sharing Groups (SGs)**
- **buses are broken into fixed length segments by Forward Units (FUs)**
- **there are different type FUs for each operand type :**
  - register -- RFU
  - memory - MFU
  - predicate -- PFU
- **both forwward and backward interconnections are provided (backward is not shown)**
- **L0 caches are distributed throughout -- one inside each of the MFUs**

a distributed L0 cache



- **MFU provides**
  - forwarding function
  - backward propagation for requests
  - filtering function
  - cache storage through L0 to satisfy backwarding requests
- **L0 may be a variety of configurations**
  - we employed fully-associative but others are possible

snoop/snarf operation



simple execution example (memory)

code fragment

label	*TT*	*instruction*
I1	0	(ma1) <= 1
I2	1	r4 <= (ma1)
I3	2	(ma1) <= 2
I4	3	r5 <= (ma1)

example execution schedule and transactions

cycle	*execute*	*forward*	*snarf*
-1		Ix{(ma1)=?}	I2{(ma1)=?}, I4{(ma1)=?}
0	I1		
1		I1{(ma1)=1}	I2{(ma1)=1}, I4{(ma1)=1}
2	I2, I4		
3	I3	I2{r4=1}, I4{r5=1}	
4		I3{(ma1)=2}	I4{(ma1)=2}
5	I4		
6		I4{r5=2}	

- **"ma1" -- some memory address, "(ma1)" -- some memory value**
- **we want *r4* to have committed value =1, from execution of I1**
- **we want *r5* to have committed value =2, from execution of I4**

- **I4 executes in clock 2 after snarfing *(ma1)* from I1, resulting in *wrong* result**
- **I4 executes again after snarfing *(ma1)* from I3, giving correct result**

experimental methodology

- used a custom simulator to handle distributed multipath execution (handles alternate and wrong paths)
- used the MIPS-1 ISA (some MIPS-2/3 instructions also)
- SGI native C language compiler, '-O' optimization
- used the Irix-6.4 standard libraries within the executables
- evaluated using SpecInt (1995 & 2000) integer benchmarks
- executed all benchmarks for 600M instructions from program start, but only accumulated data over the latter 500M instructions

- we first evaluated IPC
- then we evaluated tolerance to various memory hierarchy latencies
  - L1-D cache
  - L2-unified cache
  - main memory

default machine parameters

L0 cache	32 entries, one word each, fully associative
L1 I/D cache	1 clock hit latency
	64 KB, 32 byte block size
	2-way set associative
L2 cache	unified I/D
	10 clock hit latency
	2 MB, 32 byte block size
	direct mapped
main memory	100 clock latency
memory interleave factor	4
forward units (all)	minimum delay 1 clock
forwarding buses (all)	delay 1 clock
	4 in parallel
branch predictor	PAg. 1024 PBHT, 4096 GPHT, 2-bit sat. counter

IPC results

geometry	8-4-8-8	8-8-8-8	16-8-8-8	32-2-16-16	32-4-16-16
bzip2	4.2	5.0	5.8	5.4	5.7
go	5.1	5.9	6.7	6.5	6.8
gap	6.0	7.5	7.5	8.9	7.9
gzip	5.0	6.3	7.0	6.7	7.2
parser	4.3	4.6	5.3	5.0	5.4
HAR-MEAN	4.8	5.7	6.4	6.3	6.5
speedup over SP	50%	46%	39%	50%	41%

geometry:

- SG rows

- AS rows per SG

- SG columns

- max alternative paths

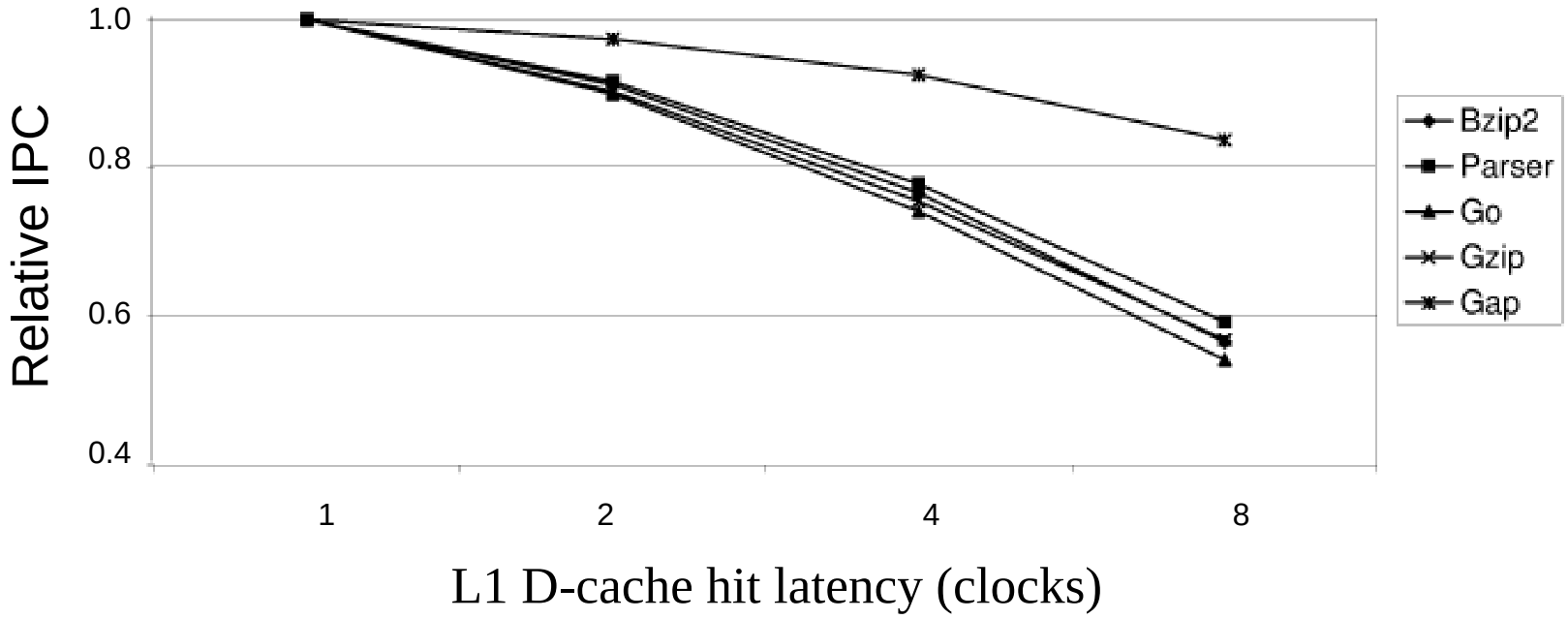
latencies:

L1 1 ck

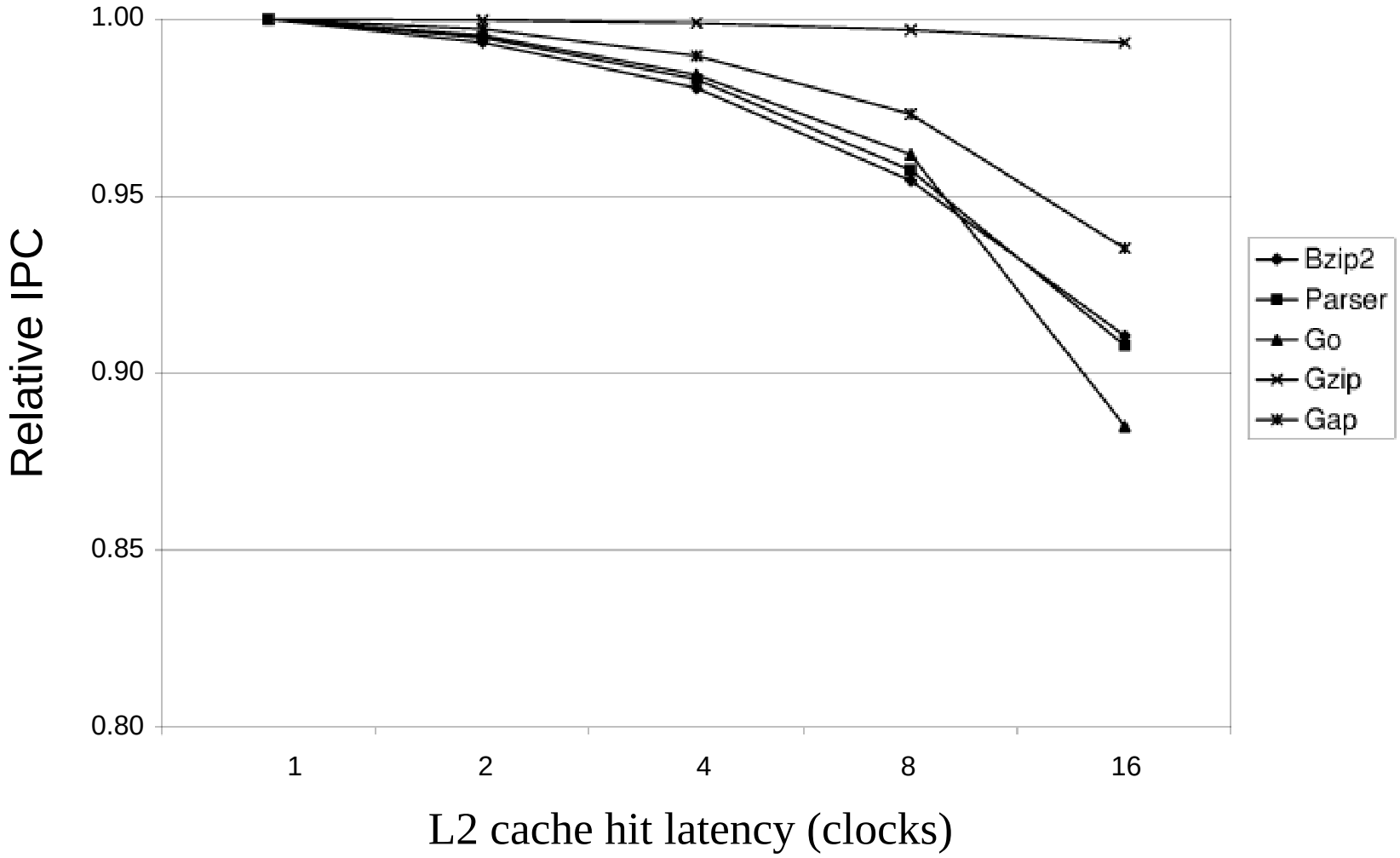
L2 10 cks

main memory 100 cks

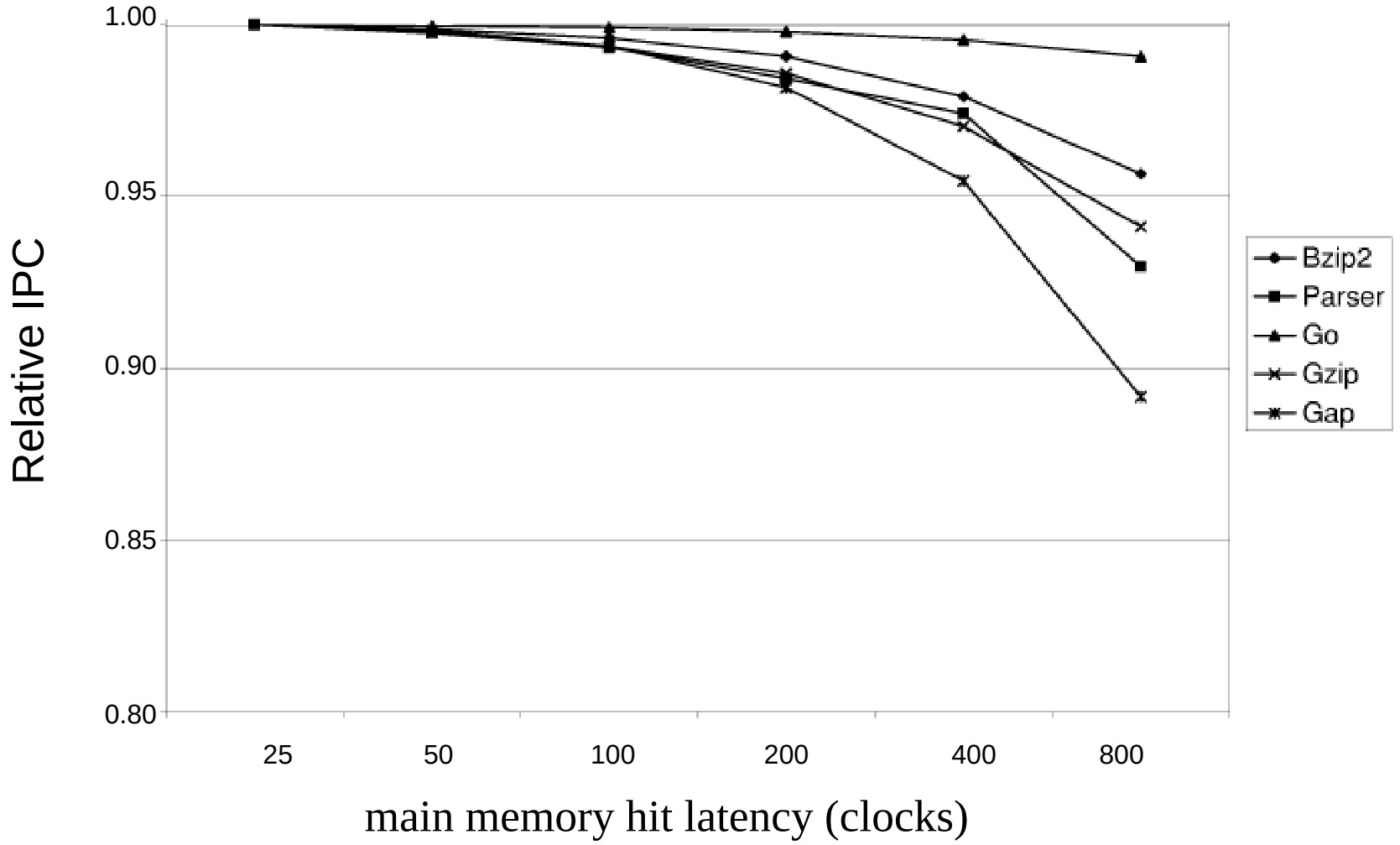
L1-D cache latency tolerance



L2 cache latency tolerance



main-memory latency tolerance



effect of L0 caches

- **memory load requests satisfied from L0 caches**

	bzip2	parser
% of all loads satisfied by L0 due to a backwarding request	3.6%	5.2%
% of all loads satisfied by L0 but w/o any backwarding request	18.2%	28.8%

- **all other memory load requests satisfied from either :**
  - an adjacent instruction in the same sharing group
  - L1 cache and beyond

summary

- **we have described a scalable and distributed microarchitecture**
  - ISA independent
  - time-tags for program ordering
  - microarchitectural predication
  - multipath execution
- **high IPC**
  - with realistic memory hierarchy
  - multipath adds about 40% speed-up over single path
- **shown a tolerance to substantial main-memory latency**
  - about 90% performance retained even out to 800 clock latency main memory
  - due in part to L1 and L2
  - but also due to L0, and operand bypass of L1 within e-window

acknowledgements

- **supported by :**
  - National Science Foundation
  - University of Rhode Island Office of the Provost
  - International Business Machines (IBM)
  - Intel
  - Mentor Graphics
  - Xilinx
  - Ministry of Education, Culture, and Sports of Spain

simple execution example (registers)

code fragment

label	*TT*	*instruction*
I1	0	r3 <= 1
I2	1	r4 <= r3 + 1
I3	2	r3 <= 2
I4	3	r5 <= r3 + 2

example execution schedule

cycle	*execute*	*forward*	*snarf*
-1		Ix{r3=?}, Iy{r4=?}	I2{r3=?}, I4{r3=?}
0	I1	Iz{r5=?}	
1		I1{r3=1}	I2{r3=1}, I4{r3=1}
2	I2, I4		
3	I3	I2{r4=2}, I4{r5=3}	
4		I3{r3=2}	I4{r3=2}
5	I4		
6		I4{r5=4}	

- we want *r3* to have value =2 after execution of I3
- we want *r5* to have value =4 after execution of I4

- I4 executes in clock 2 after snarfing *r3* from I1, resulting in *wrong* result
- I4 executes again after snarfing *r3* from I3, giving correct result