

Understanding, Exploiting and Improving Java Runtime Systems

Lizy Kurian John

Laboratory for Computer Architecture

The University of Texas at Austin

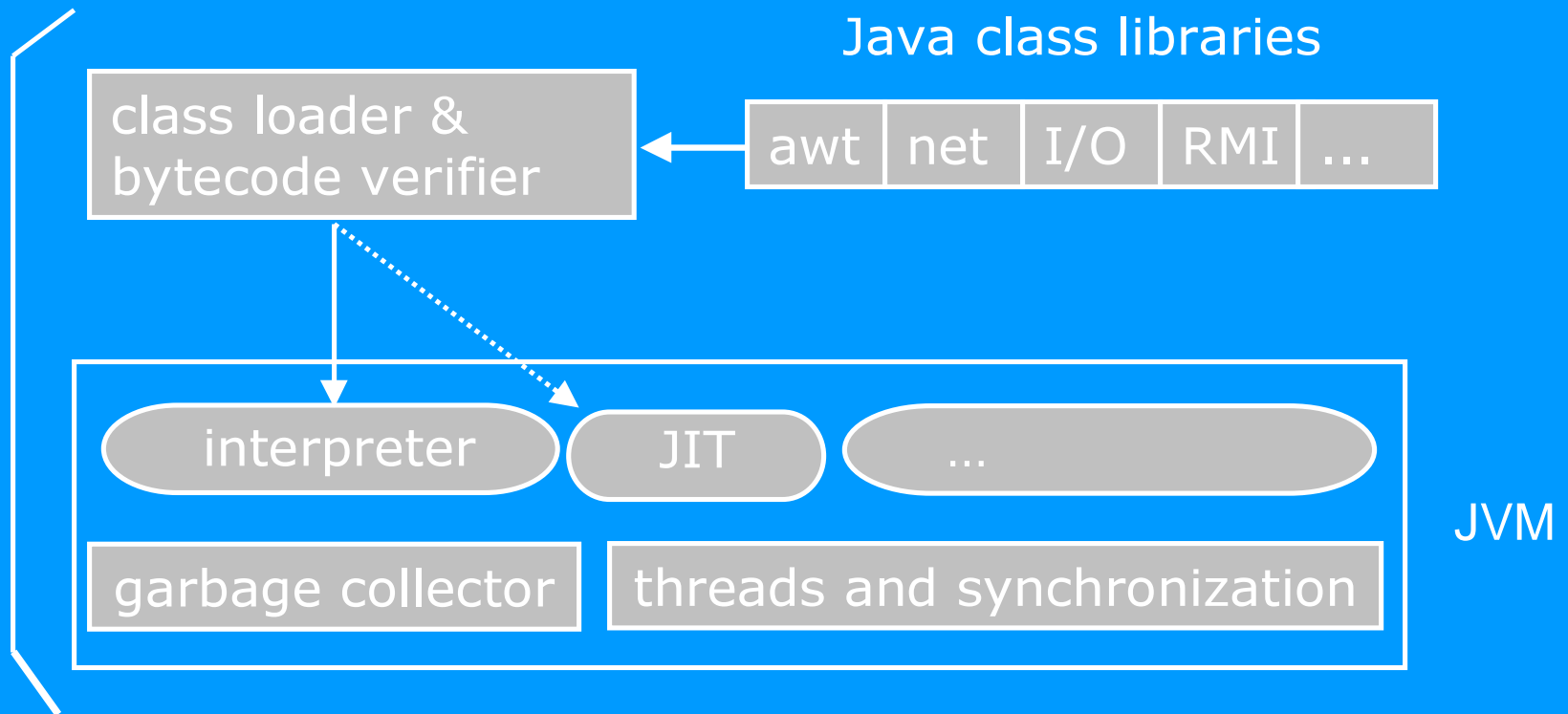
ljohn@ece.utexas.edu

<http://www.ece.utexas.edu/~ljohn>

Collaborators on this work

- Ramesh Radhakrishnan
- Tao Li
- Jyotsna Sabarinathan
- Deepu Talla
- Vijay Narayan
- Anand Sivasubramaniam

Java RunTime Environment (JRE)



JVM functional relationship to JIT compiler and the JRE

Diverse Modes of Execution

- Each mode has strengths and weaknesses
- JIT compilers require increased memory
- Interpreters are typically inefficient
- Java processors are ideal for low-end
- Understanding, exploiting and improving Interpreters, JITs and Java processors is extremely important

Java Research in the Laboratory for Computer Architecture (LCA) at UT

- Unveil execution characteristics of different Java run-time systems
- Identify run-time bottlenecks in the execution
- Derive design enhancements that can improve performance

Important Issues

- Locality and Cache performance
- Branch prediction
- Instruction Level Parallelism
- Operating System Activity

Benchmarks

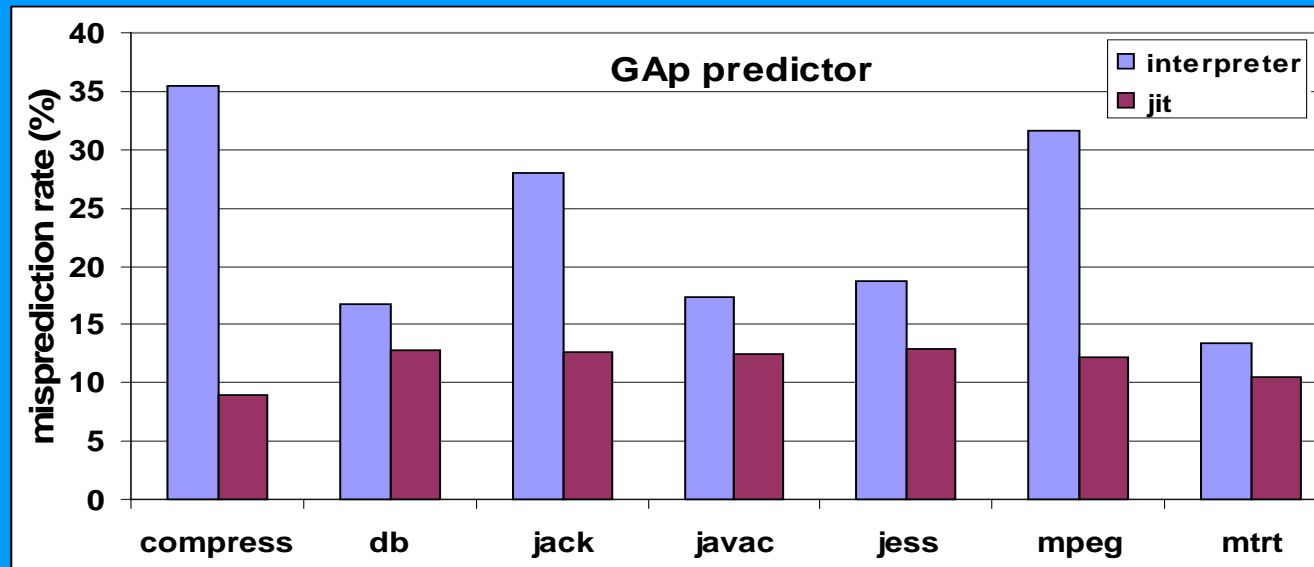
SPECjvm98

compress	A popular LZW compression program
jess	A Java version of NASA's popular CLIPS
db	Data management software written by IBM
jack	A parser-generator from Sun Microsystems
mtrt	A dual-threaded program that ray traces an image file
mpegaudio	The core algorithm for software that decodes an MPEG-3 audio file
javac	The JDK compiler from Sun Microsystems

Experimental environment

- Sun JDK 1.1.6
- Kaffe VM 0.9.2
- SPECjvm98
- Shade, cachesim5
- Supersim superscalar processor simulator
- SimOS Complete System Simulator

Branch Prediction



GAp : 2 Level predictor

[256 entries, 3 correlation bits]

- Interpreter branch prediction accuracy low due to abundance of indirect branches
- Misprediction rates with JIT comparable to traditional programs

Locality and Cache performance

- Instruction and data cache performance for the SpecJVM98

		I-Cache		D-Cache	
		Refs	Misses	Refs	Misses
Compress	interpreter	10425 M	84,398	5365 M	21 M
	JIT	1385 M	218,101	751 M	43 M
jess	interpreter	259 M	179,545	81 M	2.3 M
	JIT	188 M	616,962	45 M	3.7 M
javac	interpreter	199 M	140,239	59 M	1.8 M
	JIT	167 M	469,143	40 M	3.3 M

Locality and Cache performance

- Instruction and data cache performance for the SpecJVM98

		I-Cache		D-Cache	
		Refs	Misses	Refs	Misses
mpeg	interpreter	1314 M	92,439	544 M	1.9 M
	JIT	264 M	355,896	101 M	3.2 M
mtrt	interpreter	1531 M	252,370	521 M	8.6 M
	JIT	942 M	522,692	230 M	16 M
jack	interpreter	2662 M	124,563	1033 M	11 M
	JIT	986 M	1M	298 M	15 M

I-Cache behavior

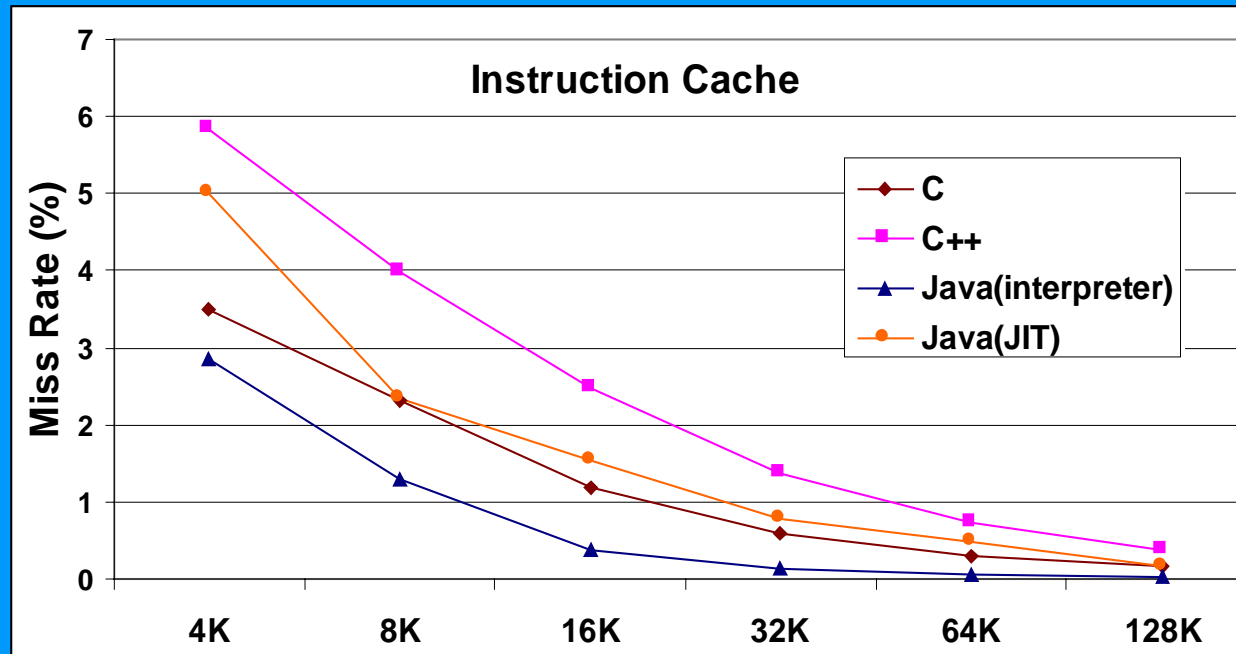


Fig: Miss rates for C, C++ & Java workloads

- Interpreter I-Cache performance is extremely good
- JIT I-Cache performance not as good, but still better than C++

D-Cache behavior

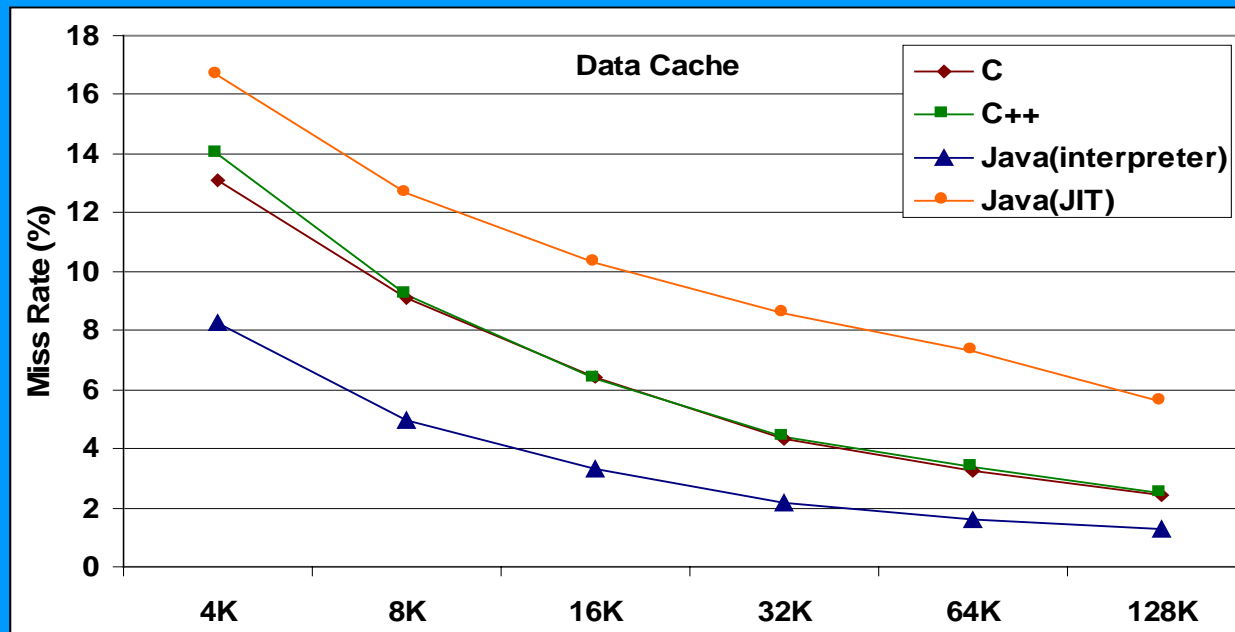
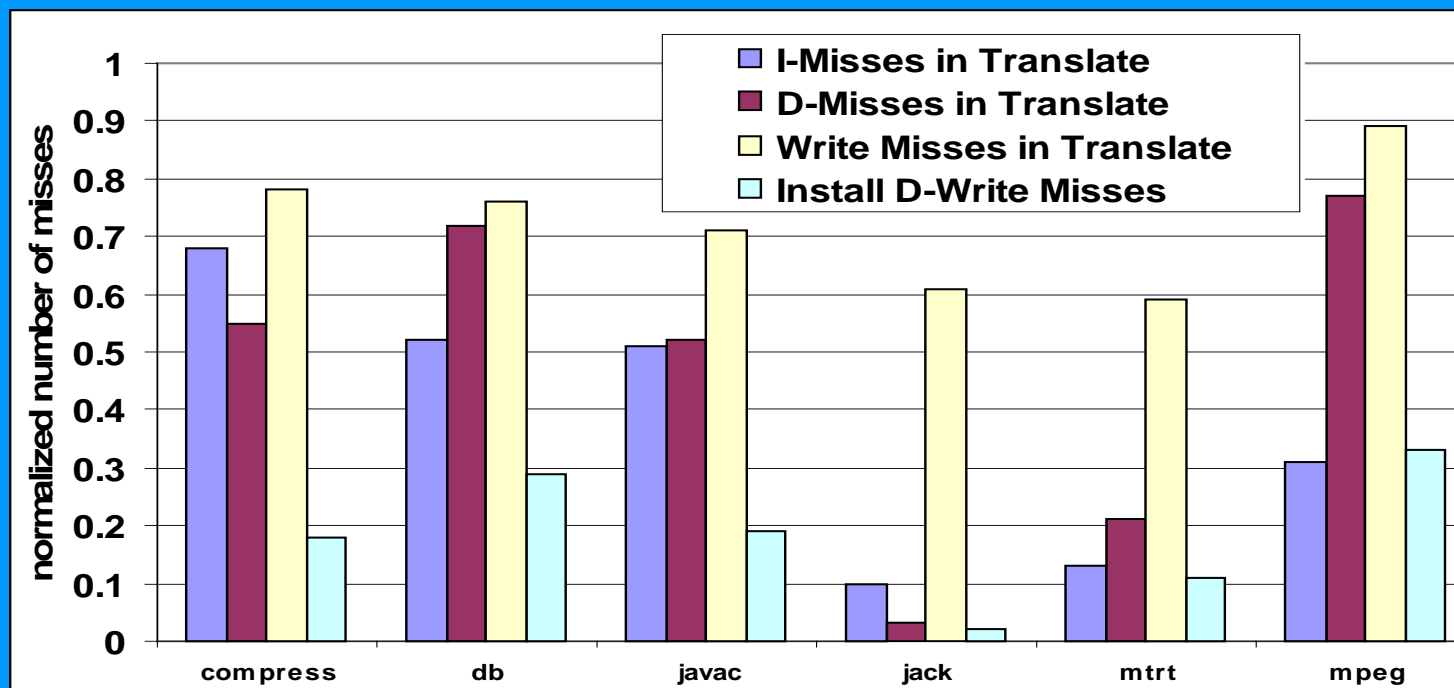


Fig: Miss rates for C, C++ and Java workloads

- Interpreter D-Cache performance is better than C & C++ programs
- JIT D-Cache performance is worse than C & C++ programs

Cache misses in translate

- Cache misses within translate portion of the JIT compiler

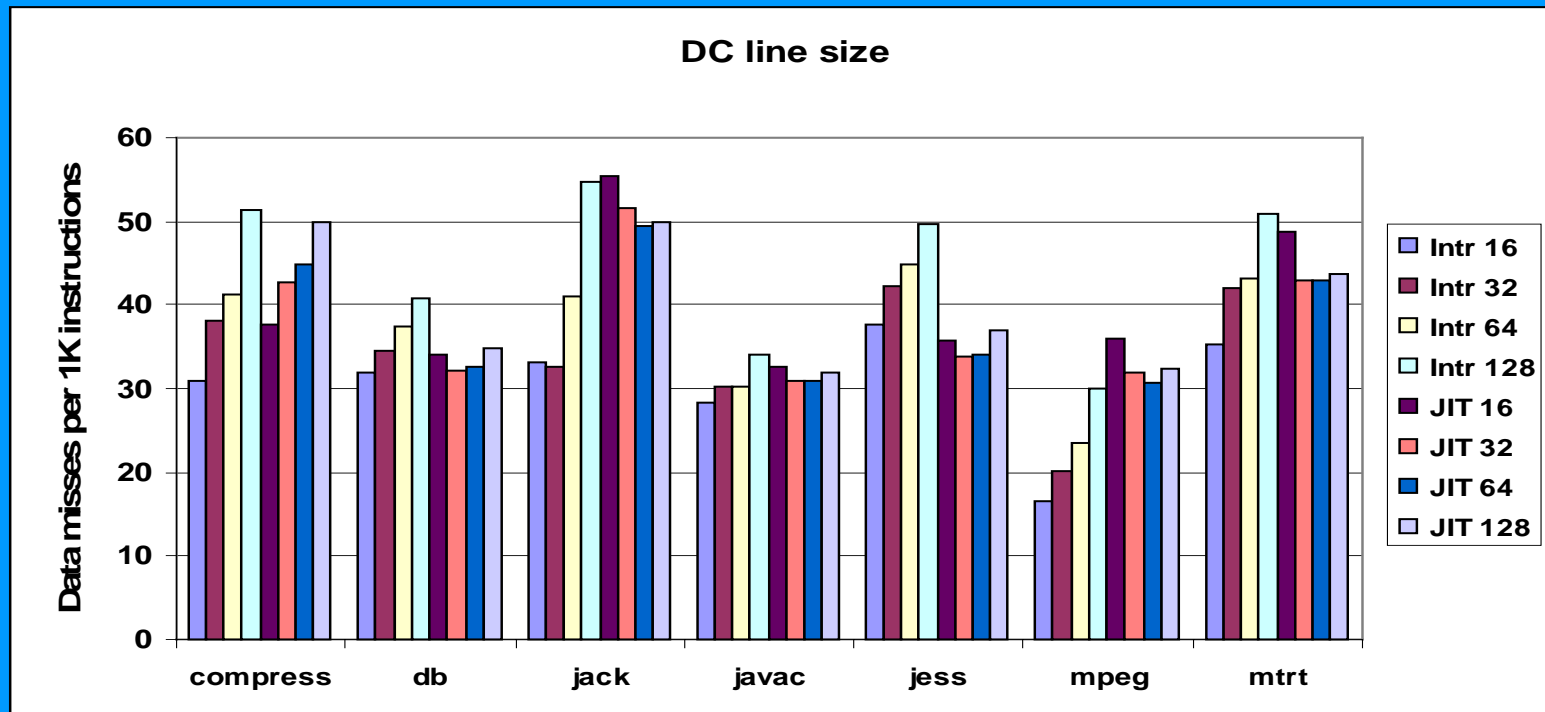


Effect of Cache Parameters

- As usual, increased associativity -> lower miss ratios
- Increased block sizes (I-cache) -> lower miss ratios
- Increased block sizes (D-cache) -> worse miss ratios
- 16 byte block size is often the best.

D-Cache line size

- Effect of changing the line size for D-Cache



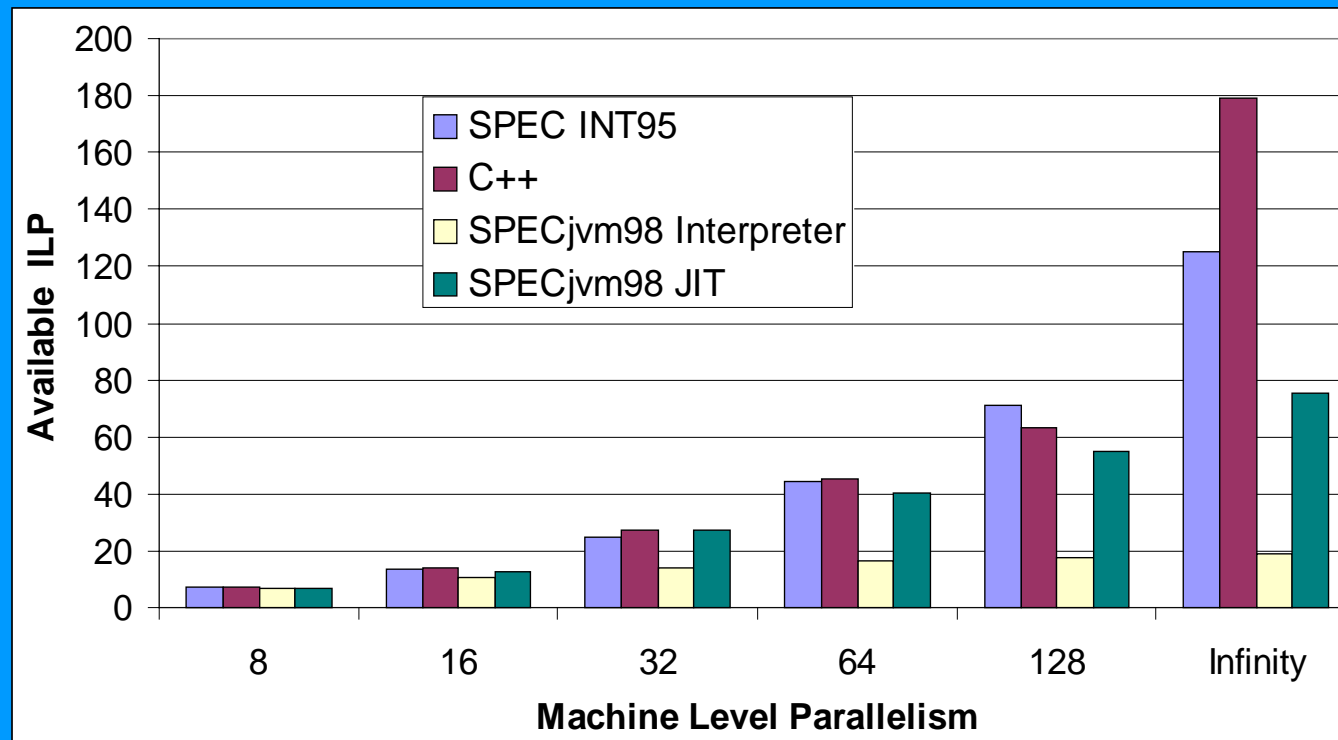
Why small block size is better?

- Data cache locality especially in interpreter depends on method locality and size
- 45% of methods - 1 or 9 bytes long
- Average bytecode size - 1.8 bytes

Data Locality in JIT mode

- Size of objects
- 16 to 23 byte objects very common in SPECjvm98
- character arrays of size 26 to 42 bytes
- The above lead to line size of 32 or 64 bytes being the best

Instruction Level Parallelism



- Stack-based nature of the JVM limits the available parallelism

Dynamic compilation

$$N_i = T_i / I_i - E_i$$

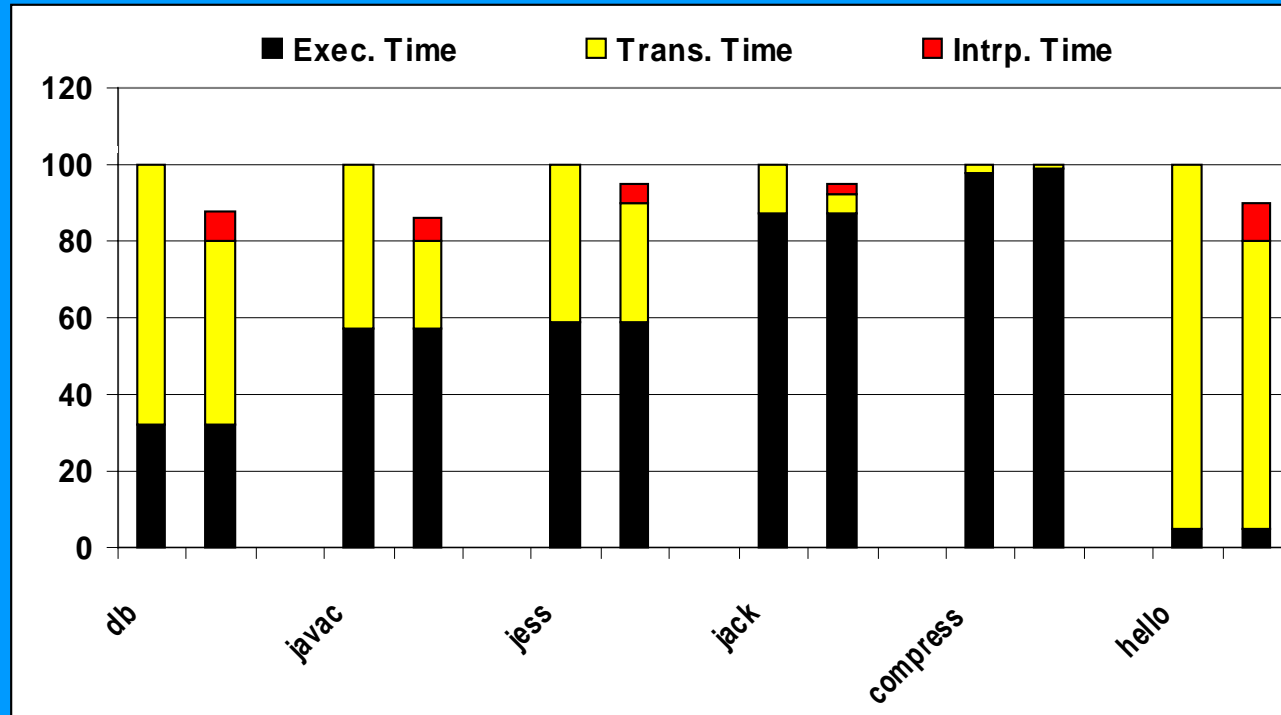
Interpret if $n_i < N_i$

Compile if $n_i > N_i$

Oracle

When or whether to translate?

Dynamic compilation: how well can we do with perfect heuristics ?



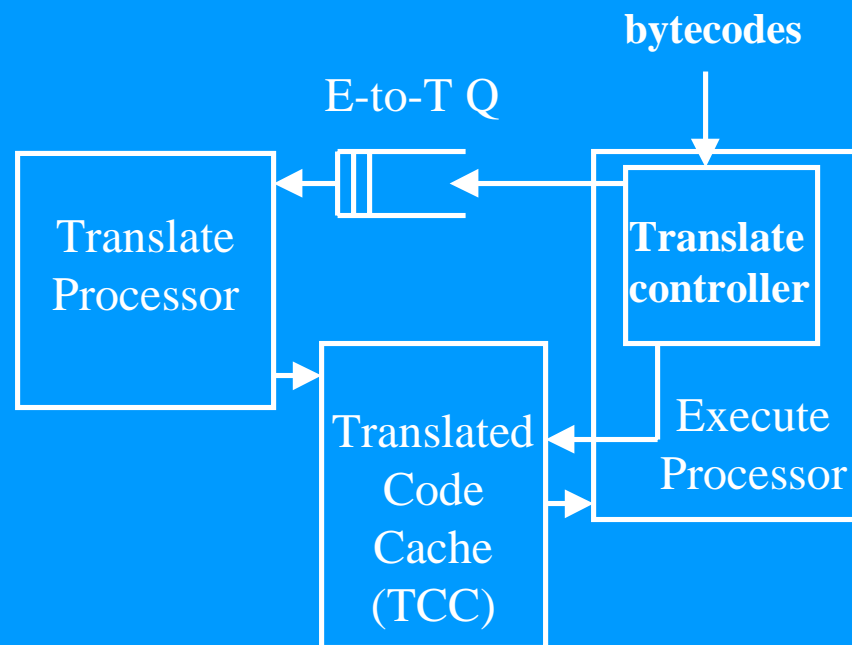
Architectural implications

- Selective translation even using an oracle can improve performance only by 10-15% in the SpecJVM98
- Effort should be expended in trying to find a way of tolerating/hiding the translation overhead

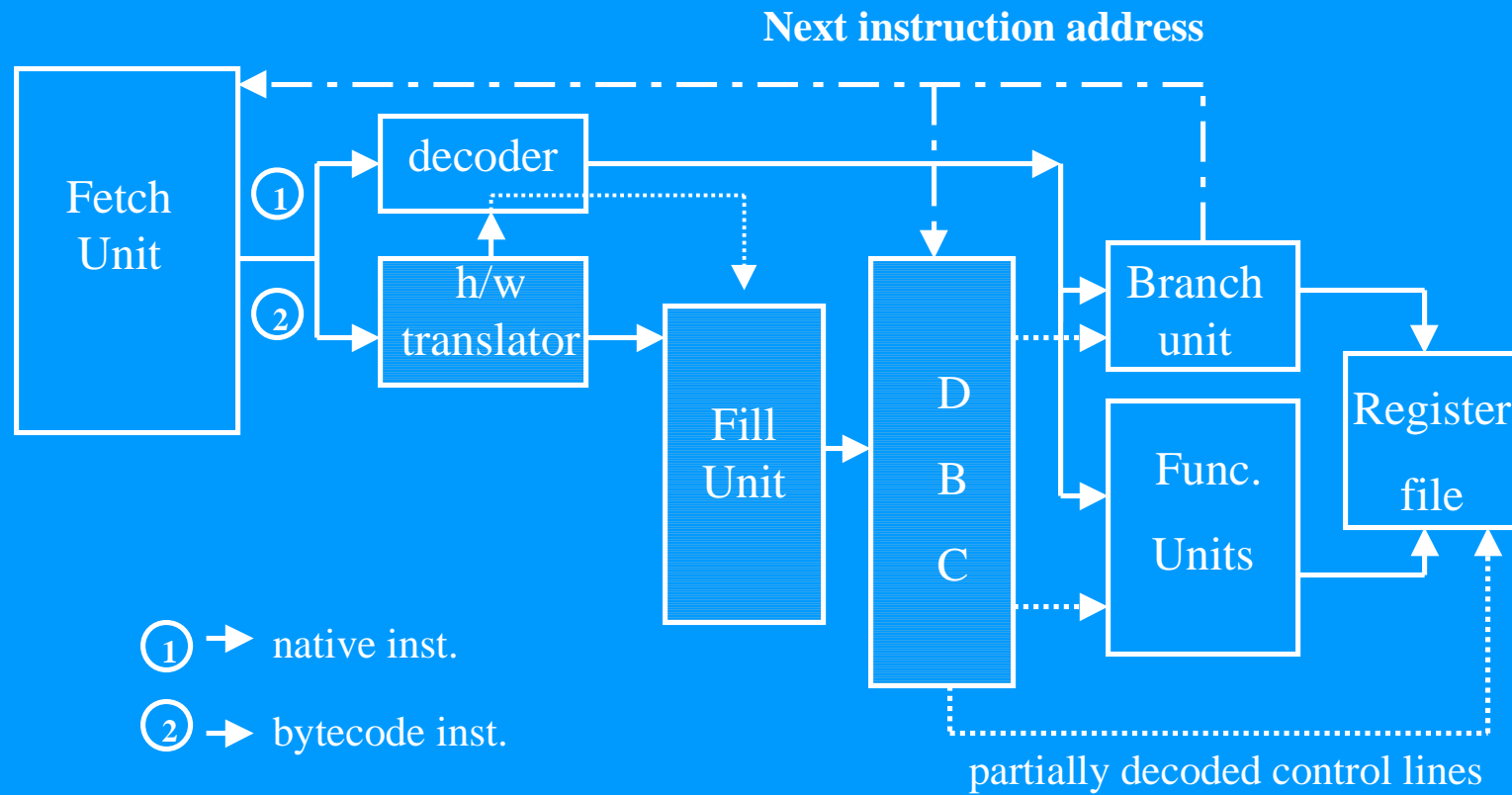
Hardware Translation Support

- Software emulation is expensive
 - interpreter generates 35 SPARC instructions per bytecode
 - JIT generates 20 SPARC instructions per bytecode
- Translation of bytecodes is a critical bottleneck in execution
- Proposal: Perform bytecode translation using hardware

DTE Architecture



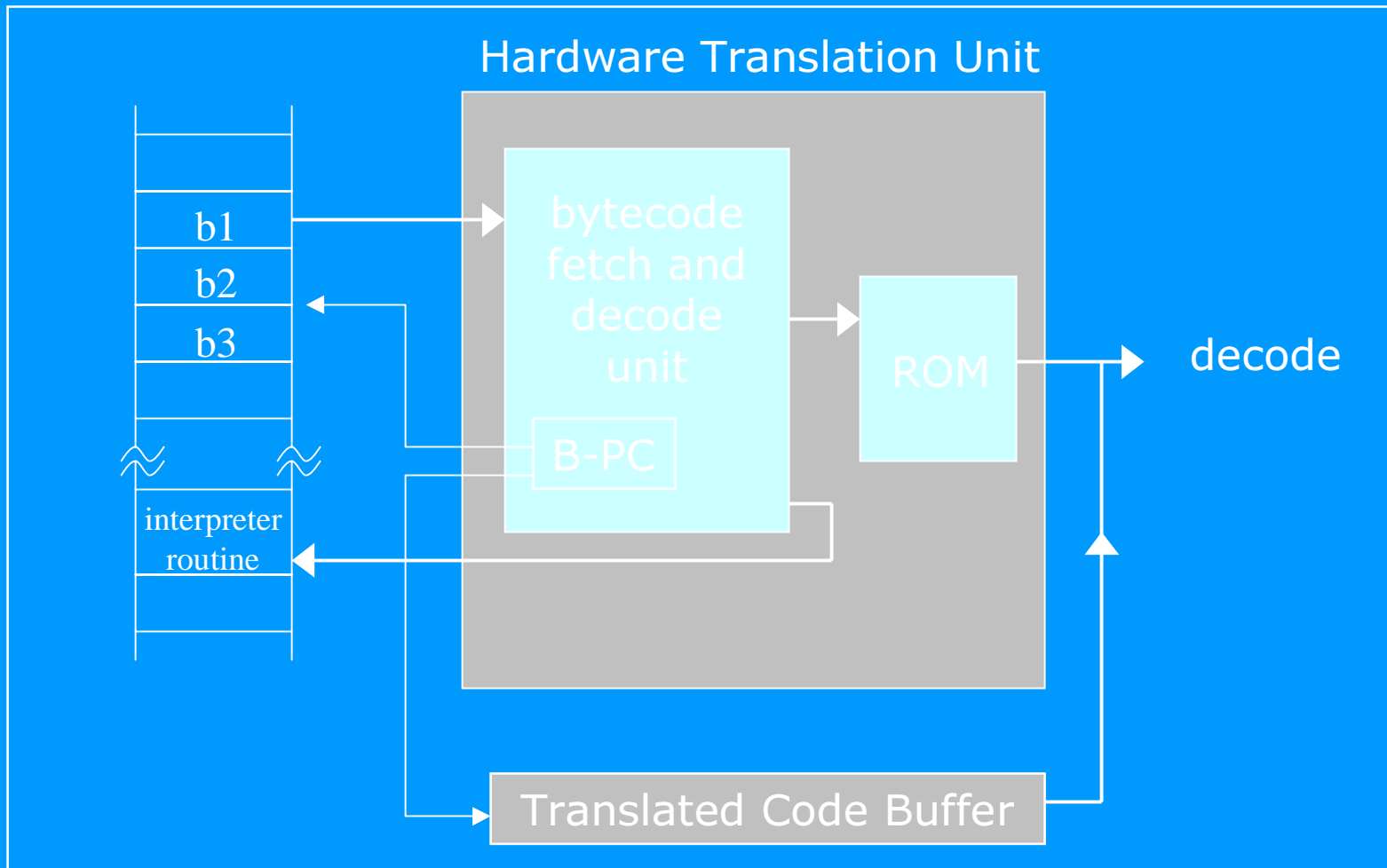
Hard-Int Architecture



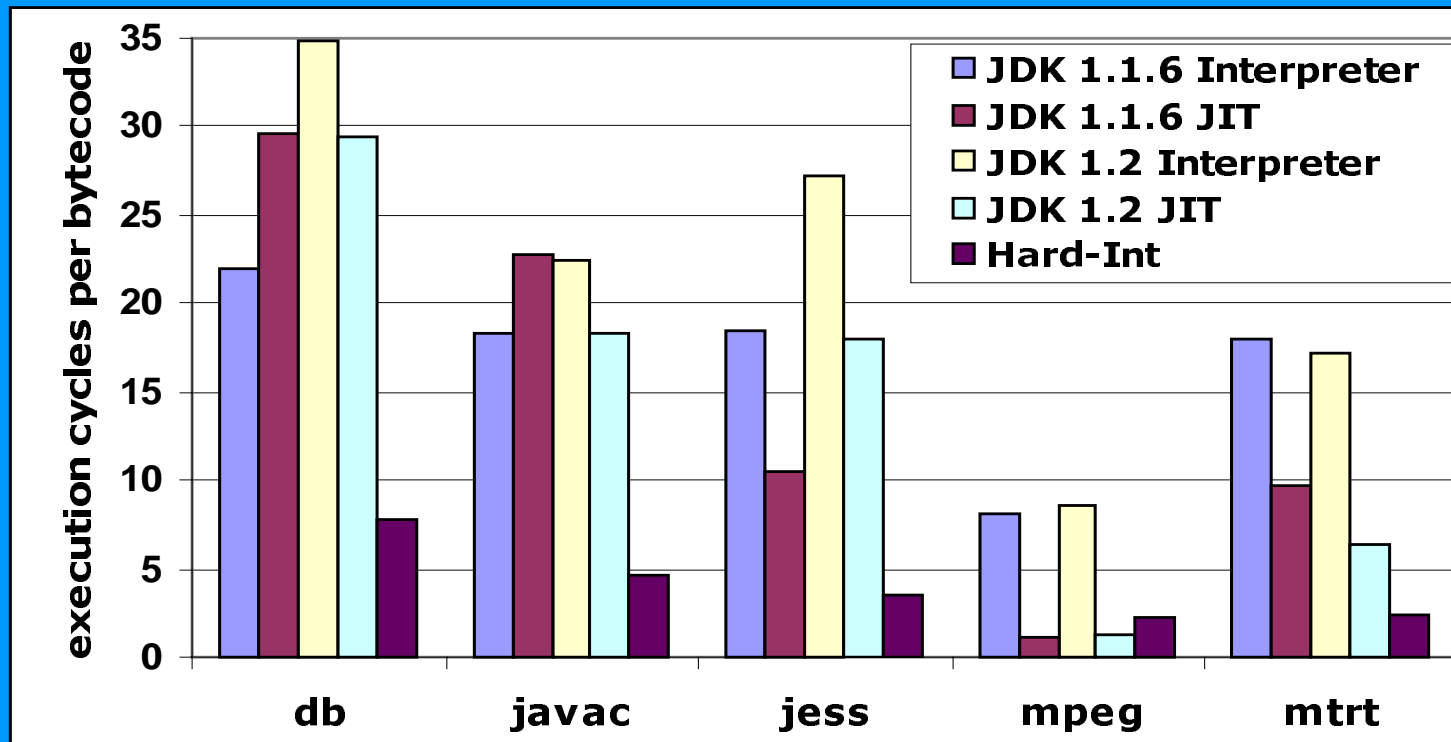
The 15 Most Frequently Used Bytecodes and Their Frequencies (Summary)

Benchmark	The 15 most frequently used bytecodes constitute:
db	59.41%
jack	85.83%
javac	58.97%
jess	56.86%
mtrt	69.54%
mpeg	68.31%

Implementation



Emulation Cost

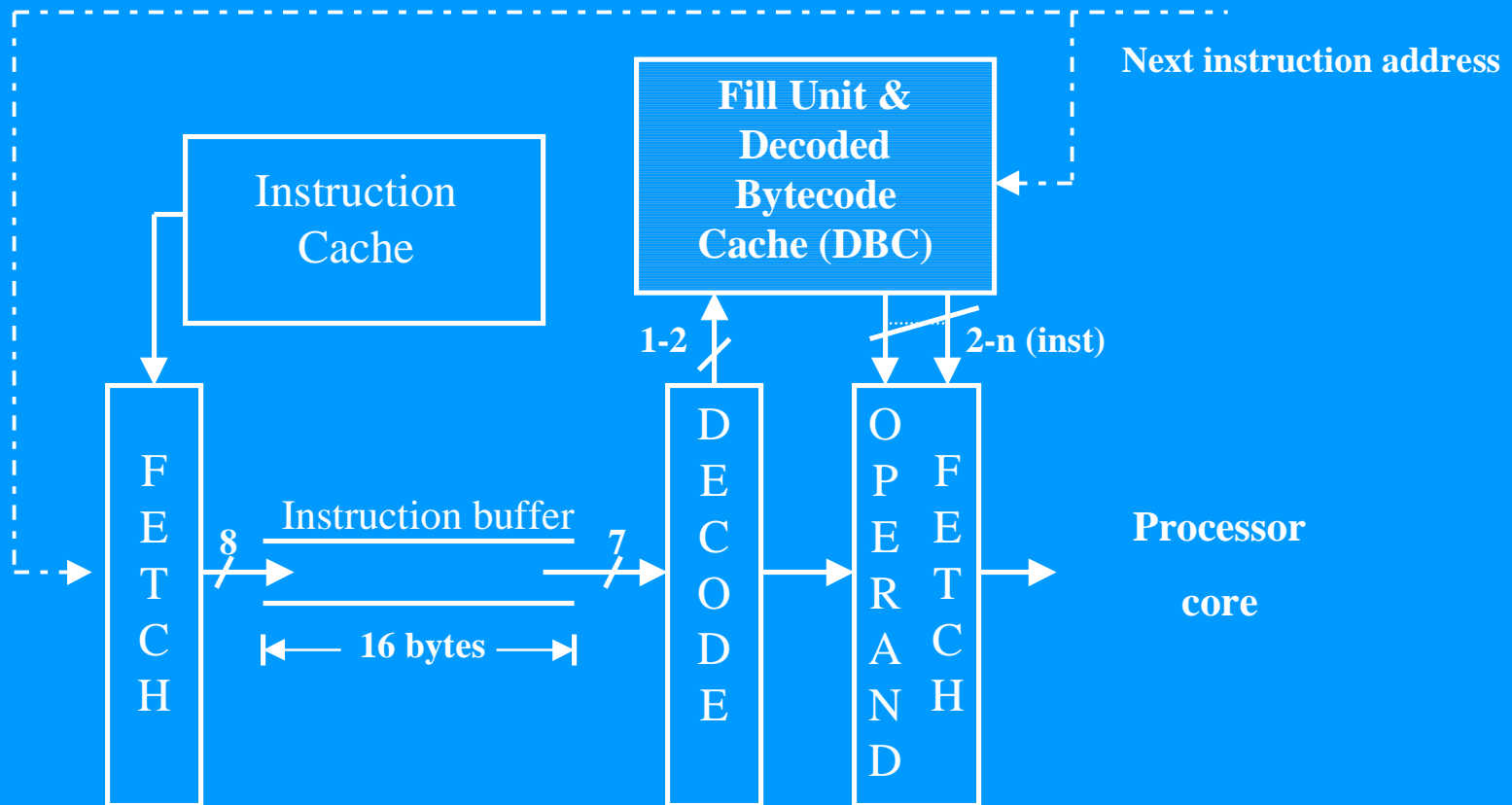


- Cost per bytecode is highest for interpreters

Summary

- Execution speedup
 - speedup of 5.43 (4.39) over JDK 1.2 (1.1.6) interpreter
 - speedup of 2.59 (2.58) over JDK 1.2 (1.1.6) JIT
- Better cache performance
 - reduced instruction and data cache misses
- Lower emulation cost
 - 2.2 to 7.8 cycles per bytecode for Hard-Int

Using a Fill Unit in the picoJava-II

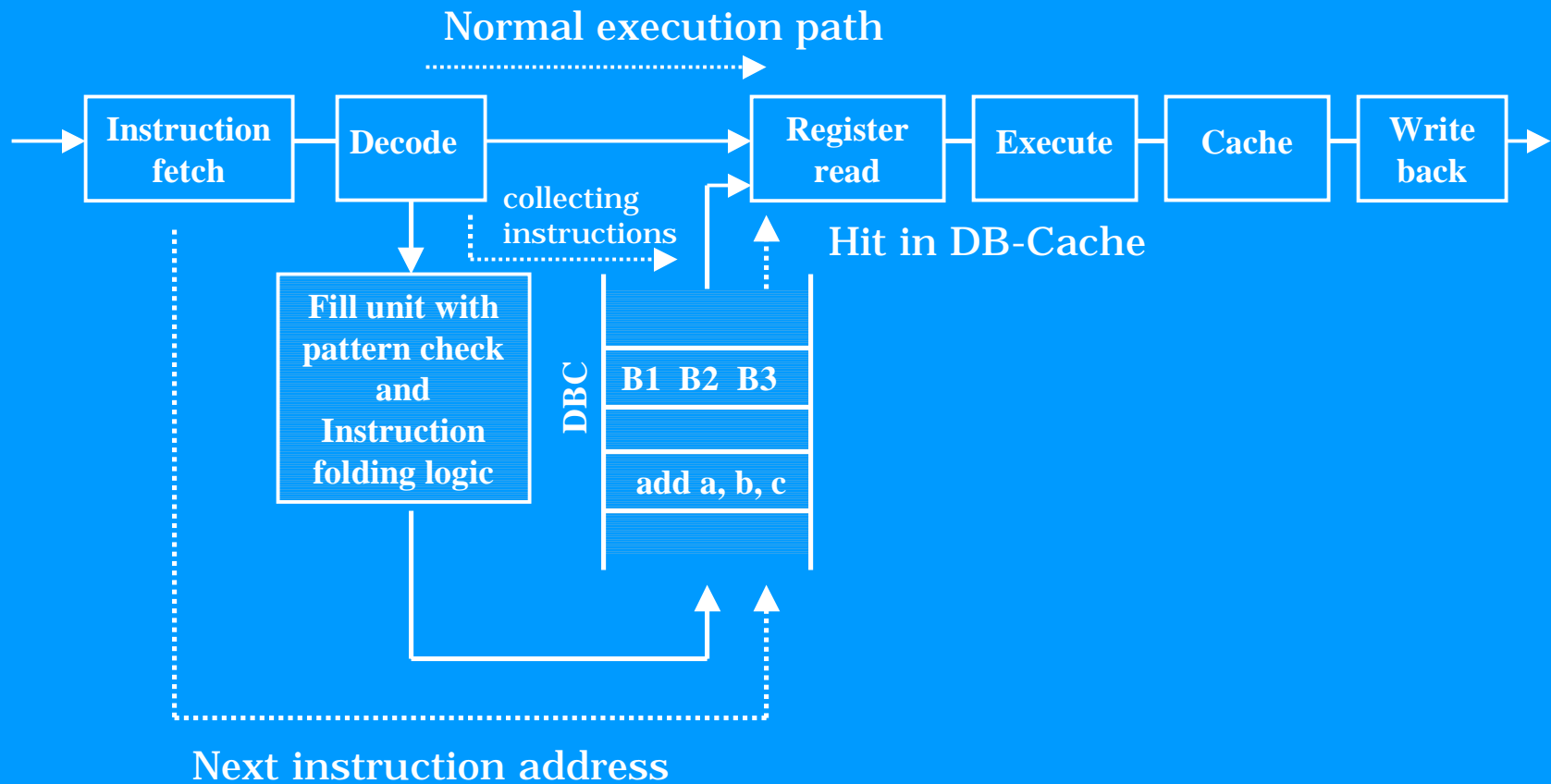


Bytecode Sequence

A sequence of bytecodes to compute the expression `c=a+b`

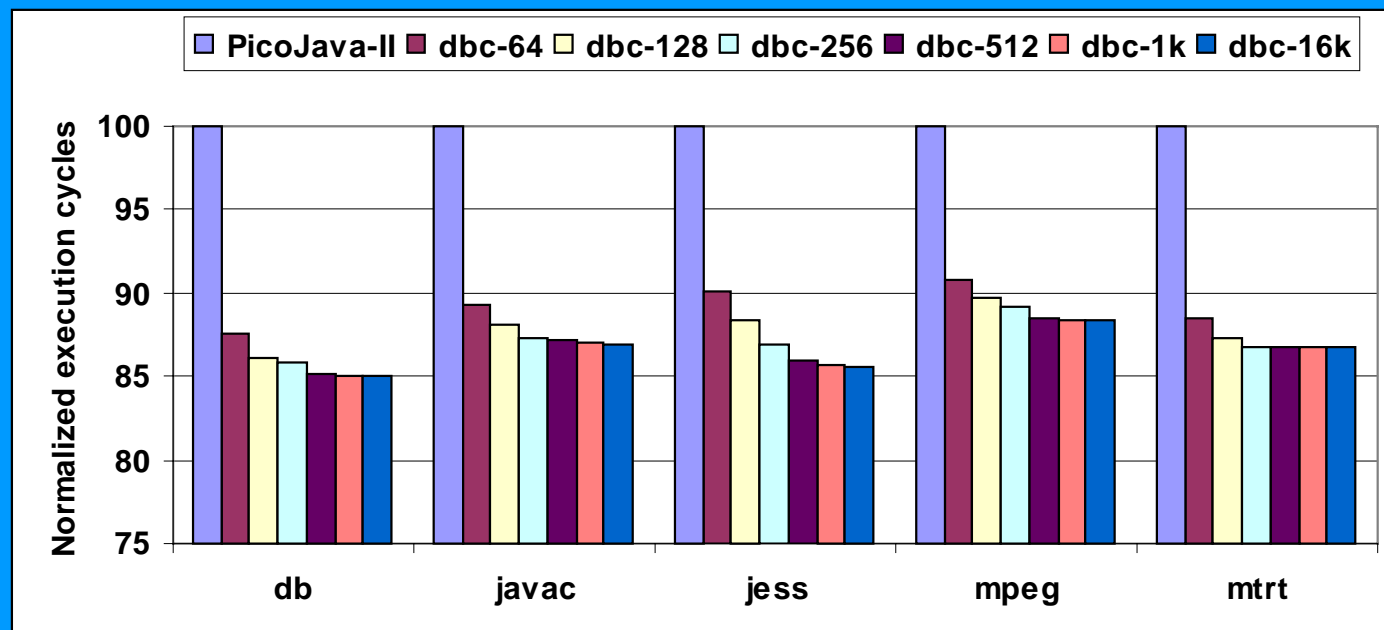
iload_a ; load local variable a into TOS (Top of Stack)
iload_b ; load local variable b into TOS
iadd ; pop top 2 elements on stack and add them.
Push result to TOS
istore_c ; store the element on TOS to local variable c

Instruction folding using the fill unit



Increasing performance using a fill unit

- Performance improvement when adding a fill unit, DB-Cache (64-16K entries) and increasing the execute width to two in a picoJava-II processor.



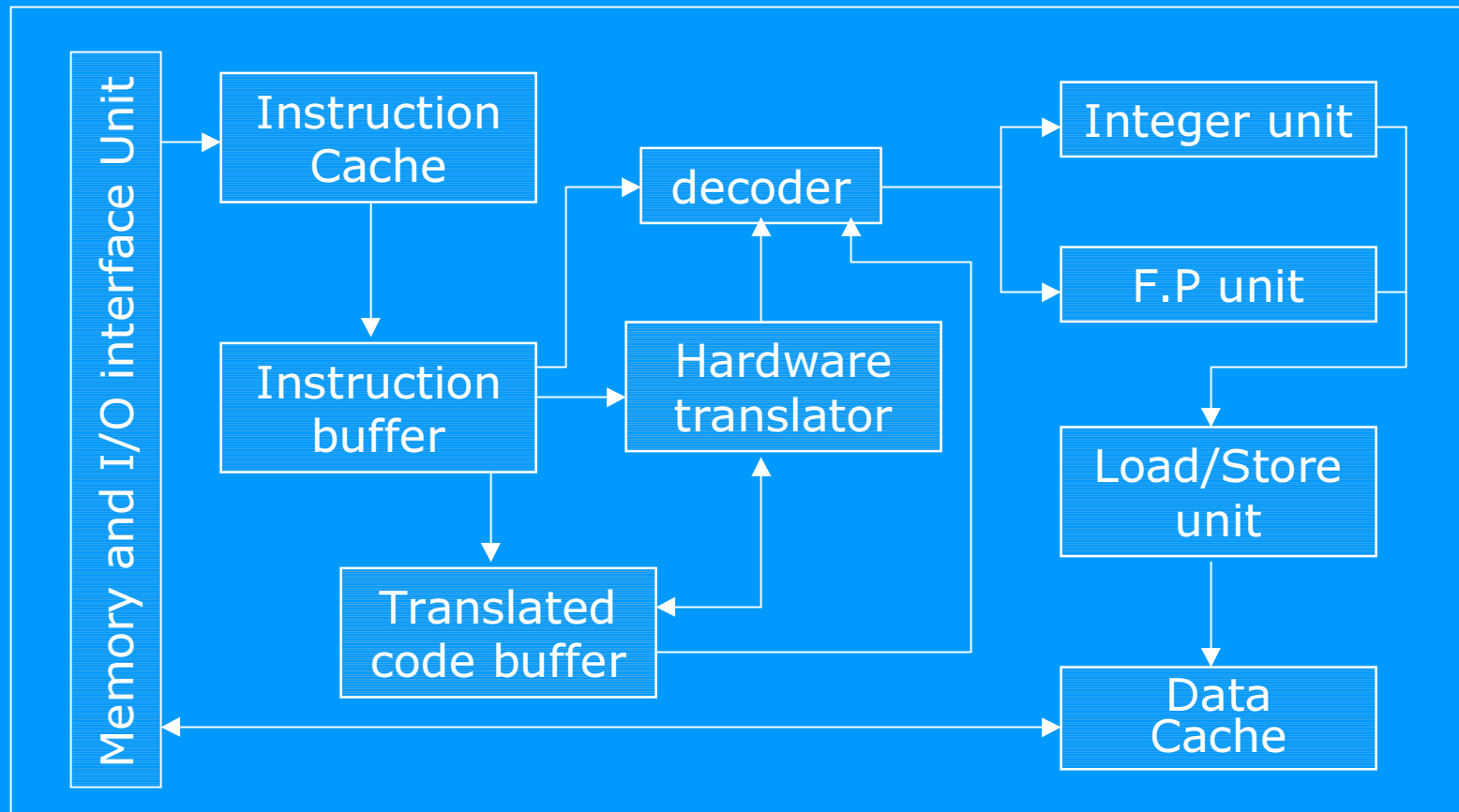
Conclusion

Are objects to be blamed?

- Cache behavior - partly yes
- Cache block size effects - yes
- Branch behavior - not really - culprit is interpretation
- ILP - not really - culprit is the stack machine

Back-up Slides

Hard-Int Architecture



Bytecode Mapping

- Map important variables to registers
 - Stack Pointer (SP), VarBase, Temp *etc.*
- Example:

swap

ld [SP], TOSB

ld [SP-4], TOSA

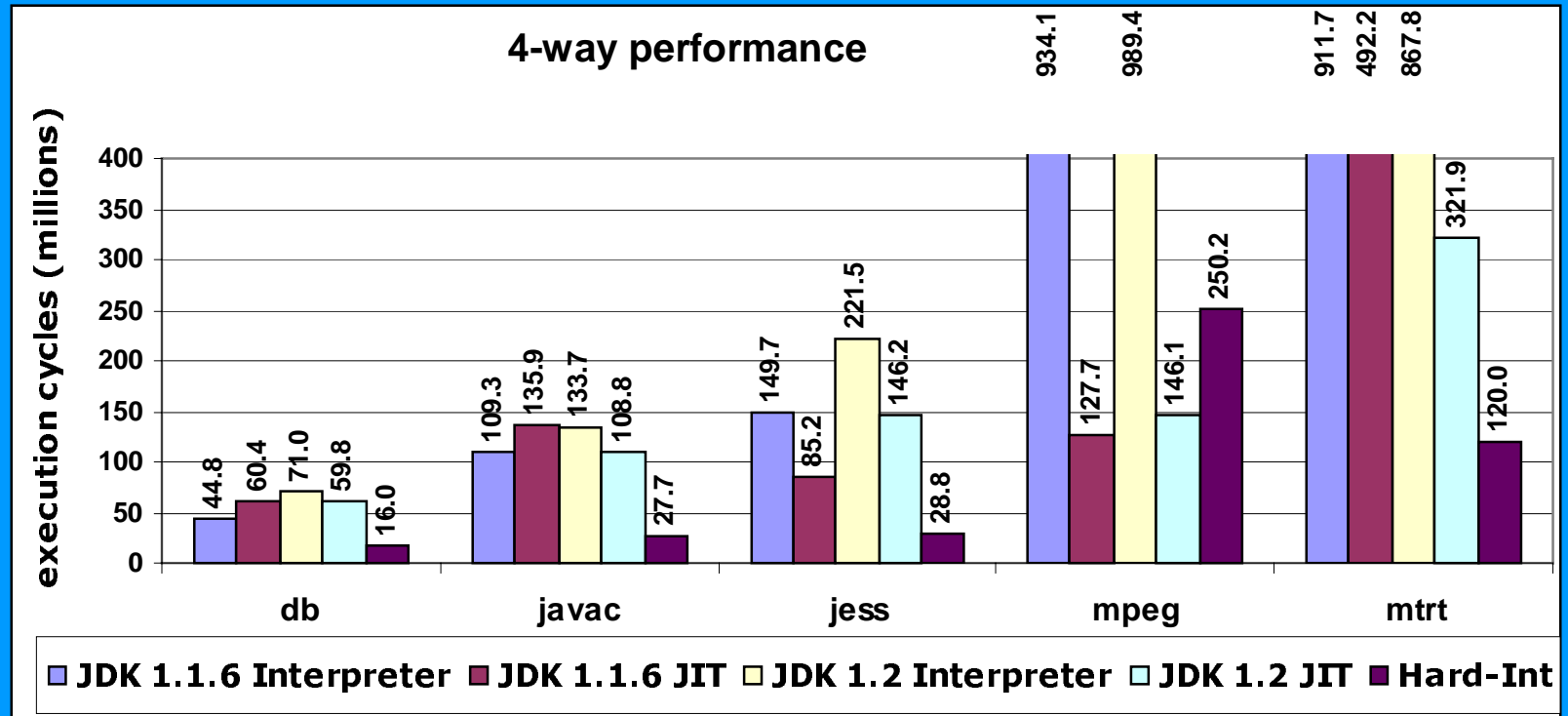
st TOSB, [SP-4]

st TOSA, [SP]

Evaluation

- 4 and 16 way superscalar processors
 - 3 cycle ROM lookup
 - 2 cycle translated code cache access
- Translated code buffer
 - 4-way, 64 entry; up to 16 SPARC instructions in each line
- JDK 1.1.6 and 1.2 (interpreter and JIT) run on the same configuration

Execution speedup



- Hard-Int performs consistently better than the interpreter
- Degradation is observed for some server-like benchmarks

Instruction Mix at the Bytecode Level

Instruction Group	BENCHMARKS						
	compress	jess	db	javac	mpegaudio	mtrt	jack
Constant Pool	19.1%	17.5%	13.1%	11.9%	10.2%	16.3%	26.2%
Stack	13.0%	7.7%	11.4%	8.5%	14.0%	8.3%	19.4%
Load	34.3%	35.5%	37.8%	37.9%	44.2%	28.2%	30.9%
Store	10.6%	6.6%	8.0%	7.5%	8.3%	3.5%	2.1%
ALU	11.2%	6.1%	8.8%	12.8%	17.1%	7.8%	5.8%
Branch	6.1%	9.6%	10.2%	8.6%	3.4%	5.1%	11.0%
Jump	0.4%	1.1%	1.1%	1.3%	0.4%	0.8%	0.5%
Method Calls	5.4%	15.7%	9.2%	10.8%	2.5%	29.3%	4.1%
Table	0.0%	0.3%	0.3%	0.7%	0.0%	0.7%	0.0%
Total Bytecodes (in thousands)	954990	8126	2035	5958	115748	50683	175740

Distinct Bytecodes that Account for 90% of the Dynamic Count

Benchmark	Number of distinct bytecodes
compress	30
jess	48
db	45
javac	45
mpegaudio	36
mtrt	39
jack	22

The 15 Most Frequently Used Bytecodes and Their Frequencies (Slide 1)

	db		jack	
1	iload	10.67%	aload_0	20.34%
2	aload_0	8.38%	getfield_quick	15.62%
3	getfield_quick	7.37%	putfield_quick	9.04%
4	invokevirtual_quick	3.99%	dup_x1	8.52%
5	aload	3.64%	iconst_1	4.77%
6	bipush	3.54%	dup	4.62%
7	iadd	3.45%	aload	4.41%
8	iload_1	3.22%	ifgt	4.35%
9	istore	2.29%	ifnull	4.33%
10	if_icmplt	2.27%	isub	4.33%
11	iload_3	2.27%	iload	1.55%
12	iload_2	2.27%	invokevirtual_quick	1.26%
13	iinc	2.09%	aload_1	1.01%
14	iconst_1	2.07%	iload_1	0.86%
15	dup	1.90%	if_icmplt	0.81%
	Total	59.41%	Total	85.83%

The 15 Most Frequently Used Bytecodes and Their Frequencies (Slide 2)

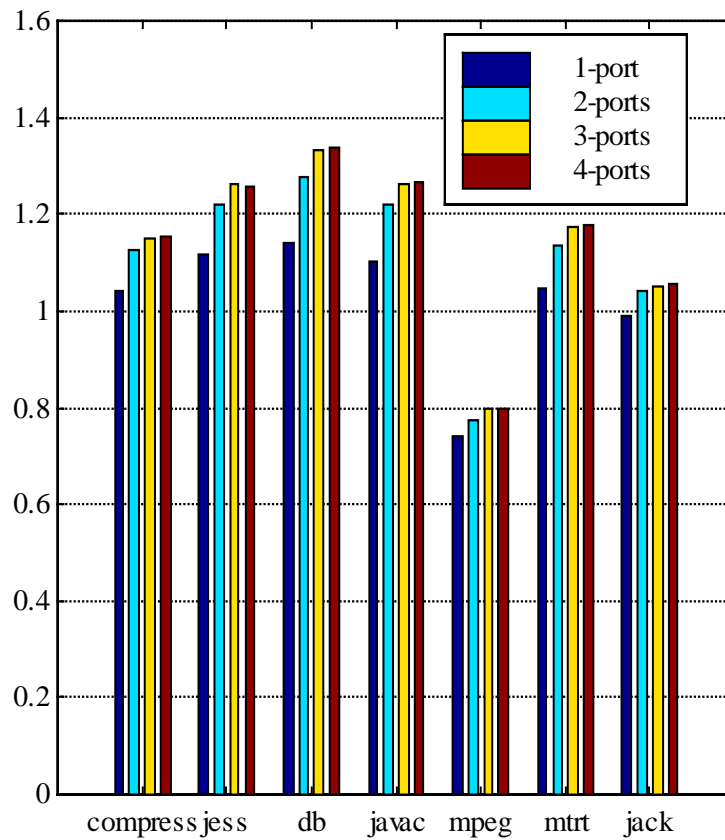
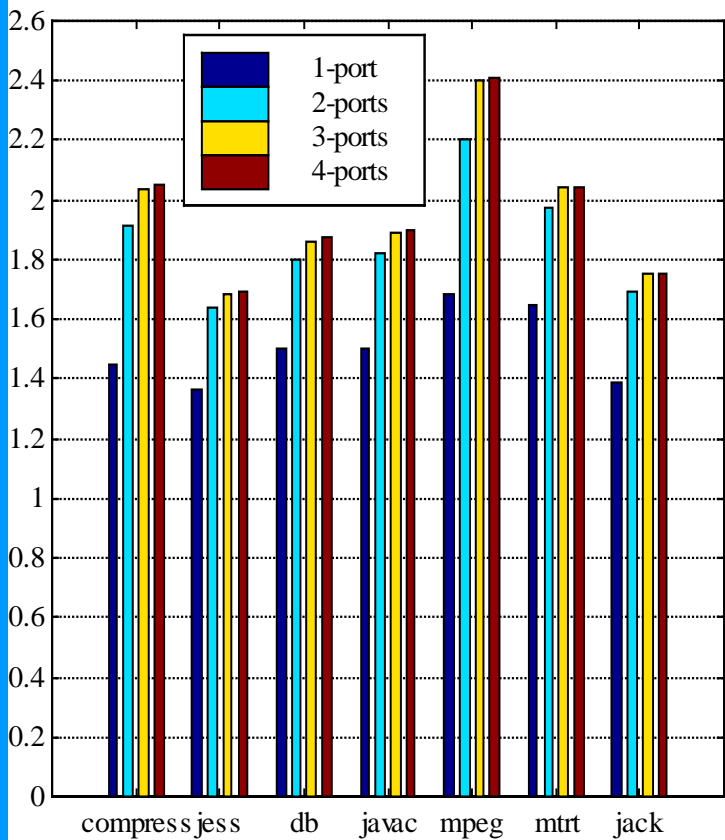
	javac		jess	
1	iload	9.21%	getfield_quick	10.66%
2	aload_0	7.19%	aload_0	9.59%
3	getfield_quick	6.61%	invokevirtual_quick	6.45%
4	iinc	4.46%	iload	4.78%
5	invokevirtual_quick	4.03%	aload_1	4.05%
6	iload_1	3.82%	iload_2	2.99%
7	iadd	3.34%	aload	2.90%
8	aload	3.17%	iload_1	2.65%
9	iload_3	2.96%	iload_3	2.11%
10	if_icmplt	2.58%	iinc	1.91%
11	iload_2	2.56%	if_icmplt	1.85%
12	caload	2.55%	aload	1.79%
13	ireturn	2.33%	getfield_quick_w	1.73%
14	aload_1	2.13%	iconst_1	1.70%
15	istore	2.05%	ireturn	1.70%
	Total	58.97%	Total	56.86%

The 15 Most Frequently Used Bytecodes and Their Frequencies (Slide 3)

	mtrt		mpeg	
1	invokevirtual_quick	18.00%	iload	10.85%
2	getfield_quick	12.83%	faload	8.59%
3	aload_0	6.42%	getfield_quick	6.90%
4	freturn	4.31%	aload_0	6.41%
5	iload	3.83%	bipush	5.44%
6	areturn	3.46%	fmul	3.82%
7	aload	3.23%	fadd	3.69%
8	aaload	3.00%	fload	3.58%
9	putfield_quick	2.58%	aload_1	3.16%
10	aload_1	2.41%	aaload	3.10%
11	iload_1	2.35%	aload	2.95%
12	iinc	2.14%	fstore	2.84%
13	if_icmplt	1.76%	iadd	2.81%
14	dup	1.76%	iconst_1	2.41%
15	iconst_0	1.47%	istore	1.75%
	Total	69.54%	Total	68.31%

Reference Counts for the SPEC JVM98

Benchmarks	Instruction References	Data References
compress	136,221,748,412	59,686,310,260
jess	38,142,463,807	13,531,453,295
db	72,772,556,697	25,769,177,220
javac	45,155,449,167	15,204,702,870
mpegaudio	124,069,494,085	52,574,044,383
mtrt	47,648,597,326	19,632,340,671
jack	44,253,877,938	17,237,217,381



Impact of Multi-Ported Cache (with JIT Compiler and use S1 Dataset)

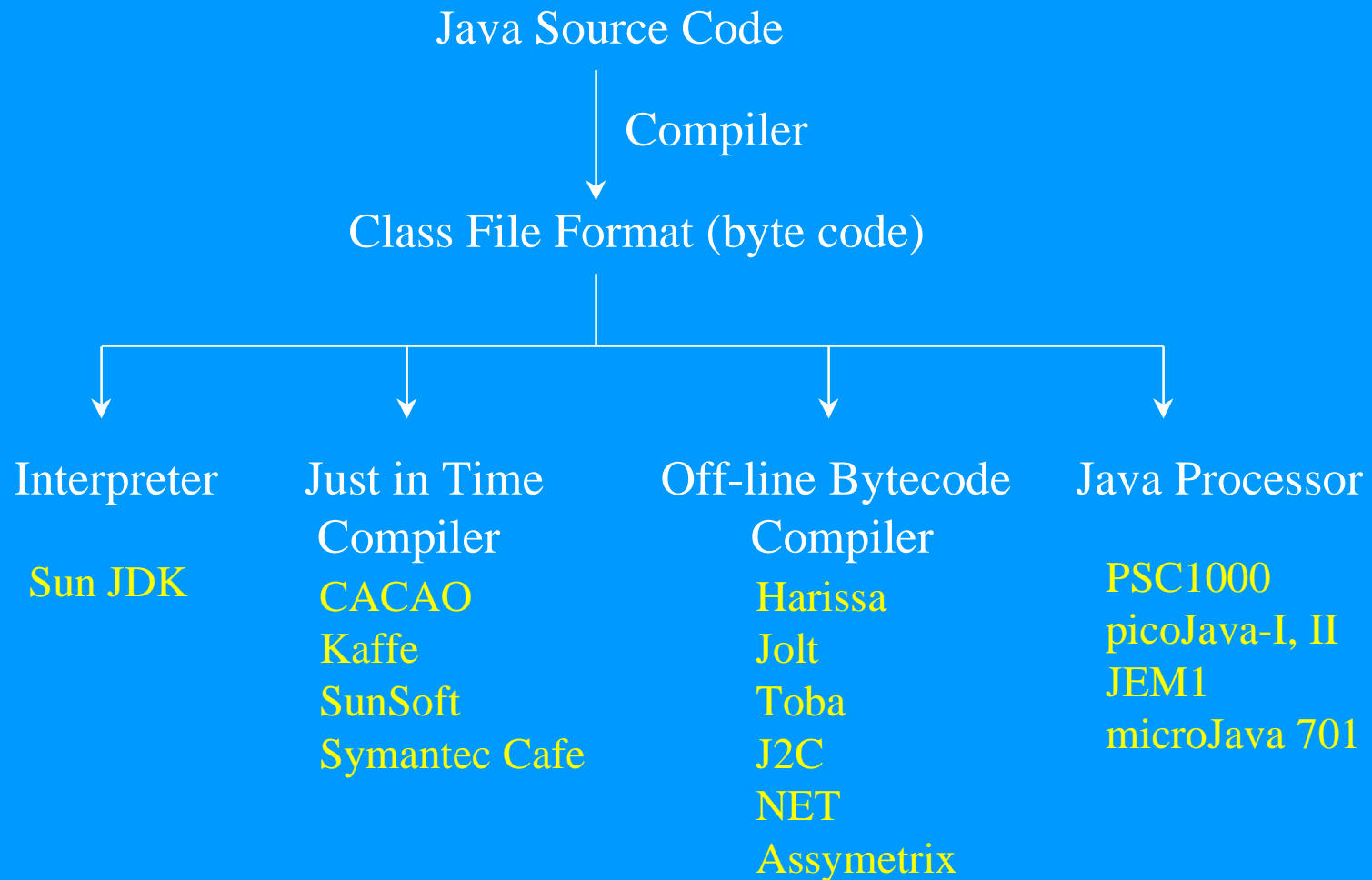
Method Reuse

	method reuse factor		
<u>benchmark</u>	<u>s1</u>	<u>s10</u>	<u>s100</u>
<u>javac</u>	154	801	16,392
<u>db</u>	102	2,498	139,442
<u>jack</u>	1,885	3,748	31,590
<u>jess</u>	339	4,339	69,787
<u>mtrt</u>	2,441	8,957	89,408
<u>mpeg</u>	1,132	9,799	110,244
<u>compress</u>	30,036	31,436	32,443

The table shows the **method reuse factor**, which is calculated as: **# method calls / # unique methods**.

As the data set size is increased the number of method invocations is seen to increase, but the number of methods in the program remains approximately the same

Java Execution Models

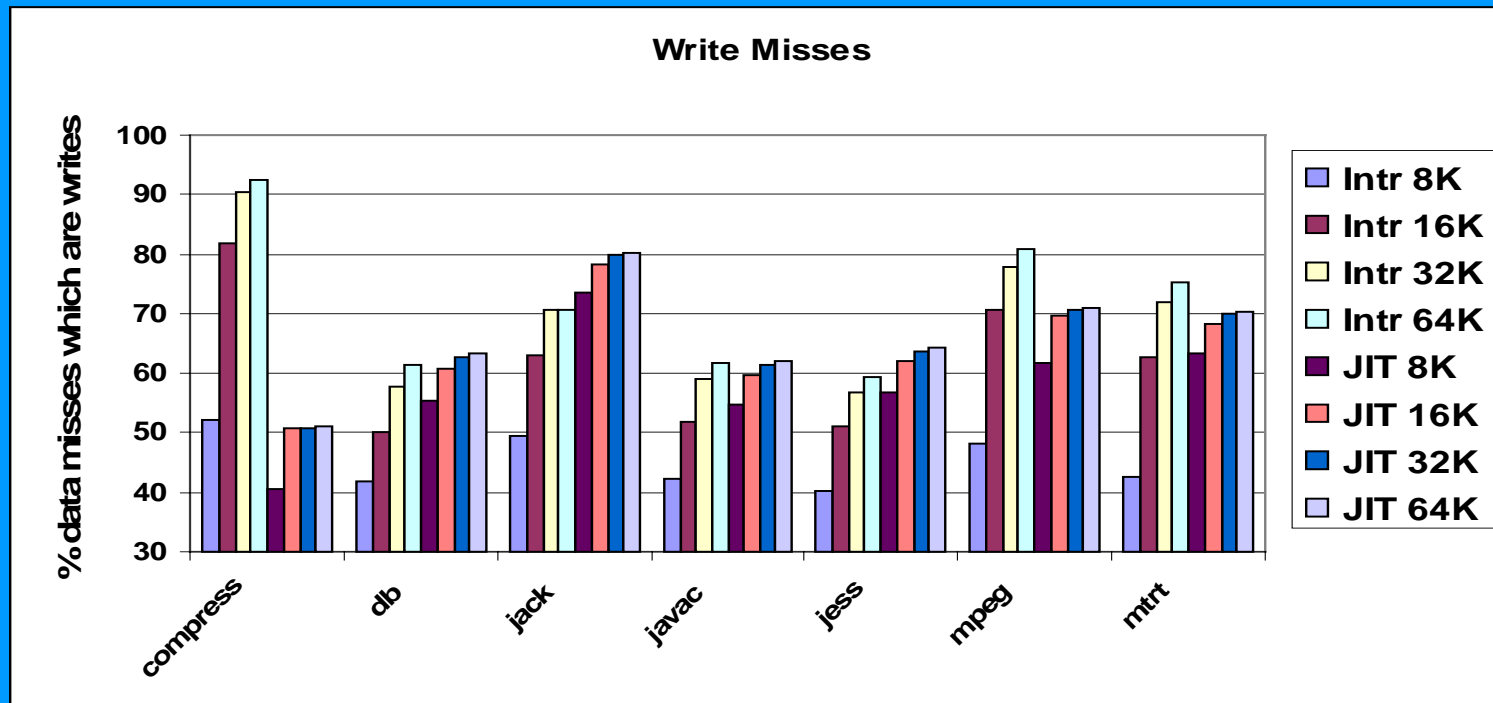


Instruction Counts for the SPEC JVM98

Benchmarks	Dynamic Instructions		
	-s1	-s10	-s100
compress	10,425,411,132	12,266,271,988	135,574,087,140
jess	259,680,029	1,883,517,080	37,278,375,650
db	86,790,209	2,563,123,834	71,418,580,006
javac	198,901,312	1,697,839,767	43,099,730,424
mpegaudio	1,314,122,732	13,169,621,773	123,754,861,802
mtrt	1,531,759,211	4,671,899,604	46,691,729,376
jack	2,668,787,739	5,254,055,974	43,832,109,516

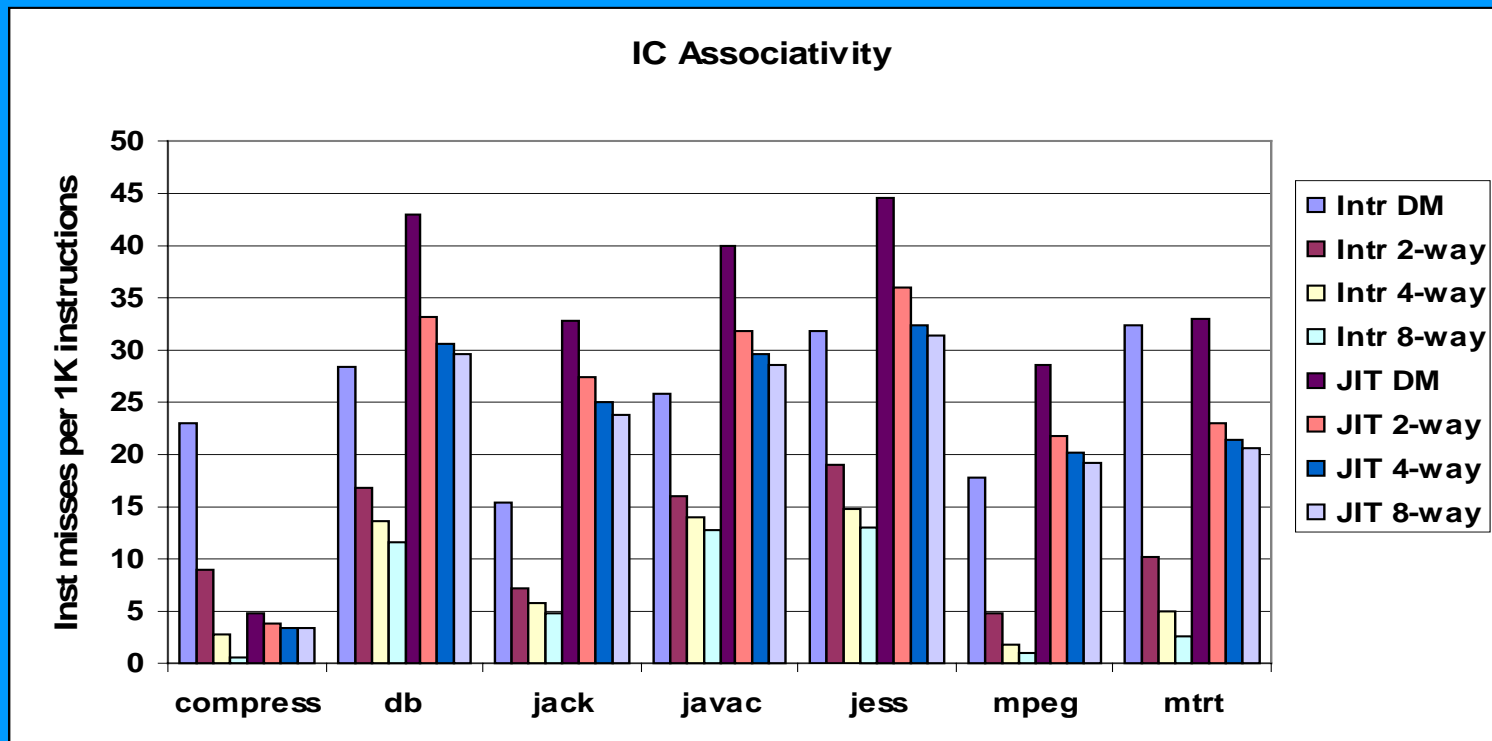
Write Misses

- Percentage of Data misses that are writes



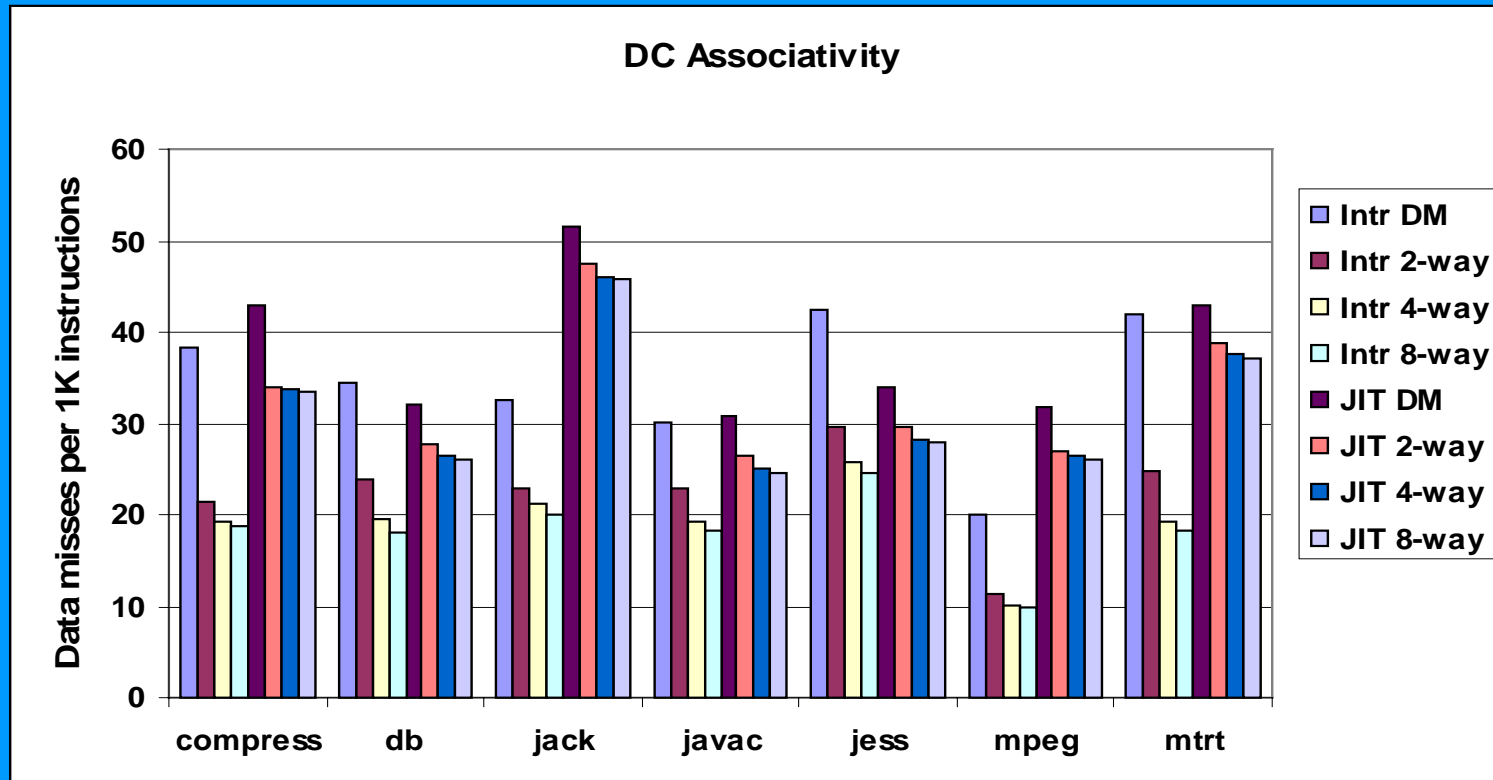
I-Cache associativity

- Effect of increasing the associativity of I-Cache



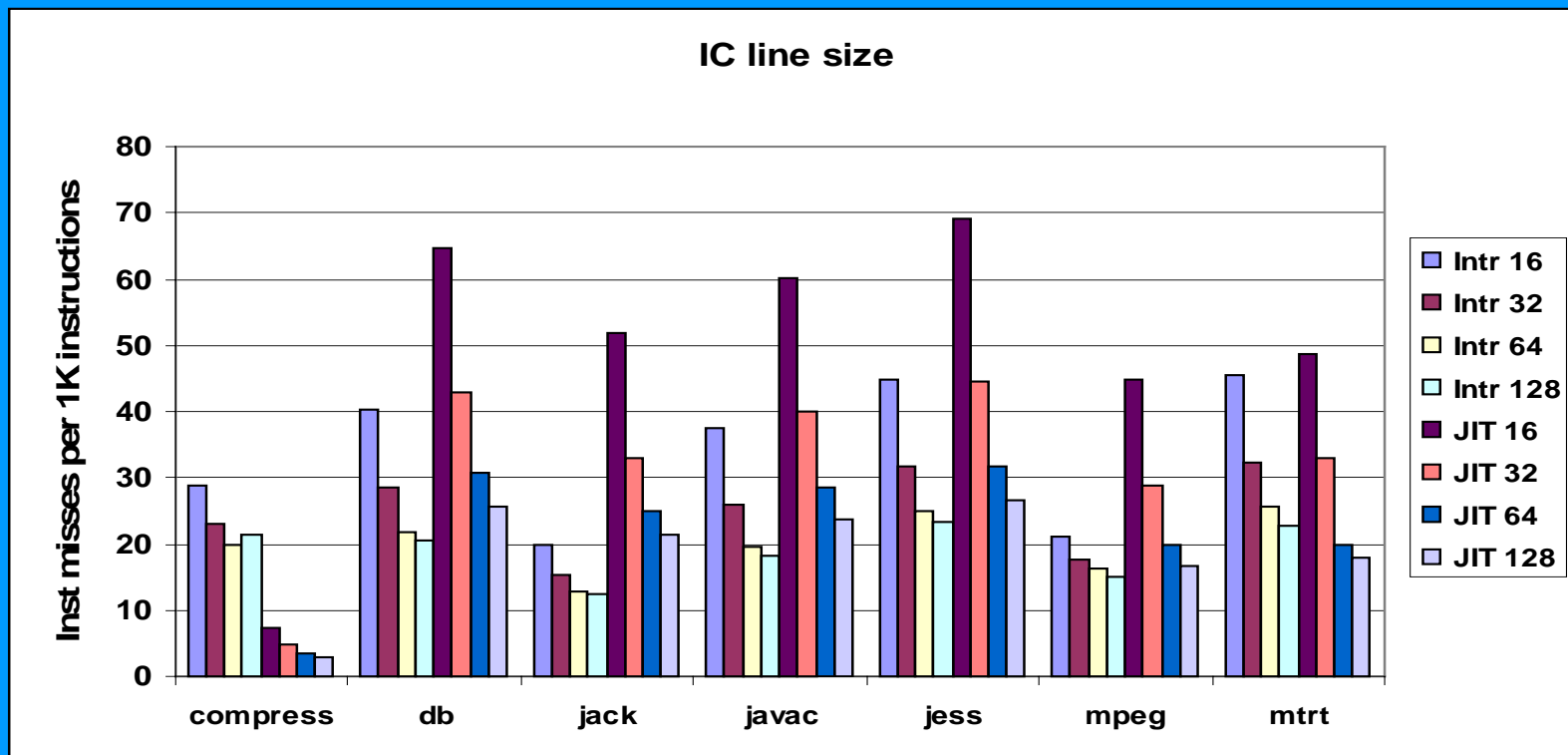
D-Cache associativity

- Effect of increasing the associativity of D-Cache



I-Cache line size

- Effect of changing the line size for I-Cache

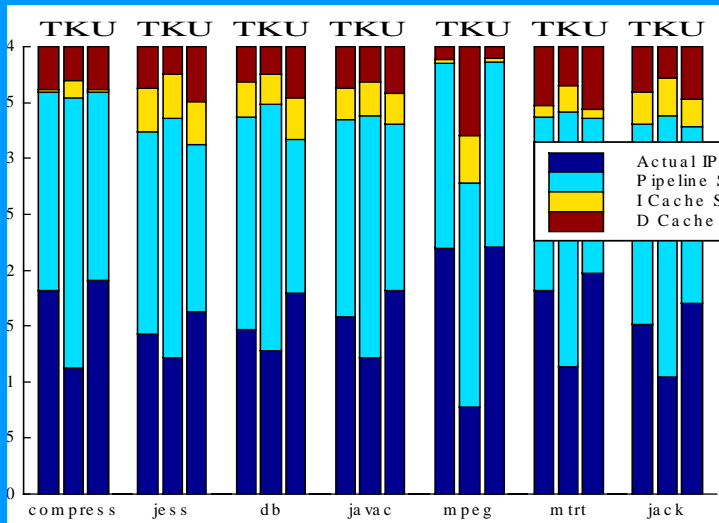


Simulation Results

Execution Time Percentages (s100 Dataset)

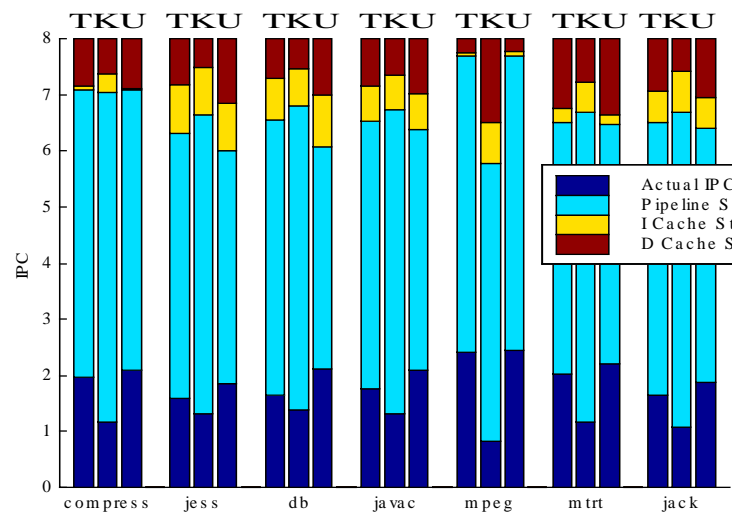
Benchmarks		User	Kernel	Idle
iess	i i t	84.95	14.90	0.15
	i n t r	92.26	7.67	0.07
d b	i i t	87.10	12.64	0.26
	i n t r	83.28	16.57	0.15
j a v a c	i i t	84.31	14.92	0.77
	i n t r	88.57	10.92	0.51
m p e g a u d i o	i i t	99.08	0.73	0.19
	i n t r	99.78	0.20	0.02
i a c k	i i t	82.94	16.90	0.16
	i n t r	83.78	16.11	0.11

- kernel exec.: up to 17% (s100) and 31% (s1)
- JIT mode: more kernel exec.%

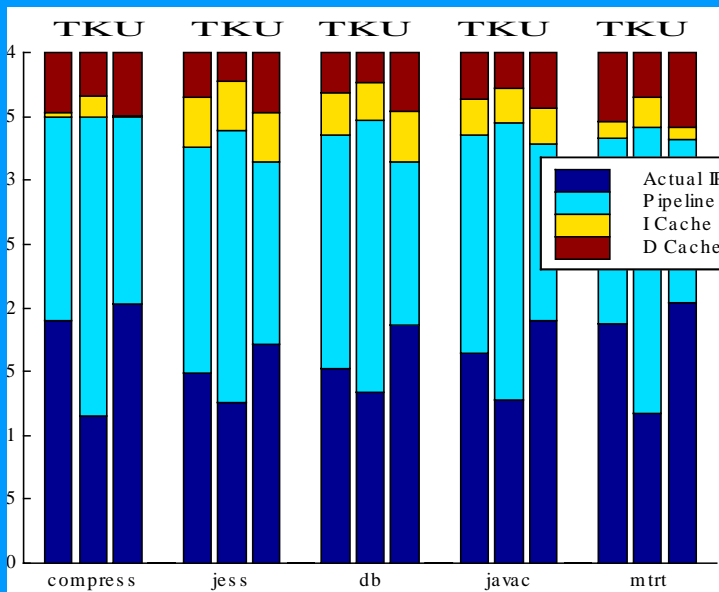


(a)

jit

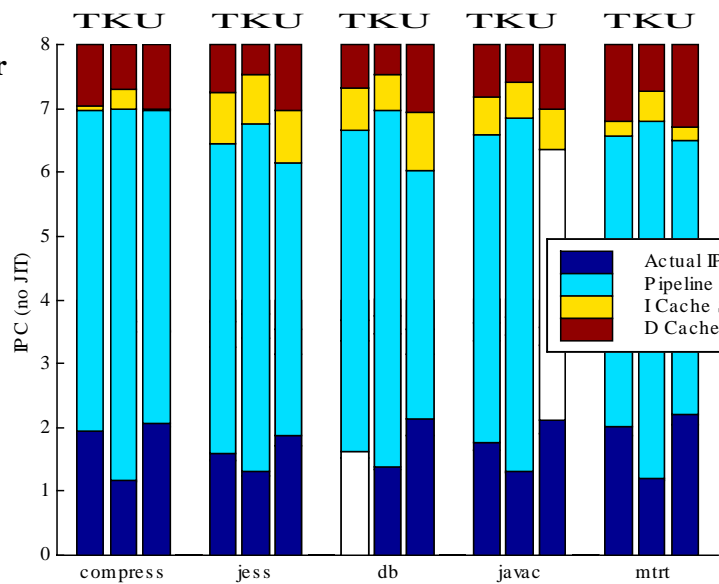


(b)



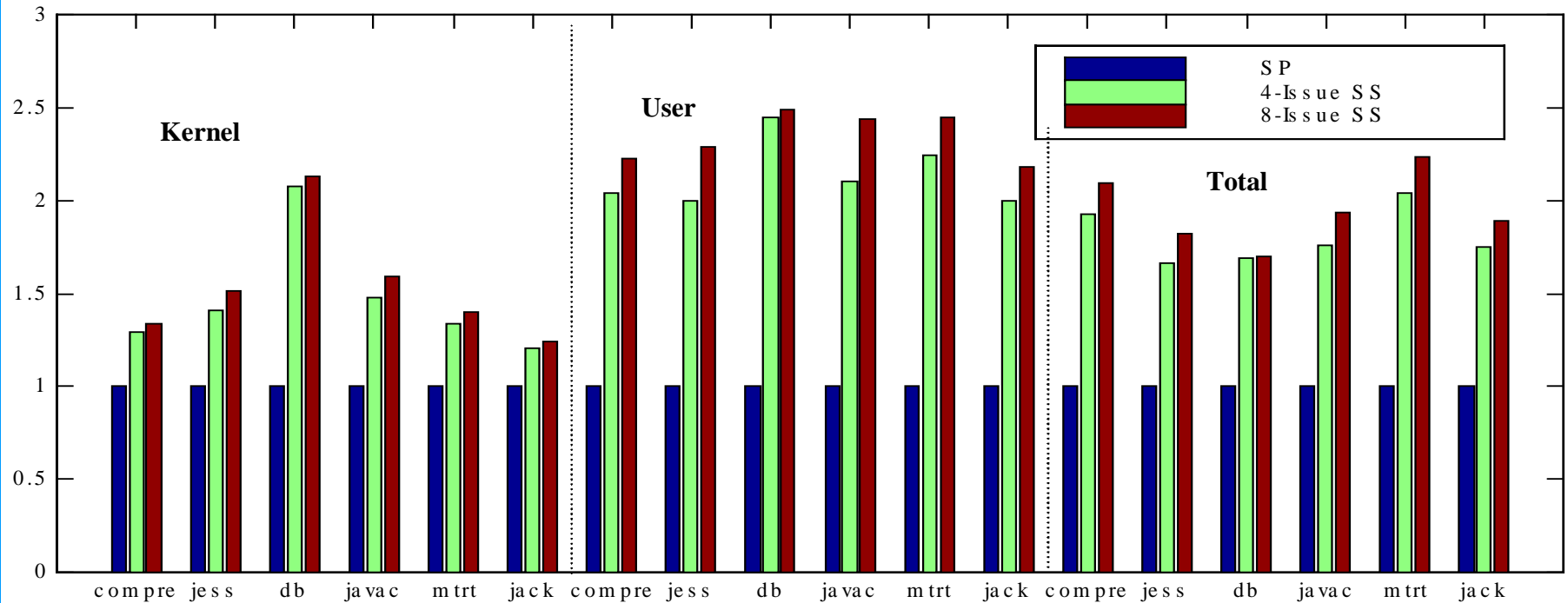
(c)

intr



(d)

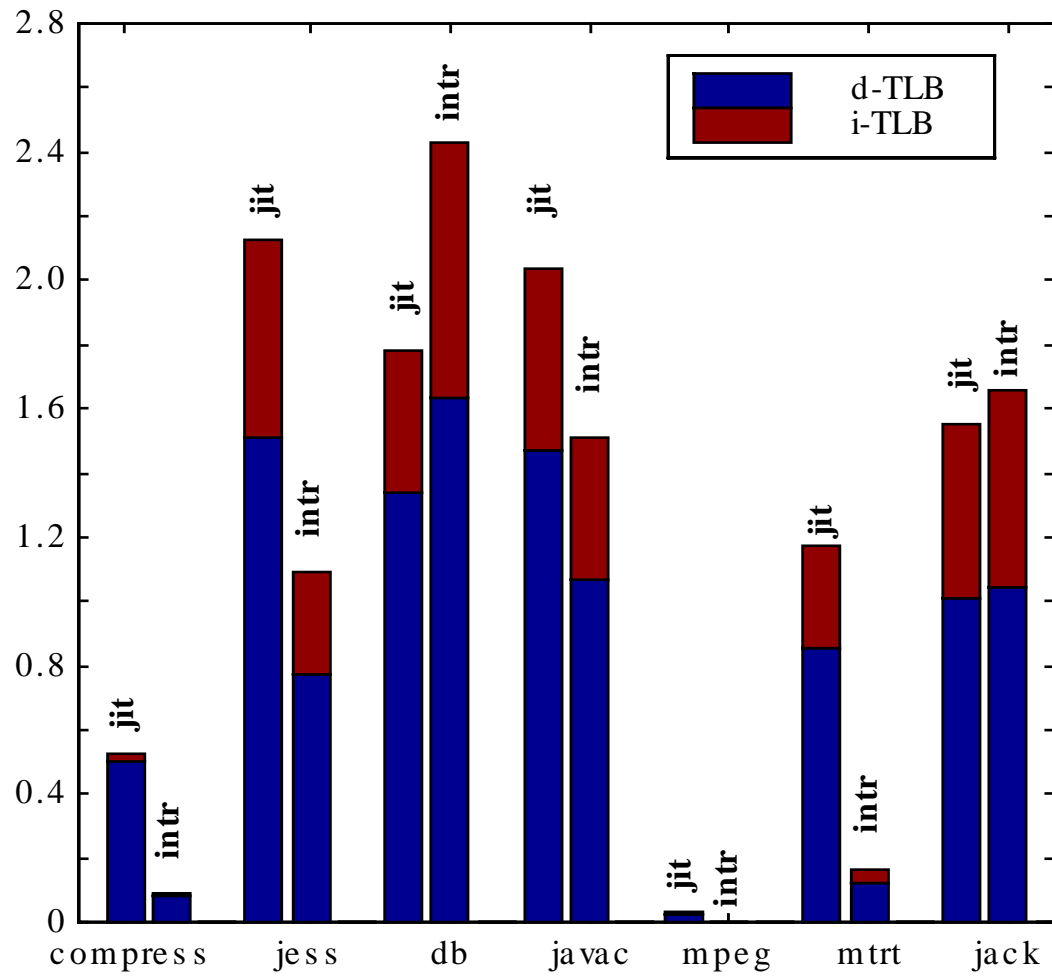
IPC Breakdown for 4-issue and 8-issue Superscalar Processors
(T: Total; K: Kernel; U: User, S1 Dataset)



ILP Speedup (with JIT and S1 Dataset)

TLB Misses Behavior for SPECInt95

Benchmarks	TLB Misses per 10^{-6} Sec.	
	d-TLB	i-TLB
compress95	6.9×10^{-1}	3.6×10^{-2}
m88ksim	6.4×10^{-3}	4.0×10^{-3}
jpeg	1.4×10^{-3}	2.5×10^{-4}
perl	7.0×10^{-5}	9.0×10^{-5}



TLB Misses Behavior for SPECjvm98 Benchmarks

Number of Instructions Executed per Bytecode

Benchmarks	Instructions per Bytecode	CTI per Bytecode	Loads per Bytecode	Stores per Bytecode
compress	10.91	1.60	3.64	1.15
jess	31.93	6.24	7.32	2.66
db	42.64	8.71	8.65	3.26
javac	33.37	6.58	7.32	2.64
mpegaudio	11.35	1.65	3.61	1.09
mtrt	30.22	5.63	7.61	2.67
jack	15.18	2.63	4.42	1.46