Top Level Data Memory Hierarchy: Java vs. C/C++

M. Martin - M. Plakal - V. Iyengar

12/12/96

CS752 -UW Madison

Java Overview

- Java is a new programming language developed at Sun
- Java shares a similar object oriented syntax with C++
- Has a different execution and memory model than C/C++

Java Differences

- Portable - device and OS independent
- Full run-time checking
- Bytecode intermediate form
- Secure
- Garbage collected
- Multi-threaded

Motivation

- Java creates new problems and provides opportunity for the memory hierarchy
- What factors are important for a memory system tuned to run a mainly Java workload?

Outline

- Overview and Motivation
- Benchmarks
- Java execution model and performance
- Java memory model
- Methodology
- Results
- Conclusions and future work

Benchmarks

- Difficult finding good benchmarks for a new language - no JavaSPEC, yet...
- Large Java programs exist, though most Java programs are “applets”
- We also compare Java to C/C++ cache behavior, so some of our benchmarks must be in both languages

Benchmarks (cont)

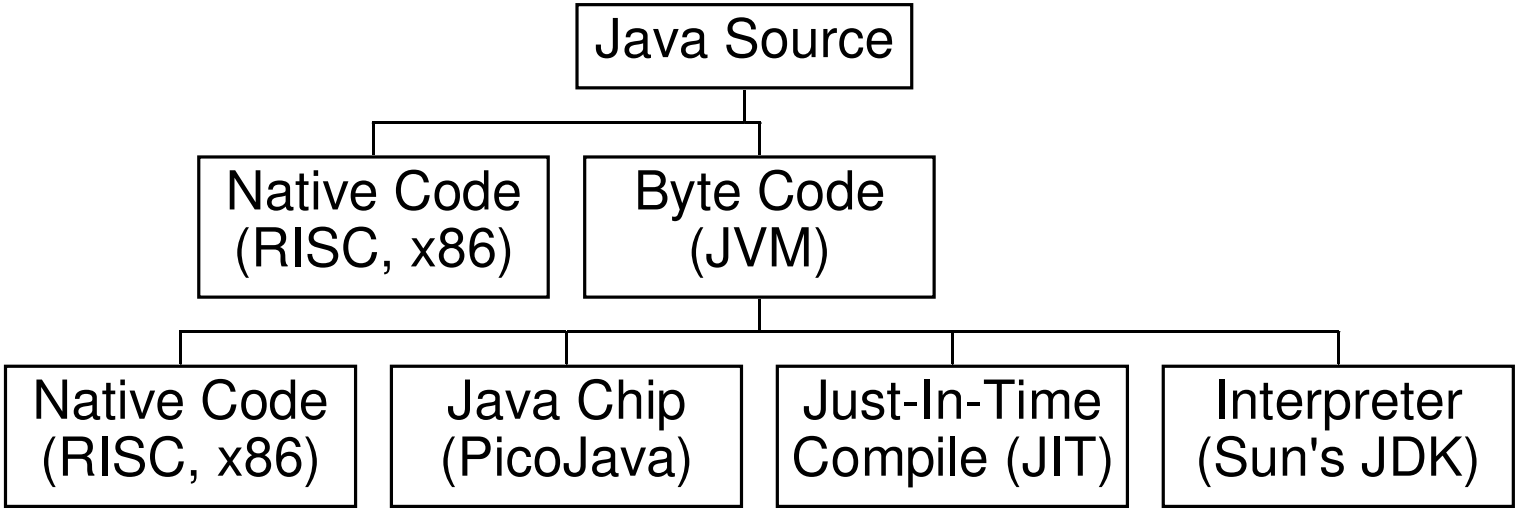
- Java and C/C++
  - Linpack
  - QuickSort
  - Automata - 1D cellular automata
  - MatMult - Matrix multiplication
  - WordFreq - Calculates word frequencies
  - 8q - Solves the eight queens chess problem
  - jb - Parser generator

Benchmarks (cont)

- Java only
  - Javac - Sun's Java to bytecode compiler
  - Kawa - Scheme interpreter
  - Jack - Parser generator
- Benchmarks avoid garbage collection and multi-threading

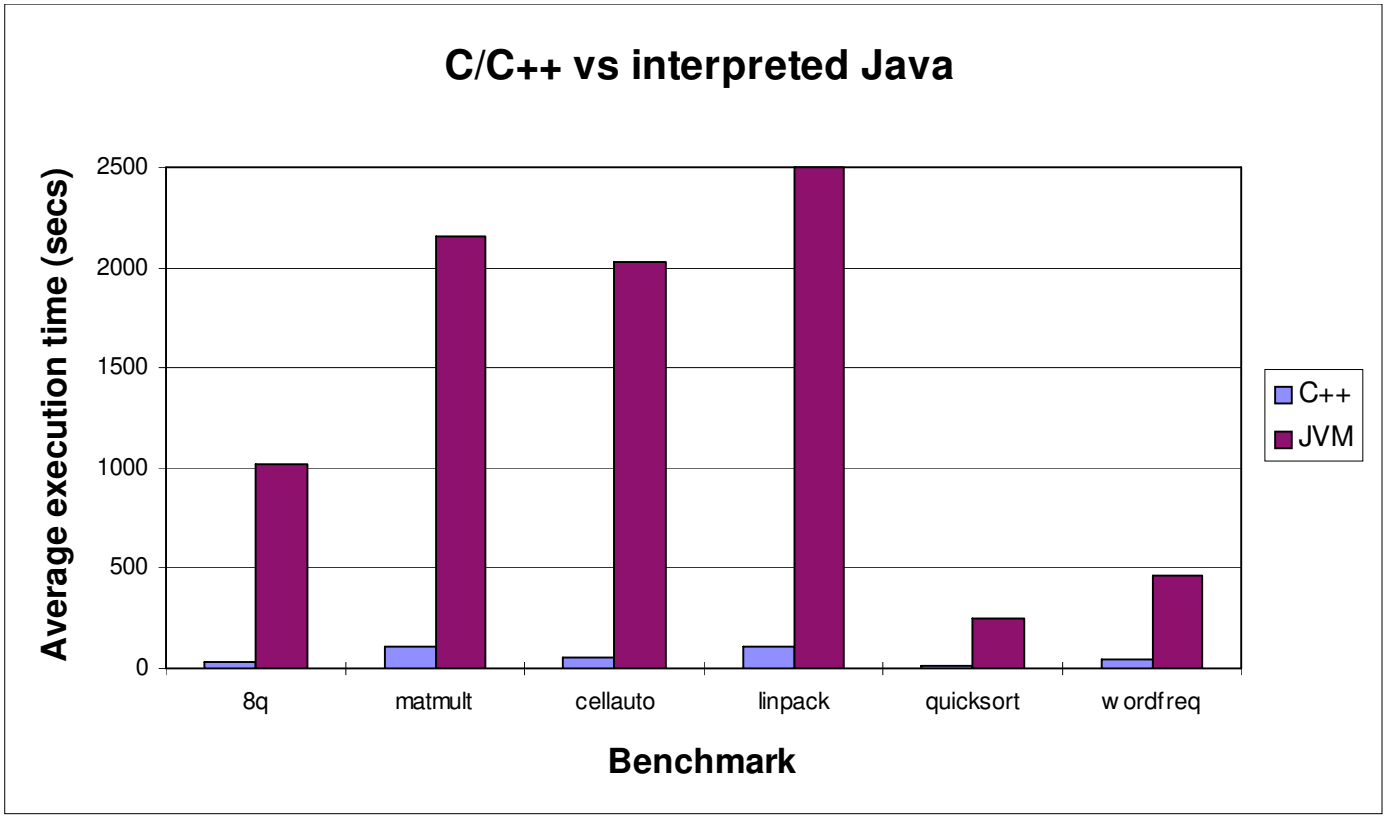
Java Execution Model

- Java source code is converted to machine independent JVM Bytecode
- JVM is a stack based virtual machine



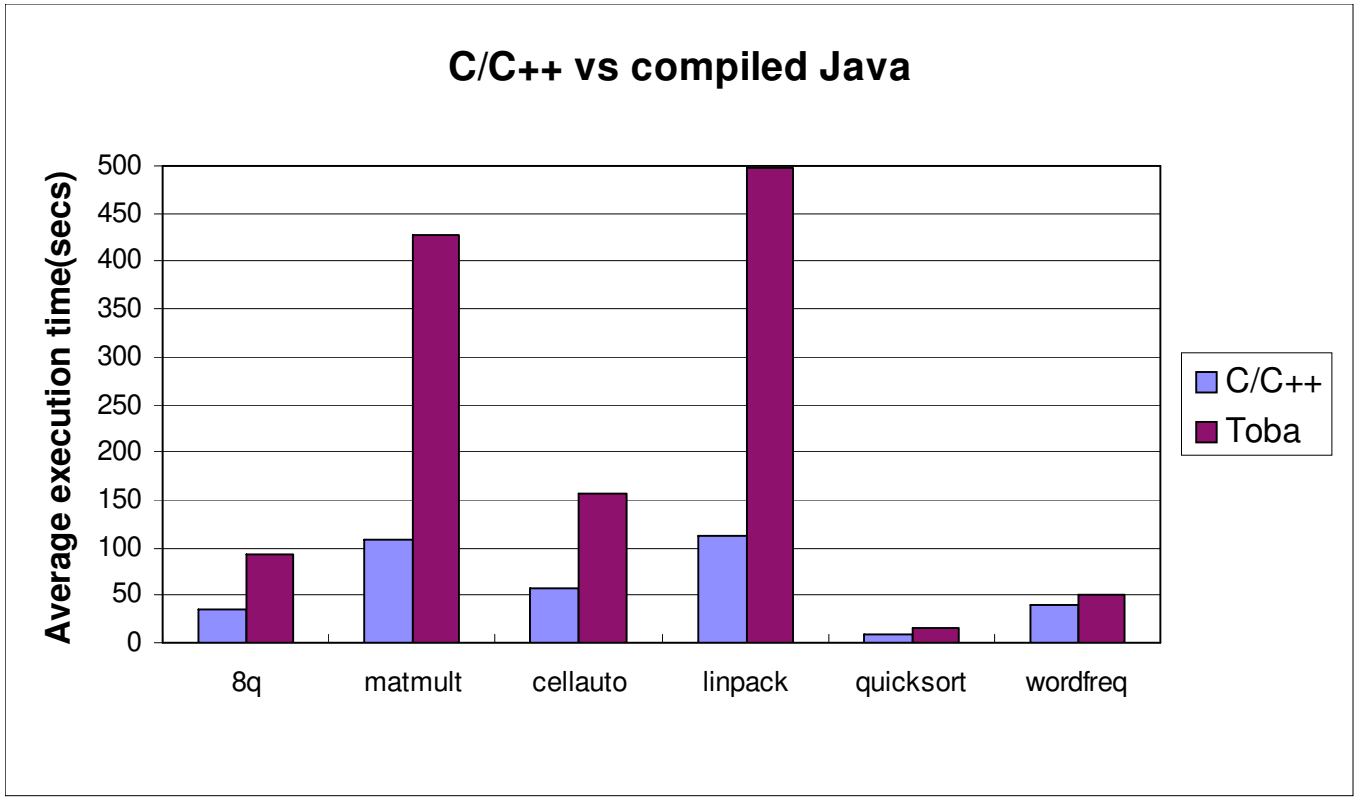
Java Performance

- Java is slow...



Java Performance (cont)

- but not too slow...



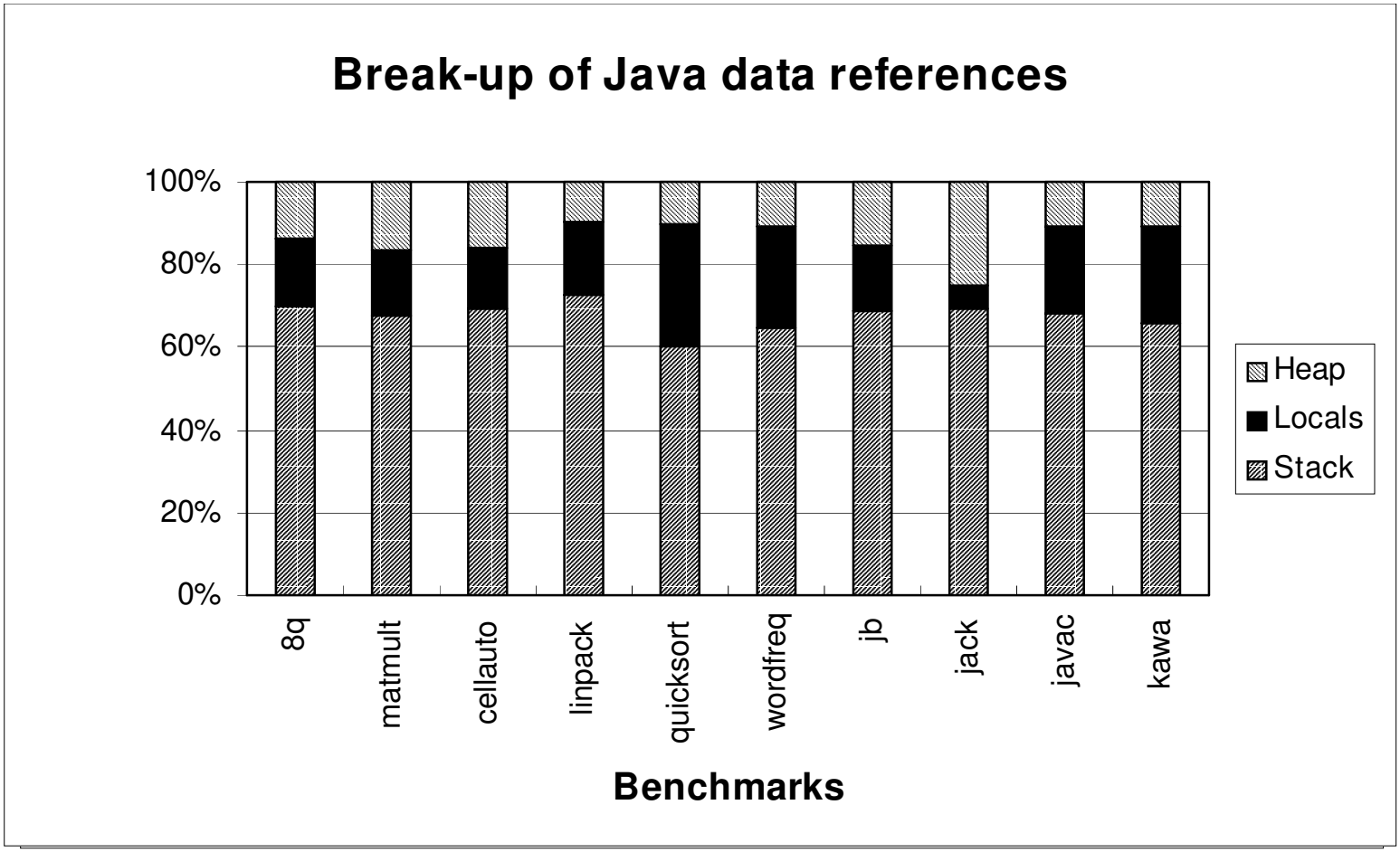
C/C++ Memory Model

- Conceptually all data is stored in memory
- Operands are moved to registers before computation
- Compilers can allocate registers at compile time to hold operand values
- Procedure call stack and heap are part of the same memory address space

Java Memory Model

- Data are stored in the call stack or heap memory
- Operands are moved to the operand stack before computation
- Procedure call stack and heap are different address spaces
- All non-scalars are stored in the heap, thus no “local” arrays

Distribution of Java Data References



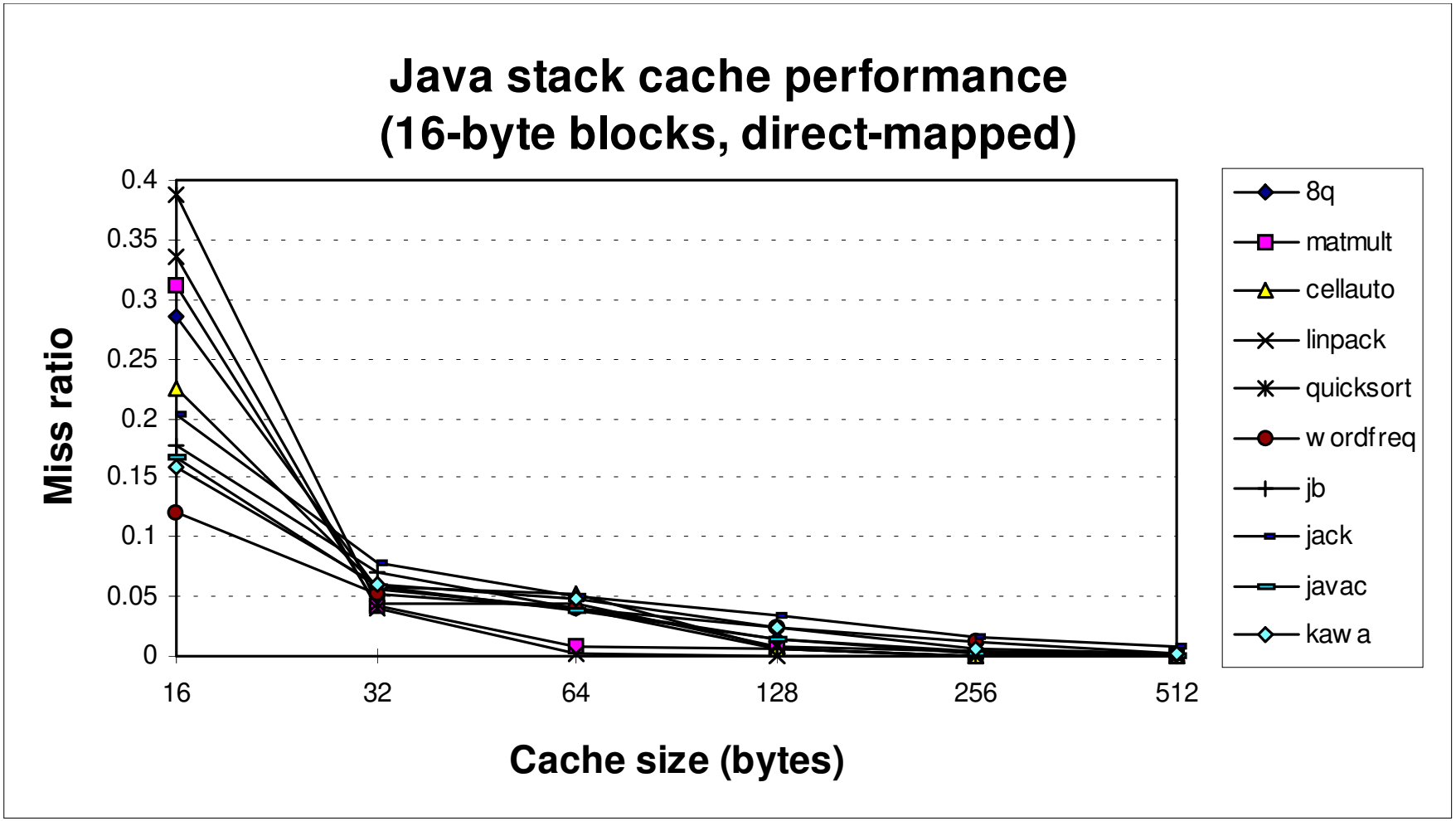
Taking Advantage of the Java Memory Model

- Separate operand stack, locals, and heap caches
- Combinations of caches above
- Unified cache

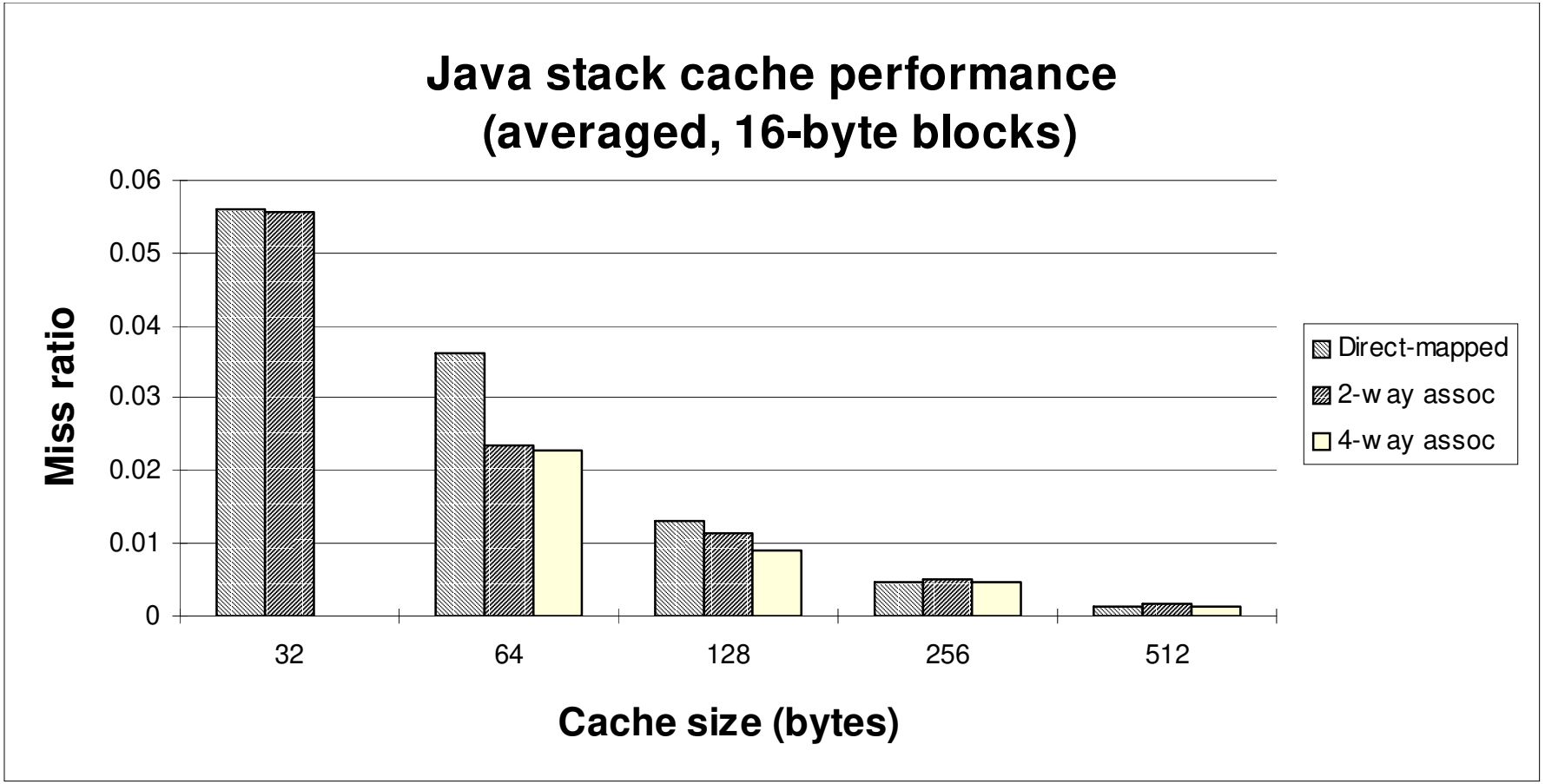
Methodology

- QPT2 was used for C/C++ traces
- We modified Sun's JDK to trace Java stack, local, and heap references
- Dinero was used for cache simulations
- Varied cache size, block size, associativity, and cache combinations

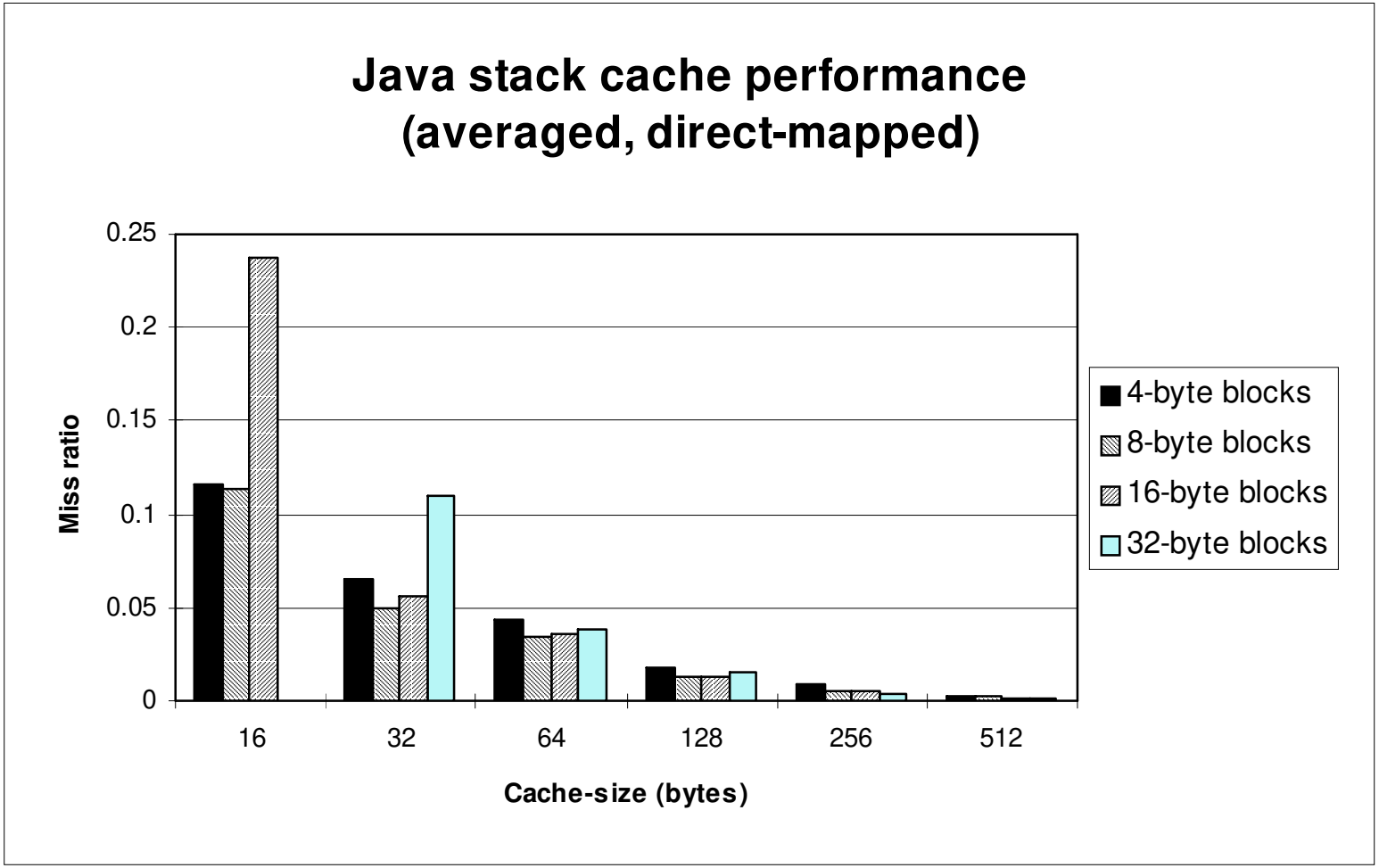
Operand Stack Cache
All Programs



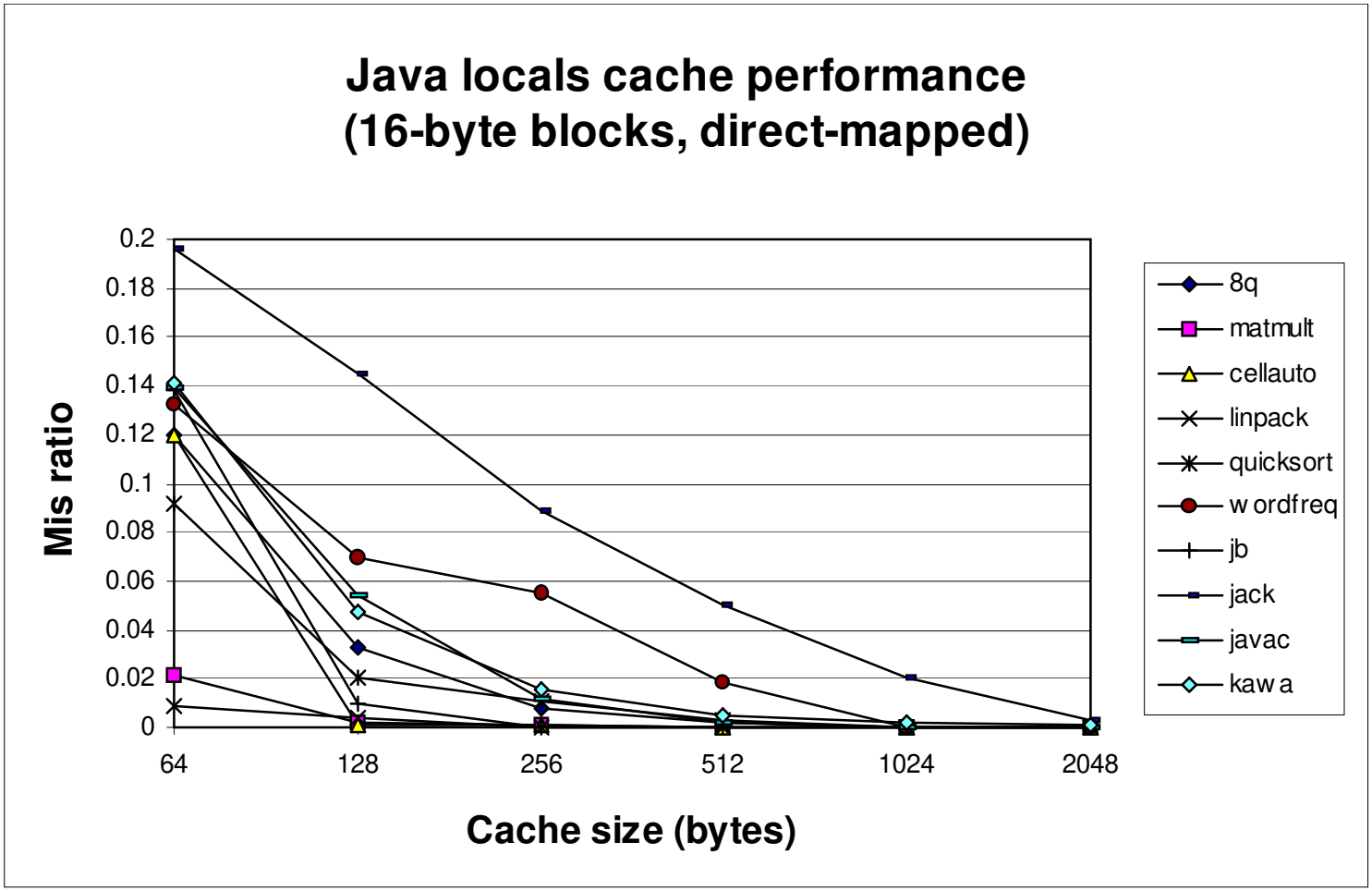
Operand Stack Cache Associativity



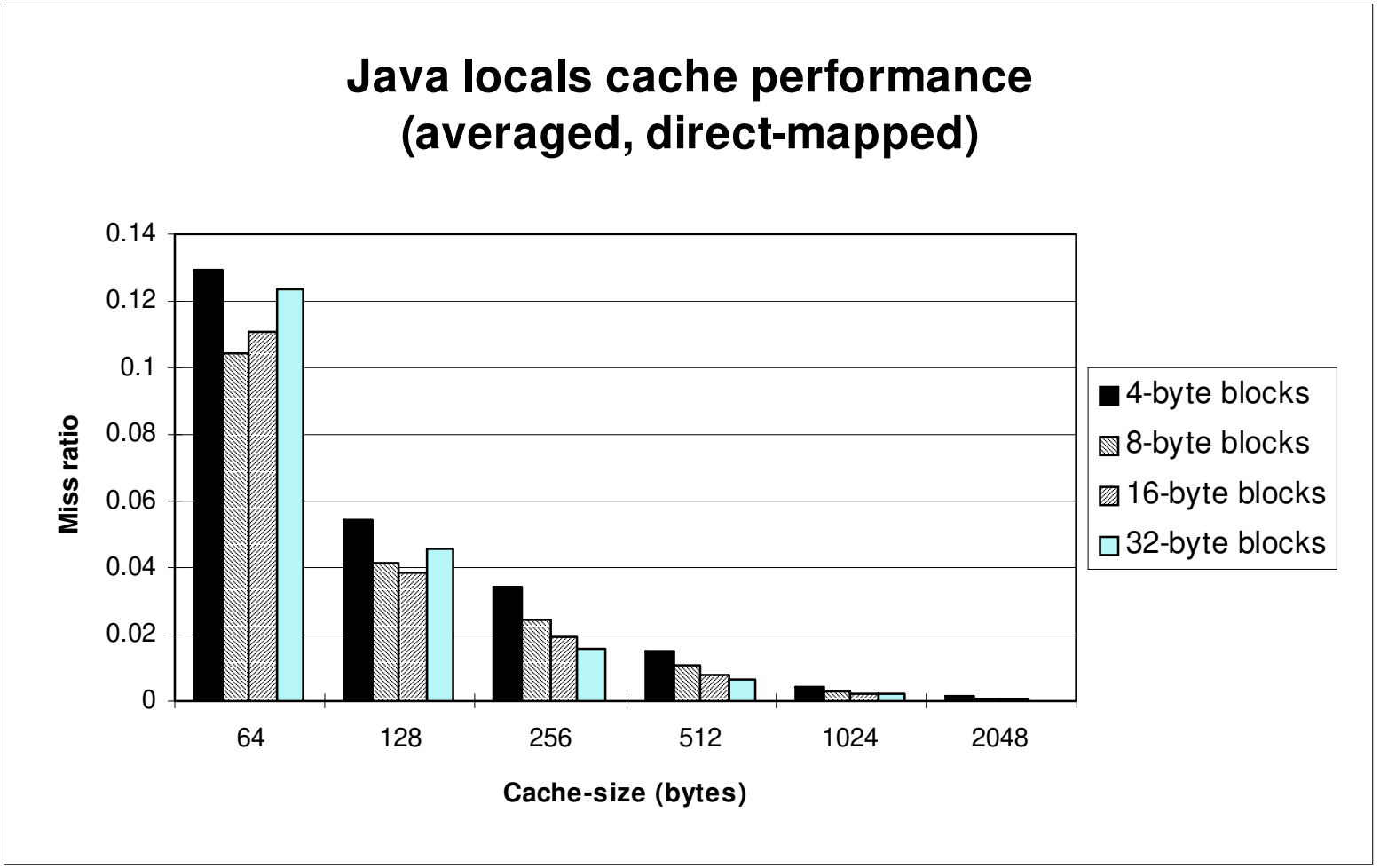
Operand Stack Cache Block Size



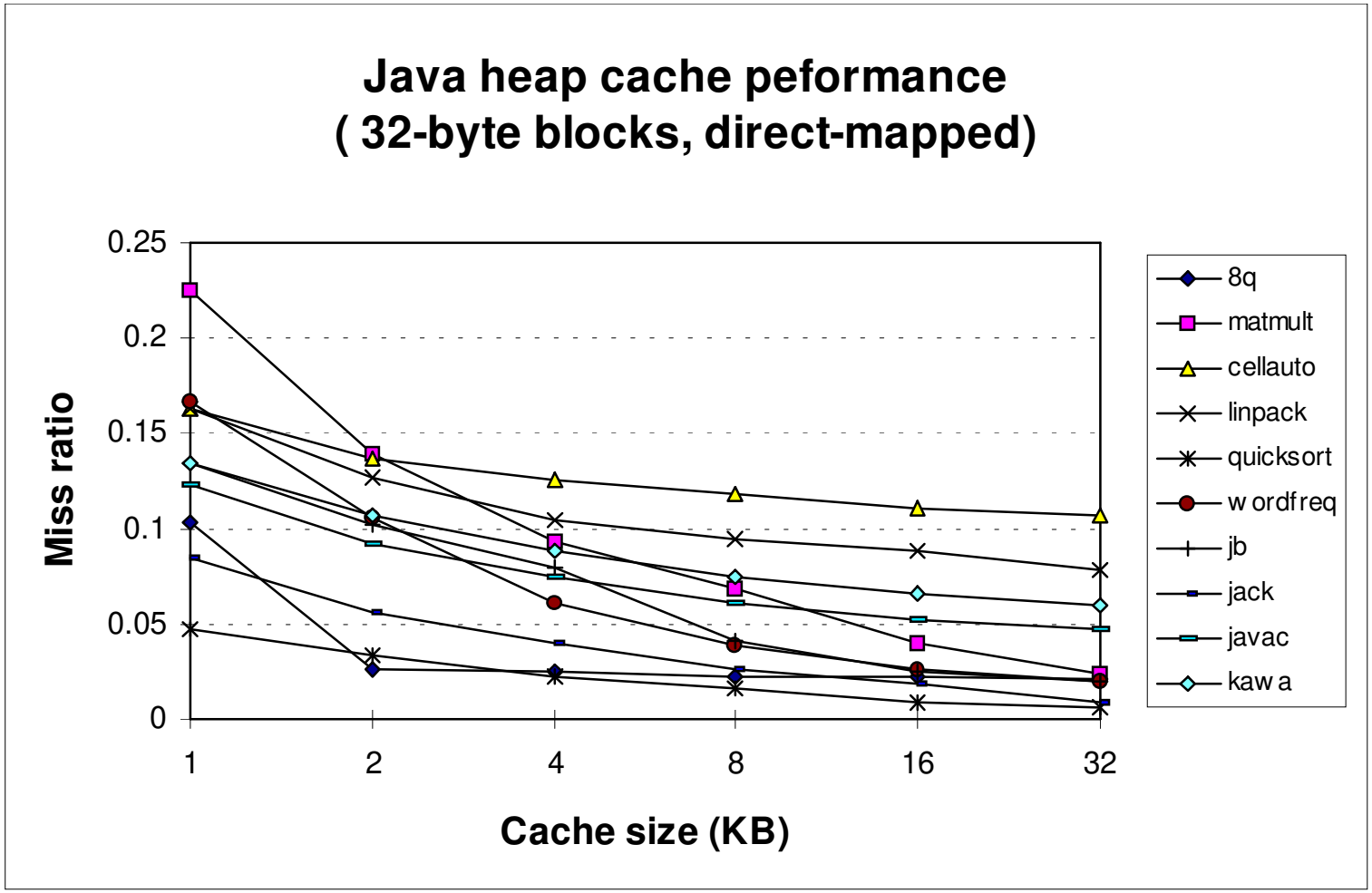
Local Cache - All Programs



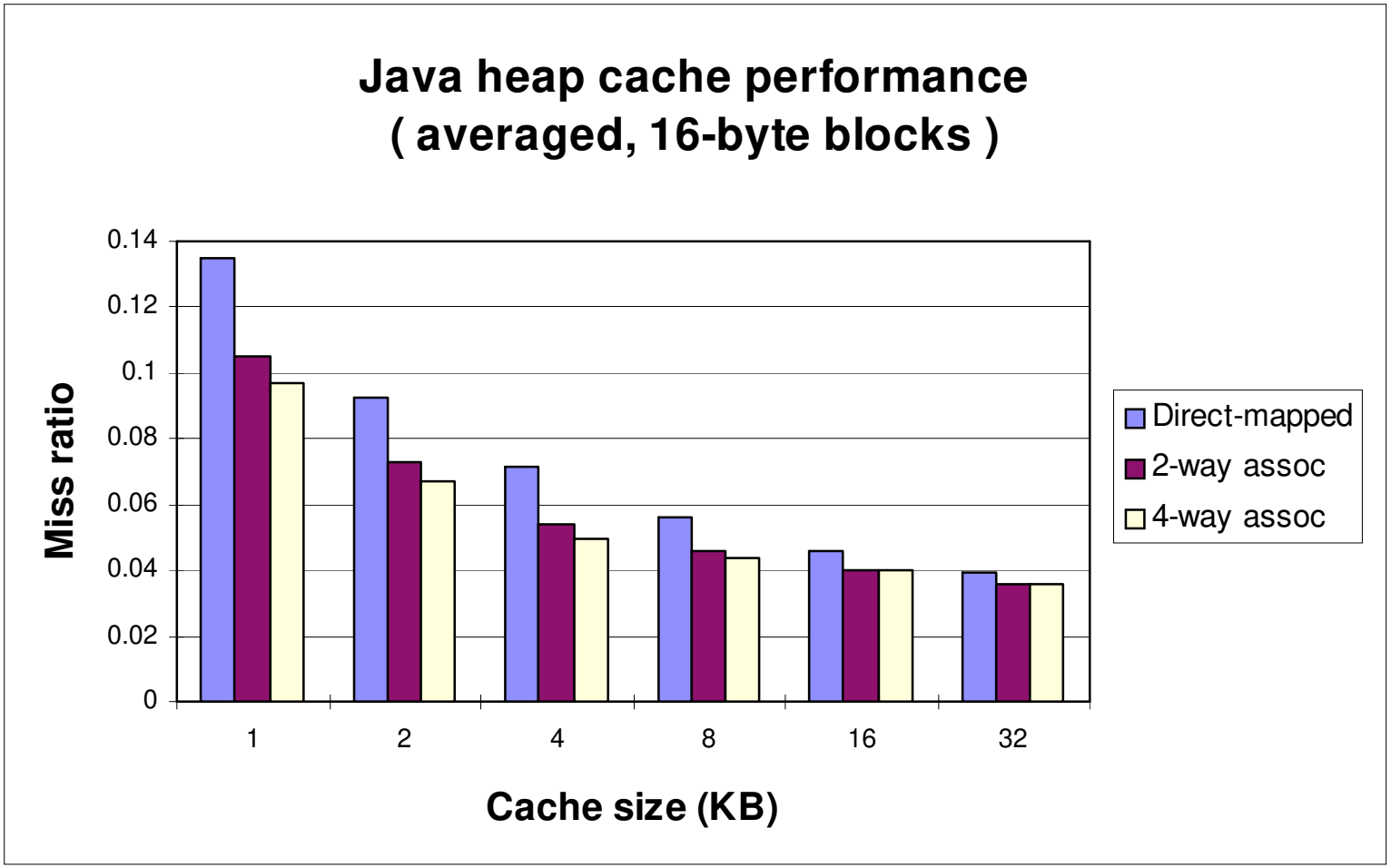
Local Cache - Block Size



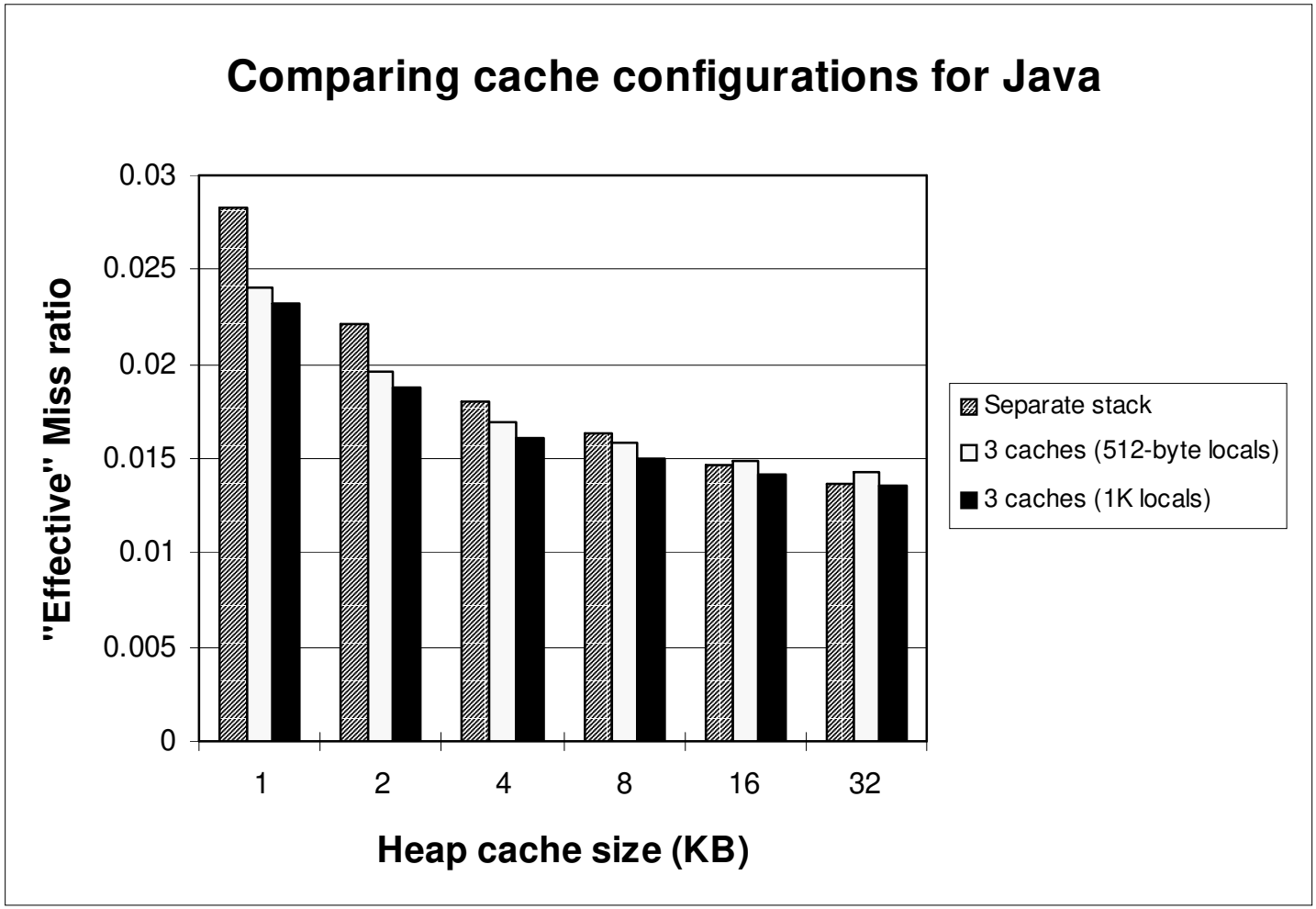
Heap Cache - All Programs



Heap Cache - Associativity



Comparing Cache Configurations

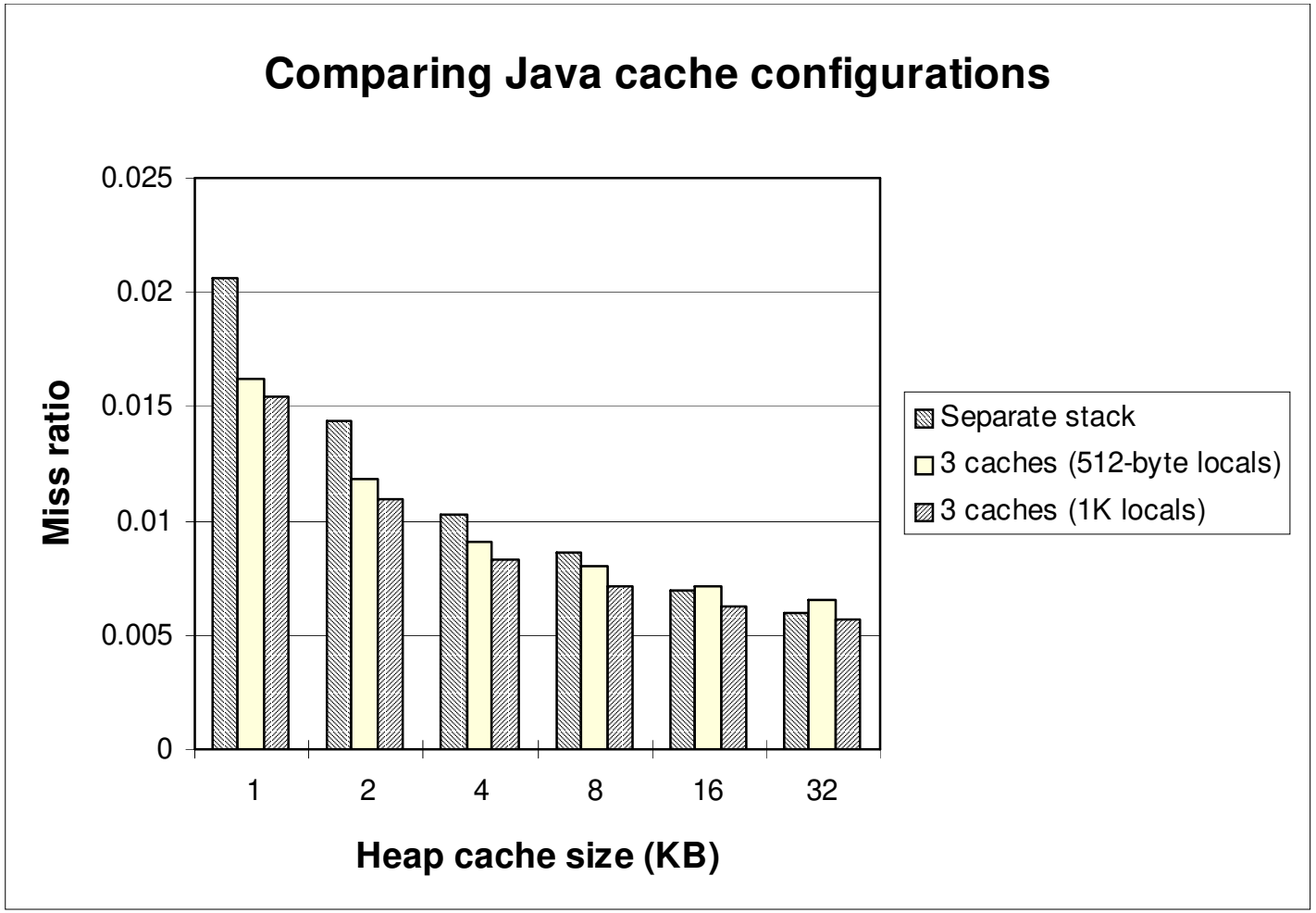


Stack Cache: 128 bytes, direct mapped, 16 byte blocks

Locals Cache: direct mapped, 16 byte blocks

Heap Cache: 2-way, 16-byte blocks

Comparing Cache Configurations

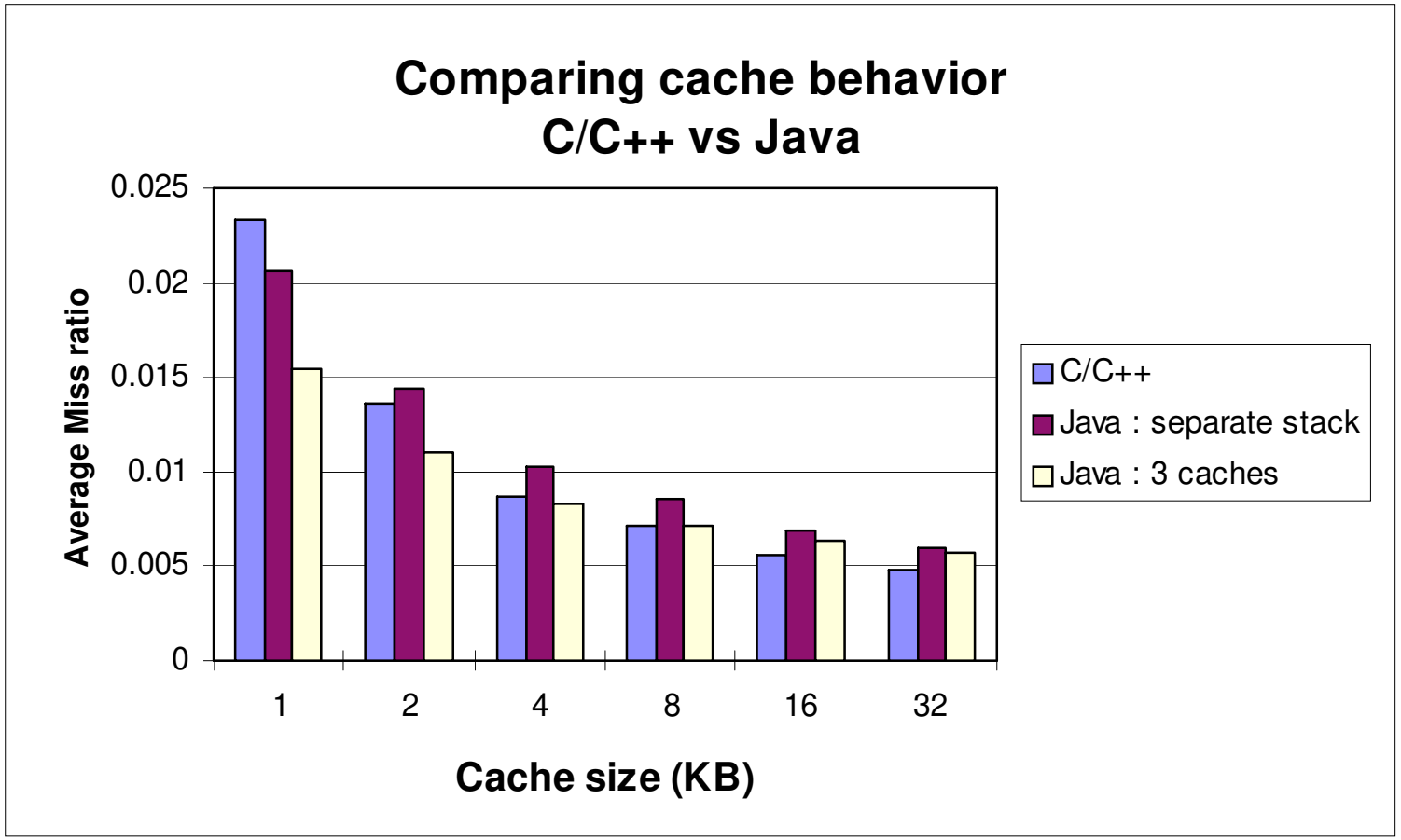


Stack Cache: 512 bytes, direct mapped, 16 byte blocks

Locals Cache: direct mapped, 16 byte blocks

Heap Cache: 2-way, 16-byte blocks

Comparing C/C++ and Java Average Miss Rate

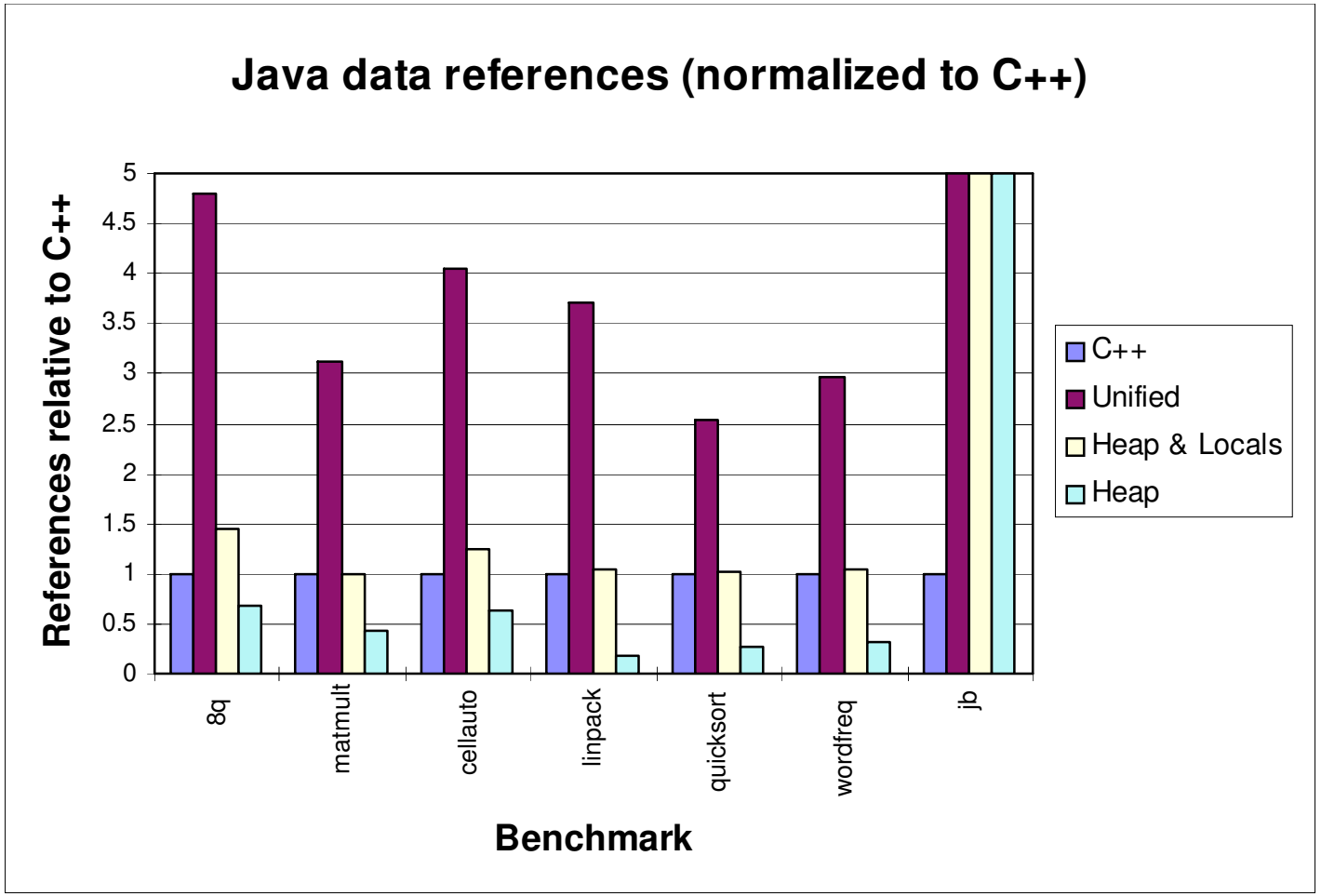


Stack Cache: 512 bytes, direct mapped, 16 byte blocks

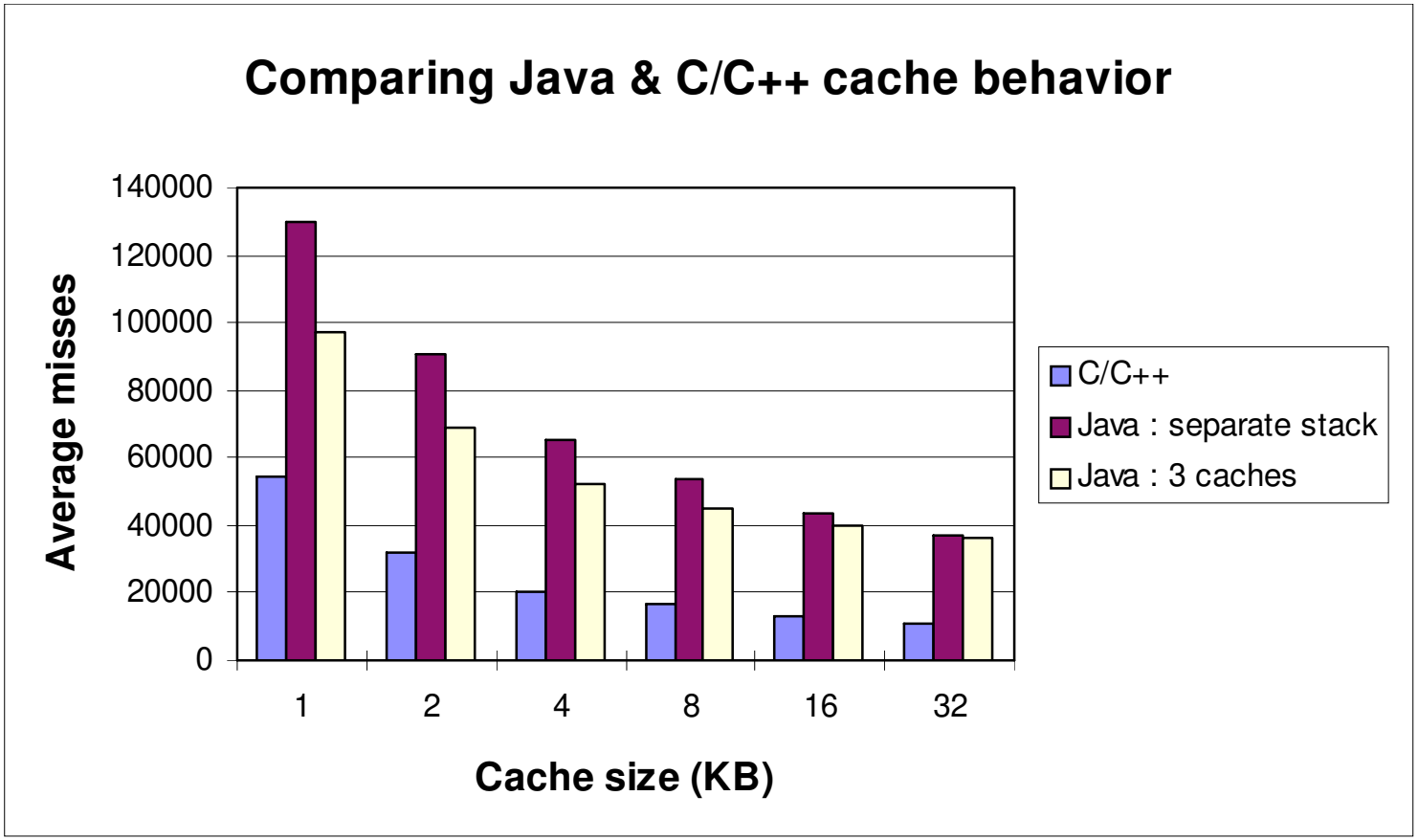
Locals Cache: 1 KB, direct mapped, 16 byte blocks

Heap Cache: 2-way, 16-byte blocks

Data References



Comparing C/C++ and Java Average Misses



Stack Cache: 512 bytes, direct mapped, 16 byte blocks

Locals Cache: 1 KB, direct mapped, 16 byte blocks

Heap Cache: 2-way, 16-byte blocks

Conclusions

- Locality of stack accesses justifies a stack cache
- Placing locals in a separate cache appears beneficial
- Java programs have more overall misses over C/C++ programs, and thus a more aggressive memory hierarchy might be justified

Future Work

- Effects of garbage collection and multi-threading on cache behavior
- JIT compiler cache behavior