LaTTe: An Open-Source Java VM Just-in-Time Compiler

S.-M. Moon, B.-S. Yang, S. Park, J. Lee,
S. Lee, J. Park, Y. C. Chung, S. Kim
Seoul National University

Kemal Ebcioglu, Erik Altman
IBM T.J. Watson Research Center

Sponsored by the IBM T. J. Watson Research Center

Introduction to the LaTTe Project

- Brief history
  - LaTTe is a university collaboration project between IBM and SNU, which begun on Nov. 1997
  - Released the source code of LaTTe version 0.9.0 on Oct. 1999
  - – Released the new version 0.9.1 of LaTTe on Oct. 2000
    - With additional performance enhancements
  - – More than 1000 copies have been downloaded so far
- Close interaction between SNU and IBM
  - – Active participants: 7 in SNU and 2 in IBM Watson
  - – Many video and phone conferences

Overview of LaTTe JVM

- Includes a fast and efficient JIT compiler
  - Fast and efficient register allocation for RISCs [PACT'99]
  - Classical optimizations: redundancy elimination, CSE,..
  - OO optimizations: customization, dynamic CHA
- Optimized JVM run-time components
  - Lightweight monitor [INTERACT-3]
  - Efficient exception handling [JavaGrande'00]
  - Efficient GC and memory management [POPL'00]

Current Status of LaTTe

- LaTTe JVM works on UltraSPARC .
  - Faster than JDK 1.1.8 JIT by a factor of **2.8** (SPECjvm98)
  - Faster than JDK 1.2 PR by a factor of **1.08**
  - Faster than JDK 1.3 (HotSpot) by a factor of **1.26**
  - Translation overhead : 12,000 cycles per bytecode on SPARC
  - Started from Kaffe 0.9.2
  - Supports JAVA 1.1

Outline

- JIT compilation technique
  - Fast and aggressive register allocation
  - Classical optimizations
  - OO optimizations: virtual call inlining
- VM run-time optimizations
  - Lightweight monitor handling
  - Efficient exception handling
  - Memory management
- Experimental results

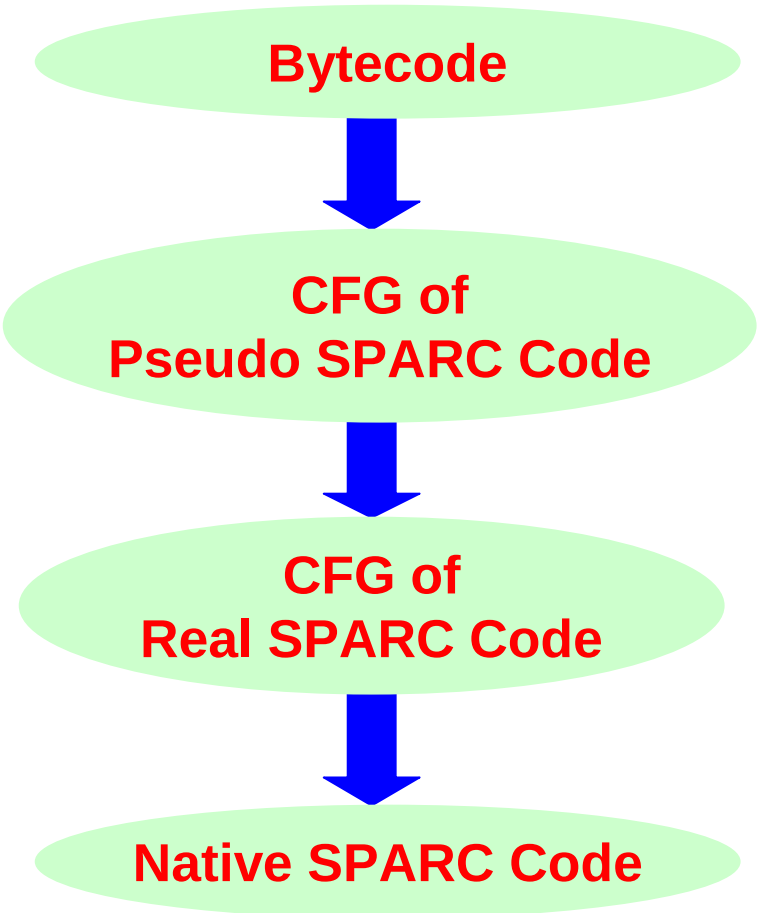
Two Issues in JIT Register Allocation

- Efficient allocation of Java stack into registers
  - Bytecode is *stack-based,* RISC code is *register-based*.
  - Map stack entries and local variables into registers
  - Must coalesce copies corresponding to *pushes* and *pops* between stack and local variables
- Fast allocation
  - Do not want to use graph-coloring register allocation with copy coalescing

Approach of LaTTe

- Two-pass code generation with CFG
  - Build CFG of pseudo code with symbolic registers
  - Allocate real registers while coalescing copies
- Slower but better register allocation than single-pass algorithms (e.g., *Kaffe* and old *VTune)*
- Our engineering solution just enough for Java JIT
  - JIT overhead consistently takes 1~2 seconds for SPECjvm98 which run 40~70 seconds with LaTTe.

JIT Compilation Phases in LaTTe



Bytecode translation

Java stack is mapped to symbolic registers.

Register allocation & Optimizations

Symbolic registers are allocated to machine registers.

Code emission

Binary image is generated from the CFG.
Determines locations of basic blocks

Bytecode Translation Example

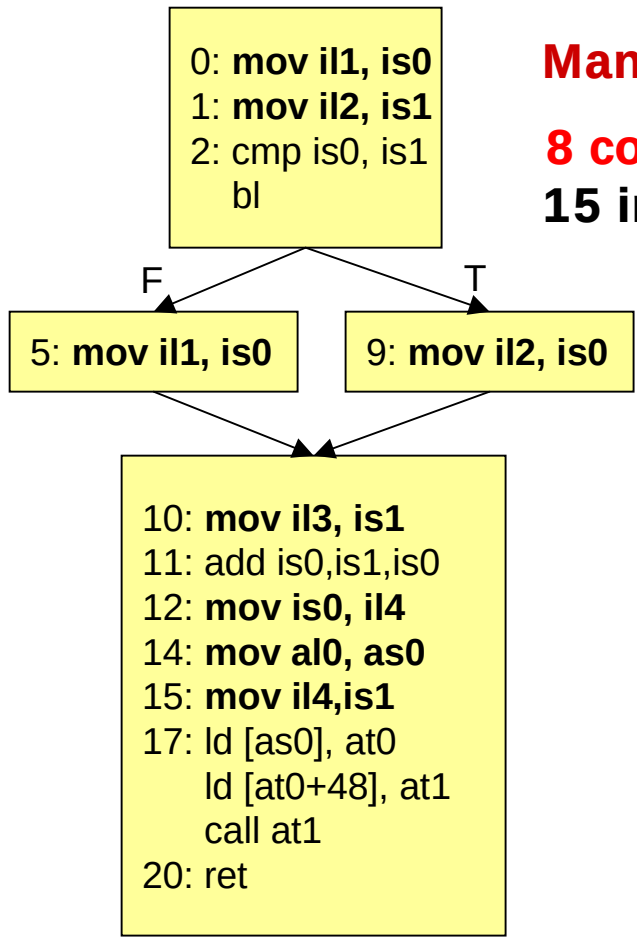
Java source

```
int work_on_max(int x,int y,int tip) {
      int val=((x>=y)?x:y)+tip;
      return work(val);
}
```

bytecode

```
0:   iload_1
1:   iload_2
2:   if_icmplt 9
5:   iload_1
6:   goto 10
9:   iload_2
10:  iload_3
12:  istore 4
14:  aload_0
15:  iload 4
17:  invokevirtual
     <int work(int)>
20:  ireturn
```

Control Flow Graph



Many COPIES!!

8 copies out of 15 instructions

Register Allocation (1)

- Enhanced left-edge algorithm [Tucker, 1975]

- Tree region
  - Unit of register allocation in LaTTe
  - Single entry, multiple exits
    (same as extended basic block)
  - Tradeoffs between quality and speed of register allocation

Register Allocation (2)

- Visit tree regions in reverse post order
  - Register allocation result of a region is propagated to next regions
  - At join points of regions, reconcile conflict of register allocation using copies
    - Similar to replacing SSA ϕ nodes with copies

Register Allocation (3)

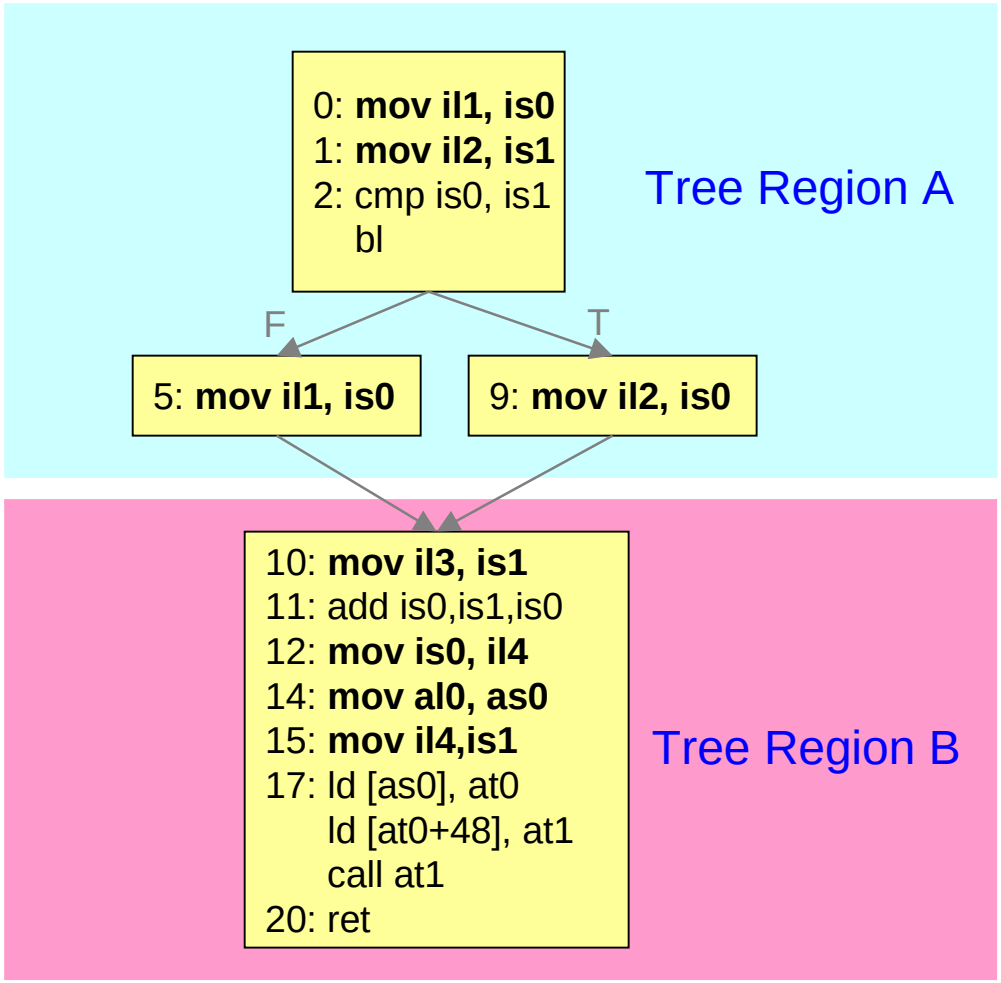
- In each region, the tree is traversed twice
  - *Backward sweep* : collects allocation hints for target registers using pre-allocation results at calls/join points (works as a local lookahead)
    - **preferred map (*p_map*) is propagated backward**

      set of (symbolic, hardware) register pairs
  - *Forward sweep* : performs actual register allocation based on the hints
    - **h/w register map (*h_map*) is propagated forward**

Register Allocation (4)

- Aggressive copy elimination
  - Pseudo code has many copies from *push* and *pop*.
  - Source and target are mapped to the same register.
    - Copies do not generate code, but only update *h_map*.
- Generation of bookkeeping copies
  - To satisfy calling conventions
  - To reconcile *h_map* conflicts at join points of regions
  - Backward sweep reduces these copies.

Register Allocation Example



Allocation Result of Region A

```
2:    cmp %i1,%i2
      bl
```

```
mov %i2,%i1
```

bookkeeping copy to reconcile allocation conflict at the join

```
10:   mov il3,is1
11:   add is0,is1,is0 (o1)
12:   mov is0,il4
14:   mov al0,as0
15:   mov il4,is1
17:   ld [as0],at0
17:   ld [at0+48], at1
17:   call at1 (as0,is1)
20:   ret
```

h={(al0,%i0) (il3,%i3) (is0,%i1)}

This map is passed from Region A to Region B.

Register Allocation Result

- After register allocation of Region B

8 copies are reduced to only 3 copies.

```
0: mov il1, is0
1: mov il2, is1
2: cmp is0, is1
   bl
```

F → 5: **mov il1, is0**

T → 9: **mov il2, is0**

```
10: mov il3, is1
11: add is0,is1,is0
12: mov is0, il4
14: mov al0, as0
15: mov il4,is1
17: ld [as0], at0
    ld [at0+48], at1
    call at1
20: ret
```

```
2:   cmp %i1,%i2
2:   bl
```

mov %i2,%i1

```
11:  add  %i1,%i3,%o1
17:  ld  [%i0],%l0
17:  ld  [%l0+48], %l0
     mov  %i0, %o0
17:  call %l0
     mov  %o0, %i0
20:  ret
```

is0 is mapped to %o1. The value of is0 will be used as the 2nd arg.

bookkeeping copies due to SPARC calling convention

Classical Optimizations

- LaTTe performs
  - Redundancy elimination : CSE, check code elimination
    - Based on value numbering
  - Loop invariant code motion/Register promotion
  - Copy propagation
  - Constant propagation, folding
  - Method inlining : *static/private/final* methods
- Unit of optimizations : tree region

Object-Oriented Optimizations

- Reduce virtual call overheads
  - Virtual calls are normally translated into ld-ld-jmpl
  - With OO optimization, virtual calls can be translated into static calls or can even be inlined
- Two kinds of VC optimizations in LaTTe
  - Customization and specialization
  - Inline of *de facto* final methods through backpatching
    - Assume a virtual method is *final* at first.
    - Create backpatching code when the method is overriden.
  - The latter *outperforms* the former.

VM Run-time Optimization

- Lightweight monitor [INTERACT-3]
  - Optimized for single-threaded programs
- Efficient exception handling [JavaGrande'00]
  - On-demand translation of exception handlers
  - Exception type prediction
- Fast mark-and-sweep GC [POPL'00]
  - Fast object allocation
  - Short mark and sweep time

Lightweight Monitor

- Make frequent operations faster
  - Frequent : free lock manipulation
  - Infrequent: lock contention or *wait*/*notify*
- lock
  - Accessed frequently
  - Embedded in the object header as a 32-bit field

nest_count[31:17]	has_waiters[16]	owner_thread_id[15:0]

- lock queue, wait set
  - Accessed infrequently
  - Managed in a hash table
  - Created only when they are actually accessed

Efficient Exception Handling

- No performance degradation of the normal flow due to exception handlers
  - Do not translate EHs on JITing a method
  - Only if an EH is to be used, translate it.

- Fast control transfer to an EH
  - Predict the exception type of an exception instruction
  - Directly connect the predicted EH to the instruction
  - No intervention of the JVM exception manager

Memory Management

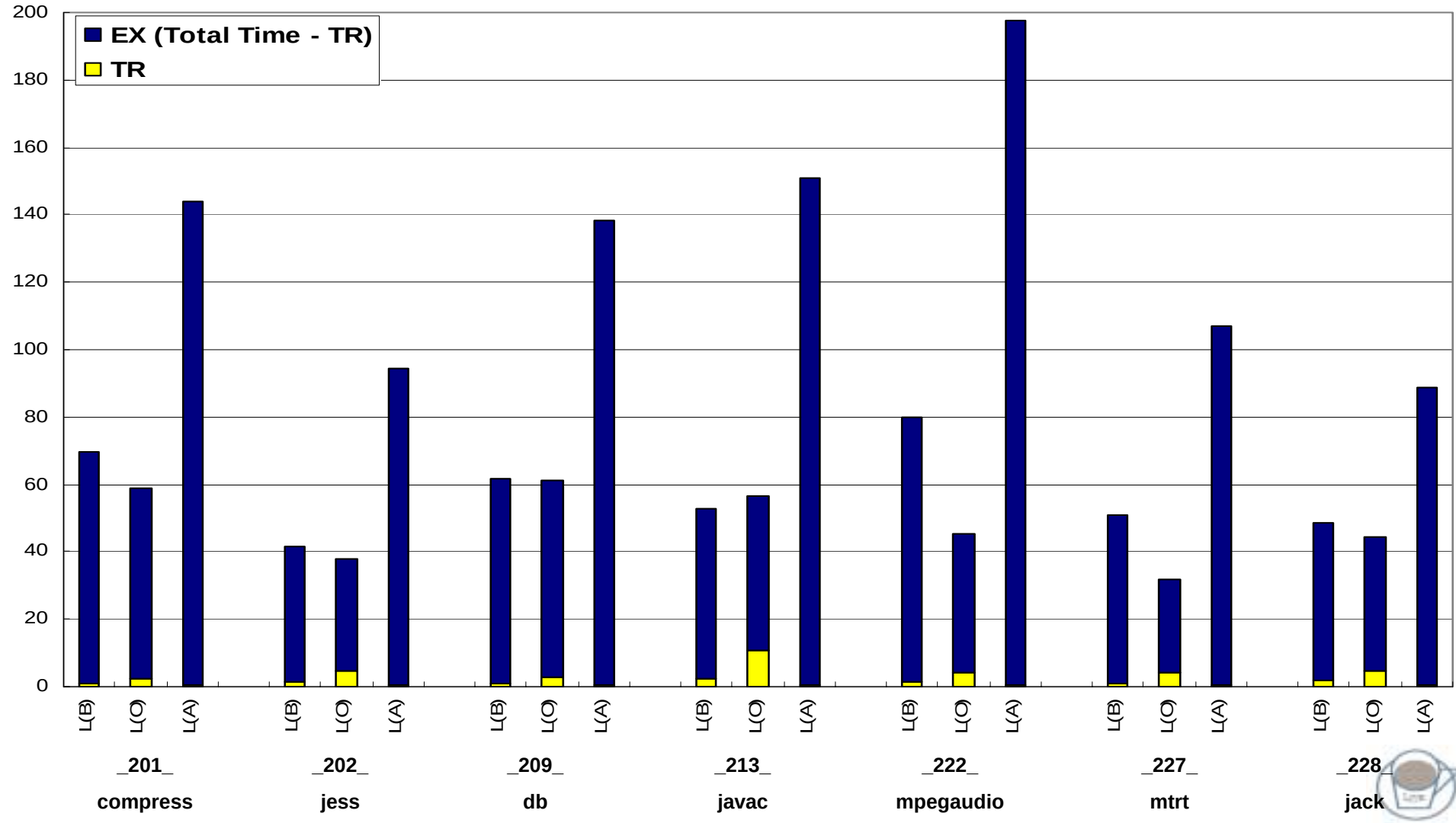
- Small object allocation
  - Very frequent : Speed is important.
  - Uses pointer increments in the most common case
  - Worst-fit if allocation failed with pointer increment
- Large object allocation
  - Not very frequent : Avoiding fragmentation is important.
  - Use a best-fit algorithm
- Partially conservative mark and sweep
  - Run-time generation of *marking* functions
  - Selective sweeping at low heap occupancies (POPL'00)

Experimental Results

- Test environment
  - SUN UltraSPARC II 270MHz with 256MB of memory, Solaris 2.6
  - single-user mode
  - run 5 times and take minimum value
- Benchmarks
  - SPECjvm98, Java Grande Benchmark
- Configuration
  - LaTTe(B) : LaTTe with fast register allocation w/o optimization
  - LaTTe(O) : LaTTe with full optimization
  - LaTTe(K) : LaTTe with Kaffe's JIT compiler

  - JDK1.1.8 : SUN JDK 1.1.8 Production Release with JIT on SPARC
  - JDK1.2 PR : SUN JDK 1.2 Production Release
  - JDK 1.3 HotSpot : SUN JDK 1.3 HotSpot Client

Total Running Time of 3 JITs in LaTTe

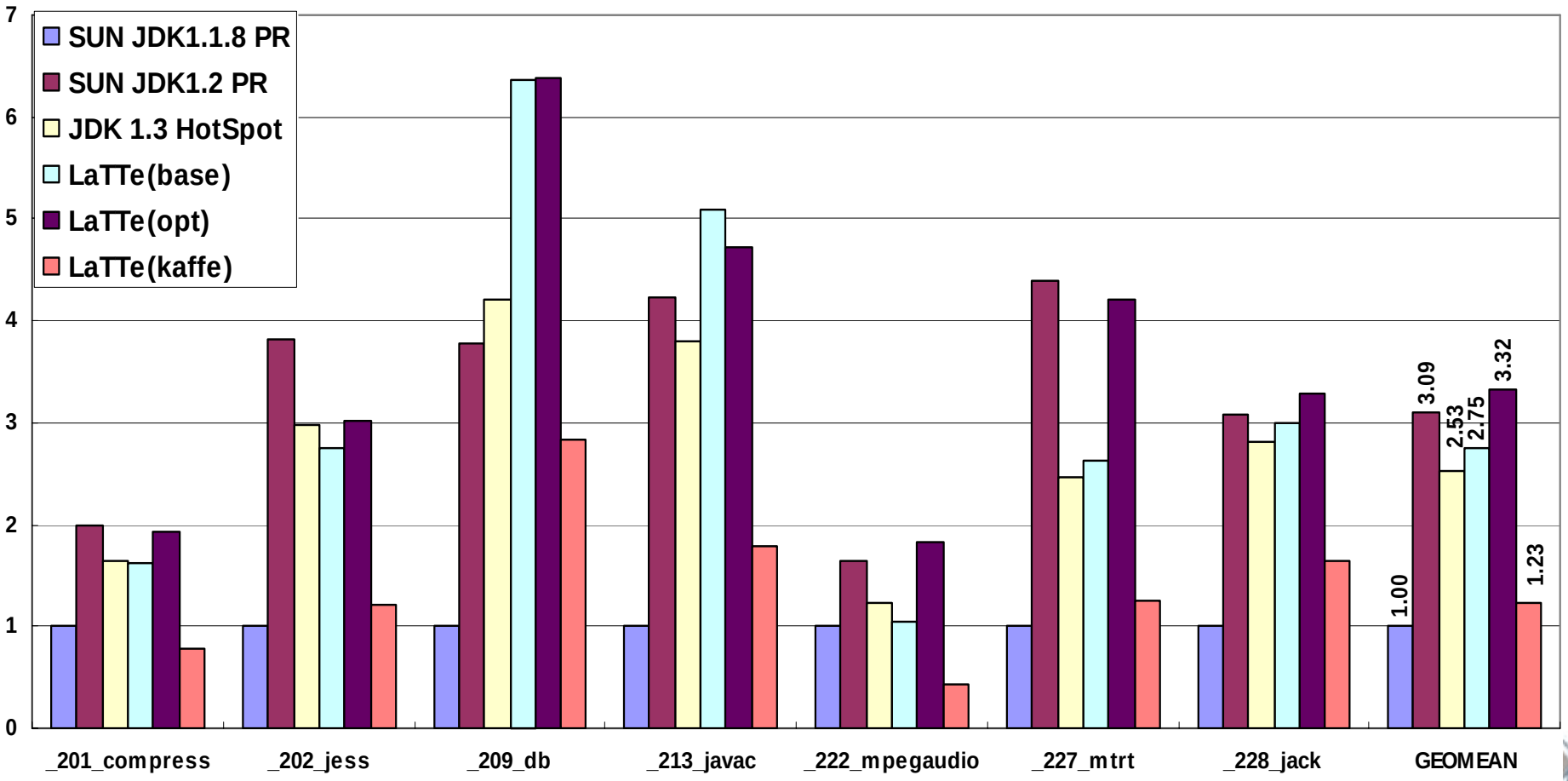


Analysis of LaTTe JIT Compiler

- TR overhead is negligible in L(B) and even in L(O)
  - TR time in L(B) takes consistently 1-2 seconds for all programs that run 30-70 seconds with LaTTe
  - Except for **_213_javac**, TR time is small even in L(O).
  - L(B) spends more TR time than L(K) by factor of 3.
    - LaTTe JIT of L(B) : 12,000 SPARC cycles/bytecode
    - Kaffe JIT : 4,000 SPARC cycles/bytecode
- Compared to Kaffe JIT, LaTTe JIT of L(B) improves JVM performance by factor of 2.2.

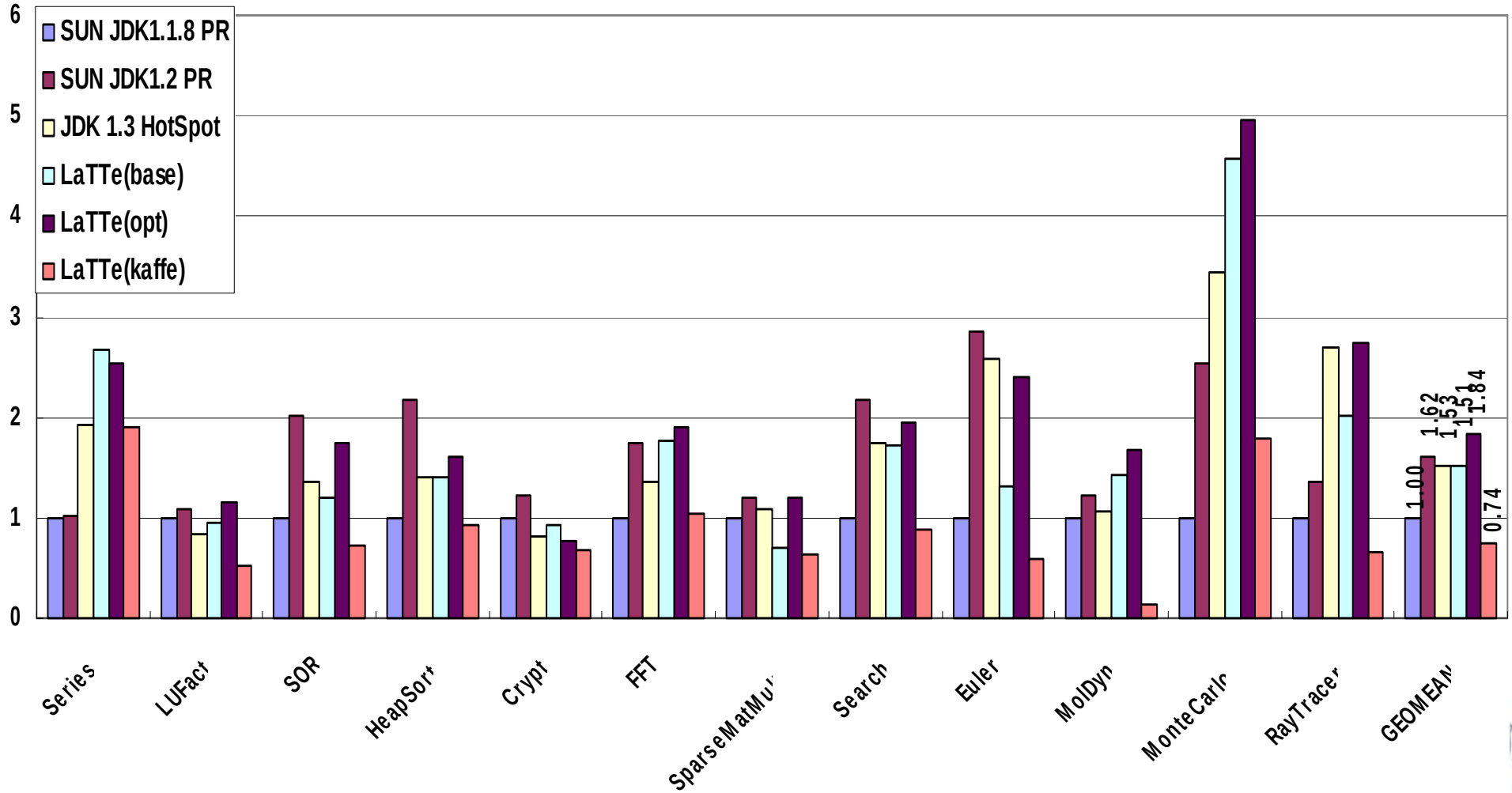
Overall Performance of LaTTe

Relative performance compared to SUN JDKs



Overall Performance of LaTTe

Relative performance compared to SUN JDKs



Summary

- LaTTe's performance is competitive, due to
  - Fast and efficient JIT compilation and optimizations
    - Virtual call overhead reduction technique
  - Lightweight monitor implementation
  - Efficient exception handling
  - Highly-engineered memory management module

- Source code available at http://latte.snu.ac.kr
  - BSD-like license

Future Work

- Proceed with further optimizations (a lot of leeway still available)
- Aggressive re-optimization of frequent code
- VLaTTe: JIT compiler for VLIW
- Re-integration with Kaffe, multiple platforms
- …
- We invite volunteers worldwide to join our LaTTe open source development team
  - and help us implement the exciting, leading edge JIT compiler, VM and instruction level parallelism optimizations to come