LaTTe and VLaTTe : Java VM Just-in-Time Compilers for RISCs and VLIWs

ByungSun Yang, Seongbae Park, Suhyun Kim,
Junpyo Lee, SeungIl Lee, Jinpyo Park,
Yoo C. Chung, Heungbok Lee
(PI: Prof. Soo-Mook Moon)

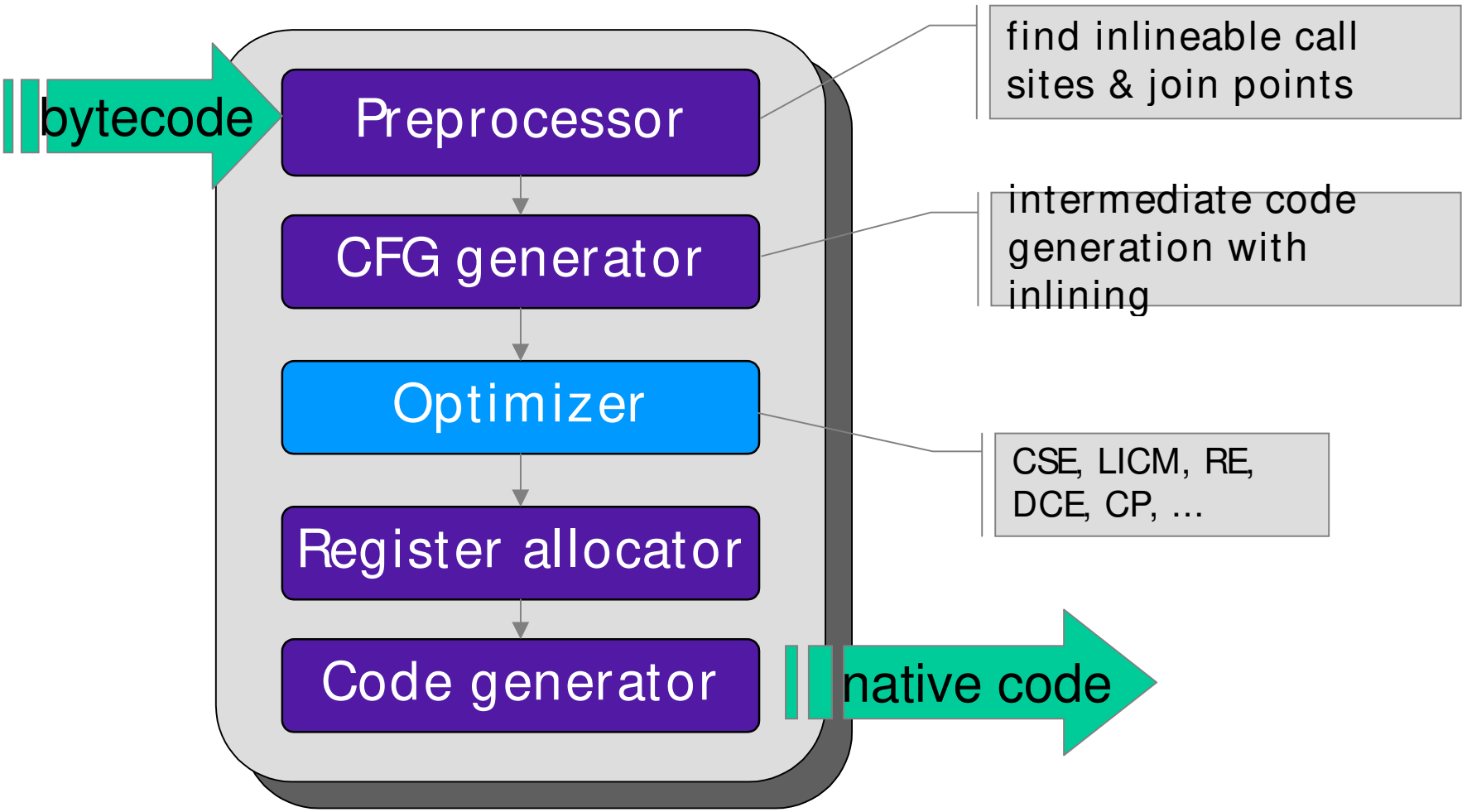
School of Electrical Engineering
Seoul National University, Korea

This work has been supported by IBM T. J. Watson Research Center

Project Overview

- Fast and efficient Java JIT Translation
  - LaTTe : Fast and efficient register allocation for RISCs
  - VLaTTe: Fast register allocation/scheduling for VLIWs
- Optimizing other JVM components
  - Lightweight monitor
  - On-demand exception handling
  - Optimized mark-and-sweep garbage collection
  - Adaptive-compilation technique
- Started from Kaffe JVM 0.9.2

LaTTe JIT Translation Overview



Fast Register Allocation

- Algorithm
  - Left-edge algorithm applied to tree regions (extended BB)
  - Two-pass algorithm with local lookahead
    - backward sweep: collects preferred target register information
    - forward sweep: allocates registers based on the information
  - Reconciling register maps at join points of tree regions
- Advantages
  - Aggressive copy elimination
  - Much faster than global register allocation

Register Allocation Example

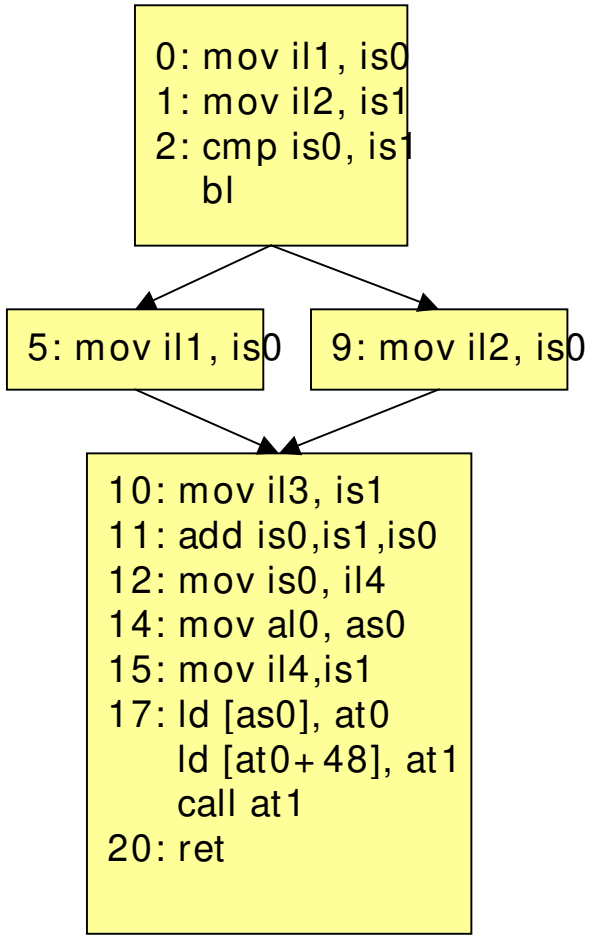
Java source

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

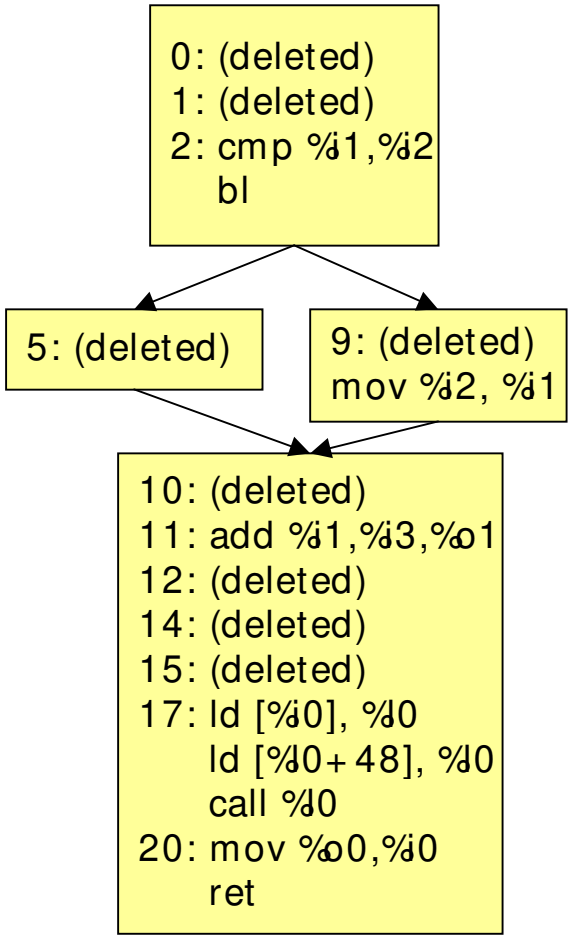
bytecode

```
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
```

before register allocation



after register allocation



Optimizing Other JVM Components

- Lightweight monitor
  - Embeds a 32-bit lock in each object
  - Inlines lock manipulation code that is highly optimized (acquisition and release: 9 and 5 SPARC instructions)
- On-demand exception handling
  - Delays translation of **`catch`** blocks until they are needed
  - Keeps consistency of register allocation with **`try`** blocks
- Optimized garbage collection
  - Conservative mark & sweep GC

VLaTTe JIT Translation

- Register allocation : left-edge algorithm
- Scheduling : list scheduling algorithm
- Register allocation and instruction scheduling are well combined in a unified framework
- Precise speculative exception handling

Current Status

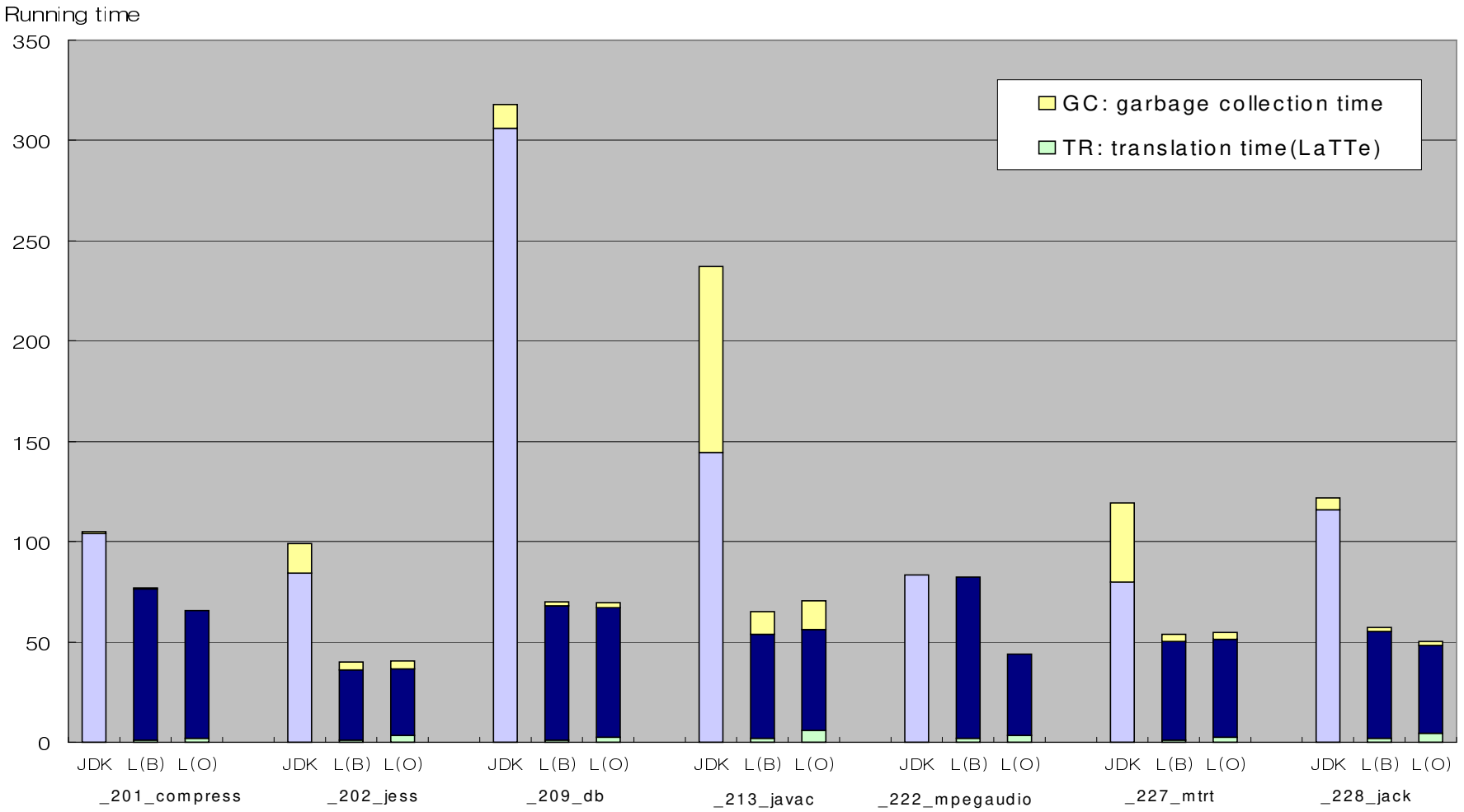
- LaTTe JVM is working on SPARC and PPC
  - Better performance than JDK 1.1.6 with JIT compiler
    (Geomean of 2.5X on SPECJVM98)
  - Comparable to JDK 1.2.1 HotSpot RC JIT Compiler
    (Geomean of 1.1X on SPECJVM98)
- VLaTTe is working on SPARC- based VLIW
  - Useful IPC on SPECJVM98
    - 2.33 on 8- ALU VLIW
    - 2.11 on 4- ALU VLIW
- Adaptive compilation technique

Experimental Results

- Test environment
  - SUN Ultra5 270MHz with 256MB of memory, Solaris 2.6
  - Single-user mode
  - Run 5 times and take minimum value
- Benchmarks

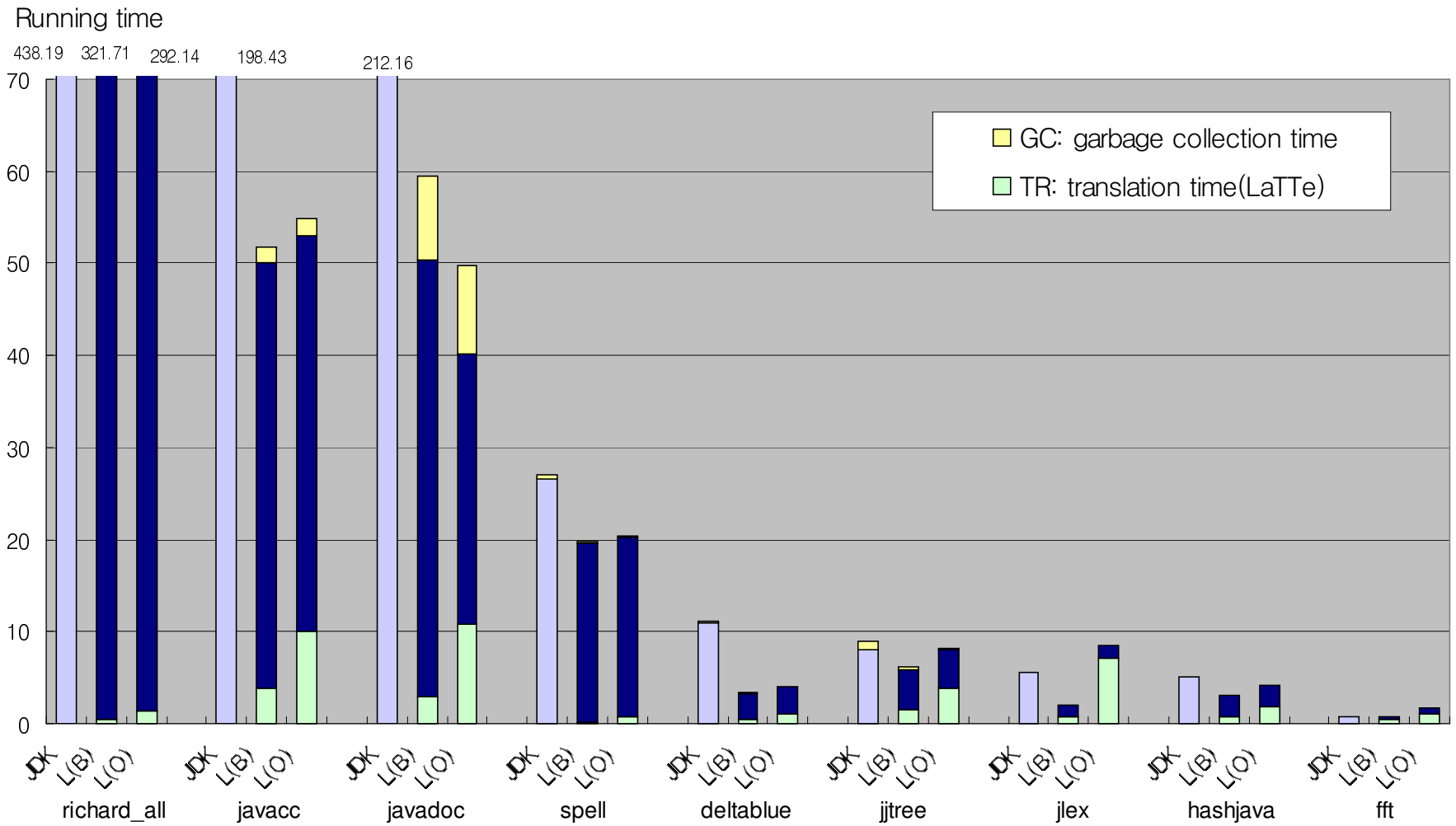
Benchmark	Description	Bytes	Benchmarks	Description	Bytes
_200_check	JVM checker	27606	_201_compress	Compress Utility	24326
_202_jess	Expert Shell System	45392	_209_db	Database	26425
_213_javac	Java Compiler	92000	_222_mpegaudio	MP3 Decompressor	38930
_227_mtrt	Raytracer	34009	_228_jack	Parser Generator	51380
richards_all	Richards Benchmark	17164	javacc	Parser Generator	118780
spell	Spelling Checker	4490	deltablue	Constraint Solver	9491
javadoc	Document Generator	71190	jlex	Lexical Analyser	25760
jjtree	JavaCC Preprocessor	60584	fft	Fast Fourier Transformer	8135
hashjava	Java Obfuscator	24412			

LaTTe vs. JDK 1.1.6

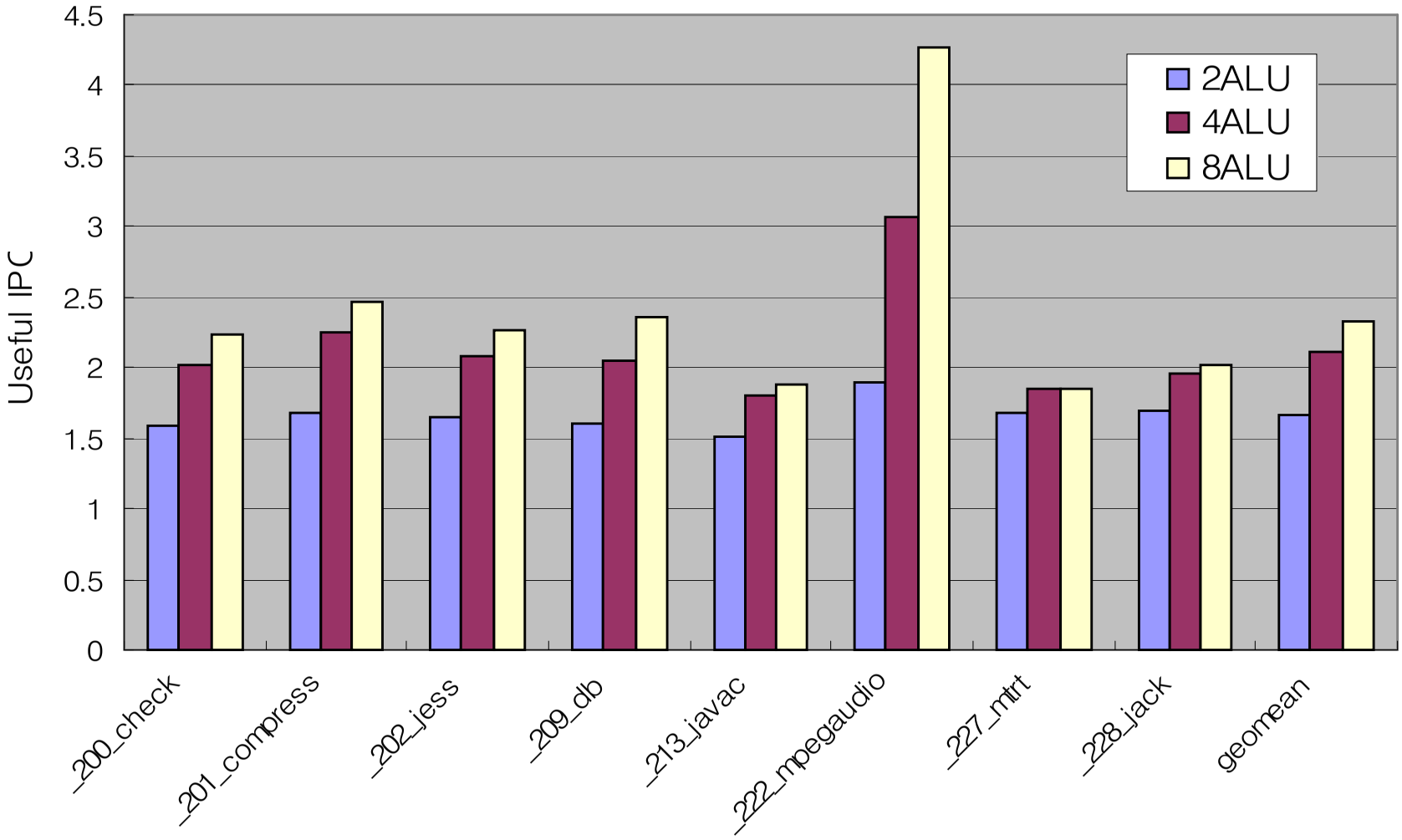


JDK: JDK 1.1.6
L(B): LaTTe (base) : fast register allocation only
L(O): LaTTe(optimizing) : L(B) + other optimizations

LaTTe vs. JDK 1.1.6(cont)



VLaTTe Useful IPC



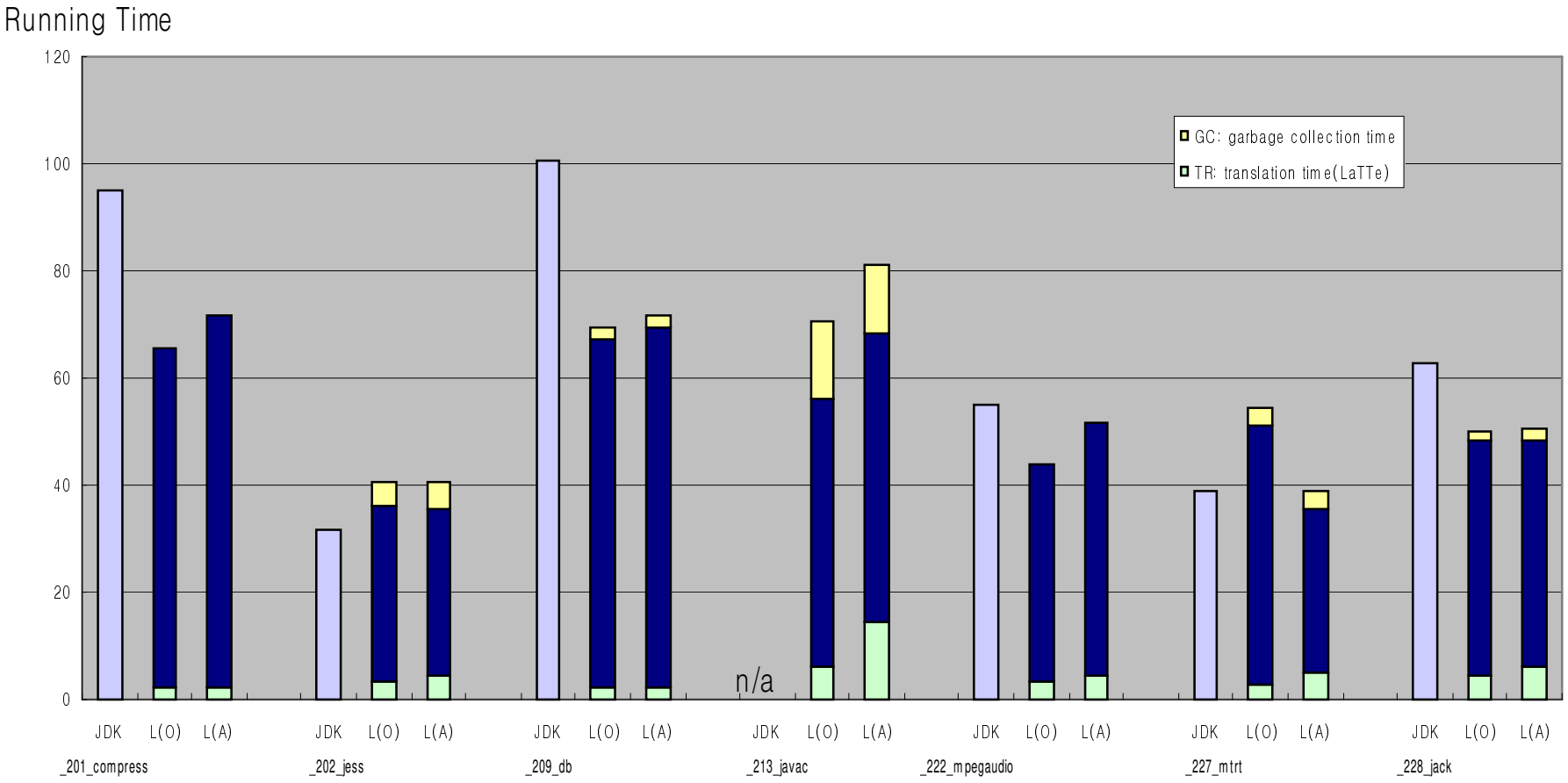
Source Code Availability

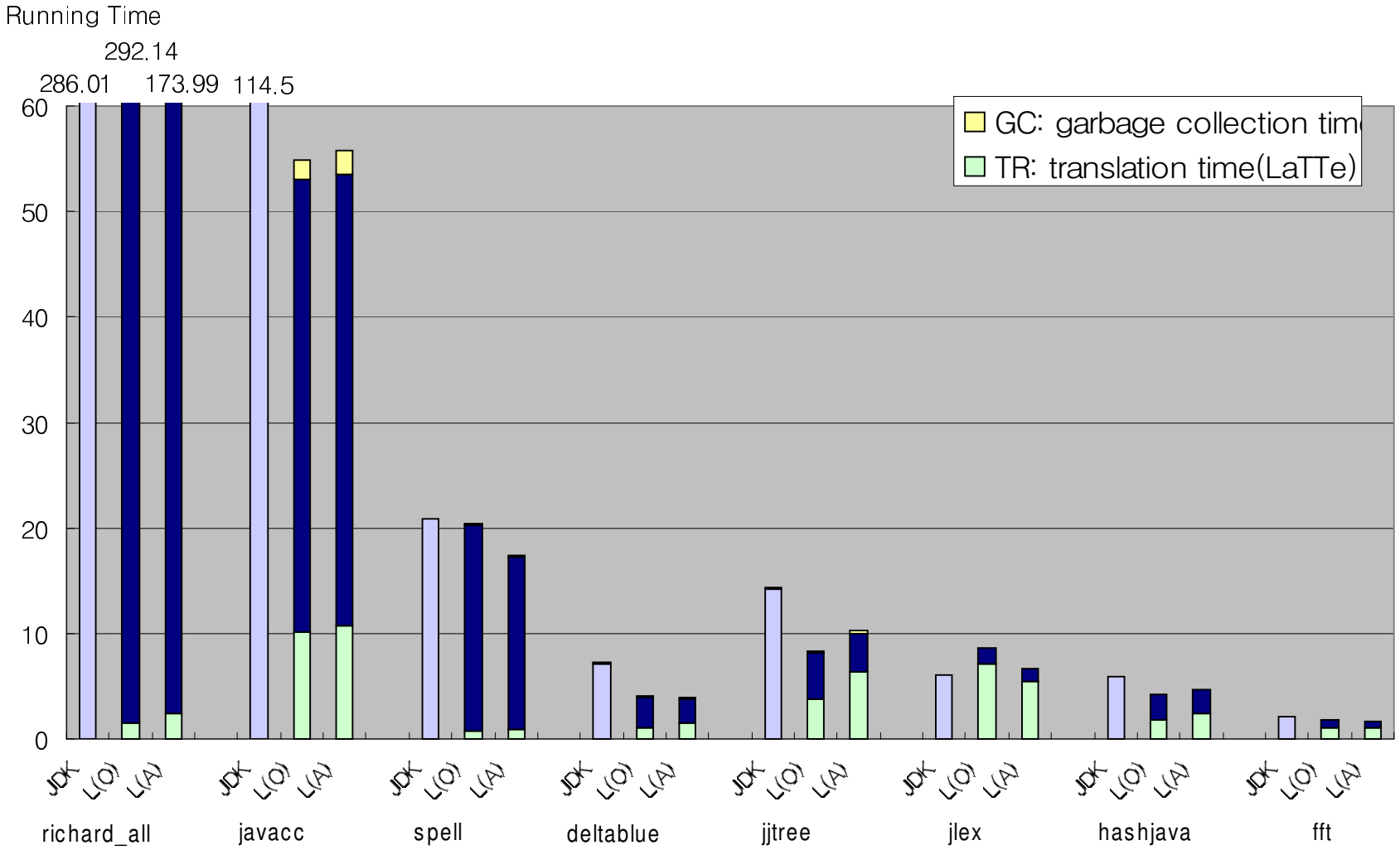
- Plan to release the source code of LaTTe (SPARC version) in August 1999
  - Visit our web page at http://altair.snu.ac.kr/latte/
- Commitment to fully support maintenance
- Support for other platforms
- VLaTTe will also be released soon

Adaptive Compilation

- Current status
  - Initial translation with minimum optimizations
  - Re-translation of frequently called methods with full optimizations (including inlining)
  - Virtual method inlining based on run-time profile
  - Apply customization & specialization by type analysis
- On-going work
  - Use interpreter for the initial execution
  - Improve hot-spot detection heuristics
  - Better register allocation (graph coloring)
  - More traditional optimization techniques

LaTTe vs. JDK1.2 hotspot RC



JDK: JDK1.2.1 HotSpot Release Candidate
L(O): LaTTe (optimizing) : fast register allocation + other optimizations
L(A): LaTTe (adaptive) : L(O) + adaptive compilation (preliminary version)

LaTTe vs. JDK1.2 hotspot RC (cont)