Performance of the SHA-3 Candidates in Java

Christian Hanser

Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a, A-8010 Graz, Austria

23. March 2012

Outline

Outline

Goals

- implement all SHA-3 finalists in Java,
- optimize them as good as possibe, and
- compare their performance

Motivation

Why Java?

very widespread,

platform-independent, and

features powerful crypto-framework

Motivation (cont.)

Why optimize Java code . . .

as Java is high-level, and

execution strongly depends on JRE ?

Motivation (cont.)

Why optimize Java code . . .

as Java is high-level, and

execution strongly depends on JRE ?

Still many ways to . . .

ruin performance,

put incentives for generation of fast JIT code

Outline

Approach

study C implementations,

port optimized implementations to Java, and

heuristically optimize Java code

Optimizing Java Code - Generic Strategies

(partial) loop unrolling,

caching intermediate results,

simplifying arithmetics,

manual method inlining, and

buffers instead of allocations in hot spots

Optimizing Java Code - Java-specific Strategies

Reduction of Boundary Checks

flattening multi-dimensional arrays,

replacing small arrays with variables, and

caching repeated array accesses

Optimizing Java Code - Java-specific Strategies

Reduction of Boundary Checks

flattening multi-dimensional arrays,

replacing small arrays with variables, and

caching repeated array accesses

Enable Automatic Inlining

private/static/final method modifiers,

splitting methods into smaller pieces, and

removing local variables

Outline

Benchmarks

Device

Intel Core i5-2540M (2x2.6GHz, 4 threads, 3MB L3, turbo-mode disabled),

8GB DDR3,

Ubuntu 11.10/amd64,

Java 6 (i386 and amd64)

Approach

warm-up before each measurement,

- allows JIT to find and compile hot spots, CPU to build jump prediction tables, ...

Benchmark Charts

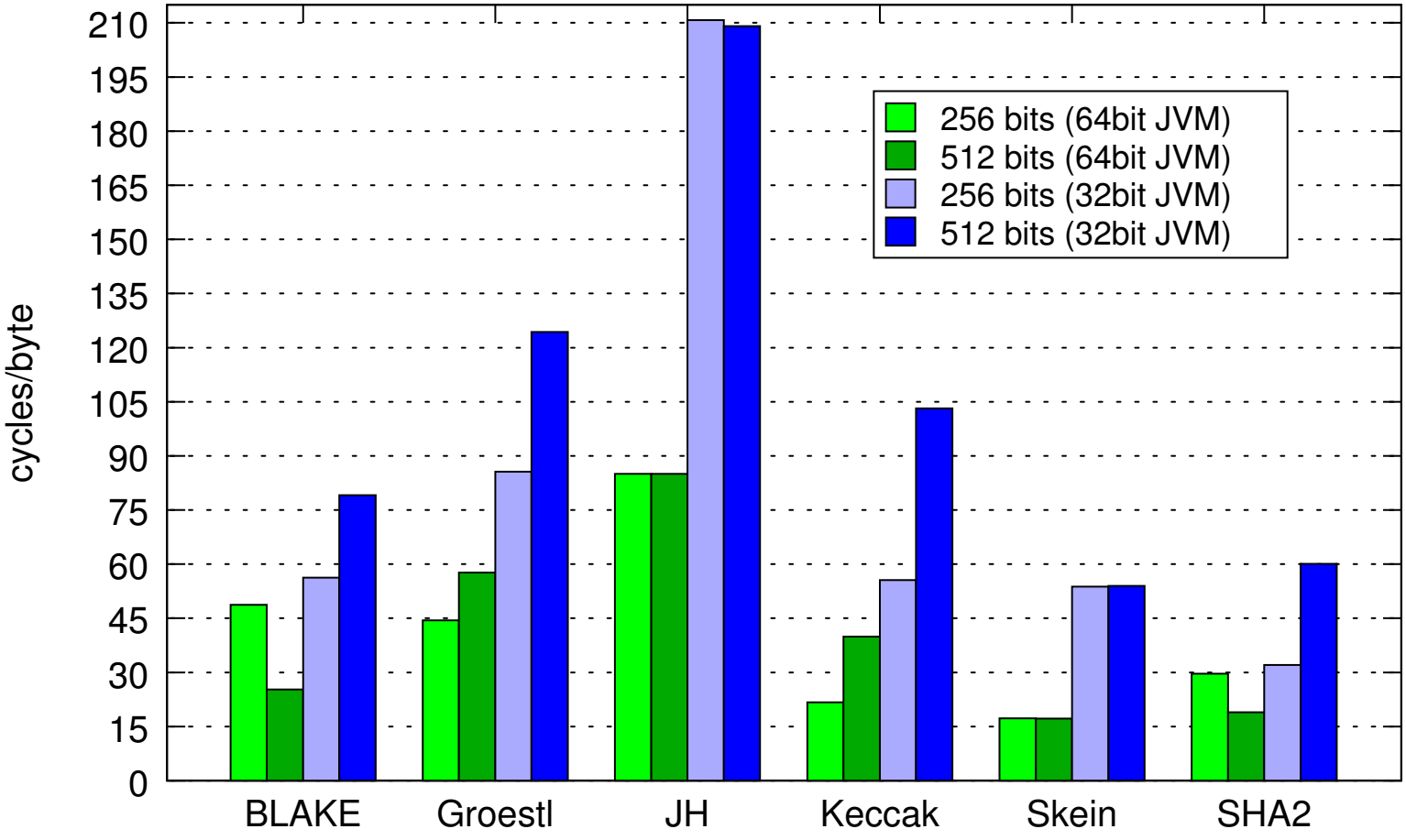

Figure: Performance of SHA-3 finalists and SHA2 in Java

Performance: Java vs. C

Java up to 3 times slower than C, e.g.

Skein-512

ANSI C: $\approx$ 6.1 cycles/byte [3],

Java: $\approx$ 17.2 cycles/byte

Optimizations Pay Off - Even in Java

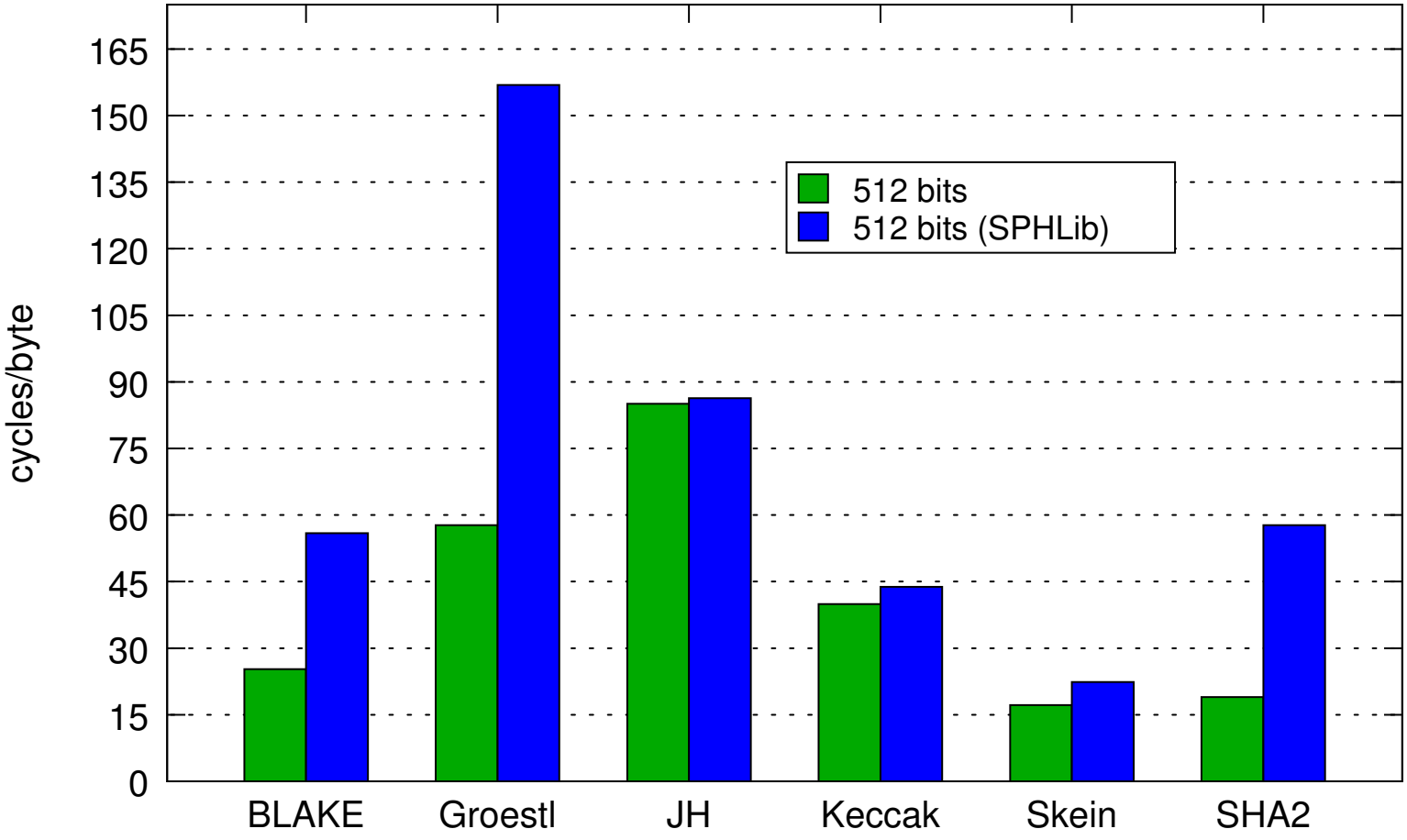

Figure: Performance Comparison with [1, 2] (Java/amd64)

Optimizations Pay Off - Even in Java (cont.)

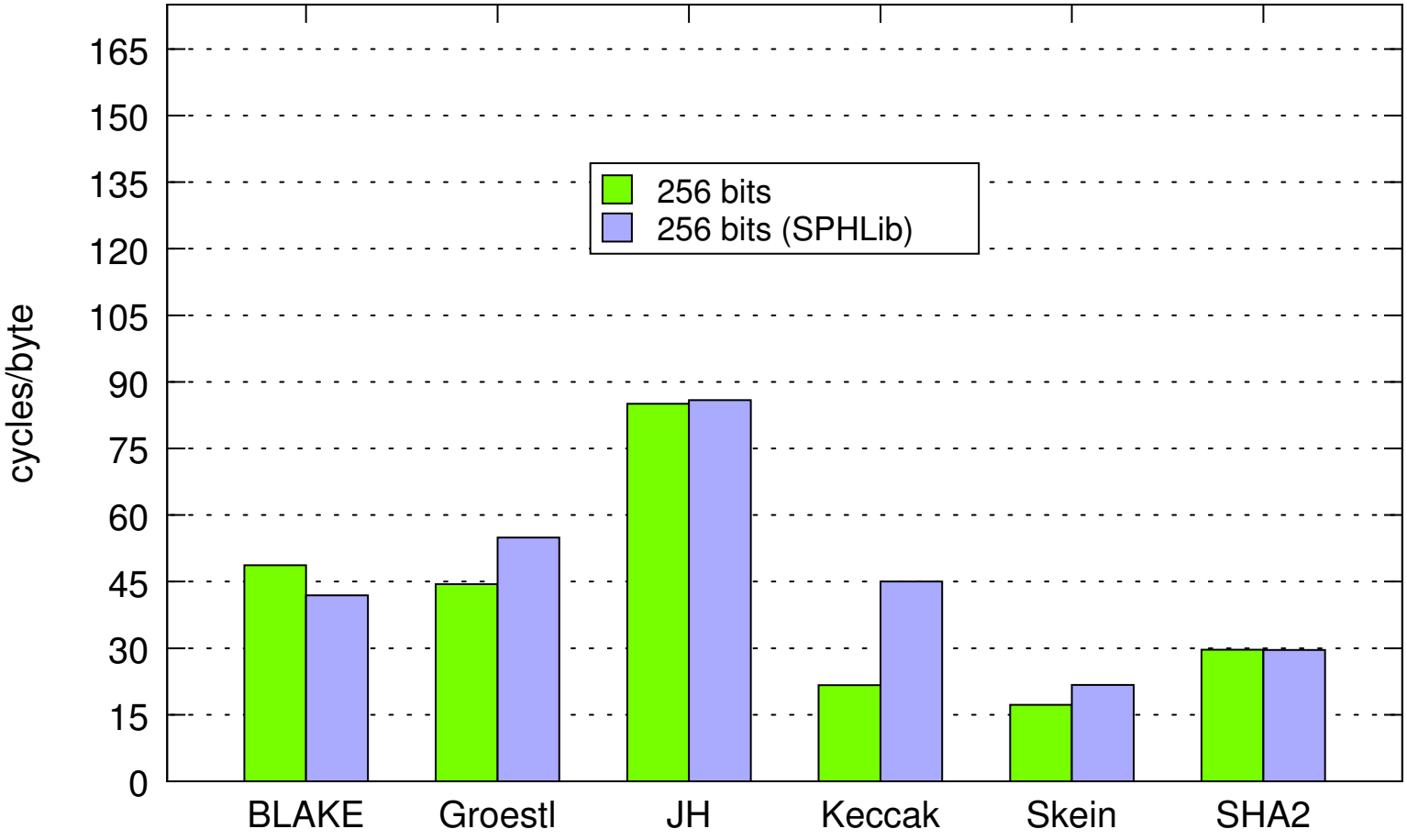

Figure: Performance Comparison with [1, 2] (Java/amd64)

Side Note: Grøstl with AES-NI via JNI

using latest AES-NI Grøstl-256 implementation,

bundle data to avoid frequent context switches,

$\approx$ 4.1 cycles/byte penalty for switching

Result: $\approx$ 15.6 cycles/byte

Side Note: Grøstl with AES-NI via JNI (cont.)

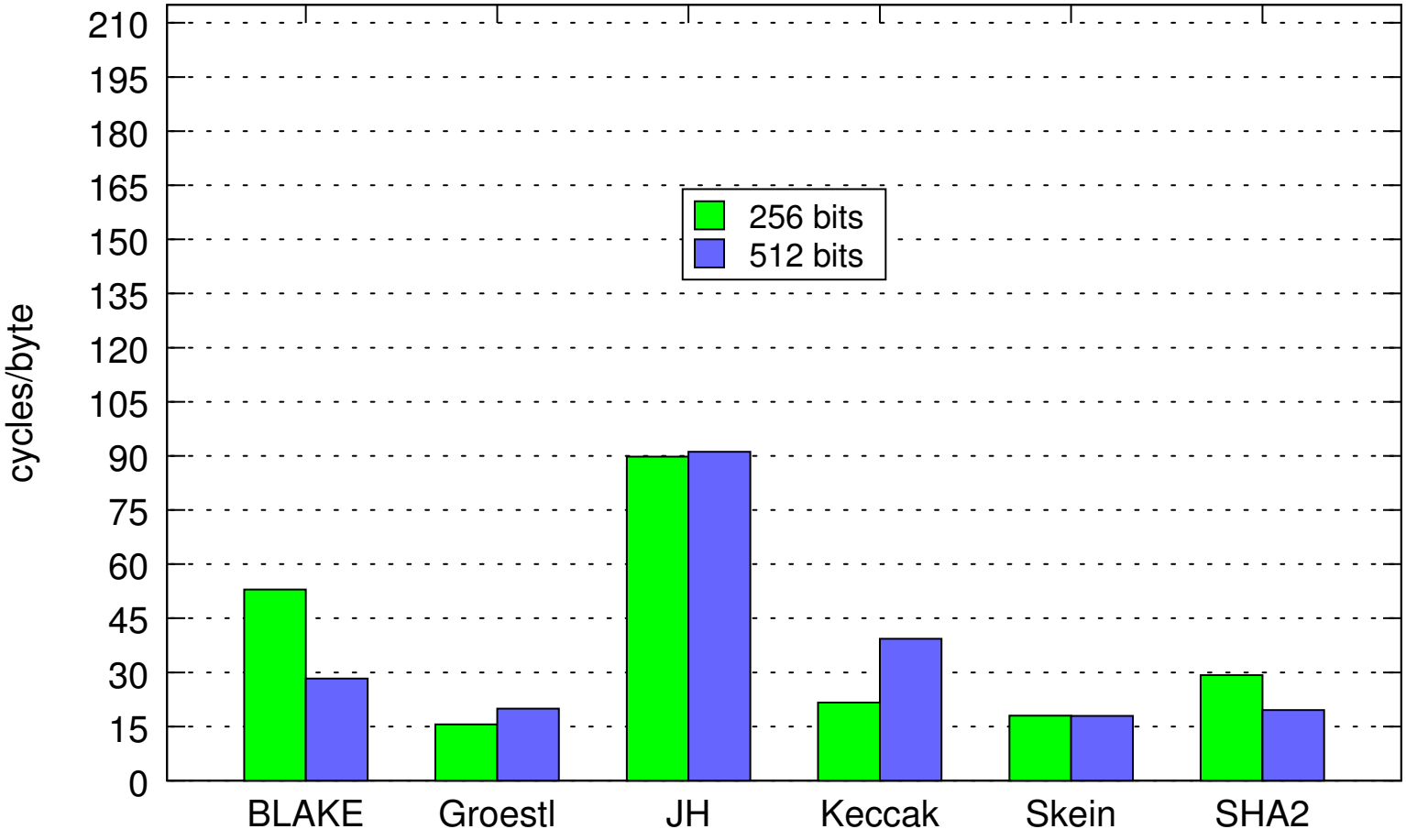

Figure: Performance Comparison with [1, 2] (Java/amd64)

Outline

Conclusions

Java up to 3 times slower than C,

optimization efforts pay off,

only Skein faster than SHA-2,

BLAKE and Keccak perform well,

Grøstl fast when using AES-NI via JNI, and

JH trails the field

Literature

Thomas Pornin

sphlib 3.0.

http://www.saphir2.com/sphlib/

Thomas Pornin

Comparative Performance Review of the SHA-3 Second-Round Candidates.

2010

Niels Ferguson et al.

The Skein Hash Function Family

2010

Thank you for your attention!

Feel free to download the implementations from:
`http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits`

Outline

Outline

BLAKE

Successful Optimizations

flattening permutation array,

per-object buffers,

simplifying index calculations, and

manual method inlining

Implementation Complexity

two distinct implementations,

little effort needed to understand and to implement

Grøstl

Successful Optimizations

flattening lookup-table,

splitting `RNDP` and `RNDQ` into smaller pieces,

partial loop unrolling,

manual method inlining,

private/static/final modifiers,

per-object buffers

Implementation Complexity

four distinct implementations,

little effort to understand implementation

JH

Successful Optimizations

replacing arrays by variables,

partial loop unrolling in `E8`,

manual method inlining,

private/static/final modifiers,

per-object buffers

Implementation Complexity

two implementations,

extensive optimization effort

Keccak

Successful Optimizations

assigning state array to local variables,

partial loop unrolling in absorb method,

private/static/final modifiers,

replacing XOR with OR in rotational shift *(sic!)*

Implementation Complexity

two implementations,

totally new notions,

confusing C implementation (extensive use of preprocessor),

additional complexity due to interleaving

Skein

Successful Optimizations

assigning state array to local variables in compress step,

replacing XOR with OR in rotational shift *(sic!)*

per-object buffers,

bundling assignments from buffer to key-schedule,

local caching of frequently used values,

private/static/final modifiers

Implementation Complexity

easy to implement,

only one implementation

Outline

Benchmark Charts for Windows 7

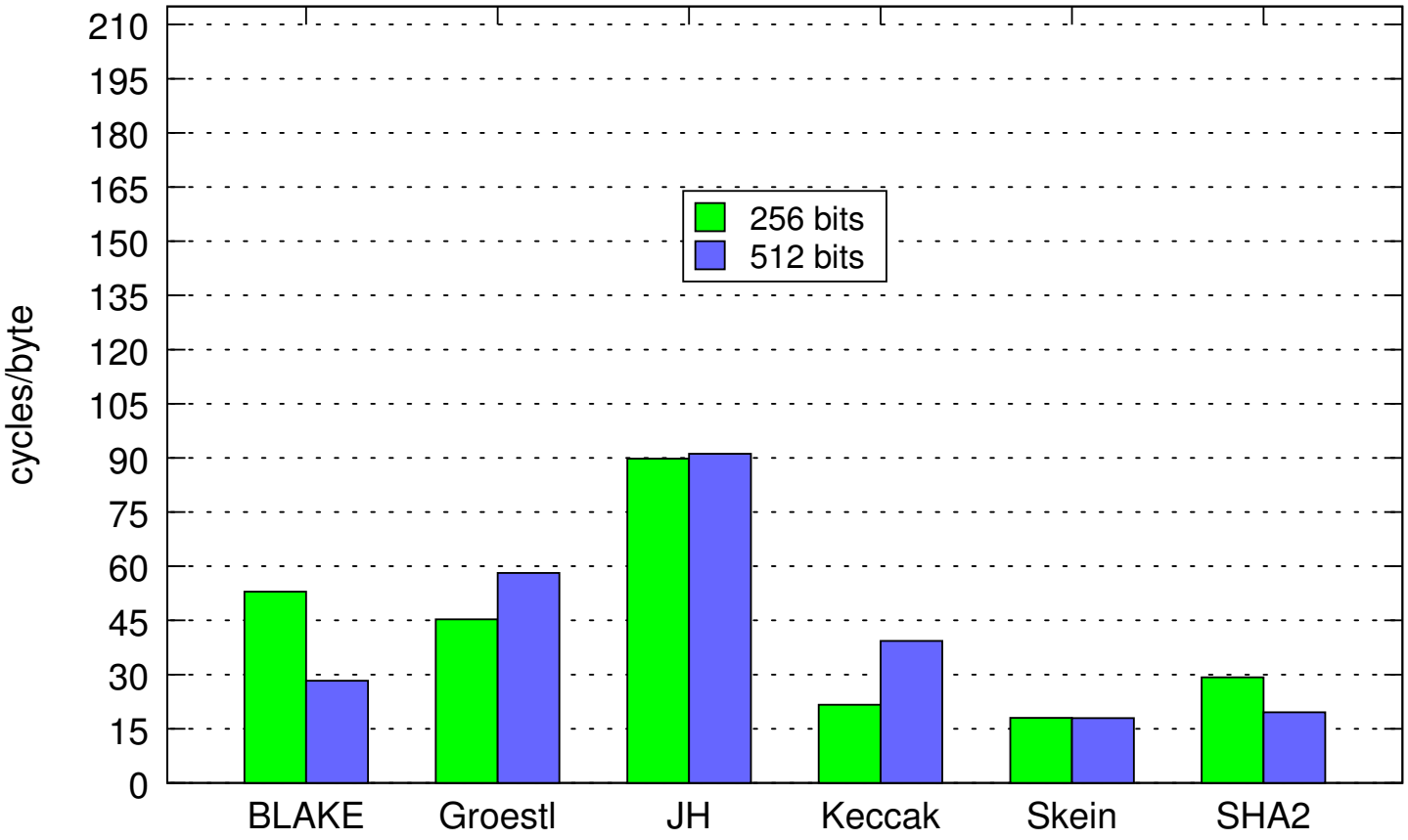

Figure: Performance of SHA-3 finalists and SHA2 on Java/amd64