Performance of the SHA-3 Candidates in Java

Christian Hanser, chanser@iaik.tugraz.at

Institute for Applied Information Processing and Communications
Graz, University of Technology
Inffeldgasse 16a, A-8010 Graz, Austria
http://www.iaik.tugraz.at

March 19, 2012

Abstract

In this paper we analyze how the five finalists of the NIST SHA-3 competition perform in Java. Our Java implementations are derived from the official, optimized implementations for both 32-bit and 64-bit platforms. However, the derived implementations did not turn out to be optimal. Hence, we have determined different optimization strategies and applied combinations of them to each algorithm. We are going to discuss these strategies and present the combinations that we consider to be best for each SHA-3 candidate. Furthermore, we present benchmark results for both 32-bit and 64-bit environments and give an estimation of the implementation complexity based on our experience. We can greatly improve the results of a related performance study and come to the conclusion that the fastest algorithm is about five times faster than the slowest.

Keywords: SHA-3, Java, implementation, benchmark, performance

1 Introduction

In 2007, NIST has started its SHA-3 competition with the aim of finding a successor to the outdated SHA-1 hash algorithm, analogously to its AES competition (1997-2001). From an initial number of 64 submissions, 51 submissions made it into round 1, 14 of these into round 2 and finally five into round 3. In 2012, one of these five finalists is going to become the new SHA-3 hash algorithm. The five candidates are called BLAKE, Grøstl, JH, Keccak, and Skein. Until now, the implementers have mainly focused on optimizing C and assembler code and less on optimizing implementations in high-level languages, such as Java. In 2010, Thomas Pornin, the author of the second round candidate Shabal, published a comparative performance study that includes Java benchmarks for all second round candidates [11]. He included these algorithms into version 2.1 of his Java library named `sphlib` [12]. In July 2011, Pornin published a new version of this library, which includes updated versions of the SHA-3 finalists. To make his results comparable to our results, we re-evaluated Pornin's latest implementations with our benchmark framework. In this paper we revise the benchmark results of Pornin and present new, greatly improved benchmark values of the SHA-3 candidates in Java.

Section 2 gives an overview of our implementation. Section 3 details our benchmark approach and our evaluation parameters, and Section 4 lists our optimization techniques. Section 5 explains how we were able to optimize each candidate, and Section 6 illustrates and lists our findings. Section 7 draws a comparison with the results of the aforementioned study, and finally Section 8 summarizes and concludes this paper.

1.1 Why Java?

Currently, Java constitutes one of the most widespread programming languages, next to C and C++. Its popularity arises from its platform independence (of both source code and byte code), its extensive development kit and its high-level features. Java has a big developer community, which is still growing

steadily, not only since Java is the programming language of choice for Google's rapidly evolving Android platform. Furthermore, Java features a powerful crypto-framework and hence is widely used for cryptographic applications. Taking all this into account, we consider it worthwile to thoroughly analyze the Java performance of the SHA-3 finalists.

1.2 Why Would One Want to Optimize Java Code?

The widespread opinion regarding Java programming is that one should not put much effort into optimizing code, since it is a high-level language and its execution strongly depends on the runtime environment and the target device. To get to the heart of it: code optimized for one setting, can be slow in another setting and besides leads to obfuscated, badly readable code. Donald Knuth summarizes these problems as follows: "Premature optimization is the root of all evil." [8].

But this idyll of a high level language is deceptive: there are many ways one can ruin the performance and many possibilities to put incentives for the generation of fast JIT code. These incentives should be chosen in such a way that there are no extra costs in the worst case. Most, but not all of our optimization strategies follow this principle. Hence, we expect many of our optimizations to be advantageous for other settings too.

2 Implementation Notes

The code was developed at the IAIK by Christian Hanser and was released in form of a library at our website, see [6]. It is licensed under the GNU Public License (GPL). The library follows the design principles of Java Cryptography Architecture (JCA), i.e. for each calculation of a message digest, a Java object of type `java.security.MessageDigest` is used. This object holds the hash context, which is being repeatedly updated with message blocks using the `update()` method. At last, the hash value can be obtained by a call to the `final()` method.

As mentioned before, all implementations are derived from the optimized reference implementations and have been subject to extensive optimization efforts.

3 Evaluation Parameters and Measures

The hash algorithms were implemented in Java and evaluated with regard to throughput and implementation complexity. This section details what we were measuring, how we were measuring and describes the environments we were using for our measurements.

3.1 Execution Speed

In general, a hash algorithm has three different execution stages: initialization, compression and finalization. Initialization involves the generation of lookup tables, parameter initialization and the like. We did not measure the initialization phase, since it requires only little time and often depends on the trade-off code-size vs. execution speed. The compression phase consists of iterative applications of the compression function to each block of input. In every iteration the internal hash state is transformed using the current state and one message block. The amount of time spent in this phase depends directly on the message length and accounts for the largest part of the overall execution time. Thus, in this study we concentrate on benchmarking the compression phase. The finalization phase usually involves the compilation of the padding block(s), the compression thereof and a final transformation on the resulting value. The time spent in this phase is negligible and was not measured.

The benchmark runs were executed consecutively, where each run starts with a warm-up phase lasting one second, which is followed by the actual benchmark phase lasting five seconds. Both phases are exactly the same, except for the execution time. The warm-up phase allows the HotSpot VM to identify and just-in-time compile the execution hot spots and the processor to build jump prediction tables and perform other optimizations.

A single benchmark phase looks as follows: as input for the compression function a byte array of the size of one message block was generated using Java's default PRNG `java.util.Random`, initialized with seed value 0. This byte array was repeatedly fed into the digest's update method for $n \in \{1, 5\}$ seconds. A second thread was used to control the timings. At the beginning as well as at the end of one

benchmark round, the current user time of the main thread was taken. Let t be the difference of these two values. Then, the benchmark result is given by:

$$\frac{t \cdot f}{c \cdot blen} cycles/byte,$$

where f, $blen$, and c denote the CPU-frequency, block length, and the iteration count, respectively.

3.2 Implementation Complexity

Next to performance characteristics we also evaluated the implementation complexity of each algorithm. Our rating comprises the time required to sufficiently understand and implement a hash function, especially the number of implementations necessary.

3.3 Setting

The results published in this paper were achieved on an Intel Core i5-2540M (2.60 GHz, 2 cores, 3 MB L3 cache, turbo mode disabled) with 8 GB DDR3 RAM running Ubuntu 11.10/amd64, which constitutes the reference platform for this paper. In order to test our 32-bit and 64-bit implementations we were using Sun JDK 1.6.0.25/i386 and OpenJDK 1.6.0.23/amd64, respectively. On the same computer we were also running the benchmark using 64-bit Windows 7 SP1 in combination with Sun JDK 1.6.0.26/amd64. In all cases the Java VMs were running in server mode, the fastest setting of the HotSpot VM.

4 Optimizing Cryptographic Algorithms for Java

Optimizing (cryptographic) algorithms for Java is by far not as straight-forward as for classic languages such as C. Due to different, not always obvious, execution strategies chosen by the runtime environment this is a rather heuristical task. In addition, not every combination of optimization strategies turned out to be successful. So, we had to determine a set of suitable optimization strategies for each implementation. Table 1 gives an overview of our optimization measures.

4.1 Generic Optimization Measures

There are classic optimizations such as (partial) loop unrolling and the caching of intermediate results, which save computations and conditional jumps. The former measure, however, must be used with caution. Only a small number of iterations should be unrolled. Otherwise, it can happen that the method code no longer fits into the CPU's instruction cache, leading to a significant loss of speed.

Other apparent modifications are the manual inlining of methods the compiler does not inline on its own, and the outsourcing of array allocations inside the compression function to per-object buffers in order to reduce the time spent in the compression function.

Optimization Strategy	**Goal**
(partial) loop unrolling caching intermediate results simplifying (index) arithmetics	save computations and conditional jumps
replacing arithmetical operations by equivalent faster operations	speeding up arithmetical computations
flattening of multi-dimensional arrays replacement of arrays by member variables caching repeated accesses to array elements	reduce number of boundary checks
introducing per-object buffers	reduce array allocations in hot spots
manual method inlining (where the compiler can not inline)	save method invocations
rewrite methods: supplementing private/static/final modifiers splitting methods into smaller pieces removing local variables	allow the compiler to inline methods

Table 1: Optimization Strategies for Java Implementations

4.2 Java-specific Optimization Measures

Then, there are modifications that are specific to the Java language. The performance can be greatly improved by allowing the compiler to inline methods. The premises to enable this are: the according method must be short, must not define local variables and must be marked either as `private`, `static`, or `final`. Another optimization measure is to reduce the number of boundary checks. On each array access, Java checks whether the array index is inside the valid bounds. This can be overcome by either replacing small arrays with an appropriate number of variables or assigning repeatedly accessed array elements (e.g. of state array) to local variables. Apparently, the performance penalty of array accesses increases if multi-dimensional arrays are involved, since the JVM must perform boundary checks for every array dimension. We reduced these costs by flattening multi-dimensional arrays.

Finally, we made a peculiar observation: replacing `XOR` by `OR` in a circular shift, which is equivalent in this context, led to a speed-up of 3 and 4 cycles per byte for Skein and Keccak, respectively. There seems to be a problem with Java's JIT, as each of these instructions maps to a single byte code instruction and the generated class files only differ in one byte, as expected. The first holds `0x83`, the byte code for the instruction `lxor`, where the second holds `0x81`, the byte code for the instruction `lor`. It would seem natural that JIT maps each one of these to its corresponding CPU instruction, yielding the same execution speed. Hence, for us there is no good explanation for this performance regression.

5 Implementation Results

Most SHA-3 candidate algorithms come along with separate implementations for 32 and 64-bit platforms. In order to get the best performance, we are following the same approach using `ints` and `longs` to represent the internal state. The optimized implementations are then chosen at runtime depending on the VM's data mode. The corresponding optimized C implementations served as starting point for each Java implementation. In the next step we applied several optimization strategies, as explained in Section 4. This way, we lowered the time spent executing the main hot spot, namely the compression function. The combination of optimization strategies turned out to be strongly dependent on the specific hash algorithm and on the VM's data mode. Thus, we were following a trial and error approach to maximize the throughput. In no case all strategies could be applied successfully. So, we had to find out the fastest combinations of such strategies for each implementation.

5.1 BLAKE

BLAKE was developed by Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C. W. Phan [1]. It is based on similar design principles as the SHA algorithm family, namely on add-rotate-xor (ARX) transformations. In the same manner as the SHA-2 family the BLAKE-family consists of a short variant for hash lengths of 224 and 256 bits and a long variant for hash lengths of 384 and 512 bits. The former uses 32-bit words, whereas the latter uses 64-bit words.

5.1.1 Optimization Strategies and Performance

Compared to some other SHA-3 candidates, BLAKE does not offer that many possibilities for performance gains. We improved its Java performance by:

1. converting the two-dimensional permutation array `SIGMA` to a one-dimensional array,
2. introducing buffers to reduce the amount of memory allocated in every iteration of the compression function,
3. simplifying index calculations, and
4. manual method inlining.

Applying the optimizations listed above to our plain C port, we were able to drop the performance from about 38 cycles/byte to 25.3 cycles/byte and over 88 cycles/byte to around 48.7 cycles per byte in the long and in the short version of BLAKE, respectively.

5.1.2 Implementation Complexity

BLAKE turned out to be relatively simple and fast to implement. With two different variants to implement, the time and the resources needed are kept within a reasonable limit.

5.2 Grøstl

Grøstl was proposed by Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Sren S. Thomsen [5]. It is a hash function based on design principles of the AES cipher. In its default implementation, there are two variants of Grøstl, one for short and one for long hash lengths, each having two implementations, one for 32-bit platforms and one for 64-bit platforms.

5.2.1 Optimization Strategies and Performance

The subsequent optimization techniques were successful:

1. converting the two-dimensional lookup-table to a one-dimensional array,
2. splitting `RNDQ, RNDP` into smaller methods to enable inlining,
3. arithmetical simplifications,
4. manual method inlining,
5. partial loop unrolling,
6. introducing buffers to reduce the amount of memory allocated in every iteration of the compression function, and
7. supplementing `private/static/final` modifiers.

Using the aforementioned code optimizations, we were able to drop the number of cycles/byte of our 64-bit implementation from roughly 98 to about 44.4 cycles/byte for both Grøstl-224 and Grøstl-256. Grøstl-384 and Grøstl-512 take approximately 57.7 cycles/byte in contrast to 160 cycles/byte of the plain C port.

5.2.2 Using AES-NI via JNI

Grøstl can be sped up tremendously by using the AES-NI instructions shipped with some models of Intel's Core i5/i7 series. Unfortunately, this is not a real option with Java and, thus, this section shall be seen as a non-competitive side note. One can make use of the AES-NI instruction set extensions by invoking native code through the Java Native Interface (JNI) at the expense of platform independence. The whole compression step moved into the native part with only the padding and the buffering left in the Java part. To our astonishment, the performance penalty of the frequent context switches between the native library and the JVM was only about 3.1 cycles/byte. The fastest native implementation of Grøstl takes 11.5 cycles per byte (see [5]) and invoked via JNI it needs around 14 to 15 cycles per byte.

5.2.3 Implementation Complexity

The amount of time necessary for optimizations and to understand Grøstl sufficiently enough to be able to implement it, is relatively low.

5.3 JH

JH is a one-man-project by Hongjun Wu [13]. Like most other SHA-3 candidates, JH is based on logical instructions. In sum, there are two implementations, one for 32-bit and one for 64-bit platforms, where only the initialization vectors (IV) vary for different hash lengths.

5.3.1 Optimization Strategies and Performance

The following optimization techniques were successful:

1. replacement of arrays by member variables,
2. partial loop unrolling of code in function `E8`,
3. manual method inlining,
4. introducing buffers to reduce the amount of memory allocated in every iteration of the compression function, and
5. supplementing `private/static/final` modifiers.

In this case, these optimizations turned out to be very effective. We were able to drop the number of cycles/byte from a total of 240 cycles/byte to 85.0 cycles/byte. Despite of these tremendous improvements, JH remains to be the slowest of all candidates.

5.3.2 Implementation Complexity

JH is comprised of a 32-bit and a 64-bit optimized version. Obtaining a valid implementation of JH was easy and straight-forward. Nonetheless, it took a long time and plenty of patience to find suitable optimizations.

5.4 Keccak

Keccak has been published by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche [2]. It is based on the so-called sponge construction, which is a "simple iterated construction building a variable-length input variable-length output function based on a fixed length permutation (or transformation)" [3]. It consists of an absorbing phase that updates the internal state using message blocks and a squeezing phase that produces the output. Both phases involve the permutation function, which is composed of logical instructions. The design of the Keccak family offers much flexibility. Two parameters, namely the rate r and the capacity c, define a member of the Keccak sponge family. The rate defines the block size and the sum $r + c$, which must be equal to 1600, defines the size of the state as well as the size of the permutation. The Keccak hash family is tailored to 64-bit systems. With a trick, however, called bit-interleaving, 64-bits of the internal state can be distributed to two 32-bit words using table-lookups. The number of table-lookups is kept within a reasonable limit. This is why, the 32-bit implementation has good performance, even in Java. The parameters r and c for the versions proposed to the NIST are chosen in such a way that c equals two times the hash length, which is necessary to achieve the required security level. So, for instance, Keccak-256 has rate $r = 1088$ and capacity $c = 512$ and Keccak-512 has parameters $r = 576$ and $c = 1024$. Since the block size is equal to the rate, this implies a difference of speed between Keccak-256 and Keccak-512.

5.4.1 Optimization Strategies and Performance

The following optimization techniques were successful:

1. replacing `XOR` in rotational shift with `OR`,
2. assigning the state array to local variables in the absorb method,
3. two-fold partial loop unrolling in the absorb method, and
4. supplementing `private/static/final` modifiers.

The result of the C to Java port can be seen as almost optimal. Our Java implementations of Keccak-256 and Keccak-512 require approximately 21.7 and 40.0 cycles/byte, respectively.

5.4.2 Implementation Complexity

It takes some time to get a comprehensive picture of Keccak, as Keccak's notions are (at least at first glance) totally different from other hash functions. Another obstacle was the C implementation of Keccak, which is confusingly complex due to the extensive use of preprocessor statements. After a time-consuming introductory phase and code simplifications achieved with the `gcc` preprocessor, the 64-bit Keccak can be implemented reasonably fast. The last obstacle the implementer is confronted with, is Keccak's 32-bit implementation, where interleaving and deinterleaving steps account for increased complexity.

5.5 Skein

Skein was developed by Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas and Jesse Walker [4]. It is based on Threefish, a tweakable block-cipher composed of ARX instructions. Skein was created with 64-bit platforms in mind and is the only algorithm that does not (yet) offer a separate implementation for 32-bit platforms. Furthermore, Skein is defined in three different variants: Skein-256, Skein-512, and Skein-1024. They differ in their memory requirements and their alleged security strength. Skein-512 is considered to be the default variant of Skein, so we were sticking to it.

5.5.1 Optimization Strategies and Performance

The following optimization techniques were successful:

1. assigning the state array to local variables in the compress routine,

2. replacing the `XOR` in the rotational shift with `OR`,
3. introducing buffers to reduce the amount of memory allocated in every iteration of the compression function,
4. bundling assignments from the buffer to the key schedule,
5. caching frequently used values locally, and
6. supplementing `private/static/final` modifiers.

With these measures the number of cycles/byte dropped from initially 25 to 17.2 cycles/byte. This is essentially the same speed that the Java reference implementation of the Skein team (see [9]) offers.

5.5.2 Implementation Complexity

Skein was very easy to implement with the least amount of time and effort needed, which is also due to the lack of a dedicated 32-bit implementation.

6 Benchmark Results at a Glance

This section gives the benchmark results we obtained on Linux and Windows.

Figures 1 and 2 show the benchmark results of the SHA-3 candidates on Linux. Two candidates have constant outcomes in every benchmark: Skein leads the field and JH comes in last. On the 64-bit VM the other algorithms perform as follows: for short hash lengths Keccak, Grøstl, and BLAKE are on the second, third and fourth place, whereas for long hash lengths Skein is followed by BLAKE, Keccak, and Grøstl in ascending order. On the 32-bit VM we get the following picture: for short hash lengths we have Keccak, BLAKE, and Grøstl on the second, third and fourth place, whereas for long hash lengths Skein is followed by BLAKE, Keccak, and Grøstl.

Figure 3 shows the benchmark results of the SHA-3 candidates on 64-bit Windows. These are similar to the according results on the Linux machine. Note that we are not dealing with the 32-bit Java Windows VM, as this specific type of VM seems to have severe legacy issues. In contrast to the other JVMs it is troublesome to optimize code for this specific VM, since some of the aforementioned optimization techniques, such as manual inlining and partial loop unrolling, often lead to performance regressions.

Finally, Table 2 summarizes the benchmark results obtained on our reference platform.

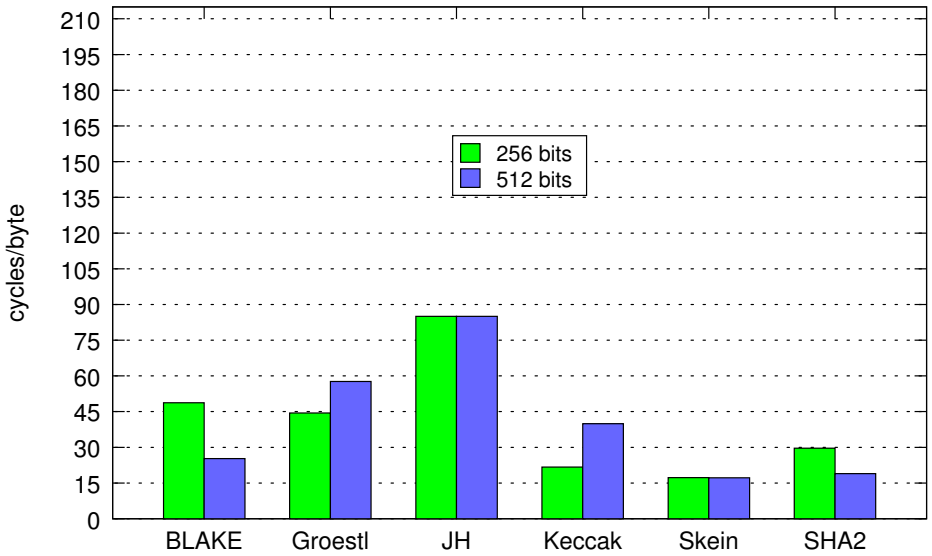

Figure 1: Performance of SHA-3 finalists and SHA2 on Java/amd64 (Ubuntu 11.10/amd64, Core i5-2540M)

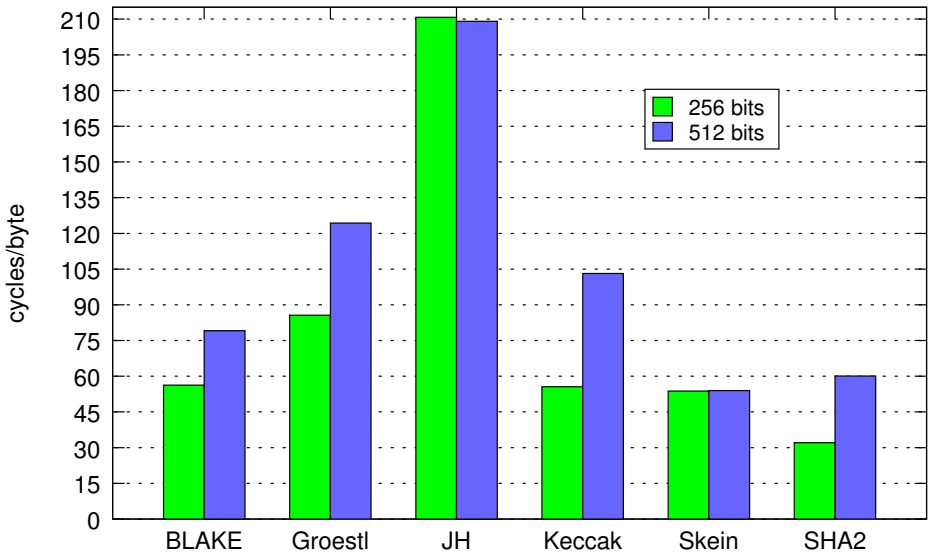

Figure 2: Performance of SHA-3 finalists and SHA2 on Java/i386 (Ubuntu 11.10/amd64, Core i5-2540M)

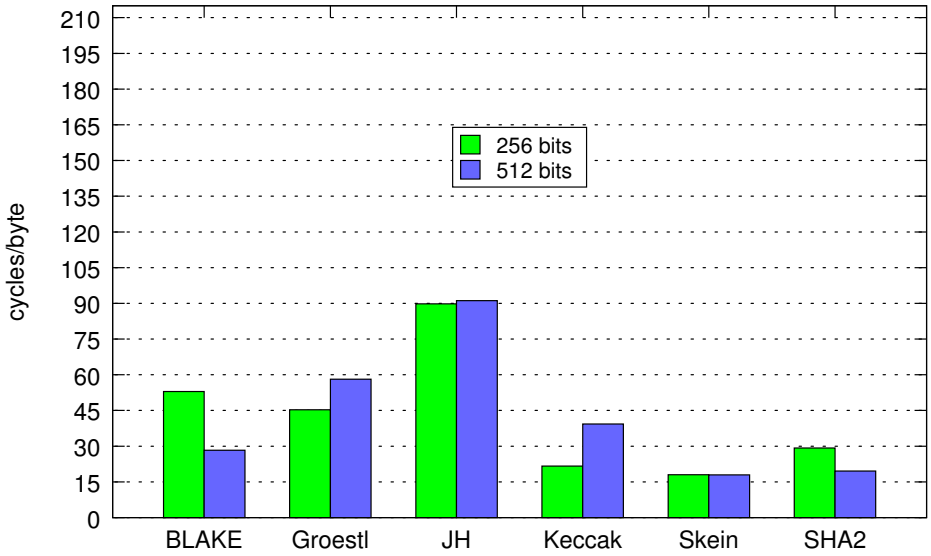

Figure 3: Performance of SHA-3 finalists and SHA2 on Java/amd64 (Windows 7/amd64, Core i5-2540M)

Function	**256 bit output**		**512 bit output**	
	cycles/byte (64-bit JVM)	cycles/byte (32-bit JVM)	cycles/byte (64-bit JVM)	cycles/byte (32-bit JVM)
SHA-2	29.6	32.0	19.0	60.1
BLAKE	48.7	56.3	25.3	79.1
Grøstl	44.4	85.6	57.7	124.3
JH	85.0	210.8	85.0	209.1
Keccak	21.7	55.6	40.0	103.1
Skein	17.2	53.8	17.2	54.0

Table 2: Overview of the Benchmark Results

7 Comparing our Benchmark Results to [11, 12]

This section provides a comparison of the results of this study with the results of Thomas Pornin [11, 12]. Our intention is to show how much leeway Java leaves to the implementer. In order to make the results comparable we benchmarked Pornin's latest implementations in the same manner as we did with our implementations. In most cases we are able to strongly improve Pornin's results.

Table 3 and Figure 4, as well as Table 4 and Figure 5 compare the results of this paper with [11, 12]. We re-evaluated the results of [11] using the latest version of `sphlib` (see [12]), which includes the updated finalist algorithms. The Linux machine, as described in Section 3.3, served as environment for the comparison. To allow a fair comparison, we also point out that version 3.0 of `sphlib` only includes implementations of Grøstl, JH, and Keccak that have been optimized for 64-bit environments. Therefore, the related 32-bit results are not directly comparable either. One can see that our results outperform `sphlib` in almost every case, except for the short variants of BLAKE.

Function	**256 bit output**		**512 bit output**	
	cycles/byte	cycles/byte ([12])	cycles/byte	cycles/byte ([12])
SHA-2	29.6	29.6	19.0	57.7
BLAKE	48.7	**41.9**	**25.3**	55.9
Grøstl	**44.4**	54.9	**57.7**	156.9
JH	**85.0**	85.9	**85.0**	86.3
Keccak	**21.7**	45.0	**40.0**	43.8
Skein	**17.2**	21.7	**17.2**	22.4

Table 3: Performance Comparison of this Paper with [11, 12] (Java/amd64)

Function	**256 bit output**		**512 bit output**	
	cycles/byte	cycles/byte ([12])	cycles/byte	cycles/byte ([12])
SHA-2	32.0	34.1	60.1[1]	146.5[1]
BLAKE	56.3	**53.8**	**79.1**[1]	128.9[1]
Grøstl	**85.6**	107.0[1]	**124.3**	321.7[1]
JH	**210.8**	219.2[1]	**209.1**	220.6[1]
Keccak	**55.5**	120.9[1]	**103.1**	118.5[1]
Skein	**53.8**[1]	54.8[1]	**54.0**[1]	54.6[1]

[1] Optimized for 64-bit environments

Table 4: Performance Comparison of this Paper with [11, 12] (Java/i386)

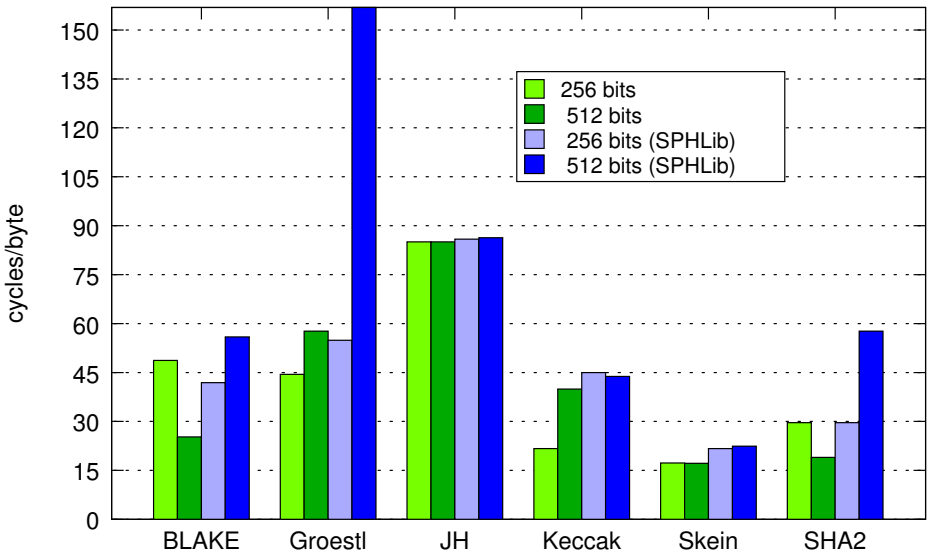

Figure 4: Performance Comparison of this Paper with [11, 12] (Java/amd64)

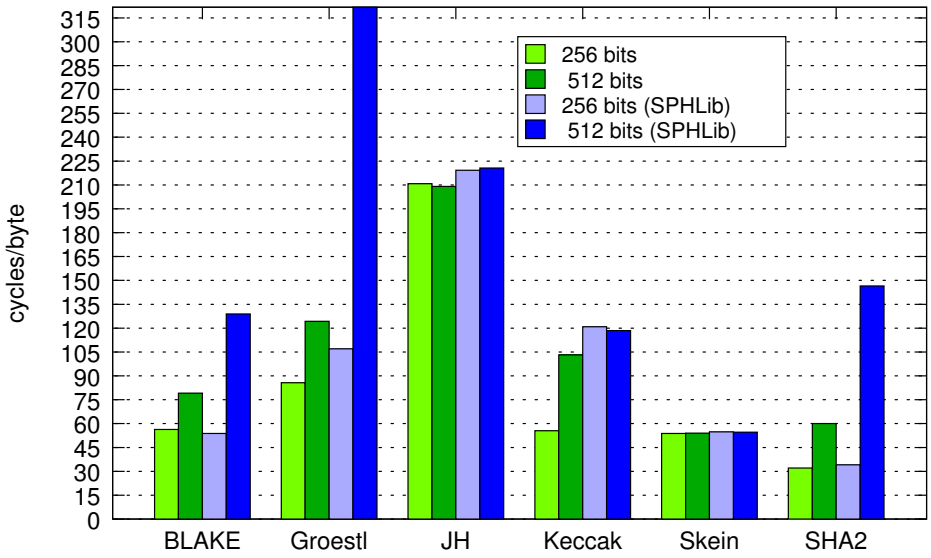

Figure 5: Performance Comparison of this Paper with [11, 12] (Java/i386)

8 Conclusions

In this paper we were dealing with Java code optimization techniques and how they can be applied to get more efficient implementations of the SHA-3 candidate algorithms. Using various combinations of the discussed optimization strategies, we were able to greatly improve the execution speed of almost all SHA-3 candidate algorithms compared to a preceding study [11, 12]. Our results show that two SHA-3 finalists feature exceptional performance in Java, namely Skein and Keccak. On the third and fourth place - depending on the digest length and the word size - come BLAKE and Grøstl. At last, JH lags far behind. The 32-bit implementations are two to three times slower than the corresponding 64-bit implementations and yet far better performing than their 64-bit analogs on 32-bit VMs. Despite extensive optimization efforts, we must conclude that hash implementations in Java are up to 3 times slower than the corresponding native implementations. When we take speed-ups achieved through specialized instruction set extensions, such as SSE, AVX, or AES-NI, into account, Java implementations drop back behind further. This can be overcome by calling native code at the expense of platform independence.

References

[1] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. `http://www.131002.net/blake/`.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The Keccak sponge function family. `http://keccak.noekeon.org/`.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Cryptographic sponge functions. Submission to NIST (Round 3), 2011.

[4] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. `http://www.skein-hash.info/`.

[5] P. Gauravaram, L. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl a SHA-3 candidate. `http://www.groestl.info/`.

[6] C. Hanser. IAIK SHA-3 Provider. `http://jce.iaik.tugraz.at/sic/Products/Core-Crypto-Toolkits`, February 2012.

[7] IAIK. The SHA-3 Zoo. `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`.

[8] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974.

[9] T. Mueller. Skein 512-512 (Java). `http://www.h2database.com/skein/`.

[10] National Institute of Standards and Technology. `http://www.nist.gov/index.html`.

[11] T. Pornin. Comparative performance review of the sha-3 second-round candidates, June 2010.

[12] T. Pornin. sphlib 3.0, July 2011. http://www.saphir2.com/sphlib/.

[13] H. Wu. Hash function JH. http://www3.ntu.edu.sg/home/wuhj/research/jh/index.html.