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

Thomas Pornin, <thomas.pornin@cryptolog.com>

June 21, 2010

Abstract

We reimplemented all round-2 SHA-3 candidates, in both C and Java. Implementations were made from scratch, not reusing the code provided with the NIST submission packages. The C code aims at portability, not using the platform specific features beyond what is available in portable C. Similar optimization techniques and efforts were applied to each function, with an emphasis on embedded low-power platforms; the resulting performance differences should then be considered as intrinsic to the functions themselves. We benchmarked our code on a variety of platforms; we present here our measures, and what conclusions can be inferred on what makes a hash function fast.[1]

1 Introduction

For the SHA-3 competition[1], NIST defined a "Reference Platform and Compiler", namely a PC with an "Intel Core 2 Duo processor", clocked at 2.4 GHz, running the Windows operating system and using Microsoft Visual Studio 2005 Professional Edition as compiler. The NIST notice introducing the competition states that the selected SHA-3 is expected to achieve its security goals with "significantly improved efficiency" over SHA-2; it also states that "the public" is encouraged to performs efficiency analysis on "other platforms".

In this paper we consider a specific class of "efficiency" on some software platforms, namely processing bandwidth (expressed in megabytes of hashed data per second or clock cycles per processed input byte) over an appropriately long message: costs inherent to hash function initialization and finalization are ignored. In practice, this means input messages are longer than ten kilobytes or so. This is the most often quoted performance figure for any function which processes streamed data, hash functions being a prime example.

Quite predicibly, most submitters concentrated on performance on PC systems, for a variety of reasons:

- The NIST had plainly announced that they would run tests *and an efficiency analysis* on their "reference platform".
- Cryptographers develop and optimize for what they can test on, and desktop systems are, by far, the easiest to use, and cheapest, development platform nowadays.
- Modern x86-compatible processors tend to have some advanced features (SSE2 unit with 128-bit registers, hard-coded AES implementation for the most recent versions) which appear to have a high potential for cryptographic applications.
- The PC is the *visible* computer. Few people are fully aware that most computing is now done on *hidden*, embedded systems, which have percolated by dozens within the most mundane-looking appliances. Embedded systems have a huge industrial significance which is widely underestimated.

[1]This work was partially supported by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014.

Therefore, we feared that high performance on the reference platform could cloud away implementation issues on other systems, resulting in a seemingly fast but actually suboptimal function selected as SHA-3. In particular, many embedded systems lack the advanced features found on modern desktop systems. As we will see later on, small systems also imply small L1 instruction cache memory, which has drastic consequences on performance. We also feared that some candidates could be unfairly dismissed as slow, despite their cryptographic merits, if their designers could not come up with decently optimized implementations: optimization is a difficult craft and a distinct skill set from what good cryptographers master.

We thus wanted to have a fair comparison between hash functions, on a variety of software platform types, with an emphasis on the non-PC-with-x86-assembly architectures. In order to smooth out discrepancies between implementer skills, we decided that all functions ought to be implemented with similar efforts and techniques by the *same* developer. Hopefully, resulting speed differences should then be representative of the intrinsic characteristics of the hash functions themselves.

This approach complements the eBASH[2] measures: our implementations focus on more portable code (to try to abstract away special architecture characteristics such as SIMD instructions), we benchmarked them on embedded systems which are not covered by eBASH (mostly due to automation issues; it is hard to run automatic benchmarks on a pocket calculator), and we added the "unique developer" rule.

Our work is the continuation of the development of `sphlib`[3], an open-source library of hash function implementations, in both C and Java. `sphlib` was written by the same developer along the same lines, and included many pre-SHA-3 hash functions, including MD5, SHA-1, the SHA-2 family and Whirlpool. We thus added implementations of the fourteen SHA-3 "round 2" candidates. `sphlib-2.1` is used here, and is available for download on the following Web site:

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

In the following sections, we describe our target architectures and development rules. We also define what we measure and how we measure it. We then give our measures, both numerically and as graphical charts. Finally, we describe what these numbers tell us, with regards to what impacts hash function performance.

2 Target Architectures

2.1 Large Systems

What we call here a *large system* is about any visible computer. This covers personal computers and workstations, but also big servers, laptops, netbooks and even smartphones. For our purposes, a large system is any software platform with a 32-bit or more CPU, with a superscalar architecture (the hardware can execute several instructions simultaneously) and a large level-1 cache for instructions (32 kB or more).

According to our tests, level-1 cache size turns out to be the most important factor for performance, while the actual processor model and instruction set impacts only moderately the relative performance of the functions (see sections 4.1 and 4.4). This is why big servers and smartphones end up together in this "large system" category.

As representative of this category, we used the three following systems:

- A PC with an Intel Core 2 Quad Q6600 processor, clocked at 2.4 GHz ("Kentsfield" core). This is similar to the NIST reference platform[2]. RAM size and operating system are not relevant for our tests (everything important happens in level-1 cache anyway). The system runs in 64-bit mode (known as "x86-64", "AMD64", "x64", "Intel 64", "EM64T" and a few other names). The C compiler is GCC[4], version 4.4.3.

- The very same PC, but this time used in 32-bit mode. Registers have size 32 bits instead of 64, and there are fewer of them. The C compiler is still GCC-4.4.3.

[2] Actually, the NIST reference platform is described a bit vaguely, since "Core 2 Duo" covers a wide range of processors from Intel, with distinct timing characteristics.

- An older PowerPC 750 (aka "G3"), clocked at 300 MHz. The processor runs in big-endian mode. The C compiler is GCC 4.1.3.

Those three platforms shall be representative of large systems because our test code is a portable C implementation, which does not use the specific features of the Intel architecture (MMX, SSE2...). Thus, they should behave similarly to other, large architectures such as the XScale and Cortex ARM-compatible processors which are often found in smartphones.

2.2 Embedded Systems

We consider here systems with a "small" 32-bit processor. The CPU still offers 32-bit registers, but with a reduced instruction set and no superscalar ability. The level-1 cache is much smaller, typically 8 to 16 kB. The clock rate is also much lower, counted in hundreds or even dozens of megahertz, not gigahertz. Of particular interest to us are:

- WiFi access points and routers. These are small, cheap appliances which must nevertheless route high-bandwidth data all day long. Routers are ideally located to run VPN software.
- Portable media players (MP3, small videos...). Cryptography, in particular signatures, is used to enforce digital rights. Such systems must be small and light, which implies drastic constraints on available CPU power (the battery is small, so the CPU is underclocked).
- Mobile phones (not the bigger so-called smartphones, which are "large systems" as far as hash function performance is concerned). Most can be used as media players, and there is a big market for data and applications running on those platforms. There again, DRM requires efficient cryptography. Also, mobile phones are inherently networked, and this implies cryptography as well (e.g. SSL connections).
- Payment terminals. These are the portable terminals which handle card-based transactions in many countries. Internally, those systems are similar to glorified smartcards, with tamper-resistant CPU and RAM. The physical and electromagnetic armor prevents these systems from using the big desktop and server processors, notably because of cooling issues. Processors are clocked at no more than 30 MHz or so. Yet those systems are in front line in valuable security systems and must rely on robust cryptography. Hardware Security Modules (HSM) are similar in that respect: despite their high price, their tamper-resistance implies drastically low CPU performance, with simple cores.

In order to represent this category, we used the two following architectures:

- A Linksys WRT54GS router, refurbished with the OpenWrt operating system[5] (a Linux clone for embedded systems). The core CPU is a Broadcom BCM3302, a MIPS-compatible 32-bit processor clocked at 200 MHz. The level-1 cache size is 8 kB for code, and another 8 kB for instructions. The C compiler is GCC, version 4.2.4.
- A Hewlett-Packard HP-50g scientific calculator. This system uses an ARM920T core (ARMv4 architecture), clocked at 12 MHz, but which can be programmatically boosted to 75 MHz (which we did for our tests). This system can be programmed in C (with a much reduced, non-standard C library), using HPGCC[6], a derivative from GCC 4.1.1.

ARM and MIPS architectures together cover the quasi-totality of 32-bit embedded systems nowadays. Our two systems should then be representative of this category of platforms.

2.3 Virtual Machines

A newer trend in computing is the use of virtual machines. Instead of compiling code for an existing, physical family of processors, a virtual CPU is defined, with its own instruction set (usually called *bytecode*). The compiled program is executed through a *Virtual Machine*. The VM interprets the bytecode, and uses JIT

("Just In Time") compilation techniques to dynamically convert the bytecode of the most used code parts into native instructions for the host CPU.

Virtual Machines are not a new concept, but widespread industrial application really began with the Java programming language, in the mid-90's. VM have a number of worthwhile characteristics:

- The virtual CPU has the same characteristics "everywhere", such as the size of integer types; representation details such as endianness are abstracted away. This provides a great deal of portability.
- Since there is a single bytecode format, software distribution is greatly eased.
- Bytecode allows a priori verification of the code with regards to data types. The VM can enforce boundary checks upon array accesses. Program safety is widely enhanced, to the point that potentially hostile code can be run in a sandbox, which supports the Applet model.
- The VM model (theoretically) allows for some performance improvements: the JIT compiler can optimize code for the specific CPU on which it happens to run, possibly using dynamically collected profiling information (if code is first interpreted, the JIT compiler running only for the most used parts), and the type safety allows for the more efficient types of garbage collection algorithms. In practice, these improvements compensate some of the overhead implied by the interpreter and JIT compilation process.

VM are extremely popular, and VM-based languages are said to be more used in servers than classical compiled languages. VM provide an API to interact with "native code" (e.g. compiled C code), but native code nullifies the advantages of VM which we described above. It thus makes sense to study the performance of hash functions when implemented in bytecode.

We use the following two architectures:

- Our PC (Intel Core 2 Q6600, 2.4 GHz, 64-bit mode), running the Java VM edited by Sun Microsystems (now Oracle), version 1.6.0_20[7].
- The same PC, in 32-bit mode, with the 32-bit version of the Java VM from Sun.

The Java VM is the most widely used VM. Its more recent competitor (the .NET VM from Microsoft, coupled with the C# programming language) should provide similar runtime characteristics. As will be explained in section 4, the VM and JIT compiler have their own specific overhead, which impacts hash function performance in various ways, which depend on the hash function itself.

2.4 Compilation Options

All the C compilers for our test architectures are variants of GCC. Optimization in GCC is triggered with the `-On` command line flags, where `n` is a digit from `1` to `9`. Theoretically, a higher `n` implies better performance (and longer compilation time), but in practice things are not that clear. Code optimization relies on a number of heuristics, which are not necessarily valid for all codes, and especially not for hash function implementations which tend to rely on heavy loop unrolling and a plethora of local variables.

In practice, the `-O1` compilation flag often offers better performance than `-O2`, sometimes dramatically so (e.g. +50% speed with SHA-1). Thus, the default compilation flags for `sphlib` are set, in the build script, to "`-O1 -fomit-frame-pointer`". Nevertheless, for some hash functions, `-O2` does a better job than `-O1`, so we also compiled all functions with "`-O2 -fomit-frame-pointer`" and we kept the best of the two measured performances. This was done individually for each function and each hash output length.

Another tunable option of `sphlib` is the selection of the "small footprint" implementations. These are activated by defining the `SPH_SMALL_FOOTPRINT` macro. Small footprint implementations are optimized to fit within the 8 kB of level-1 cache of our MIPS-based test architecture; they offer much better performance on that system, but worse performance on large systems. There again, for each platform, each optimization level, each function and each hash output length, we tried the "normal" and the "small footprint" implementations, and kept the best measure among the two.

An additional parameter is found on the ARM platform: the ARMv4 architecture supports *two* instruction sets, the "normal" set (inherited from older ARM versions) and the "thumb" mode. The thumb mode features simpler instructions, and some of the registers are not available through thumb mode; but instructions are also twice smaller, which makes the code more compact and helps it fit within the level-1 cache. Normal and thumb code can be freely mixed within a given application (the CPU can switch conventions in a single clock cycle, when calling a function). Hence, we doubled our tests, once in normal mode and once in thumb mode.

For the Java architectures, the VM does not offer many tunable options. We ran our tests with the `-server` command-line option, which should encourage the JIT compiler to think harder. Our test procedure includes a bit of "warm-up" so that measures are taken *after* the JIT compiler has acted.

2.5 Out of Scope Architectures

Our work does not cover the following kinds of architectures, which are nonetheless important with regards to hash function performance:

- 8-bit and 16-bit architectures. These are found mostly in smartcards. They are usually programmed in assembly, not C, and their characteristics vary quite widely. A comparison effort such as the one we describe here would require the use of many test platforms, each with its own development and optimization effort.
- FPGA and ASIC. Hardware follows its own rules, in particular parallelism, and circuit size is at least as important as resulting bandwidth. `sphlib` is a *software* library.
- GPU. The most recent graphics accelerators, normally used for rendering 3D images and animations, actually include arrays of processors in SIMD mode (single instruction, multiple data), which can be programmed "generically". The market for GPU is still too new to define "representative" systems. Also, GPU programming has some specificities, with languages which are similar, but not identical to C. We did not investigate GPU implementation of hash functions (yet).

Also, we have not devoted any effort to the production of compact implementations. We always targetted the size of the level-1 cache for instructions. In practical situations (as opposed to ideal benchmark conditions), the hash function code must share the caches with the other elements in the applicative data processing path. Very small implementations (e.g. less than 1 kB of compiled code) are also very important in some situations where overall code size is constrained (for instance, in a boot ROM). `sphlib` implementations do not cover that kind of optimization, although the cache size issues illustrate how hash function performance is impacted by constraints on code size.

3 Measures

For each function, the hash function performance was measured on a *long message*. Basically, this means that we hash a single message of length bn bytes, where b is the *block size* and n is the number of blocks. The function implementation is given one block of size b per call. b is equal to 8192 (this is the "traditional" buffer size for I/O and it fits within the level-1 cache for data on all our test architectures). n is dynamically adjusted so that the processing time exceeds two seconds. Elapsed time is measured with the `clock()` function on an otherwise idle machine (`System.currentTimeMillis()` in Java). Achieved precision is about 2% (in the tables below we give more digits, but only the first two or three are really significant).

Before that measure, the function is "warmed-up" by computing many digests on messages of various lengths. The point is to allow every part of the hardware and software to reach its top speed (cache memory, jump prediction tables in the processor, JIT compilation).

This measure does not take into account the initialization and finalization times. Although `sphlib` code tries not to do anything stupid, initialization and finalization procedures have not been optimized with as much care as the main loop. Thus, performance of `sphlib` might not be really representative of hash function

performance on that matter. Not all functions are equivalent in that respect; but `sphlib` is not (yet) the right tool to evaluate hash function performance on very short messages.

The tables below (tables 1 to 7) contain our measures, expressed in megabytes per second (one megabyte is 10^6 bytes), cycles per byte, and speed relative to the average SHA-2/3 speed. That average speed is defined to be the average speed of the fifteen SHA-2 and SHA-3 candidate functions with the same output size, on the same machine. The charts (figures 1 to 7) list the same functions, sorted by bandwidth (in megabytes per second).

We give figures for 256-bit and 512-bit outputs. For all tested functions except Keccak, the function with a 224-bit output offers the same performance than the 256-bit variant. For all tested functions except Fugue, Keccak and Luffa, the 384-bit function yields the same bandwidth than the 512-bit function.

It shall be noted that whenever applicable, `sphlib` performance was compared with the submitted optimized code and the performance figures reported in the submission packages; `sphlib` was always on par with those figures. More precisely, if the submitted code kept to "portable C" (no MMX/SSE2), then we compiled and benchmarked it on our x86 PC, in both 32-bit and 64-bit mode, using the same compiler and optimization flags than for `sphlib`. The `sphlib` code always offered comparable performance, or better performance in a few cases (e.g. with SHAvite-3 and a 512-bit output, `sphlib` is twice faster, because the submitted optimized code has an unrolled loop which exceeds the 32 kB of level-1 instruction cache of our Intel Q6600 – the SHAvite-3 submitters optimized for an AMD CPU which has a 64 kB level-1 cache). This validates the effectiveness of the optimization strategy of `sphlib`.

Of course, for the "large" architectures (mostly x86) and most of the SHA-3 round 2 candidates, there exist published, faster implementations which use the special features of the platform[2]. The figures below should not be taken as a claim for the fastest achievable performance on the specific platforms on which we ran our tests. Rather, the reported measures:

- are a token of the quality of `sphlib` code, within its set constraints of portability;
- are meant to allow prediction of hash function performance on other comparable architectures;
- can be used to analyze some effects such as the importance of 64-bit native types for a given function (by comparing the x86 platform in 32-bit and 64-bit modes).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	144.71	16.58	1.093	187.72	12.78	1.306
BLAKE	230.42	10.42	1.740	341.96	7.02	2.379
BMW	238.61	10.06	1.802	462.82	5.19	3.219
CubeHash	60.19	39.87	0.455	60.19	39.87	0.419
ECHO	62.14	38.62	0.469	33.55	71.54	0.233
Fugue	79.18	30.31	0.598	41.30	58.11	0.287
Grøstl	94.85	25.30	0.716	48.11	49.89	0.335
Hamsi	41.55	57.76	0.314	14.65	163.82	0.102
JH	60.19	39.87	0.455	59.65	40.23	0.415
Keccak	120.37	19.94	0.909	64.22	37.37	0.447
Luffa	96.56	24.86	0.729	55.01	43.63	0.383
SHAvite-3	105.27	22.80	0.795	61.57	38.98	0.428
Shabal	318.62	7.53	2.407	319.57	7.51	2.223
SIMD	46.93	51.14	0.354	40.92	58.65	0.285
Skein	286.33	8.38	2.163	365.22	6.57	2.540

Table 1: Performance of `sphlib` on x86-64 (Intel Q6600, 64-bit, 2.4 GHz).

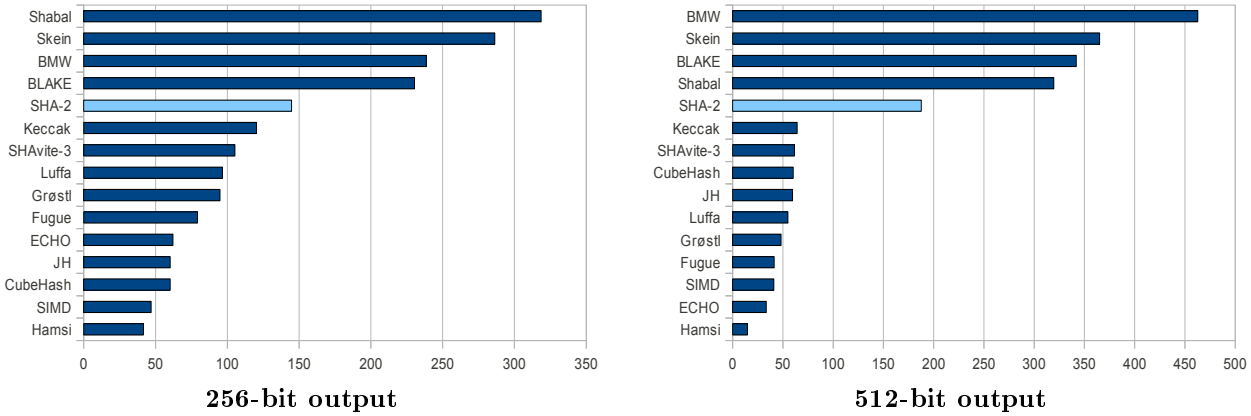

Figure 1: Bandwidth of `sphlib` on x86-64 (Intel Q6600, 64-bit, 2.4 GHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	133.55	17.97	1.521	47.43	50.60	0.821
BLAKE	199.58	12.03	2.273	44.30	54.18	0.767
BMW	208.09	11.53	2.370	154.27	15.56	2.672
CubeHash	37.49	64.02	0.427	37.39	64.19	0.648
ECHO	50.27	47.74	0.572	27.06	88.69	0.469
Fugue	55.69	43.10	0.634	36.97	64.92	0.640
Grøstl	29.31	81.88	0.334	21.17	113.37	0.367
Hamsi	33.22	72.25	0.378	12.86	186.63	0.223
JH	19.85	120.91	0.226	19.97	120.18	0.346
Keccak	41.05	58.47	0.467	22.60	106.19	0.391
Luffa	84.41	28.43	0.961	47.59	50.43	0.824
SHAvite-3	71.97	33.35	0.820	48.28	49.71	0.836
Shabal	258.11	9.30	2.939	258.11	9.30	4.470
SIMD	30.64	78.33	0.349	26.21	91.57	0.454
Skein	63.91	37.55	0.728	61.85	38.80	1.071

Table 2: Performance of `sphlib` on i386 (Intel Q6600, 32-bit, 2.4 GHz).

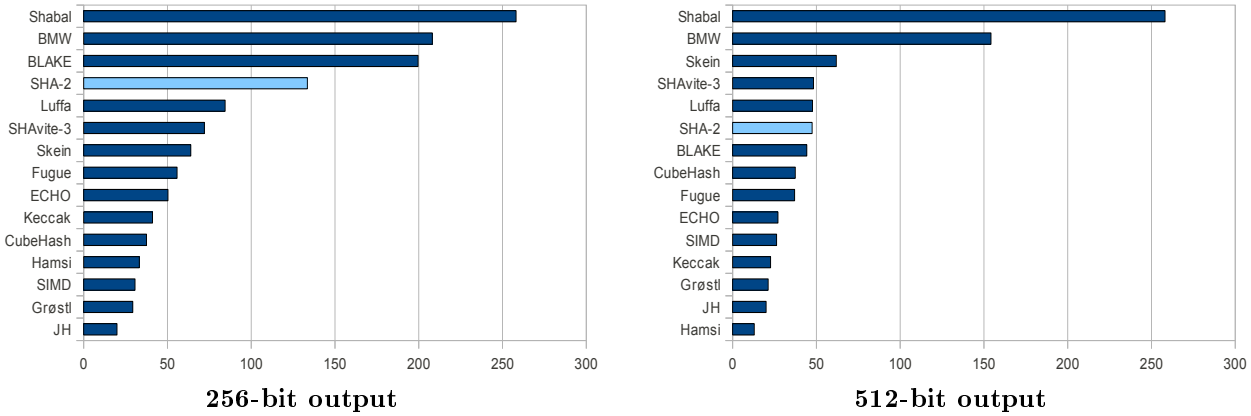

Figure 2: Bandwidth of `sphlib` on i386 (Intel Q6600, 32-bit, 2.4 GHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	14.04	21.37	1.619	4.58	65.50	0.880
BLAKE	21.44	13.99	2.472	4.95	60.61	0.951
BMW	17.03	17.62	1.963	8.07	37.17	1.550
CubeHash	5.81	51.64	0.670	5.81	51.64	1.116
ECHO	3.92	76.53	0.452	2.14	140.19	0.411
Fugue	6.74	44.51	0.777	3.94	76.14	0.757
Grøstl	2.10	142.86	0.242	1.45	206.90	0.279
Hamsi	3.85	77.92	0.444	1.17	256.41	0.225
JH	1.98	151.52	0.228	1.97	152.28	0.378
Keccak	2.34	128.21	0.270	1.27	236.22	0.244
Luffa	8.47	35.42	0.977	4.15	72.29	0.797
SHAvite-3	8.18	36.67	0.943	5.02	59.76	0.964
Shabal	25.42	11.80	2.931	25.42	11.80	4.883
SIMD	3.19	94.04	0.368	2.46	121.95	0.473
Skein	5.59	53.67	0.645	5.69	52.72	1.093

Table 3: Performance of `sphlib` on G3 (PowerPC 750, 32-bit, 300 MHz).

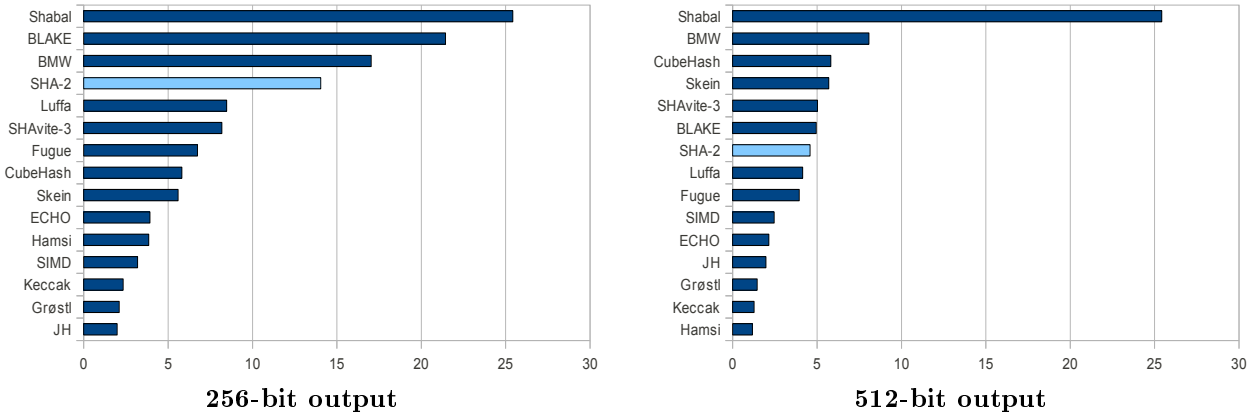

Figure 3: Bandwidth of `sphlib` on G3 (PowerPC 750, 32-bit, 300 MHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	2.82	70.92	1.372	1.47	136.05	1.064
BLAKE	3.04	65.79	1.480	1.61	124.22	1.165
BMW	5.31	37.66	2.584	2.60	76.92	1.881
CubeHash	1.11	180.18	0.540	1.11	180.18	0.803
ECHO	1.09	183.49	0.530	0.59	338.98	0.427
Fugue	1.83	109.29	0.891	0.95	210.53	0.687
Grøstl	0.50	400.00	0.243	0.33	606.06	0.239
Hamsi	0.87	229.89	0.423	0.25	800.00	0.181
JH	0.64	312.50	0.311	0.64	312.50	0.463
Keccak	0.79	253.16	0.384	0.44	454.55	0.318
Luffa	1.78	112.36	0.866	1.03	194.17	0.745
SHAvite-3	1.58	126.58	0.769	0.93	215.05	0.673
Shabal	6.74	29.67	3.280	6.74	29.67	4.877
SIMD	0.79	253.16	0.384	0.47	425.53	0.340
Skein	1.93	103.63	0.939	1.57	127.39	1.136

Table 4: Performance of `sphlib` on Broadcom BCM3302 (MIPS, 32-bit, 200 MHz).

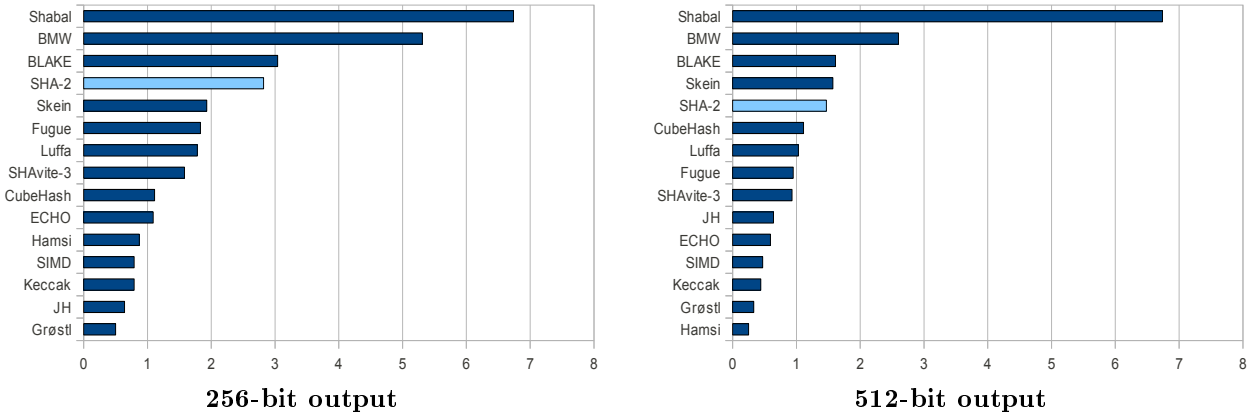

Figure 4: Bandwidth of `sphlib` on Broadcom BCM3302 (MIPS, 32-bit, 200 MHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	1.82	41.2	1.814	0.55	136.4	0.887
BLAKE	1.76	42.6	1.754	0.76	98.7	1.226
BMW	2.46	30.5	2.452	1.08	69.4	1.742
CubeHash	0.63	119.0	0.628	0.63	119.0	1.016
ECHO	0.45	166.7	0.449	0.24	312.5	0.387
Fugue	0.73	102.7	0.728	0.39	192.3	0.629
Grøstl	0.26	288.5	0.259	0.11	681.8	0.177
Hamsi	0.27	277.8	0.269	0.10	750.0	0.161
JH	0.23	326.1	0.229	0.23	326.1	0.371
Keccak	0.25	300.0	0.249	0.13	576.9	0.210
Luffa	1.02	73.5	1.017	0.56	133.9	0.903
Shabal	3.22	23.3	3.209	3.22	23.3	5.194
SHAvite-3	0.71	105.6	0.708	0.45	166.7	0.726
SIMD	0.34	220.6	0.339	0.27	277.8	0.435
Skein	0.90	83.3	0.897	0.58	129.3	0.935

Table 5: Performance of `sphlib` on ARM920T (ARMv4, 32-bit, 75 MHz).

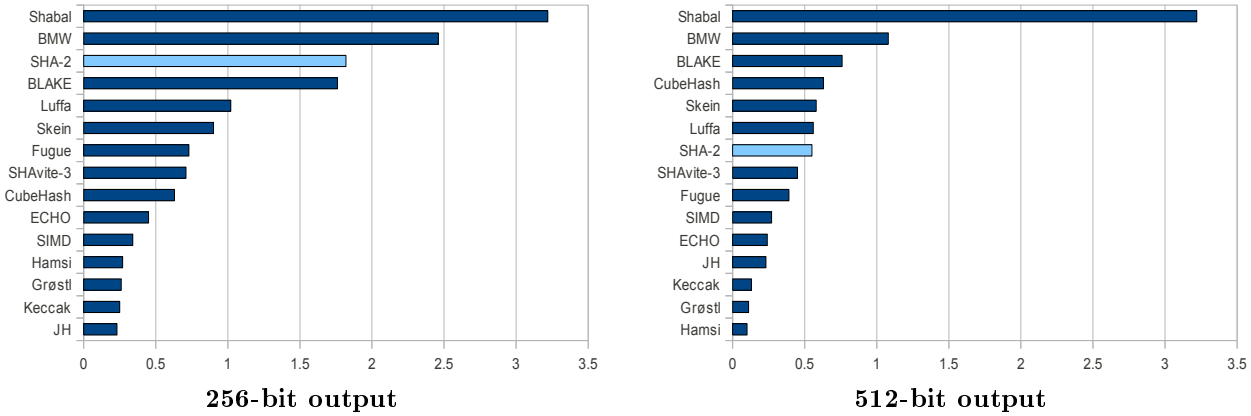

Figure 5: Bandwidth of `sphlib` on ARM920T (ARMv4, 32-bit, 75 MHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	83.26	28.83	1.436	77.20	31.09	1.369
BLAKE	69.42	34.57	1.197	91.00	26.37	1.613
BMW	79.09	30.35	1.364	186.87	12.84	3.313
CubeHash	45.24	53.05	0.780	45.24	53.05	0.802
ECHO	22.35	107.38	0.385	11.89	201.85	0.211
Fugue	41.96	57.20	0.724	21.39	112.20	0.379
Grøstl	36.83	65.16	0.635	24.44	98.20	0.433
Hamsi	24.75	96.97	0.427	8.73	274.91	0.155
JH	30.69	78.20	0.529	30.69	78.20	0.544
Keccak	52.04	46.12	0.897	28.27	84.90	0.501
Luffa	59.34	40.44	1.023	35.00	68.57	0.620
SHAvite-3	46.85	51.23	0.808	28.16	85.23	0.499
Shabal	153.35	15.65	2.645	153.35	15.65	2.718
SIMD	18.49	129.80	0.319	1.69	1420.12	0.030
Skein	106.10	22.62	1.830	102.26	23.47	1.813

Table 6: Performance of `sphlib` with Java (Intel x86 Q6600, 64-bit, 2.4 GHz).

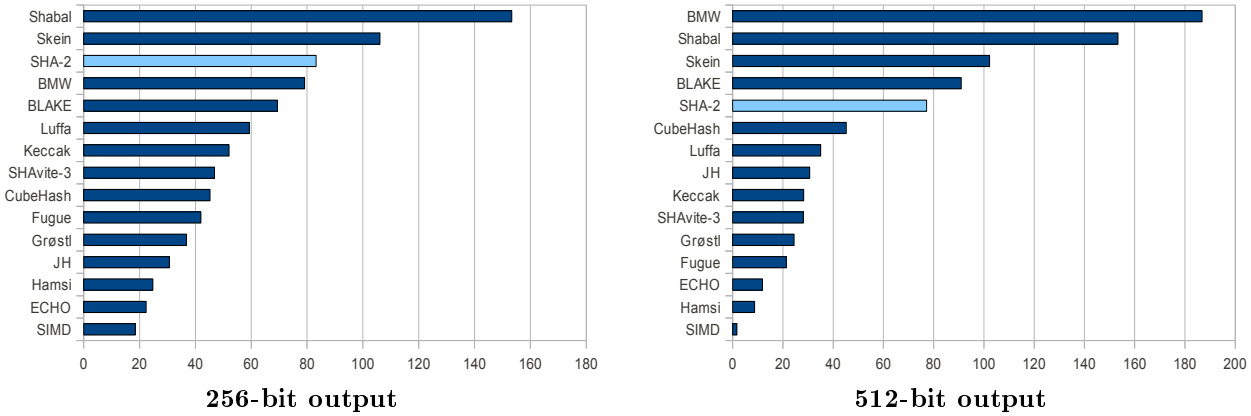

Figure 6: Bandwidth of `sphlib` with Java (Intel x86 Q6600, 64-bit, 2.4 GHz) (megabytes per second).

Function	256-bit output			512-bit output		
	Mbytes/s	cycles/byte	normalized	Mbytes/s	cycles/byte	normalized
SHA-2	68.34	35.12	1.590	26.63	90.12	0.845
BLAKE	62.89	38.16	1.463	44.06	54.47	1.398
BMW	74.67	32.14	1.737	66.41	36.14	2.108
CubeHash	35.33	67.93	0.822	35.33	67.93	1.121
ECHO	18.18	132.01	0.423	9.91	242.18	0.315
Fugue	35.75	67.13	0.832	18.25	131.51	0.579
Grøstl	18.89	127.05	0.439	12.41	193.39	0.394
Hamsi	20.08	119.52	0.467	7.88	304.57	0.250
JH	10.65	225.35	0.248	10.65	225.35	0.338
Keccak	15.64	153.45	0.364	8.30	289.16	0.263
Luffa	42.77	56.11	0.995	24.89	96.42	0.790
SHAvite-3	38.67	62.06	0.900	24.50	97.96	0.778
Shabal	142.75	16.81	3.321	142.75	16.81	4.531
SIMD	15.85	151.42	0.369	1.57	1528.66	0.050
Skein	44.30	54.18	1.031	39.08	61.41	1.240

Table 7: Performance of `sphlib` with Java (Intel x86 Q6600, 32-bit, 2.4 GHz).

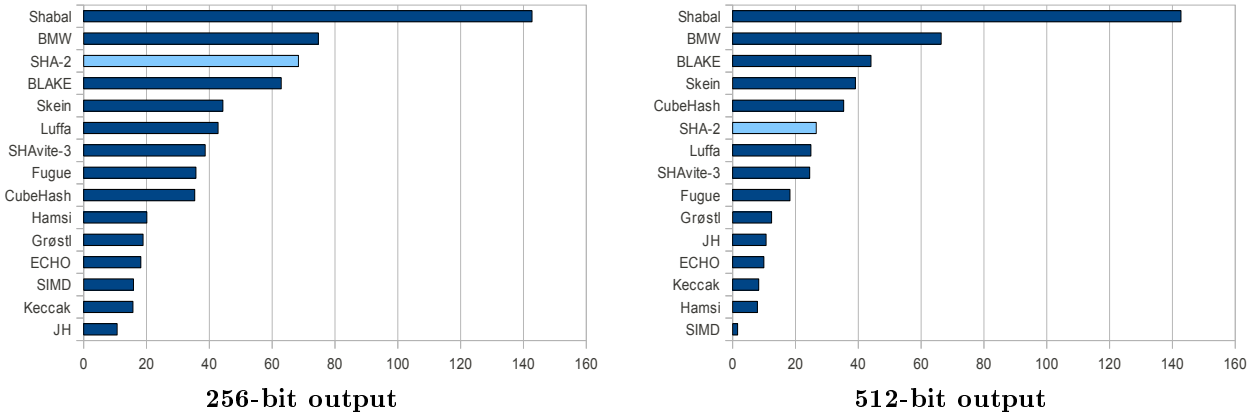

Figure 7: Bandwidth of `sphlib` with Java (Intel x86 Q6600, 32-bit, 2.4 GHz) (megabytes per second).

4 What Makes a Hash Function Fast

4.1 Cache Size

The most important parameter for hash function performance is cache size, precisely the size of the level-1 cache for instructions. The level-1 cache for data is much less important, except for the functions which use tables in their implementation (e.g. ECHO).

Failure to fit the core algorithm loop within the level-1 cache implies severe performance degradation, usually by a factor of 2 or 3, possibly much more (we witnessed a degradation of a factor of 22 on a Skein implementation !). This impacts loop unrolling.

Loop unrolling is a technique which consists in duplicating the code for one function "round"; several successive rounds are thus converted to executable opcodes, presented successively in RAM. The main point of loop unrolling is that it nullifies data routing cost. For instance, consider SHA-256: at each round, the eight state words are "rotated". If you unroll eight successive rounds, then each round may use the proper variables, and the rotation becomes virtual (i.e. free: no more data copying between variables). Moreover, if you unroll 64 successive rounds of SHA-256, then round constants can be hardcoded instead of being obtained from a table, which saves one memory indirection per round.

Thus, unrolling can be viewed as a generic tool for suppressing data routing cost (at runtime), at the expense of a larger code footprint[3]. We say that the implementation is *fully unrolled* when there is nothing more to gain, with regards to data routing, by unrolling more. For instance, a fully unrolled SHA-256 has 64 copies of the round code.

If the fully unrolled function fits in level-1 cache, then everything is fine. Otherwise, the function must be partially re-rolled; thus, some routing operations again imply some runtime cost, through data copying or indirections. In the implemented functions, only CubeHash, Fugue, Luffa and Shabal fit in the 8 kB level-1 cache of our MIPS-based test architecture, when fully unrolled. All other functions must be only partially unrolled, which explains their poorer performance on that platform (with regards to what they can do on large systems).

4.2 64-bit Types

64-bit integers are efficient on systems which have native 64-bit registers. On 32-bit systems without such registers, 64-bit integer types are very inefficient. Any 64-bit value, on a 32-bit system, requires two registers; most operations require two or more opcodes; additions need manual carry propagation between the lower and upper words. Rotations are especially expensive, since they cannot use the rotation opcodes provided by the CPU for 32-bit registers (at least on the architectures which have such opcodes).

This is easily seen on the charts about the functions with 512-bit output. On the 64-bit architectures (either in C or Java), BMW-512 is the fastest hash function; Skein, BLAKE-512 and SHA-512 are also quite fast. On 32-bit systems, these functions lose much ground. Comparatively, Shabal efficiency remains high on 32-bit systems, because that function uses only operations on 32-bit values.

On the x86 architecture, the lack of a 64-bit integer type is often compensated, in hash function implementations, with the use of special units which offer 64-bit registers (MMX, SSE...) but this requires inline assembly or non-portable compiler intrinsics; this does not work on other architectures, in particular the Java VM.

We also note that using 64-bit types makes porting to some architectures quite difficult. The "`long long`" integer type has been added to C in the 1999 standard, but in the previous standard (C89, aka "ANSI C"), the biggest available type is not guaranteed to exceed 32 bits. Fortunately, most embedded architectures nowadays use development kits derived from some version of GCC, which has offered the "`long long`" type for more than 15 years.

[3] Compilation time is also greatly enlarged.

4.3 Endianness

Endianness appears to be a non-issue. Sticking to the native endianness of the machine can yield an improvement, but only a very slight one. For MD4 (which is extremely fast, more than twice faster than SHA-1), on big-endian Sparc v9 systems, using the special assembly opcodes for little-endian access may speed up the function by about 30%, with regards to the generic code (which must byte-swap the input words). The relative speedup decreases correspondingly with the function performance.

In some corner cases, e.g. when implementing under strict code and RAM size constraints, using the native endianness can help a bit, mostly because it avoids having to store decoded words in local variables; hence it makes sense to define I/O with the little-endian convention, which is now prevalent[4]. Most of the SHA-3 round 2 candidates use little-endian. Nevertheless, at least BLAKE uses big-endian and still offers very decent performance. Generally speaking, endianness is *not* an important factor for hash function performance.

4.4 Instruction Set

For most functions, the actual instruction set is not important. In large systems, assembly opcodes are dynamically translated to internal elementary instructions, on which the CPU applies its optimizations (parallel execution, reordering, speculative execution...). The dialect, as seen by the assembly programmer and the C compiler, impacts performance only insofar as it changes the code size, hence the ease with which the code fits in level-1 cache.

However, we can see, in CubeHash, a case of register starvation. On our x86 CPU, CubeHash runs at about 60 MB/s in 64-bit mode, but only 37 MB/s in 32-bit mode. This is surprising, since CubeHash uses only 32-bit operations. To understand what happens here, one can use the following view:

- The hash function state consists in a number of values; for CubeHash, these are 32 32-bit words. Since the processor has less registers than that, the compiler will have to emit extra opcodes to swap values between registers and local variables in RAM.
- On a superscalar processor, several instructions can be executed concurrently. But parallelism is constrained by dependencies between successive operations: some operations take as operand the result of previous operations, and thus must occur *after* those previous operations. This results in a number of "free slots", in which some parts of the processors are idle for one clock cycle.

The "free slots" are used by the compiler to perform the data swap operations between the registers and the RAM. If there are not enough free slots with regards to the needed swap operations, performance is degraded. This effect is most visible on x86 CPU in 32-bit mode, which has only seven available general purpose registers. It is especially severe with CubeHash, which is amenable to local parallelism, i.e. has very few dependencies between successive operations, thus reducing the number of free slots. This explains the performance drop on 32-bit x86. As a side-note, the local parallelism is also what makes CubeHash fast when using SSE2 opcodes.

Another instruction set effect which we observed is on the MIPS architecture, which does not have a 32-bit rotation opcode. Rotations must therefore be emulated with two shifts and a boolean combination, i.e. three opcodes. This impacts both execution speed and code size. An affected function is Keccak: the Keccak specification describes an optimization method (called "interleaving") which can be applied to 32-bit architectures, and allows some of the rotations (expressed over 64-bit words) to be translated into 32-bit rotations. Interleaving is a net gain on 32-bit systems which have a 32-bit rotation opcode; but on a MIPS, the gain is very slight.

[4] The little-endian convention makes it easier to port non-endian-neutral code from the PC/Windows world, where such code is common. Also, the little-endian convention makes it simpler to *simulate* the target system on a PC.

4.5 Java Specific Issues

It appears that the JIT compiler produces very fat code. This is due to the constraints under which the JIT compiler operates: it must work fast, and produce code which the garbage collector can inspect, and which is amenable to runtime patching, based on code which may be dynamically loaded afterwards. The net effect is as if the level-1 instruction cache was severely reduced. In practice, one must implement partial unrolling in a way very similar to what is done for the C implementation for our MIPS test platform. Note that level-1 data cache is unaffected.

Another Java-specific overhead is about table accesses. Java implements an abstraction layer over memory; one cannot access RAM directly. Instead, tables use Java *arrays*, where every access is checked with regards to the actual array size. The JIT compiler tries to merge such checks, but array accesses are nevertheless more expensive than the already costly table accesses which are used in C. Functions which use many table accesses thus incur an extra slowdown on the Java VM.

4.6 Complexity

CPU do not fear executing code with thousands of distinct opcodes. Programmers, however, do. When a function consists of several distinct stages, each with many subtle and changing details, then the intellectual resources that the programmer can devote to optimization get diluted. The prime example is SIMD. This is the function which took the most time to implement, about twice more development time than any other of the candidates. In the end, in order to comply with our stated rule of "similar optimization efforts", we had to give up with making SIMD-512 fast in Java. This is why performance of SIMD-512 appears to be abysmal on Java platforms: we simply lacked the time to solve its severe L1-cache issues.

Regular designs are much easier to implement. For instance, BLAKE was very easy[5]. This yields more time to tinker with optimization. Design regularity should also be a valuable asset when trying to optimize for code size instead of raw bandwidth.

4.7 Consistency

Most of the round 2 candidates are actually pairs of functions, one "small" function for 224-bit and 256-bit outputs, and one "big" function for 384-bit and 512-bit outputs. Some functions (e.g. Fugue and Luffa) are actually *three* functions, the third one being used for the 384-bit output. Keccak is *four* functions (albeit sharing the same core).

Such a dichotomy has some negative effects:

- Implementing the whole family of functions requires more resources from the programmer, reducing correspondingly the resources allocated to actual optimization.
- Code size increases, when all the functions must be implemented together (e.g. for interoperability reasons).
- Performance issues and security cease to be orthogonal: the decision of which function output size to use, nominally a security-related issue, can no longer be taken independently of performance.

We note that SHA-2 already featured such a dichotomy. Among the SHA-3 candidates, three (CubeHash, JH and Shabal) avoid that dichotomy and provide consistent performance for all output sizes.

4.8 On the Importance of Speed

It shall be noted that many of the candidates, and SHA-2 as well, offer *good enough* performance on large systems. A good hard disk, or a gigabit network interface, in ideal conditions, tops at about 100 MB/s. Any hash function which offers at least that bandwidth will be sufficient for most purposes, in particular since we

[5] The clarity of the function specification considerably helped, too.

are talking about hash function bandwidth using a single core, and a PC which processes 100 MB of data per second is likely to have at least four cores. In brief, it is quite unlikely that even a not-that-fast hash function like SHA-2 becomes a bottleneck in any practical situation involving a large system.

On the other hand, our MIPS router has 100baseT connectivity, and thus may receive data at 10 MB/s speed. But the fastest SHA-3 candidate on that platform is Shabal (which is faster than SHA-1), and its bandwidth is below 7 MB/s. This means that hash function performance *is* a very real issue on that kind of system.

This is why we consider performance measures on embedded systems to be much more relevant than those on large systems.

4.9 About Special SIMD Units

Our code is portable, because it does not use the specific SIMD units that some processors offer (MMX, SSE2, AltiVec...). These units may be accessed through inline assembly or compiler-specific builtin functions, but they are not portable accross architectures. One could argue that on a specific system where hashing speed proves to be a bottleneck, it would make sense to use the platform-specific features, even at the expense of portability.

However, the platforms on which performance issues are most likely to appear are the small, embedded systems: precisely the platforms which do *not* offer SIMD units. On these small RISC processors, there is very little to gain in making non-portable code. The RISC concept is that a reduced instruction set makes it easier for compilers to produce efficient code. Using portable C code, as we do in `sphlib`, should then be the optimal implementation strategy.

Of course, SIMD units do not apply to the Java VM. The very essence of Java is about not using platform-specific opcodes.

5 Conclusion

Our measures show that most of the round 2 candidates were designed for maximum performance on large systems, preferably 64-bit architectures. For some of them, adequate performance is achieved only with the help of the special x86 units, such as the SSE2 unit (with its 128-bit registers and parallel execution features), or the AES-NI instructions which compute an AES round in two clock cycles on the newest Intel processors (all of them being newer than the NIST reference platform). In our tests, such extra units are not available, since we use portable code and try to obtain measures which can be interpolated to many distinct architectures in the same category.

Some candidates achieve very good performance on 64-bit platforms through the use of 64-bit integer operations, in particular BMW-512 (BMW-256 uses only 32-bit words). BMW-512 is the fastest function, by far, on 64-bit architectures. Unfortunately, its performance decreases quite a lot on 32-bit systems. In our view, BMW breaks records precisely on those architectures where breaking records matters the least.

It would be natural to suspect us of having chosen the specific test platforms on which Shabal would shine; after all, we are part of the Shabal design team, and this *is* a competition. However, it is really the other way round. When Shabal was designed, we had already conceived the idea that performance was most important on embedded 32-bit systems, and we made sure that Shabal was fast on those. It seems that few candidates followed the same path.

Development of `sphlib` will keep on. Other areas where performance issues should be investigated include:

- performance on hashing very small messages (less than one "elementary block");
- compact implementations, for constrained environments;
- script languages, in particular Javascript (which has almost nothing in common with Java).

References

[1] *Cryptographic Algorithm Hash Competition*, http://csrc.nist.gov/groups/ST/hash/sha-3/

[2] *eBASH: ECRYPT Benchmarking of All Submitted Hashes*, http://bench.cr.yp.to/ebash.html

[3] *sphlib 1.1*, http://www.crypto-hash.fr/modules/wfdownloads/singlefile.php?cid=13&lid=10

[4] *GCC, the GNU Compiler Collection*, http://gcc.gnu.org/

[5] *OpenWrt*, http://openwrt.org/

[6] *HPGCC*, http://sourceforge.net/projects/hpgcc/

[7] *Java SE Downloads*, http://java.sun.com/javase/downloads/index.jsp