Data-Oriented Performance Analysis of SHA-3 Candidates on FPGA Accelerated Computers

Zhimin Chen, Xu Guo, Ambuj Sinha, and Patrick Schaumont
ECE Department, Virginia Tech
{chenzm, xuguo, ambujs87, schaum}@vt.edu

***Abstract*—The SHA-3 competition organized by NIST has triggered significant efforts in performance evaluation of cryptographic hardware and software. These benchmarks are used to compare the implementation efficiency of competing hash candidates. However, such benchmarks test the algorithm in an ideal setting, and they ignore the effects of system integration. In this contribution, we analyze the performance of hash candidates on a high-end computing platform consisting of a multi-core Xeon processor with an FPGA-based hardware accelerator. We implement two hash candidates, Keccak and SIMD, in various configurations of multi-core hardware and multi-core software. Next, we vary application parameters such as message length, message multiplicity, and message source. We show that, depending on the application parameter set, the overall system performance is limited by three possible performance bottlenecks, including limitations in computation speed, in communication band-width, and in buffer storage. Our key result is to demonstrate the dependency of these bottlenecks on the application parameters. We conclude that, to make sound system design decisions, selecting the right hash candidate is only half of the solution: one must also understand the nature of the data stream which is hashed.**

I. Introduction

A cryptographic hash algorithm converts a variable-length message into a unique fixed-length digest. Hash algorithms are used to verify the integrity of digital storage such as disk files and memory pages. They are further used for authentication purposes, such as in message authentication (MAC) applications. The relationship between a message and its digest are unique. However, when two messages hash into the same digest, a collision between those messages exists. A cryptanalyst will attempt to find such collisions. If this can be done more efficiently than using a brute-force search for colliding messages, the hash algorithm is considered cryptographically broken.

The SHA-3 competition organized by NIST is motivated by recent advances in identifying collisions in the SHA-2 standard. The competition is executed in three phases, with the final phase starting in late 2010. Burr points to security, diversity, and performance as key selection factors for this new NIST standard [1]. In recent months, especially the performance evaluation of competing SHA-3 proposals has received significant attention from the research community. This may be because performance is easier to benchmark than security. Several crypto-benchmarking environments have been proposed. For example, EBACS [2] and Cryptopp [3] are aimed at general-purpose processors. sphlib [4], and XBX [5] are aimed at embedded processors. Athena [6] collects metrics on reconfigurable hardware.

We noticed that all these crypto-benchmarking proposals pay little or no attention to the system integration effects of a hash primitive. For example, they optimistically assume that the messages originate from a high-bandwidth data source. They also assume a random-access storage architecture that provides fast access to the messages regardless of their length. Hence, the aforementioned benchmarking systems aim to identify the *computational* bottleneck of each hash candidate under evaluation.

Our efforts are not aimed at building another cryptographic benchmark. Instead, we analyze the impact of non-ideal computer architectures on the performance of hash algorithms. Real computers have real limits, such as slow disks, limited cache memories, and computer busses with limited bandwidth. Moreover, exploiting parallelism in a computer architecture is complex, even for a simple task such as hashing multiple messages in parallel. We demonstrate that, on real computer architectures, there is a highly non-linear dependence of computer performance on the major hash application parameters (message size, message multiplicity, and hash candidate). This dependence has so far not been covered by crypto-benchmarks.

Our results are based on actual measurements. We use a computer with a quad-core Xeon processor and a tightly-coupled, high-end Field-Programmable Gate Array (FPGA) accelerator. We program multiple instances of a hash algorithm in hardware on the FPGA, or in software on the Xeon. We then evaluate the system-performance for a wide range of application parameters and message characteristics. We did this for two different SHA-3 candidates, Keccak and SIMD.

A major insight from this paper is that the implementation of hash algorithms on multi-core architectures may experience three different bottlenecks (computation, communication, or storage). Traditional benchmarks only reveal the computation bottleneck. In contrast, our results show that a hash application on a multi-core architecture in hardware or software may be in three operating modes, each characterized by a different bottleneck. We also show that the transition of one mode to the other depend on the hash message length.

An additional contribution of this paper is to compare

TABLE I
DESIGN SPACE EXPLORATION

Application	
Algorithm	Keccak-256, SIMD-256
Message Length (bytes)	8, 64, 576, 1536, 4K, 8K 16K, 32K, 64K, 128K, 256K
Message Source	Memory, Hard disk
Message Multiplicity	Single Message Hashing (SMH) Multi-Message Hashing (MMH)
Platform	
Computer	Intel server, 4 sockets, FSB
CPU Socket 1	Xeon E7310 Quad-Core 1.6 GHz
CPU Socket 2	Xilinx Virtex-5 SX-240T 200 MHz
OS (Kernel)	CentOS 5.4 (Linux 2.6.18-164.el5)
C Compiler	gcc version 4.12
HW Compiler	Xilinx ISE 12.2
Main Memory	8 GByte, peak bandwidth 21GB/s
Hard Disk	460 GByte, peak bandwidth 100 MB/s

hardware and software implementations of a hash candidate next to one another, in a single computer architecture and under the same application parameters. This comparison shows under what conditions an FPGA implementation of hashing will outperform a Xeon processor.

This paper is organized as follows. In the next section, we introduce the design space of the hash application under analysis. We also describe the computer architecture which we used to take measurements. Section 3 presents the results. We systematically describe how each of the three bottlenecks limits the hashing performance of the system. Section 4 concludes the paper.

II. DESIGN SPACE

In this section, we introduce the design space of our experiments. There are two aspects. The first is the application design space, which defines the application parameters of interest. The second is the target platform, and the strategy used to map the hash application. Table 1 summarizes the major features of the application design space and the target platform.

A. Application Design Space

1) Algorithm: We selected two hash candidates out of the 14 second-round SHA-3 candidates: Keccak and SIMD-256, both of which generate a 256-bit digest. In each case, we selected optimized source code for the hardware and software implementations of these algorithms. The software implementations of Keccak and SIMD are derived from the best implementations found for our server using EBACS [2]. The hardware implementations are derived from the source code of the AIST SHA-3 website [7].

2) Message Length: Message length is an important parameter for hash algorithm performance because of two reasons. First, hash algorithms may use additional post-processing to generate a digest, which makes short messages more expensive to handle. Second, practical applications use a wide range of message lengths. For example, Internet packet hashing requires mostly short messages, since half of the internet packets is shorter than 64 bytes [8]. Disk file hashing will require large message lengths. We selected a wide range of message lengths. We started from the packet lengths used by EBACS and extended those to include messsages up to 256 Kbytes.

3) Message Source: A third parameter in the application design space is the message source. We selected two likely source locations for a message in a typical computer architecture: main memory, and the harddisk. The difference between these two is substantial. The data bandwidth from a harddisk, available to an application, is two orders of magnitude smaller than the bandwidth from main memory.

4) Message Multiplicity: The final parameter considers the potential of parallel hashing: calculating the hash digests of multiple messages at the same time. This is beneficial for applications such as tree-hashing and hash collision search. We capture this parameter with the same terminology as used by Savas [9]. Single-Message Hashing (SMH) evaluates the hash for a single message at a time. Multi-Message Hashing (MMH) evaluates the hash for multiple messages at a time. The MMH case is suitable for task-level parallelization. Our computer architecture supports task-level parallel implementations in software as well as in hardware. The Xeon processor is a quad-core CPU and can execute 4 hash threads in parallel. The accelerator FPGA can accommodate up to 12 instances of Keccak or 10 instances of SIMD.

B. System Architecture

We now describe the architecture of the target platform, as well as the application mapping strategy used for software and hardware targets.

1) System Overview: Figure 1 provides a summary of the system architecture. A fast North Bridge integrates high-speed components, including a Xeon E3710 processor, an FPGA Accelerator, and main memory. A slower South Bridge integrates peripherals into the system, including the harddisk. Both the Xeon and the FPGA can directly access system memory using a Front Side Bus (FSB). The communication between Xeon and FPGA is implemented using a shared region in the system memory, called a *workspace*. Within this system architecture, the hash application is implemented as follows. Messages are stored either in system memory, or on hard disk, and flow over the South and North Bridge to the Xeon or the FPGA.

2) System Bottlenecks: A system architecture as shown in Figure 1 can experience three different bottlenecks. A *compute* bottleneck occurs when the performance of hash operations is limited by the speed of computations on the Xeon or the FPGA. A *communication* bottleneck occurs when the performance is limited by the bandwidth of a bus or interface, such as the FSB, or the hard-disk interface. A *storage* bottleneck occurs when the performance is limited by the size of local memories, including the local message buffers used inside of the FPGA.

3) Use Model: Within this system, there are two targets for hash computations. Designers can implement a hash on

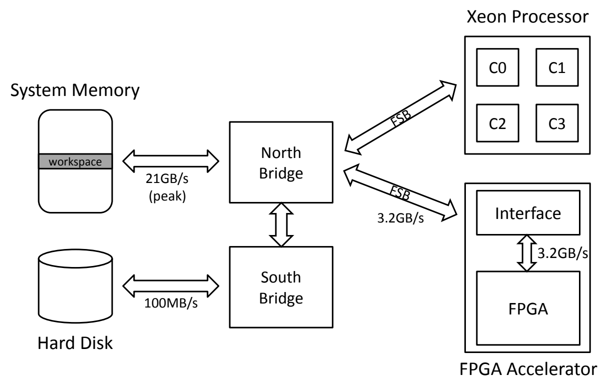

Fig. 1. System overview of the FPGA accelerated computer system.

the 4-core Xeon, on the FPGA accelerator, or on both of them. We explored two use-models. The first one is to map the hash computation in software on the Xeon. This model leads to the *Software Solutions*. The other model is to utilize the FPGA accelerator for hash computation and one core in Xeon to control the data communication between the memory or disk and the FPGA accelerator. This use model leads to the *Hardware Solutions*.

4) Software Solutions: The *software solutions* utilize only the Xeon processor. As discussed in II.A, we distinguish Single Message Hashing (SMH) from Multi-Message Hashing (MMH). In SMH, a hash candidate runs as a single thread on one core of the Xeon. In MMH, up to 4 parallel hash threads can execute on the Xeon processor, with one thread per core. By using POSIX threads, the kernel ensures that threads are evenly distributed over the available cores of the Xeon.

The application design space considers messages stored in system memory as well as on disk. Messages are stored in system memory as a contiguous array of bytes. In MMH mode, multiple arrays are used, one for each message. When messages are stored on disk, they are first read into system memory as a contiguous array. In MMH mode from disk, multiple disk files are used, one for each message. Each disk file is read into a contiguous array in system memory.

5) Hardware Solutions: The *hardware solutions* require hash implementations on the FPGA, as well as supporting control software running on one Xeon core.

The FPGA is operating as a slave to the Xeon. The Xeon first allocates a workspace to store the messages. If messages are stored on disk, the Xeon will then read the files, and copy them in the workspace. The Xeon then shares the workspace with the FPGA. For each hash, the FPGA is notified of the starting address and length of each message, and the destination address for the resulting digests. The FPGA will compute the digest, fill out the result, and then notify the Xeon.

Figure 2 illustrates the FPGA accelerator architecture. The data bandwidth between the FPGA and the FSB is 3.2 GB/s in each direction. Each hash core configured in the FPGA has a local buffer to store messages. To support the MMH mode, the FPGA also includes a controller that distributes messages to different hash cores. In our experiments, we evaluated both Keccak as well as SIMD. Table 2 shows the major characteristics of each hash core implemented in the accelerator FPGA.

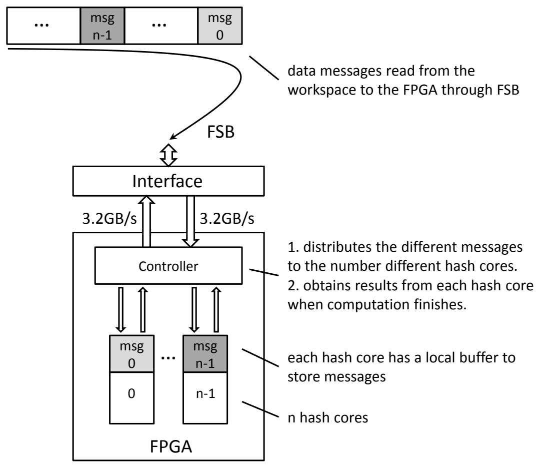
...
msg n-1
...
msg 0
data messages read from the workspace to the FPGA through FSB
FSB
Interface
3.2GB/s
3.2GB/s
Controller
1. distributes the different messages to the number different hash cores.
2. obtains results from each hash core when computation finishes.
msg 0
msg n-1
0
n-1
each hash core has a local buffer to store messages
n hash cores
FPGA

Fig. 2. FPGA accelerator with n hash cores integrated.

TABLE II
KECCAK AND SIMD DESIGNS IN FPGA

	Keccak	SIMD	
Hash Block Size	1024	512	Bit
Latency of 1 Block	28	48	Cycles
Area per Core	1541	4864	Slices
Operating Freq	100	50	MHz
Cores per FPGA	10	12	
Performance (UET)	2.2	15	ns/Byte

III. RESULTS

In this section, we present both the experimental results and their analysis. We group the discussion according to two application parameters: the message multiplicity and the message source. This gives us 4 categories: single-message hash from memory (SMH-MEM), multiple-message hash from memory (MMH-MEM), single-message hash from hard disk (SMH-HD), and multiple-message hash from hard disk (MMH-HD). Inside each category, we expand our discussion by taking additional parameters into account, such as the message length, sw/hw solutions, the number of Xeon cores or FPGA hash cores and so on.

Software benchmarks such as EBACS measure hash performance in cycles/byte, while hardware benchmarks such as Athena express performance as the throughput in Mbyte/second. In order to have an apples-to-apples comparison of hardware and software performance, we measure the performance of hashing as the average execution *time* to process one message byte. We call this the Unit Execution Time (UET), and it's measured in ns/Byte. The throughput (GB/s) can be calculated as the reciprocal of UET.

A. SMH-MEM

SMH-MEM is the application scenario assumed by most of the software benchmarks. Thanks to the FPGA accelerated

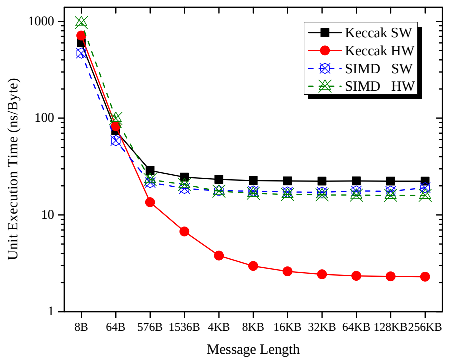

Fig. 3. Time to process one message byte in the SMH-MEM scenario.

system, we are able to investigate this application scenario from both software and hardware perspective. In SMH-MEM, we only use a single Xeon core for the software solution and a single hash core for the hardware solution.

The results are shown in Figure 3. As the message length increases, the UET of each solution decreases and then remains stable. This is consistent with the results shown by the software benchmark works. The software implementations of both Keccak and SIMD have similar performance. For long messages, software implementations of Keccak and SIMD achieve 22.36 ns/Byte and 19.1 ns/Byte, corresponding to a throughput of 44.72MByte/s and 52.36MByte/s. This is much lower than the system memory bandwidth, showing that the hashing performance is limited by the Xeon core. The SMH-MEM scenario in software is therefore performance-limited by a *computational* bottleneck.

The UET of the hardware implementations of Keccak and SIMD are 2.30 ns/Byte and 15.94 ns/Byte respectively, for long messages. This is very close to the stand-alone performance of the stand-alone hardware modules (Table II). It is also well below the FSB bandwith of 3.2 GB/s. Therefore, the SMH-MEM scenario in hardware is computation-limited for long messages. On the contrary, when messages are short, the performance degrades significatnly (713.44 ns/Byte for Keccak and 971.25 ns/byte for SIMD). Two factors are responsible for this: software padding and the communication overhead. Communication here includes both the interaction between the Xeon and the FPGA (shared-memory exchange overhead) and the data transfer between the system memory and the FPGA. According to our measurements, software padding takes around half of the UET and the communication overhead is responsible for the other half. Thus, even a hardware implementation of the padding function can only provide a limited improvement. We conclude that, for short messages in the hardware SMH-MEM scenario, system performance is limited by the FSB communication bottleneck.

Comparing the software and the hardware, we can see that hardware solutions only outperform software solutions for long messages. This is because software does not have the communication overhead that bothers the hardware. While the communication overhead can be averaged out over long messages, but not for short messages. Hence, hardware acceleration does not pay back on short messages.

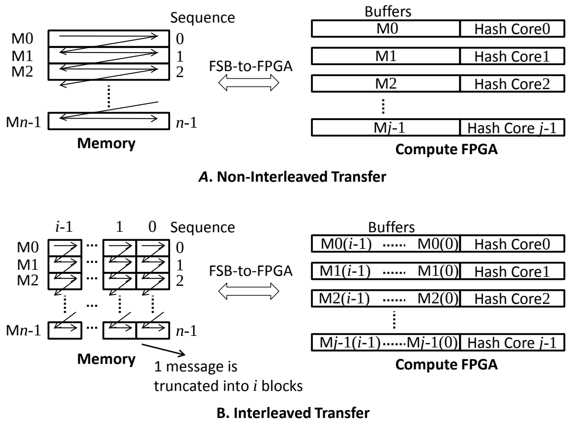

Fig. 4. Two different data transfer approaches: (a) non-interleaved transfer; (b) interleaved transfer.

B. MMH-MEM

In the MMH-MEM scenario, multiple messages are processed by multiple Xeon or FPGA cores. The software solutions use POSIX threads to execute multiple hash operations in parallel. Each thread runs on a separate Xeon core. Similarly, the hardware solutions configure multiple hash cores in the FPGA and execute hash operations in parallel. In such a parallel execution, the latency per message does not decrease, but the throughput over all messages does increase. As a result, UET becomes lower.

Despite the similarities between the software and the hardware solutions, there still exists a major difference. In the software solutions, the Xeon has random-access to the entire system memory. In the hardware solutions, the FPGA operates as a slave to the Xeon. Hash cores in the FPGA can only process the data stored in the local buffer, where it is copied to from the system memory. Since the hash cores do not have random-access to the system memory, the data format transferred from the system memory has a great impact on the system performance. This leads to the buffer storage bottleneck, discussed below.

1) Non-interleaved Transfer: The straightforward implementation of MMH hashing on the hardware accelerator is to use a non-interleaved transfer, illustrated in Figure 4a. In this case, the messages are processed in their entirety, one by one. In this case, hash core $\#c$ will not get its message $\#m$ until the message $\#m-1$ is all stored in hash core $\#c-1$'s local buffer. This requires a large local buffer for each hash core.

2) Interleaved Transfer: A solution which is more efficient from a hardware perspective is to use an interleaved transfer. In this case, data blocks from different messages are interleaved. A controller in the FPGA distributes the data blocks belonging to different messages to different hash cores. This process is shown in Figure 4b. The advantage of the interleaved transfer is that each hash core gets data to process on time and the local buffer can be small. On the other hand, interleaved transfers require an interleaved data organization in system memory, which is more complex from an application viewpoint.

Figure 5 presents the results of the software solutions. Besides the effects related to the message length that are also observed in Figure 3, we also see that the UET decreases linearly according to the reciprocal of the number of cores for both Keccak and SIMD. Therefore, the MMH-MEM software solutions still are limited by a computational bottleneck.

For hardware solutions, as the application has multiple messages to process, we are able to first pad c messages (given there are c hash cores), then transfer them within one transfer operation, and finally transfer the results back at one time. With non-interleaved transfer, we divided the design space into two parts depending on whether the message length is larger than the size of each hash core's local buffer (8KBytes). The results are shown in Figure 6. The processes on the first part have a message-handling bottleneck, similar to the SMH-MEM case. Beyond a message length of 8Kbyte, a serious non-linearity shows up. No matter how many hash cores we use, the UET always converges to the single-core's UET. The reason is that when the local buffer on each core is not large enough to accommodate one message, the rest of that message will stay on the FSB channel and block the transfer of all the other messages. For large messages, there remains only one FPGA core that has data to process, which degrades the multi-core FPGA to a single-core FPGA. Therefore, we conclude that, with non-interleaved transfer in the MMH-MEM scenario, hashing short messages (shorter than buffer) still has bottleneck on communication while hashing long messages (longer than buffer) has a bottleneck on buffer storage.

Figure 7 shows the results of hardware solutions with interleaved data transfer. It shows an increase of the available architectural parallelism and an elimination of the local-buffer bottleneck discussed before. However, more hardware parallelism will also require more data bandwidth. This leads to another bottleneck as follows. As the number of cores increases, the UET of SIMD always decreases linearly. However, this linearity does not hold for Keccak after 8 or more cores are integrated, since the UET curves for 8- and 12-core Keccak almost overlap. The UET of 8- and 12-core Keccak implementations are both around 0.32 ns/byte, which means a throughput of 3.13 GB/s. This is very close to the FSB communication throughput limit offered by the computer system. Therefore, for the long-message case in the MMH-MEM scenario and with the interleaved data, SIMD hardware solutions are limited by a compute-bottleneck, which the Keccak hardware solutions are limited by the FSB communition-bottleneck.

C. SMH-HD

The SMH-HD scenario corresponds to hashing a file on the disk. We first read the file from the hard disk to the system memory and then process it either with a Xeon core or a FPGA core. So in addition to the time cost shown in the SMH-MEM scenario, there is an additional cost for data transfer from disk to memory. This section provides a quantitative view of this problem. Figure 8 illustrates the results.

Compared with Figure 3, we see UET increases more than 0.1 ms for short messages while only around 10 ns increase for long messages. This can be explained as follows. To access the messages on disk requires not only an overhead at the beginning but also the time to transfer data. Similar to the communication between the system memory and the FPGA accelerator, the overhead degrades the performance

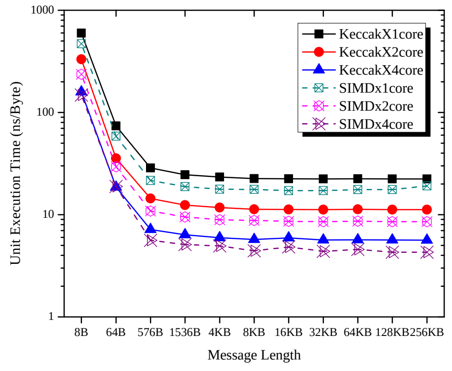

Fig. 5. Time to process one message byte with software in the MMH-MEM scenario.

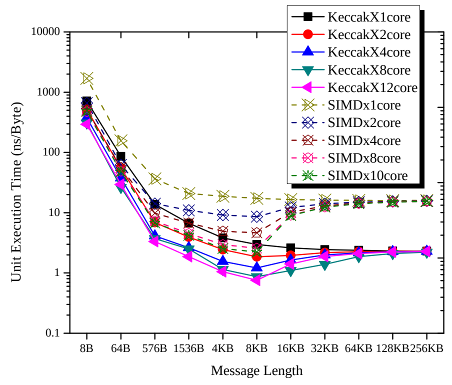

Fig. 6. Time to process one message byte with FPGA hardware in the MMH-MEM scenario with non-interleaved data.

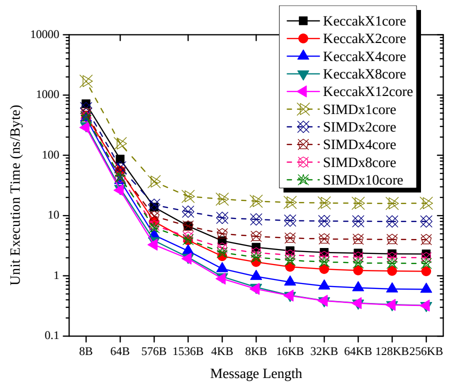

Fig. 7. Time to process one message byte with FPGA hardware in the MMH-MEM scenario with interleaved data.

TABLE III
DOMINATING BOTTLENECK DISTRIBUTION ACCORDING TO APPLICATION PARAMETERS

	software		hardware	
	short msg	long msg	short msg	long msg
SMH-MEM	Xeon comp.	Xeon comp.	FSB comm.	FPGA comp.
MMH-MEM	Xeon comp.	Xeon comp.	FSB comm.	BUF storage[1], FSB comm.[2], FPGA comp.[3]
SMH-HD	disk comm.	disk comm.	disk comm.	disk comm.
MMH-HD	disk comm.	disk comm.	disk comm.	disk comm.

[1] Non-interleaved transfer for both Keccak and SIMD.
[2] Interleaved transfer for Keccak.
[3] Interleaved transfer for SIMD.

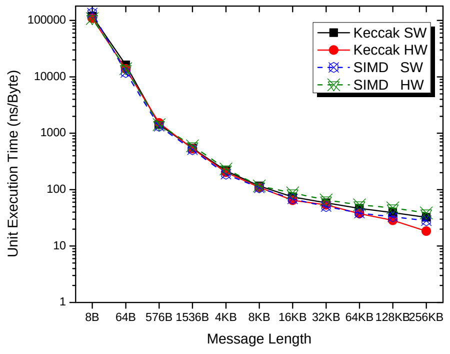

Fig. 8. Time to process one message byte in the SMH-HD scenario.

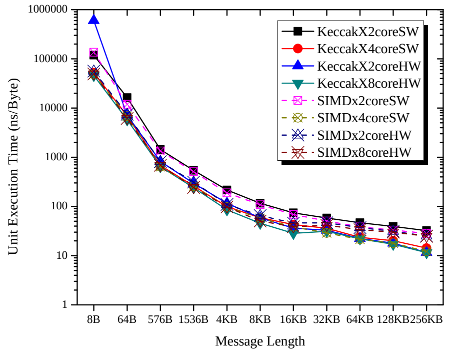

Fig. 9. Time to process one message byte in the MMH-HD scenario.

on short messages while only exerting a ignorable influence on long messages. For long messages, the 10-ns increase on UET comes from the limited communication bandwidth of the hard disk, which is around 100MB/s. In comparison, the harddisk communication cost is more dominant than the computation cost. Therefore, harddisk communication is the dominant bottleneck in the SMH-HD scenario.

D. MMH-HD

Similar to the SMH-HD applications, MMH-HD applications also need to first move the data from disk files to the memory. Since this operation can only support transferring one message at one time, the cost on this part should be similar to the one in the SMH-HD scenario, regardless how many cores can be used for processing. Compared with the SMH-HD case, the computation is faster, but the communication bottleneck remains the same. This is shown in Figure 9. We can see Figure 9 and Figure 8 show very similar curves. This demonstrates that, under the harddisk communication bottleneck, the value of additional parallelism is very limited.

IV. CONCLUSIONS

We use Table III to summarize our experiments. We see that the bottleneck of a system has a high dependency on the application parameters. In addition, we show that due to the slave-execution model of the accelerator hardware, hardware solutions are not as good as software solutions even though the hash cores are faster. Moreover, besides the commonly-mentioned bottlenecks, such as computation and communication, we find a third one: the buffer storage. This bottleneck can be avoided using message interleaving, but this will definitely increase the FSB communication penalty. Thus, message data organization definitely deserves attention when integrating a hash algorithm into a parallel computer architecture. We consider the solution is out of the scope of this work and will continue this research in the future.

V. ACKNOWLEDGEMENT

This research was supported in part through NSF Grant no. 0644070.

REFERENCES

[1] W. E. Burr, “A New Hash Competition,” *Security Privacy, IEEE*, vol. 6, pp. 60 –62, may. 2008.
[2] D. J. Bernstein and T. Lange, “eBACS: ECRYPT Benchmarking of Cryptographic Systems,” 2010. http://bench.cr.yp.to/.
[3] W. Dai, “Security Policy FIPS 140-2 Level 1 Validation Crypto++ Libarary,” 2003. Available at http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp343.pdf.
[4] “sphlib 2.1,” available at http://www.saphir2.com/sphlib/.
[5] C. Wenzel-Benner and J. Gräf, “XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework,” in *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, vol. 6225, p. 294, 2010.
[6] K. Gaj, E. Homsirikamol, and M. Rogawski, “Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs,” in *CHES*, pp. 264–278, 2010.
[7] AIST, “SHA-3 Hardware Project,” 2010. Available at http://www.rcis.aist.go.jp/special/SASEBO/SHA3-en.html.
[8] C. A. for Internet Data Analysis (CAIDA), “Packet Length Distribution,” 2010. Available at http://www.caida.org/research/traffic-analysis/AIX/plen_hist/.
[9] A. Akin, A. Aysu, O. C. Ulusel, and E. Savas, “Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa and Blie Midnight Wish for Single- and Multi-Message Hashing,” in *2nd NIST SHA-3 Conference*, 2010.