Optimization and Benchmark of Cryptographic Algorithms on Network Processors

THIS WORK COMPARES AND ANALYZES ARCHITECTURAL CHARACTERISTICS OF MANY WIDESPREAD CRYPTOGRAPHIC ALGORITHMS ON THE INTEL IXP2800 NETWORK PROCESSOR. IT ALSO INVESTIGATES SEVERAL IMPLEMENTATION AND OPTIMIZATION PRINCIPLES THAT CAN IMPROVE OVERALL PERFORMANCE. THE RESULTS REPORTED HERE ARE APPLICABLE TO OTHER NETWORK PROCESSORS BECAUSE THEY HAVE SIMILAR COMPONENTS AND ARCHITECTURES.

Zhangxi Tan
Chuang Lin
Hao Yin
Tsinghua University

Bo Li
Hong Kong University of Science and Technology

Information security is an indispensable concern owing to the growing demands for trusted communication and electronic commerce. For example, applications such as those for secure IP (IPSec) and virtual private networks (VPNs) have been widely deployed in nodal processing. On the other hand, network processors that provide a balance between performance and flexibility become fundamental building blocks in nodal devices to handle diverse applications and protocols.

As requirements for communication security grow, cryptographic processing becomes another type of application domain. However, cryptographic algorithms are all computationally intensive.[1] Furthermore, networks must apply them to every packet crossing a secure link, so cryptographic processing can have a significant impact on overall performance.

To address this problem and add security functions to network processors, a straightforward approach—one that achieves comparable performance—is to implement them in hardware. Unfortunately, many security chips or coprocessors can only handle a few algorithms, while most Internet security standards allow flexibility in algorithm selection. In addition, cryptographic hardware is not cheap or readily exportable. To compensate for these drawbacks, vendors often build security functionality directly into the same silicon as the network processor. But this method is still inflexible in that it cannot implement multiple algorithms (the Intel IXP2850 network processor, for example, has only two block ciphers and one hash algorithm). Besides, data must traverse shared memory and buses at least four times. So the resource contention problem actually prevents those inline cryptographic units from reaching their claimed performance.

Hence, the implementation of cryptographic applications on network processors via software is still necessary. Clearly, the most challenging work for software implementations is to provide performance guarantees, for instance, covering the handling packets at high speed. On general-purpose processors, traditional optimization techniques emphasize

0272-1732/04/$20.00 © 2004 IEEE Published by the IEEE computer Society

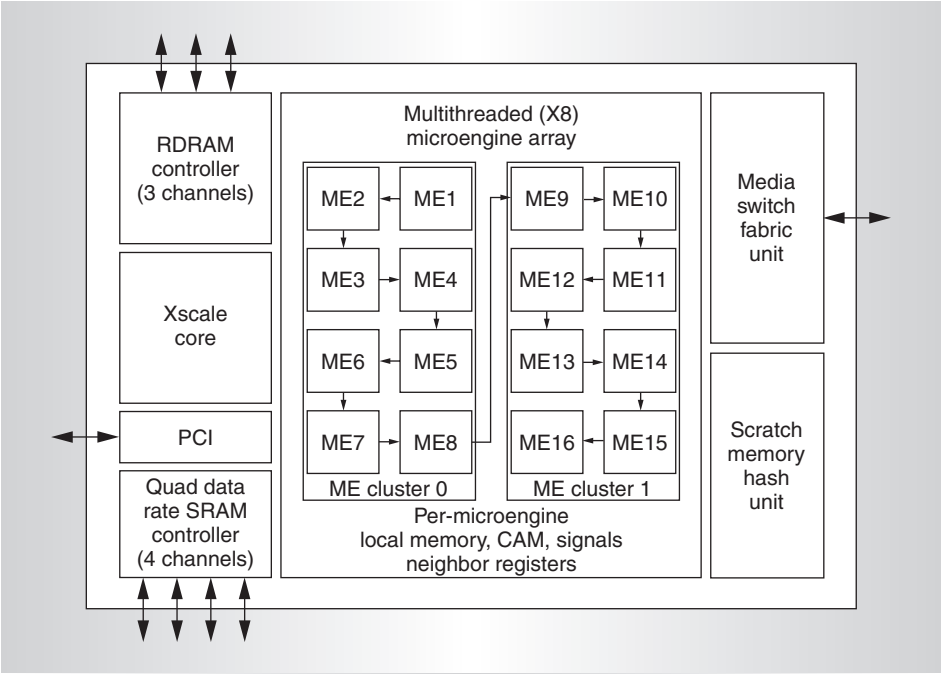

Figure 1. Intel IXP2800 hardware architecture.

improving instruction level parallelism (ILP) and cache performance.[2] However, most network processor architectures employ multiple processing engines (PEs) that work on data plane processing; these multiple PEs go by several names: microengine, channel processor, or task-optimized processor. Based on RISC cores, such PEs provide little ILP and offer no cache or a small cache. What complicates the problem is that PEs often communicate directly with I/O. Accordingly, they must be able to account for even small variations in I/O operations. What's more, the impact of security applications performed on such platforms is still unclear.

Most recent studies of cryptographic issues on network processors assume a symmetric multiprocessor (SMP) or superscalar architecture with multilevel caches.[2,3] Such architectures, however, are more similar to general-purpose processors; those studies thus ignore many characteristics such as hardware multithreading and asynchronous I/O that are common in actual network processors. In addition, such studies tend to measure results using simulators for general-purpose processors; these simulators cannot cope with special memory and I/O features unique to network processors.

Our work studies architectural properties for several widespread cryptographic algorithms on an actual Intel IXP2800 network processor. In this article, we propose several implementation and optimization principles and also discuss three topics in benchmarking the optimized algorithms.

First, can multi-thread and multi-PE scale up the processing? Second, what are the potential bottlenecks and what are their causes and solutions? Third, what optimizations can be made to escalate the performance?

Intel IXP2800 architecture

Closely examining the IXP2800's hardware architecture, shown in Figure 1, helps to elucidate our implementation and optimization. IXP2800 is a member of Intel's second-generation network processor family. Like its predecessor, IXP1200, IXP2800 is also a 32-bit RISC-based multicore system that exploits the system-on-chip (SOC) technique for deep packet inspection, traffic management, and high-speed forwarding. The 700-MHz XScale core is a general-purpose processor used for exception handling, slow-path processing, and other control plane tasks. The sixteen 1.4-GHz microengines (MEs) are data plane PEs, connected in two clusters. MEs in the same

Table 1. Characteristics of IXP2800 register and memories.

Name	Size	Transfer size (bytes)	Reference latency (no. of cycles)*	Application
Per-ME storage				
General-purpose register	256*4 bytes	4	1	General programming
Transfer register	512*4 bytes	4	1	Transferring data from other MEs, memory, or I/O devices
Next-neighbor register	128*4 bytes	4	1	Communication with an adjacent ME
Local memory	640*4 bytes	4	5	Caching data needed by ME
Shared-memory among all MEs				
Scratch	16 Kbytes	4	100/40**	Fast access to processing state or data shared by all MEs
Quad data rate SRAM	64 Mbytes	4	130/53**	Low-latency access to smaller data structures
RDRAM	2 Gbytes	16	295/53**	Large storage for packet buffers and bulk data transfer

* Reference latencies are measured in ME cycles, configuring ME at 1.4 GHz, SRAM at 200 MHz, and RDRAM at 400 MHz.

** These average memory access latencies are for read/write.

cluster share a common command bus, which they use to forward memory and I/O requests to other relevant units. Adjacent MEs (referred to as *next neighbors*) connect together in a pipeline with their nearest neighbors to provide one-way communication.

Intel designed the 32-bit media switch fabric and PCI interface to connect to a media access controller and external devices. Unlike general-purpose processors—which rely heavily on a large cache and efficient cache replacement policies to improve performance—the lack of locality in packet processing has forced network processor designers to come up with innovative memory and PE architectures. For example, IXP2800 has a distributed, shared-memory hierarchy that supports two types of external memory: RDRAM and quad-data-rate SRAM. In addition, the processor includes a 16-Kbyte, on-chip, scratch SRAM (shared among all MEs), plenty of registers, and a small amount of local memory per ME. In Table 1, we list the capacity, transfer size, reference latency, and the typical usage of these registers and memories. As shown in the table, memory access latencies have not kept pace with ME processing speed. For instance, the minimum read latency for fastest shared SRAM (scratch) is 100 ME cycles.

To solve this problem, the IXP architecture uses eight *zero-thread-switching-overhead* hardware threads for interleaved operation—one thread does computation while others block, waiting for memory operations to complete. Thread swapping can be software controlled, and MEs can perform asynchronous memory and I/O operations using multiple signals that indicate the completion of these references. Moreover, each ME supports a single-cycle arithmetic logic unit (ALU) with shifter, a multiply unit, and other specially designed I/O instructions.

Cryptographic algorithms

Cryptographic processing is the art and science of transforming regular data (plaintext) into encrypted data (ciphertext). Such processing primarily occurs in three application domains:

- *Public-key ciphers.* In this domain, a sender uses an openly published public key for encryption. For decryption, the receiver uses a private key that only it knows.
- *Private-key ciphers.* Private-key applications rely on the sender and receiver sharing of a common private key for use in encryption and decryption.
- *Hash functions.* These functions transform a variable-size input to an irreversible, fixed-length hash code. Both

Table 2. Selection and characteristics of cryptographic algorithms.

Name	Block size (bits)	No. of rounds	Table size (bytes)	Description or applications	Special requirements
Block ciphers					
Data Encryption Standard (DES)[4]	64	16	256	First modern, commercial-grade cipher	None
Advanced Encryption Standard (AES)[5]	128	10	5,120	Incorporated into 802.11i	None
International Data Encryption Algorithm (IDEA)[6]	64	9	0	Pretty Good Privacy (PGP) and Secure Shell/Secure Sockets Layer (SSH/SSL)	Multiply unit
RC5 (Ron's Code 5)[7]	64	16	136	Wireless transport security layer in the Wireless Application Protocol	32-bit variable rotation engine
RC6 (Ron's Code 6)[8]	128	20	176	AES candidate, improved version of RC5	Multiply unit and 32-bit variable rotation engine
Blowfish[9]	64	16	4,168	Incorporated into Norton Utilities	None
Stream ciphers					
RC4 (Ron's Code 4)[10]	NA	NA	256	SSL/TSL, 802.1x	None
SEAL (Software-Optimized Encryption Algorithm)[11]	NA	64 + 2	less than 4,096*	Disk encryption	None
Hash functions					
MD5 (Message Digest 5)[12]	512	64	0	Digital signatures	None
SHA-1 (Secure Hash Algorithm 1)[13]	512	80	0	Digital signatures	None

* SEAL's table size is variable concerning the output length. Table 2 lists the upper bound.

sender and receiver must know the hash algorithm used.

In this article, we do not address public-key ciphers, because they require storing a large amount of code and have slow execution speeds, making them impractical for implementation on the fast path of a network processor. Besides, typical cryptographic systems like the Secure Sockets Layer (SSL) use only public-key ciphers for short sessions and private-key management, while private-key ciphers are critical for long-session performance.

Consequently, we focus on private-key ciphers and hash functions, further classifying the former into block and stream ciphers. *Block ciphers* transform a fixed-length block of plaintext into a block of ciphertext, using a process that has several rounds. In contrast, *stream ciphers* take data of variable length and randomly generate key streams to "XOR" with plaintext. Of the many algorithms, we selected a subset of 10 algorithms based on their popularity, availability, and how representative they were of encryption processing in general. Table 2 summarizes the characteristics of these algorithms. Some algorithms—such as AES and RC5—allow variable block sizes and number of rounds.

In our experiments, we select recommended values for characteristics, according to algorithm standards or which ones are most popular

in commercial applications. Most algorithms include several key-dependent or key-independent tables, called S- or P-boxes. Their sizes not only affect security from a cryptographic standpoint, but also have a substantial influence on encryption speed. For this evaluation, our selection combines small or no table algorithms (such as hash function) and large-table algorithms (such as AES and SEAL).

Experiments

To observe the architectural characteristics of cryptographic algorithms and their utilization of internal resources and to detect performance bottlenecks, we conducted our experiments under Workbench 3.1, a cycle-accurate IXP2800 simulator. Our experiments covered 1.4-GHz ME configurations, 200-MHz SRAMs, and 400-MHz RDRAMs. We compiled all the source codes using the Intel Microengine C compiler 3.1 with optimization level -O2, which offers the basic instruction and language optimizations. Some operations, such as rotation, are not directly expressible using C operators but are supported by IXP instructions. So we implement these operations with inline assembly codes. Hence, our optimization principles do not focus on specific instructions unique to one target but general features applicable to a wide range of network processors. In addition, to test the scalability of parallel optimization, we used up to 8 MEs (64 threads total) in one ME cluster, as described earlier.

Almost every known cryptographic algorithm includes several small inner loops that consume the vast majority of all processing time. Other operations, such as key scheduling and table initialization, can be precalculated at set-up time by slow path processors (such as the XScale in the IXP2800). Thus, our work focuses on the algorithm characteristics of the inner loops, while many related statistics include other portions, even those that contribute little to overall performance. To obtain benchmark results, we manually apply the proposed optimization principles to each algorithm. We store both the input and output data in RDRAM, according to the IXP2800 memory usage locations in Table 1.

Instruction characteristics

In this section, we present experimental statistics on the instruction distribution of these algorithms. These metrics are essential information in understanding the dynamic properties of these algorithms and developing implementation and optimization principles. Figure 2 illustrates the instruction mix profile and code size of all selected algorithms. Table 3 depicts average statistics compared with Commbench[14] and NpBench,[15] two existing network processor benchmark suites that contain typical header and payload processing applications.

Based on this information, we see that most block and stream ciphers need only a little code storage (less then 200 lines of code). The only exception is DES because it has several complex bit operations that you must implement using a set of instructions indirectly. Hash functions usually need more code storage. On IXP2800, the codes for MD5 and SHA-1 occupy 17.4 and 31 percent of the overall 4-K instruction storage.

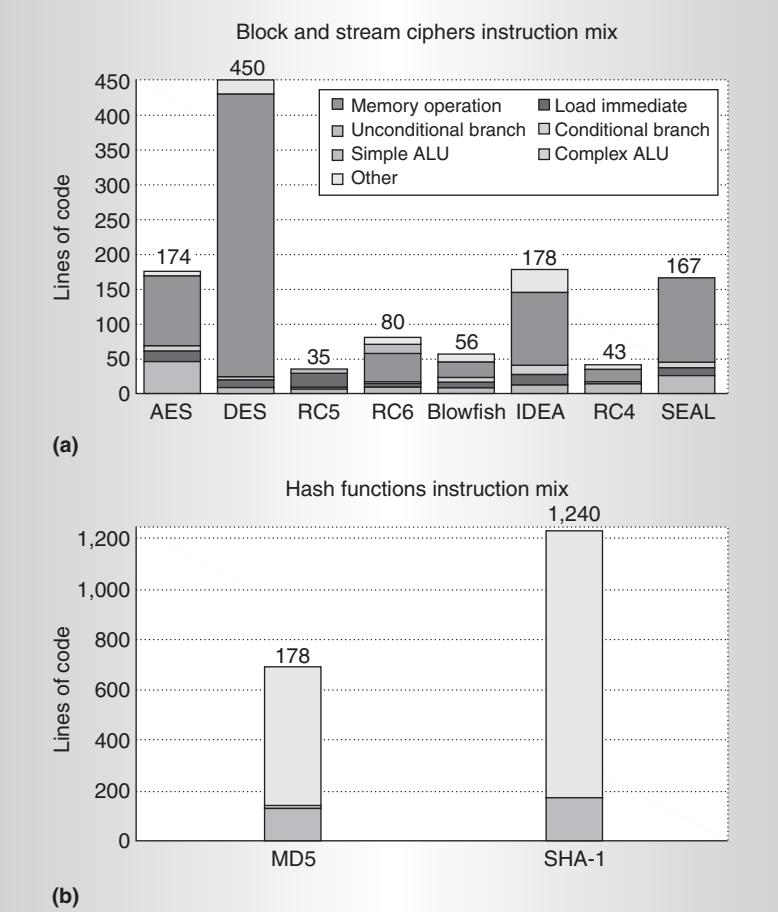

Figure 2. Raw code sizes and instruction mix.

Table 3. Comparisons of average instruction mix.

Instruction type	Average mix	Commbench PPA*	Commbench HPA*	NpBench
Memory	4.0	26	33	28.2
Load immediate	12.2	1	2	NA
Unconditional branch	0.7	1	1	16.2
Conditional branch	0.8	13	18	
Simple ALU	78.6	58	41	53.5
Complex ALU	1.3			
Other	2.4	1	5	2.2

* Commbench has two groups of benchmarks: payload processing applications (PPA) and header processing applications (HPA).

The most frequently used instructions are ALU instructions, especially simple ALU instructions like add, shift, and logic. Only algorithms with special requirements (as indicated in Table 2) need a few complex ALU instructions like multiply and complex shift. As a whole, ALU instructions occupy a significant share of the total instruction mix, which is 79.9 percent on average followed by Commbench PPA (58 percent), NpBench (53.5 percent), and Commbench HPA (41 percent) instructions. Hence, cryptographic algorithms will consume more of network processor's computing power than other payload or header processing applications.

Compared to other applications, every cryptographic algorithm uses fewer branch instructions in every algorithm. The average percentage decrease is 1.5 percent, which is much lower than that of Commbench PPA (15 percent), Commbench HPA (20 percent), and NpBench (16.2 percent).

Memory and load-immediate instructions exhibit significant differences from algorithm to algorithm. Stream ciphers and some block ciphers (AES and Blowfish) tend to have a relatively higher percentage of memory instructions than hash functions, which primarily consist of ALU and load-immediate instructions. Except for AES, which has a percentage of memory instructions (26 percent) comparable to that of header processing applications (33 percent in Commbench HPA), the average percentage of memory instructions for the 10 algorithms is only 4 percent. However, because of the long access latency described earlier, memory operations should still be carefully handled, as we will describe in detail in the following sections.

Optimization principles and benchmarks

We describe our optimization and benchmarks in two subsections. The first focuses on general implementation and optimization principles for a single thread within one ME. The second section considers multithreading and the results from scalability tests with multiple MEs.

Single threads: Generic optimizations

Generic implementation and optimization rules are independent of the particular network processor, and most are suitable for optimizing general-purpose processors. Their goal is to minimize the overall computation complexity and decrease the number of expensive operations.

Take advantage of memory. For optimum performance, single threads must take full advantage of rich register resource and the distributed memory hierarchy. To minimize access latencies, place some frequently used tables into registers and per-ME local memories as much as possible. This situation is similar to prefetching and using data caches on general-purpose processors; but in network processors, this memory use is controlled directly by software. The C compiler used in our benchmarks can automatically select the fastest memory storage for the given table size. To further improve the optimization, we also manually split large tables. For instance, two of the five total tables of AES go into local memory and "unused" registers, while other tables remain in scratch memory. Our optimizations handle Blowfish tables in a similar fashion.

Avoid using complex instructions. Avoid instructions, like multiplication, that consume more than one cycle. For example, replace a power of 2 multiply with shift operations.

Precalculated parts of algorithms. Aside from the table initialization and key scheduling mentioned earlier, immediate data used in inner loops can also be preloaded. Moreover, on architectures like IXP2800, loading a 32-bit immediate requires two instructions. As an example, hash functions use the C statement "a += b + c + int32" extensively, where variables *a*, *b* and *c* all fit in registers; int32 is a 32-bit immediate. To reduce extra cycles loading the immediate, replace this statement by "a += b + c + d," in which *d* is a register preloaded with the value int32.

Unroll loops. This can prevent the pipeline flushing and save extra clock cycles. On IXP2800 (which has 6 pipeline stages), this can save 3 cycles for each pipeline stall. Besides, unrolling loops can reduce calculations that concern iteration variables and make addressing in arrays more efficiently. For instance, the inner loop of SEAL is

```
for(i=0;i<64;i++) {
...
a+= N[2*(i&1)];
c+= N[2*(i&1)+1];
...
}
```

Unrolled twice, this loop becomes

```
for(i=0;i<32;i++) {
...
a += N[0]; c += N[1];
...
a += N[2]; c += N[3];
...
}
```

The second code segment greatly simplifies addressing in array *N*. It also halves the total number of branches in the loop. However, this improvement is at the cost of code size. Fortunately, the small code size of most cryptographic algorithms provides a good opportunity for fully unrolling (eliminating) the loop.

Single threads: Network-processor-dependent memory optimizations

These principles use special optimized memory and I/O units on network processors to increase ME utilization rate and stretch the computation capacity to the utmost level.

Align-memory operations. Table 1 shows transfer size for different memory units. Access to data sizes smaller than those supported by the hardware incurs overhead. Thus, to achieve optimal performance, an optimized application should align tables at hardware boundaries. For instance, an application would store 8-bit table entries in RC4 as 32-bit variables in per-ME Local Memory, at the expense of more storage space.

Memory burst read and write. On most network processors, a PE can issue memory burst operations directly at the instruction level. IXP2800 allows a 32-byte scratch or SRAM, or 64-byte RDRAM burst reference within one instruction. Employing this mechanism further reduces memory instructions. In our benchmark, reading plaintext and writing ciphertext are all burst operations at their block sizes.

Latency hiding and I/O parallelization. This makes use of asynchronous memory operations to hide long memory access latencies and to improve the ME utilization rate. The core idea is to continue calculating while waiting for the completion of references. Further, with the mechanism of complete signals and command queues, the microengines can issue multiple memory references simultaneously. Figure 3 shows an example of this simultaneous issue.

Figure 4 presents single-thread throughputs for selected algorithms with the application of different optimization principles. Figure 5 (on p. 63) gives related internal statistics for MEs. As is evident from the plot, hash functions have the best performance (see MD5 1,219 Mbits/s) followed by stream ciphers and block ciphers. DES achieves the lowest throughput (32.2 Mbits/s after optimization) because it works at the bit level; the 32-bit IXP2800 has weak support for bit-level instructions.

Although ALU instructions are the most frequently used in every algorithm, generic opti-

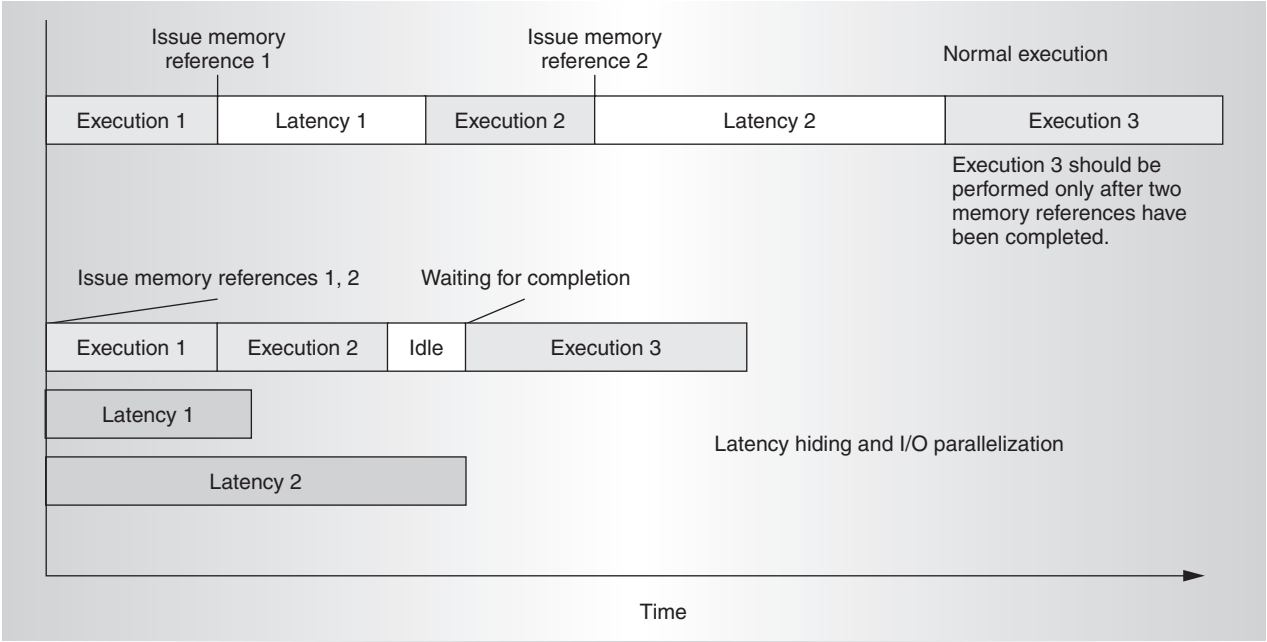

Figure 3. Example of latency hiding and I/O parallelization.

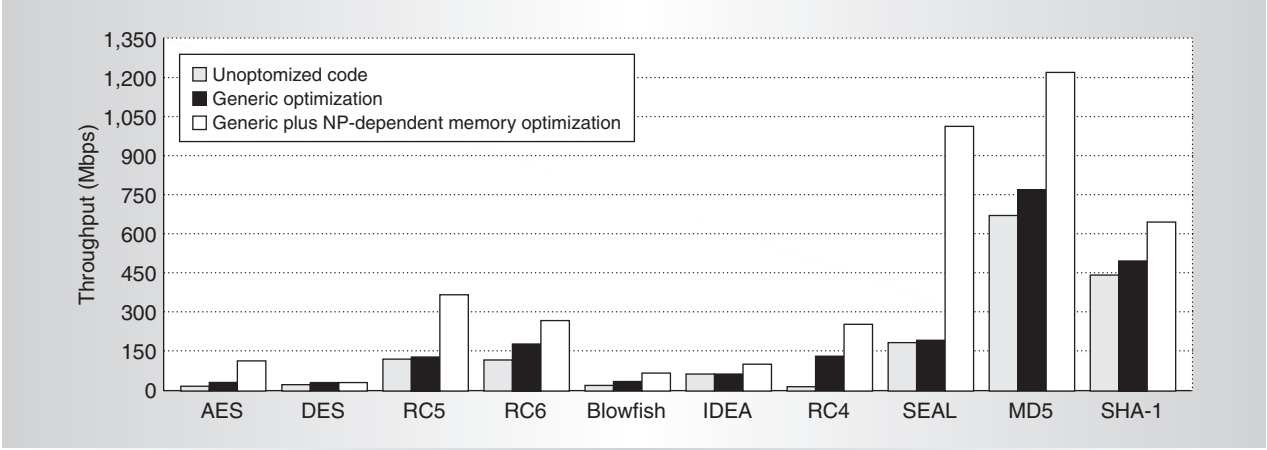

Figure 4. Single-thread performance with different optimization principles.

mization techniques, which aim to improve computational efficiency, are less effective than network-processor-dependent memory optimizations. For most algorithms, generic optimizations can only provide less than a 15 percent improvement. The exceptions are RC4, AES, and Blowfish. Because RC4 is 8-bit aligned, after loop unrolling, this algorithm can process the data at 32-bit boundaries, reducing overall memory operations. Both AES and Blowfish have large tables, so their performance enhancements come mainly from manually splitting large tables.

The effect of pipeline optimizations seems quite limited. This is because of the short pipeline architecture of network processors and low percentage of branch instructions (less than 2 percent) in cryptographic algorithms. The execution statistics also support these observations. Before any optimization, an ME was only aborted (threads experience a pipeline flush because of branch or thread swapping) for a small percentage (0.2 to 2.2 percent) of ME utilization time.

Most stream and block ciphers suffer from a low ME utilization rate (see the low levels of

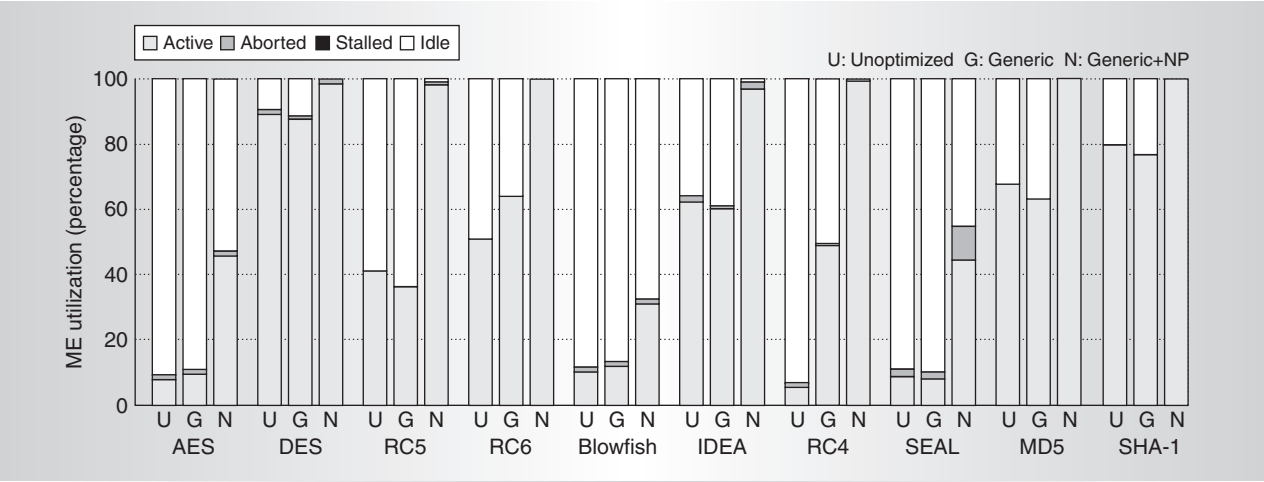

Figure 5. Internal statistics for MEs.

active utilization for these algorithms in Figure 5), but generic optimizations do not account for long memory reference latencies. On the other hand, network-processor-dependent memory optimizations effectively hide them and significantly increase the ME utilization rate, especially for algorithms, which have a high percentage of memory operations. Even though hash functions consist of less than 1 percent of memory instructions, memory optimizations still yield more speedup than generic optimizations. Thus, for network processors, hiding memory latencies should have a high priority, unlike related discussions on general-purpose processors, which put less emphasis on memory systems.[2]

From Figure 5, we also observe that all algorithms except AES, Blowfish, and SEAL have a near 100 percent ME utilization rate after memory optimizations. Hence, the computing power of MEs remains their bottleneck.

In contrast, long access latency, rather than memory bandwidth, limits the throughput of AES, Blowfish, and SEAL, because no algorithm tested had its ME stall because of full target memory queues or ME command queues.

One negative effect of network-processor-dependent memory optimizations is that they involve more signal synchronizations and cause extra pipeline abort cycles. However, for most algorithms, an ME is aborted less than 1 percent of the time, which is negligible compared to the speedups provided by these optimizations.

Multiple threads: Using parallelism

An obvious way to improve cryptographic applications on network processors is to use parallelism. Three types of parallelism are useful: flow-level, block-level, and intrablock. Flow-level parallelism is straightforward and applicable to every algorithm, but does not improve the throughput of a single flow. Block-level parallelism focuses on block ciphers only and works differently with encryption modes, such as Electronic Code Book (ECB) and Cipher Block Chaining (CBC). ECB allows the encryption of separate blocks in parallel but is susceptible to simple-substitution attacks and cut-and-paste forgery. Therefore, most applications use CBC mode, which XORs the output of each ciphertext with the next block of plaintext thus allowing less parallel operations.

Intrablock parallelism operates on a single block but only works with algorithms that have less-serialized operations.

Flow- and block-level parallelism. We experimented with flow- and block-level parallelism to see how well the overall throughput scaled using multiple threads and IXP2800 MEs. We implemented all block ciphers in CBC mode. When encrypted with CBC mode, block read/write operations are parallelizable, handled by single thread using I/O parallelization. Thus, we assign one hardware thread to one flow, which requires no thread communication.

Figure 6 presents the overall throughputs of

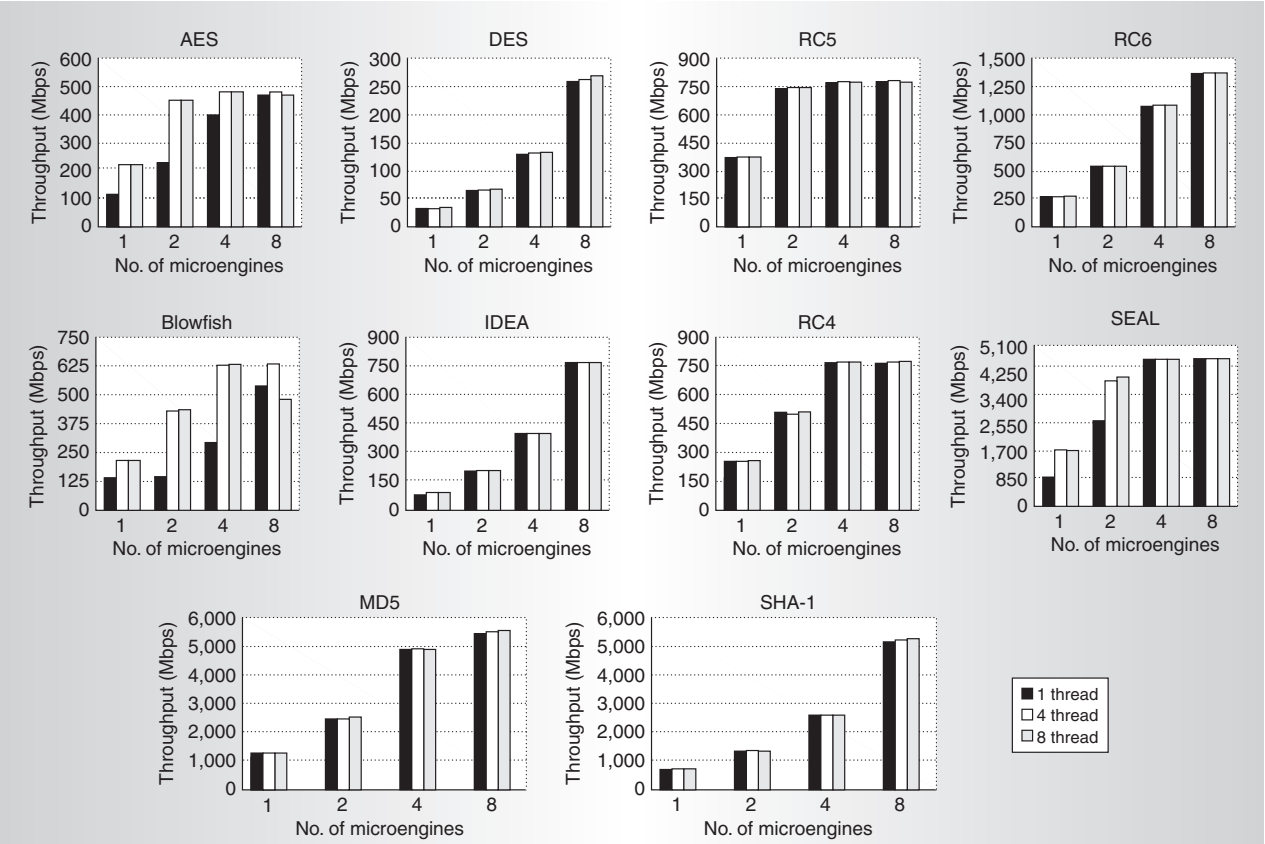

Figure 6. Throughput of selected algorithms with varying numbers of threads and MEs.

the selected algorithms with our multiME, multithread implementation. First, we observe the results of one and two MEs. Naturally, the throughput of two threads in two MEs is around double that of a single thread in one ME. However, the throughput of four or eight threads in a ME is not four or eight times that of one thread because they share the same computing power. For algorithms that already achieve a near 100 percent ME utilization rate for a single-thread benchmark (via latency hiding), multiple threads cannot increase the overall throughput. In contrast, multithreading enhances the throughput of AES, Blowfish, and SEAL, algorithms that involve many memory references. But, increasing the number of threads from four to eight generates no extra throughput because MEs are already saturated. On an overall basis, the bottleneck is again an ME's computing power.

Another observation from the figure is that having many MEs (greater than four) does not always yield close to linear speedups, as does the move from one to two MEs. Only DES, IDEA, and SHA-1 maintain a near linear speedup with an eight ME configuration. Other algorithms improve very little in moving from four to eight MEs, even though the computing power is double. So apparently, the bottleneck is no longer the same.

Identifying bottlenecks. To identify the new bottleneck, we inspected internal statistics for eight ME configurations. We also measured performance statistics for command bus arbiters; Figure 7 shows those results. Eight MEs share two command bus arbiters, sending memory requests to relevant controllers, one for an SRAM request and one for scratch and RDRAM requests. Except for the three algorithms that improve with eight MEs, other algorithms—such as MD5 and RC6—suffer from low ME utilization rates with one thread per ME. This is because the memory access latency increases under heavy load, and the compute cycles cannot hide them anymore. Table 1 only

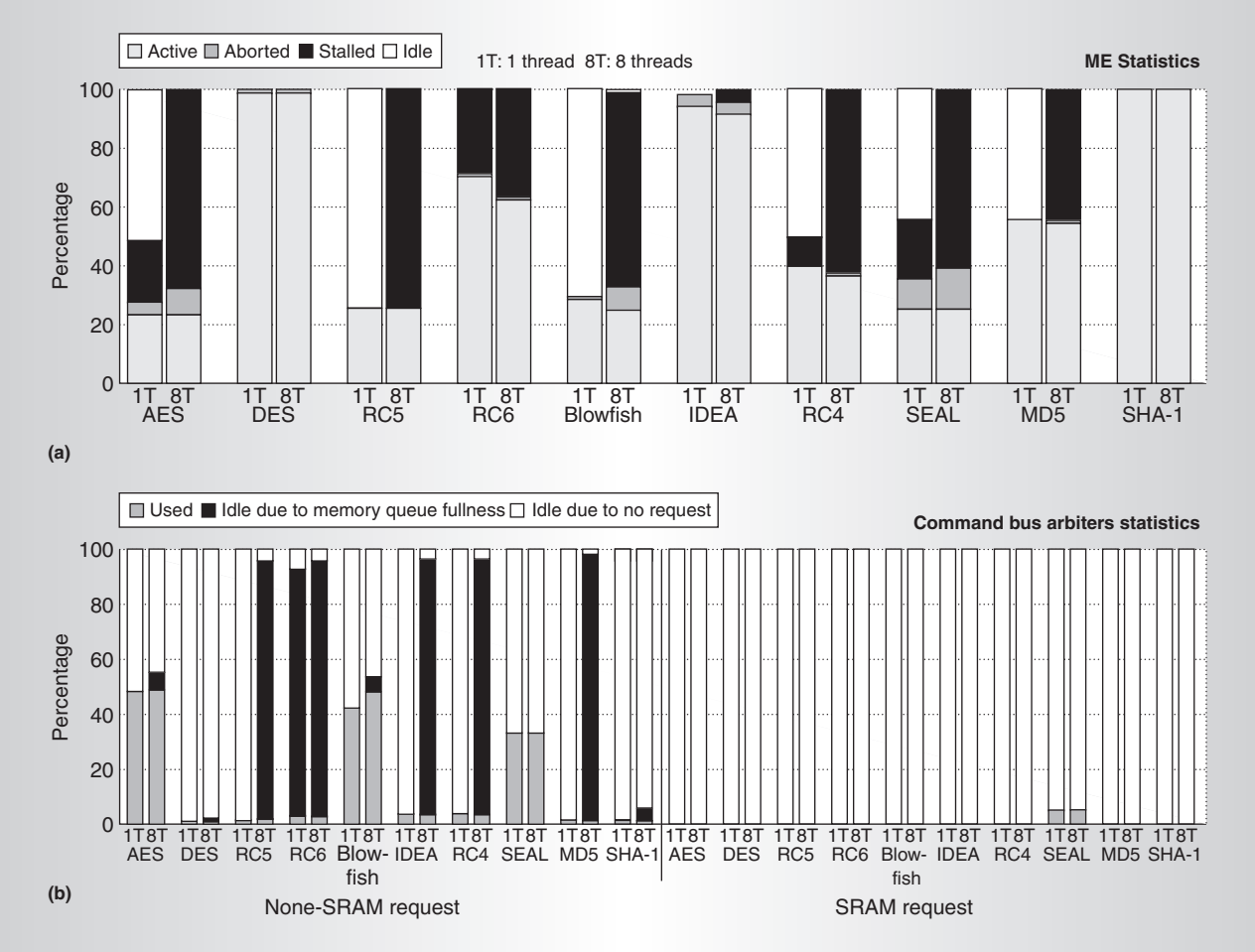

Figure 7. Internal statistics of ME and command bus arbiters with an eight ME configuration.

depicts the average latency under modest load. If the request is queued, its latency will be several times longer. We initially thought that multithreading might help to hide these extra latencies. Actually, our results show that the use of more threads does not convert the idle cycles to active cycles, but to stalled cycles. The bottleneck might differ for different algorithms:

- For AES, Blowfish, and SEAL, which have the most shared-memory requests among all the selected algorithms, the bottleneck is the shared bus that connects multiple MEs, because command bus arbiters only idle a small percentage (less than 6 percent) because of the fullness of memory queues.
- For RC4, RC5, RC6, and MD5, the bottleneck is RDRAM. Unlike the preceding three algorithms, the only off-ME memory access is the reading or writing data from RDRAM, which occurs twice at most when encrypting one block. In the results for these algorithms, command bus arbiters are also idle for most of the time, but because of a higher percentage (greater than 90 percent) of memory queue fullness.
- For DES, IDEA, and SHA-1, the bottleneck is once again the computing power, because these algorithms have an approximate 100 percent ME utilization.

Memory system optimizations. As we just discussed, for many algorithms, shared memory becomes the new system bottleneck when working in parallel with many MEs. To mitigate the contention, it is possible to optimize the memory system using the following hardware or software approaches:

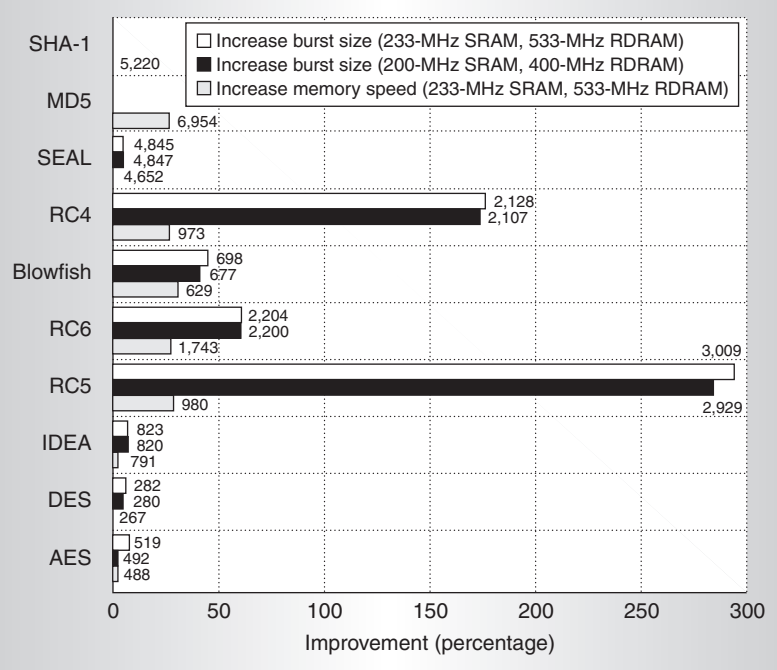

Figure 8. Throughput and improvement of memory system optimizations with eight-thread and eight-ME configuration.

Here, we demonstrate the effect when we enlarge the burst to the maximum size supported by IXP2800.

Figure 8 shows optimized throughput (marked in numbers) and improvements from applying the optimizations with eight thread and eight ME configurations. Increasing burst size is not possible on MD5 and SHA-1, because their block sizes are already at the maximum burst size. Algorithms that suffer from the RDRAM bottleneck benefited most. Although not shown here, we also test other possible burst sizes and find that the larger the burst size, the greater the improvement.

Additionally, increased burst size contributes more than increased memory bus speed. Raising the burst size successfully shifts RC4, RC5, and RC6 away from the RDRAM bottleneck, because memory bus speed cannot convey more significant improvements. For those whose bottleneck is shared-bus or ME computing power, memory optimizations afford relatively little improvement, usually less than 10 percent.

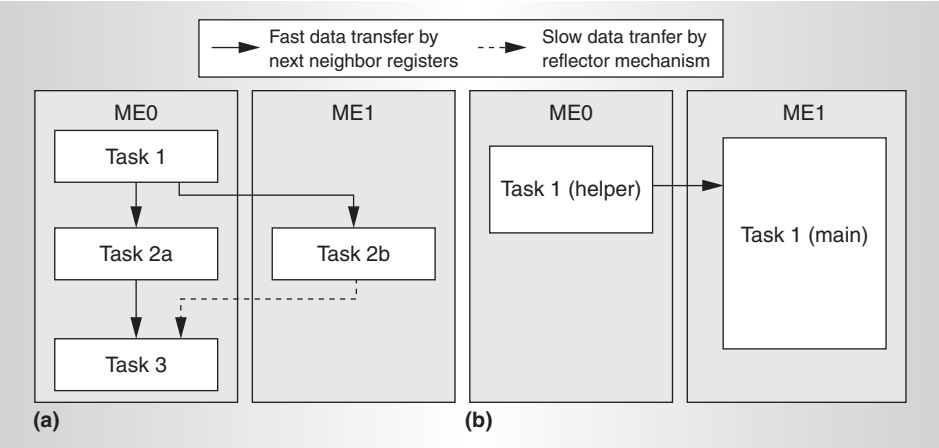

Figure 9. Execution modes for intrablock parallelism: Fork-join (a); Helper-ME (b).

- Increase the memory bus speed to reduce access latency. We investigated this approach by increasing IXP2800's SRAM to 233 MHz (a 16.5 percent increase) and RDRAM to 533 MHz (a 33.5 percent increase).
- Increase the memory read/write burst size to reduce memory traffic, which will also reduce the pressure on the shared bus. In previous tests, all memory references burst at the block size of their algorithms.

Intrablock parallelism. For some algorithms, several threads in different MEs can work together to enhance the performance of a single flow, in which the granularity of flow- and block-level parallelism is one thread within one ME. Besides, these optimizations not only increase computation resources but also increase high-speed storage, such as registers and local memory. Here, we do not use several threads in one ME because they share the same computing power. However, working with different MEs requires extra communication among MEs. In our implementations, we consider the use of dual MEs (two adjacent MEs) as the building block of intrablock parallelism. Because IXP2800 only organizes MEs in a pipeline, it supports fast communication between two adjacent MEs.

Depending on the algorithm, we use two execution modes of intrablock parallelism, illustrated in Figure 9:

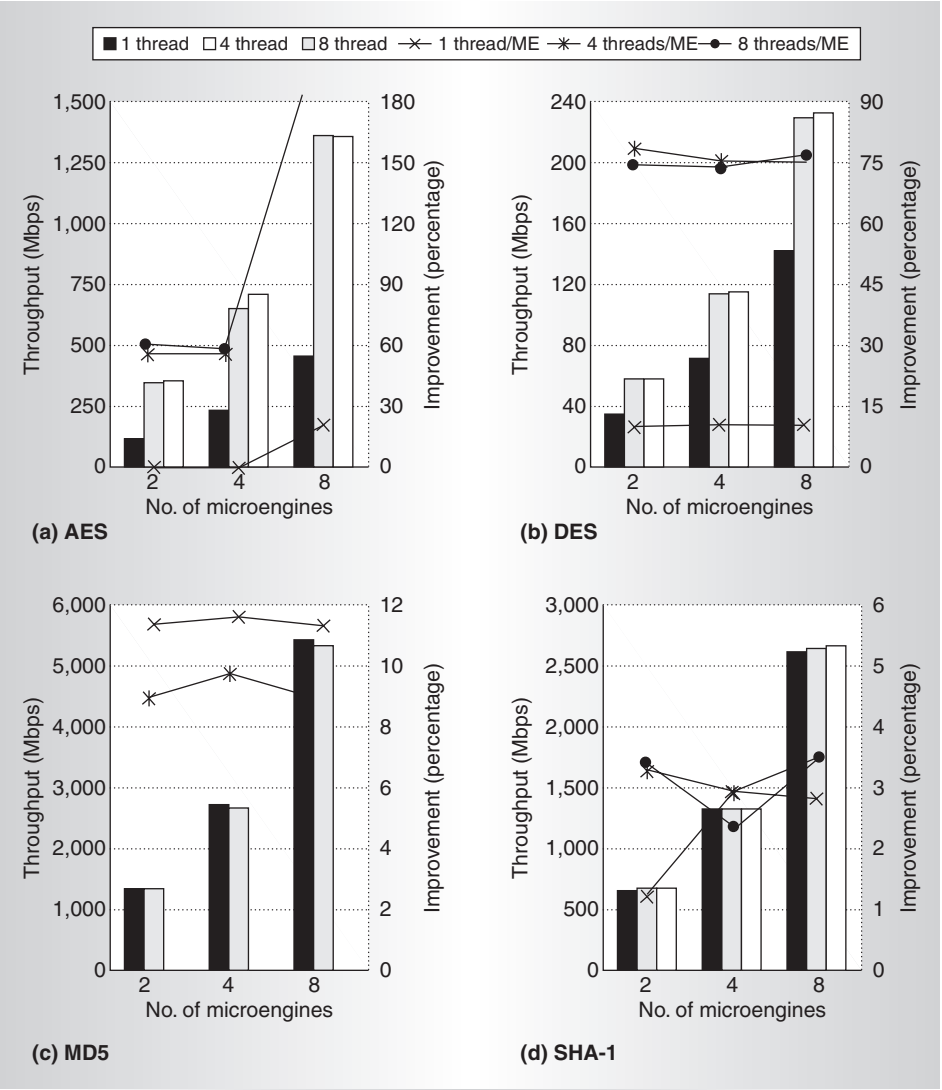

Figure 10. Throughput and per-flow improvement. We cannot configure MD5 in eight-thread mode because of a register shortage.

- *Fork-join.* Some tasks can be split in two parallel streams and deployed on two MEs. They then require bidirectional communications for synchronization. This *fork-join mode* works with some block ciphers.
- *Helper ME.* This mode creates two copies of the same task, one on each ME. One ME, the helper, runs the reduced task; the other (main ME) runs the full task. The helper ME goes ahead of the main ME and performs calculations with data loaded from shared memories, before the reference by the main ME. Many hash functions can execute in this mode.

IXP2800 hardware restricts fast data transfer by next-neighbor register in only one direction (indicated by the solid line in Figure 9). Communication in the reverse direction must use a slower mechanism, called the *reflector* (dashed line in Figure 9). We benchmarked AES and DES in fork-join mode, and MD5 and SHA-1 in helper-ME mode. Figure 10 presents their throughputs and per-flow improvements over flow- and block-level parallelism. Intrablock parallelism involves many inter-ME communication latencies, especially working in fork-join mode. But, we can hide these latencies using the hardware thread. Consequently, throughputs and per-flow improvements for

AES and DES with four- or eight-thread configurations are much higher than for single-thread configurations.

So although the computing power has doubled, some algorithms cannot entirely exploit the increase. MD5 and SHA-1 have the lowest improvements, because only a small percentage of their codes can run in the helper-ME mode. In contrast, AES obtains a 186 percent per-flow improvement with an eight-thread and eight-ME configuration. This is because we split its large tables into the local memories of different MEs. Splitting the tables reduces the shared-memory requests, which put great pressure on the command bus when the number of MEs is large. Besides, unlike other algorithms, AES obtains a much higher overall throughput with intrablock parallelism than with other forms of parallelism.

As a whole, our benchmarks show that hash functions can achieve relatively high throughputs exceeding 5 Gbps. Stream and block ciphers run slower, having a throughput of 1.5 Gbps on average. Nevertheless, all the stream ciphers and some block ciphers in our test still attain over 2-Gbps throughputs. Because shared resources such as memory and bus are a major factor that affects the overall performance, we doubt whether employing a shared, cryptographic-application-specific chip or coprocessor in a network processor is the only way to meet high-throughput demands. However, several hardware improvements to current network processors can help software implementations on data path PEs:

- Increase the cache size on PEs to hold large tables and lessen the pressure on the shared memory and bus.
- Organize PEs in smaller clusters and enhance the shared-bus performance.
- Enlarge the size of the memory and command queues to reduce the possibility of PE stalls.
- Improve communications among different PEs to help intrablock parallelism.
- Distribute support for cryptographic-specific hardware in each PE or share this support among a few PEs.
- Adopt a new memory system to shorten the access latency.

We believe that in combination, the proposed implementation and optimization principles can go a long way toward improving cryptographic processing performance on network processors.

Acknowledgment

This research was supported by Intel IXA University Research Plan (No. 9077), the Natural Science Foundation of China (No. 90104002, 60173012, 60273009, and 60372019), NSFC and RGC (No. 60218003), the Projects of Development Plan of the State Key Fundamental Research (No. G1999032707 and 2003CB314804), and China Postdoctoral Science Foundation (No. 2003034152). Bo Li's research was supported in part by a NSFC/RGC joint grant under the contract N_HKUST605/02; grants from RGC under the contracts HKUST6402/03E and HKUST6104/04E; and a grant from Microsoft Research under the contract MCCL02/03.EG01. This work was performed while Bo Li was a visiting scientist in Microsoft Research Asia, Beijing.

References

1. M. Merkow and J. Breithaupt, *The Complete Guide to Internet Security*, Amacom, 2000.
2. H. Xie, L. Zhou, and L. Bhuyan, "Architectural Analysis of Cryptographic Applications for Network Processors," http://www.cs.ucr.edu/~bhuyan/papers/np1.pdf.
3. P. Dongara and T.N. Vijaykumar, "Accelerating Private-Key Cryptography Via Multithreading on Symmetric Multiprocessors," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 03), IEEE Press, 2003, pp. 58-69.
4. *Data Encryption Standard (DES)*, Nat'l Institute of Standards and Tech., Federal Information Processing Standards Publication 46-3, 25 Oct. 1999; http://csrc.nist.gov/cryptval/des.htm.
5. *Specification for the Advanced Encryption Standard (AES)*, Nat'l Institute of Standards and Tech., Federal Information Processing Standards Publication 197, 26 Nov. 2001; http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
6. X. Lai, *On the Design and Security of Block Ciphers*, Hartung-Gorre Veerlag, 1992.
7. R.L. Rivest, "The RC5 Encryption Algo-

rithm," *Proc. 2nd Int'l Workshop on Fast Software Encryption,* Springer-Verlag, 1995, pp. 86-96.
8. J. Nechvatal et al., "Report on the Development of the Advanced Encryption Standard," Nat'l Institute of Standards and Tech., 2 Oct. 2000; http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf.
9. B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher," *Proc. Cambridge Security Workshop*, Springer-Verlag, 1994, pp. 191-204.
10. B. Schneier, *Applied Cryptography*, 2nd ed., John Wiley & Sons, 1996.
11. P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," *Proc. the Cambridge Security Workshop,* Springer-Verlag, 1994, pp. 56-63.
12. R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, Apr. 1992; http://www.faqs.org/rfcs/rfc1321.html.
13. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
14. T. Wolf and M. Franklin, "CommBench: A Telecommunication Benchmark for Network Processors," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 00), IEEE Press, 2000, pp. 154-162.
15. B.K. Lee and L.K. John, "NpBench: A Benchmark Suite for Control Plane and Data Plane Applications for Network Processors," *Proc. IEEE Int'l Conf. Computer Design* (ICCD 03), 2003, pp. 226-233.

Zhangxi Tan is an ME student in the Department of Computer Science at Tsinghua University, China. His research interests include computer architecture, network processors, performance evaluation, and resource management in computer networks. Tan has a BE in electronic engineering from Tsinghua University. He received the 2002 Outstanding Graduate Student Award from both Tsinghua University and the City of Beijing, and the 2004 Outstanding Chinese Student Scholarship from IBM.

Chuang Lin is a professor and the head of the Department of Computer Science at Tsinghua University. His research interests include computer networks, performance evaluation, network security, and Petri net theory and applications. Lin has a PhD in computer science from Tsinghua University. He serves as the general chair for ACM SIGCOMM's 2005 Asia workshop and the associate editor of the *IEEE Transactions on Vehicular Technology*. He is a senior member of the IEEE.

Hao Yin is an assistant professor with Department of Computer Science, Tsinghua University. His research interests include multimedia communication, video coding, and network security. Yin has a BS, an ME, and a PhD, all in electrical engineering, from the Huazhong University of Science and Technology, China. He is a member of IEEE.

Bo Li is an associate professor with the Department of Computer Science, Hong Kong University of Science and Technology; he is also an adjunct researcher for Microsoft Research. Li has a PhD in electrical and computer engineering from the University of Massachusetts at Amherst. He was cochair of IEEE Infocom 2004's technical program committee.

Direct questions and comments to Zhangxi Tan, Department of Computer Science and Technology, Tsinghua University,100084 Beijing China; xtan@csnet1.cs.tsinghua.edu.cn.