Accelerating Private-Key Cryptography via Multithreading on Symmetric Multiprocessors

Praveen Dongara[1] and T. N. Vijaykumar

Performance Analysis and Architecture, Computation Products Group
Advanced Micro Devices
praveen.dongara@amd.com

School of Electrical and Computer Engineering
Purdue University
vijay@ecn.purdue.edu

Abstract

Achieving high performance in cryptographic processing is important due to the increasing connectivity among today's computers. Despite steady improvements in microprocessor and system performance, private-key cipher implementations continue to be slow. Irrespective of the cipher used, the main reason for the low performance is lack of parallelism, which fundamentally comes from encryption modes such as the Cipher Block Chaining (CBC) mode. In CBC, each plaintext block is XOR'ed with the previous ciphertext block and then encrypted, essentially inducing a tight recurrence through the ciphertext blocks. To deliver high performance while maintaining high level of security assurance in real systems, the cryptography community has proposed Interleaved Cipher Block Chaining (ICBC) mode.

In four-way interleaved chaining, the first, fifth, and every fourth block thereafter are encrypted in CBC mode; the second, sixth, and every fourth block thereafter are encrypted as another stream, and so on. Thus, interleaved chaining loosens the recurrence imposed by CBC, enabling the multiple encryption streams to be overlapped. The number of interleaved chains can be chosen to balance performance and adequate chaining to get good data diffusion. While ICBC was originally proposed to improve hardware encryption rates by employing multiple encryption chips in parallel, this is the first paper to evaluate ICBC via multithreading commonly-used ciphers on a symmetric multiprocessor (SMP). ICBC allows exploiting the full processing power of SMPs, which spend many cycles in cryptographic processing as medium-scale servers today, and will do so as chip-multiprocessor clients in the future. Using the Wisconsin Wind Tunnel II, we show that our multithreaded ciphers achieve encryption rates of 92 Mbytes/s on a 16-processor SMP at 1 GHz, reaching a factor of almost 10 improvement over a uniprocessor, which achieves 9 Mbytes/s.

1 Introduction

Information security is an important concern due to the increasing connectivity among today's computers. Cryptographic algorithms (ciphers) form the underpinnings of providing security assurances for both information communicated via a public medium such as the Internet, and data stored in modern multi-user computers. With the advent of Secure IP [2] and virtual private networks [10], cryptographic processing will become more important, even beyond its pervasive role in the world of e-commerce. Hence, achieving high performance in cryptographic processing is increasingly important.

Cryptographic processing involves applying a key to transform regular data (or plaintext) into encrypted data (or ciphertext), communicating the ciphertext, and then decrypting the ciphertext back to plaintext. Two categories of cryptographic algorithms are private-key and public-key ciphers. In private-key ciphers, both the encrypting and decrypting ends use the same key. Private-key ciphers require a means of securely sharing the common private key between the encrypting and decrypting ends. Unfortunately, secure exchange of the private key requires encrypting the key itself using another private key! Public-key ciphers avoid this problem by encrypting using an openly-published public key, and decrypting using a private key known only to the decrypting end.

Because public-key ciphers usually involve calculations using 1024-bit precision numbers, they are computationally more expensive, by as much as a factor of 1000, than equivalent private-key ciphers. Consequently, typical cryptographic systems, such as the Secure Sockets Layer (SSL), use a public key to exchange the private key among the communicating parties, and then use the private key for session data. Except for short sessions where public key processing dominates, private key processing speed is key to achieving fast response times, especially for typical session lengths of 20KB-32KB [1].

Despite steady improvements in microprocessor and system performance, private-key cipher implementations continue to be slow. For instance, encrypting 50 bytes of data takes more than 1000 processor cycles on the Alpha 21264 for many of the commonly-used ciphers. While private-key ciphers are computationally intensive (although not as much as their public-key counterparts), the main rea-

1 This work was started when Praveen Dongara was at the School of Electrical and Computer Engineering, Purdue University.

son for the low performance is lack of parallelism and not computational intensity [5]. Modern computers exploit parallelism to achieve performance, and the lack of parallelism severely obstructs achieving fast response times.

Irrespective of the cipher used, the lack of parallelism fundamentally comes from encryption modes such as the Cipher Block Chaining (CBC) mode. In CBC each plaintext block is XOR'ed with the previous ciphertext block and then encrypted, essentially inducing a tight recurrence through the ciphertext blocks. Other modes such as the Propagating Cipher Block Chaining mode, Cipher Block Chaining with Checksum mode, and Cipher FeedBack mode also have the recurrence property. Because such recurrence thoroughly randomizes (diffuses) the ciphertext while concealing any patterns present in the plaintext, CBC is approved by NIST [29] and widely used in SSL ciphers IDEA, RC2, DES and 3DES, and in authentication systems such as Kerberos 5. Electronic CodeBook and Plaintext Block Chaining modes do not have this recurrence property, and are not used as widely.

To deliver high performance while maintaining high level of security assurance in real systems, the cryptography community has proposed Interleaved Cipher Block Chaining (ICBC) mode [23,], which is approved by the recently-announced FIPS 140-2 [31] for 3DES [30,14], and deployed in real-world systems [13]. This mode creates multiple interleaved encryption streams instead of just one. In four-way interleaved chaining, the first, fifth, and every fourth block thereafter are encrypted in CBC mode; the second, sixth, and every fourth block thereafter are encrypted as another stream, and so on. Thus, interleaved chaining loosens CBC's recurrence, enabling the multiple encryption streams to be overlapped.

For typical input sizes, ICBC diffuses the ciphertext quite well. First, ICBC does not change the algorithm, whose mathematical properties are what provide the key security guarantees. Second, applying ICBC to small inputs results in chaining of only a few blocks. For instance, a 1KB plaintext with 8-way interleaving results in chaining of only 128-bytes per chain. While such small inputs will need to use CBC, such small inputs do not gain much performance by using ICBC anyway. Also, the processing time may be dominated by the session's public key processing, and not private key processing, for such input sizes [5]. Third, for large inputs, ICBC diffuses data well. For instance, for the median web object of about 21KB [1], 8-way interleaving allows chaining of more than 2.5KB per chain. Fourth, web objects are increasing and not decreasing in size, so ICBC will be more effective in the future. Fifth, ICBC affords a balance between performance and cryptographic security by allowing a number of interleaved chains which gives good diffusion with reasonable speedups. Real systems have always had to maintain this balance: even though public keys are more secure, private keys are more commonly used due to their performance advantages.

While ICBC was originally proposed to improve hardware encryption rates by employing multiple encryption chips in parallel as done for IDEA [13], this is the first paper to evaluate ICBC via multithreading commonly-used ciphers on a symmetric multiprocessor (SMP). Using ICBC, ciphers need not be interleaved into as many threads as there are processors in the SMP, if the data per chain seems insufficient. For instance, a 16-processor SMP can use 8-way interleaving if the input is too small to be 16-way interleaved.

There are other advantages to using SMPs: First, SMPs spend many cycles in cryptographic processing, deployed as medium-scale servers (e.g., the Sun Enterprise 6000 is a popular choice for web servers). Multithreading ciphers using ICBC allows exploiting the full processing power of the machines. It is true that improving throughput is important for such servers, and ICBC reduces latency. However, when the server is not saturated and not all the processors are busy with messages, ICBC reduces per-message latency by multithreading a single message on the available processors. Second, many clients in the future will use microprocessors with support for multithreaded execution, such as Simultaneous Multithreading [27] or Chip Multiprocessors [19]. Therefore, not only servers but also clients will have the ability to exploit multithreaded ciphers. Third, our approach is low cost because it requires little beyond standard multiprocessor/multithreaded hardware. Only the software implementation of the ciphers (perhaps in the SSL layer of the OS networking code) changes to being multithreaded.

The main contributions of the paper are:

- Multithreaded ciphers using ICBC are a good match for SMPs because there is no data sharing and no communication among the threads. The threads are naturally load balanced and computationally intensive, spending hundreds of cycles per cache line of input data brought from memory.
- Our multithreaded implementation achieves encryption rates of 92 Mbytes/s on a 16-processor SMP at 1 GHz, reaching a factor of almost 10 improvement over a uniprocessor, which achieves 9 Mbytes/s.
- The number of compute cycles per cache line of input is large enough both to hide prefetch of the next cache line completely, and to allow the bus to service prefetch requests from up to 16 processors.
- Serial initialization code, software barrier costs, and bus occupancy prevent multithreaded ciphers from achieving perfect speedups on SMPs.

The rest of the paper is organized as follows. We describe how we multithread the ciphers in Section 2, and discuss the key implementation issues in Section 3. In Section 4, we present our results, and in Section 5 we discuss related work. Finally, we conclude in Section 6.

2 Multithreading Private-Key Ciphers

Ciphers are designed to encrypt plaintext into ciphertext and decrypt ciphertext back to plaintext. While the specifics of the computation done by the cipher depends on the algorithm used, there are several commonalities among ciphers. Most recent ciphers operate on blocks of data instead of individual bytes, typically using a block size of 128 bits (16 bytes). We refer to this block used by the algorithms as Ablock. If the input to be encrypted is not an exact multiple of 128 bits, it is made so by padding.

Another commonality among ciphers is the encryption mode. While the algorithm defines how an Ablock of plaintext is transformed into an Ablock of ciphertext, the mode defines how the preceding ciphertext or plaintext Ablock affects the encryption of the next plaintext Ablock. The idea behind modes is to increase the diffusion of the ciphertext and conceal any patterns in the plaintext. Electronic Codebook (ECB) mode encrypts a plaintext Ablock without combining with any previous ciphertext Ablock, and hence achieves less diffusion. Cipher Block Chaining (CBC) mode first XORs the plaintext Ablock with the previous ciphertext Ablock, and then encrypts the XOR result. Figure 1(a) illustrates CBC. The Cipher Feedback mode (CFB) first uses the previous ciphertext Ablock to generate some pseudorandom numbers which are combined with the next plaintext Ablock, and then encrypts the combined result. CBC, CFB, and other chaining modes use a random initialization vector (IV) to combine with the first Ablock of the input.

Because such chaining of the plaintext Ablocks with previous ciphertext Ablocks distributes the resulting ciphertext bit-values evenly among all possible combinations, the chaining modes prevent codebook attacks. Consequently, CBC is widely used and non-chaining modes like ECB are not. Unfortunately, CBC results in low performance due to lack of parallelism. CBC creates a tight recurrence where the encryption of the next Ablock depends on the result of encrypting the previous Ablock, eliminating any possibility of overlapping the encryption of multiple Ablocks.

A relatively simple approach to break the recurrence induced by CBC is to interleave multiple encryption streams. Instead of chaining all the input Ablocks through a single CBC chain, interleaved cipher block chaining (ICBC) uses multiple chains, interleaving the input Ablocks. For instance, in a four-way interleaved ICBC, the first, fifth and every fourth block thereafter are encrypted in CBC mode with one IV. The second, sixth, and every fourth block thereafter are encrypted with another IV, and so on, as illustrated in Figure 1(b). Thus, four-way-interleaved ICBC can be thought of as encrypting four different inputs with the

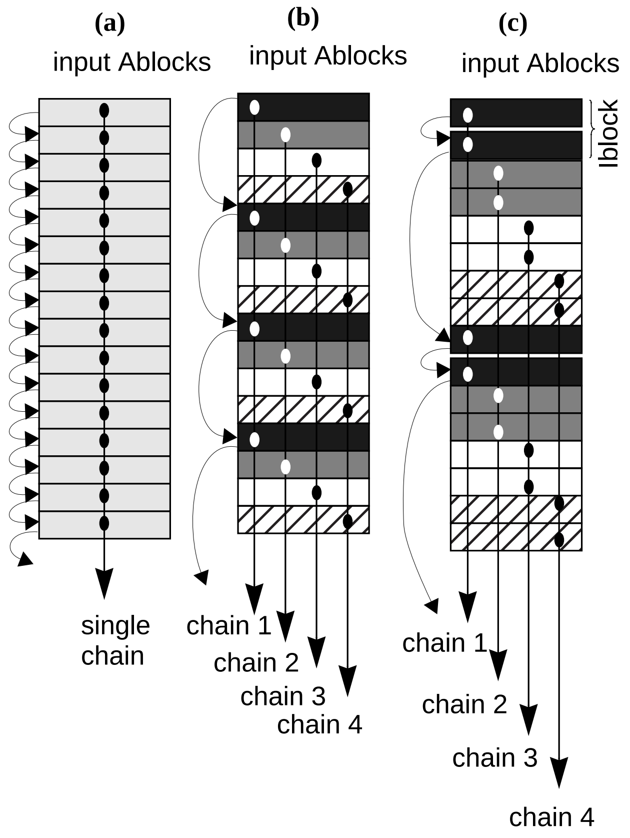

FIGURE 1. (a) Cipher block chaining, (b) interleaved cipher block chaining using Ablocks, and (c) interleaved cipher block chaining using Iblocks.

same key and four different IVs.

Fortunately, ICBC uses the same key as CBC, so there are no additional overheads in public-key processing (i.e., public-key encryption of the private key at the time of the session setup) due to ICBC. However, ICBC's total IV is longer than CBC's IV. Because the overhead of CBC's IV generation compared to the encryption computation is small to begin with, ICBC's longer IV does not impact execution time.

CBC induces a recurrence only for encryption, and not for decryption. Decryption using CBC is parallelizable because all of the ciphertext Ablocks can be decrypted in parallel and then XORed with the previous cipher text to obtain plaintext back. However, if encryption uses ICBC then decryption also needs to use ICBC. ICBC's decryption can be multithreaded exactly like ICBC's encryption, with identical parallelism.

3 Multithreading Issues

Our multithreaded code employs a straightforward strategy for parallelization using the standard shared-memory application programming interface to multithread encryption. One processor executes the serial part of the initialization code and initializes like 3DES's SBOX, PBOX, and other arrays. Parts of the initialization code can also be multithreaded. Then the code simply forks as many threads as there are processors. Each thread encrypts its portion of the input using CBC within its portion. As such, there is no data sharing among the threads, implying that multithreaded ciphers are naturally amenable to high performance through parallelization.

From a performance standpoint, there are many issues with multithreaded ciphers, such as false sharing, global synchronization, data locality, and serialized initialization, much like any multithreaded application.

3.1 True and False sharing

Although there is no true sharing of data among the threads, false sharing due to cache line granularity needs to be avoided. The way ICBC is defined, one encryption thread encrypts an Ablock and writes its result at location A, and another thread encrypts the next Ablock and writes its result at location A+Ablocksize (Ablocksize is 128 bits usually). If locations A and A+Ablocksize fall within the same cache line, then the two threads incur false sharing of that cache line upon the writing of the encrypted Ablocks.

We avoid false sharing by assigning a set of contiguous input Ablocks, called an *Iblock*, to one processor and the next Iblock to the next processor. Iblocks need to be as large or larger than cache lines to be effective, but should not be so large as to reduce the number of interleaved chains. When the processors write the encrypted Ablocks they are guaranteed to write to different cache lines, without incurring any false sharing. For instance, for a system using 32-byte cache lines, Iblock sizes of 32 or 64 bytes would work well. With such input data assignment, a processor can chain the Ablocks within every Iblock assigned to it, and then interleave-chain with the Ablocks in the next Iblock assigned to it, as illustrated in Figure 1(c). This chaining within each Iblock ensures that the overall chaining of ICBC using Iblocks is no less than that of the original ICBC using Ablocks. Thus, ICBC using Iblocks differs from the original ICBC in only the grouping of Ablocks into each chain.

3.2 Data Locality

Because each input Ablock is accessed only once, encrypted, and written to output, cipher codes do not exhibit much temporal locality in the input data. This access pattern implies that caches are not effective for input data, but internal data structures such as the SBOX and PBOX are accessed repeatedly and benefit from caching. However, cipher codes do exhibit spatial locality in the input data. That is, on an input Ablock access, the entire cache line containing the Ablock is brought in, so that subsequent input

```
fork(); /* spawn n threads */
initialize();
/* multithread init if needed */
global_barrier();

/* Pick each thread's Iblock */
Iblock = &input_start+Iblock_size*threadID;
Iblocks_per_thread = Input_size/
                 (Num_threads*Iblock_size);
for (i = 0; i < Iblocks_per_thread; i++) {
   /* prefetch next Iblock */
   Prefetch(Iblock+Iblock_size*Num_threads);
   /* first Ablock within Iblock */
   Ablock = Iblock;
   for (j = 0; j < Ablocks_per_Iblock; j++) {
      /* encrypt current Ablock */
      Encrypt(Current_Ablock);
      (Num_threads*Iblock_size);
      /* jump to next Ablock within Iblock*/
      Ablock += Ablock_size;
   }
   /* jump to next Iblock*/
   Iblock += Iblock_size*Num_threads;
}
```

FIGURE 2. High-level multithreaded cipher pseudocode.

Ablocks within the cache line are accessed fast from the cache.

We address the lack of temporal locality by prefetching the next Iblock. Because cipher codes predictably march down the input data, determining the address of the next Iblock to perform the prefetch is trivial. While a thread is encrypting one Iblock, a prefetch is issued for the next Iblock and the latency of bringing the next Iblock into the cache is hidden under the current Iblock's encryption. If memory latency or contention is more, then prefetch would have to be initiated earlier (e.g., two Iblocks ahead). This prefetching is accurate in that every block prefetched is actually accessed, and does not generate any unnecessary memory traffic.

3.3 Initialization Serialization

Initialization of data structures is typically sequential in multithreaded applications. In the case of some of the ciphers such as Rijndael, Twofish, and IDEA, initialization is long and contributes significantly to the execution time. Fortunately, parts of the initialization is amenable to multithreading. In multithreading the initialization part, there is a trade off between performance gain due to parallelization and overhead introduced due to the extra barriers which we might have to use. We do semi-parallelization of the initialization code of Rijndael, Twofish, and IDEA. Unfortunately, not all of the initialization is parallelizable. Blowfish has a high fraction of initialization code, which uses CBC chaining to initialize later array elements by chaining earlier array elements. Consequently, Blowfish's initialization cannot be parallelized. Note that the non-parallel nature of the initializations are inherent to the algorithm and are not an artifact of the implementation.

3.4 Barrier Synchronization

While there is no synchronization among the encryption threads, there is a global synchronization (i.e., a barrier) at the end of the initialization, so that the encryption threads do not start before the initialization is complete. In the cases where parts of the initialization is parallelized, more barriers are inserted to ensure dependent loops of the initialization do not start before previous loops finish. Because barriers are global synchronization among all the processors in the system, barriers usually cost many processor cycles (e.g., hundreds to thousands of cycles). For small input sizes, the cost of the barrier may not be amortized over the encryption of the input, resulting in the barrier accounting for a large fraction of the overall encryption time. We employ efficient barrier implementations in software using MCS locks [16] to reduce the cost of barriers. We also show improvements achieved by hardware barrier implementations.

3.5 High-level pseudocode

Figure 2 gives the high level pseudocode of the multithreaded ciphers. After forking the threads, each thread works on its share of the cipher initialization. If the initialization is not parallel, a single thread does the initialization. The global barrier ensures that the initialization is complete before any of the threads embark on encryption. As part of encryption each thread is assigned a certain number of Iblocks, which is calculated as the input size divided by the number of threads and Iblock size. Each Iblock consists of multiple Ablocks. Each thread encrypts Ablock size of data at a time. After working on all the Ablocks in an Iblock, the thread moves on to the next Iblock.

4 Results

Our experimental results are organized as follows. First, we show that our multithreaded approach achieves good speedups over a uniprocessor. We analyze the speedups in light of the maximum achievable speedups taking into consideration the fraction of the multithreaded code that is sequential (for initialization) and the barrier overheads. Then, we show that the number of compute cycles to encrypt each cache line of input data is significantly larger than typical memory latencies, allowing us to prefetch the next cache line while the processor is busy encrypting the current cache line. This timely prefetching hides the memory latency of the input data almost entirely. We further show that the number of compute cycles per cache line is large enough for the bus to service prefetch requests from even 16 processors in the time a processor encrypts one cache line. This measurement implies that even in a 16-processor SMP, there is not any significant queuing at the bus for input data prefetches. Finally, we show that we achieve encryption rates of 92 MB/s using 16 processors and 32 MB/s using 4 processors, compared to 9 MB/s achieved by a uniprocessor, assuming clock speed of 1GHz.

CPU clock	1 GHz
L1 cache	32 Kbytes, 32-byte lines
Memory	100 cycles, n-way interleaved for a n-CPU SMP
Processor to Bus cycle ratio	5
Bus cycles per transaction	4
Memory bus	32-byte, split transaction, pipelined

TABLE 1. System configuration parameters.

4.1 Methodology

To simulate the multithreaded ciphers on symmetric multiprocessors (SMPs), we use the Wisconsin Wind Tunnel II (WWT2) [18]. We configure the WWT2 simulator to simulate an SMP with the parameters shown in Table 1. Because the ciphers do not exhibit temporal locality, as explained in Section 3.2, having bigger caches or a deeper hierarchy (two or three levels of caches) do not improve performance. A 32-KB L1 cache is sufficient to hold all the data except for the plaintext data. Plaintext data, however, does not have any temporal locality and hence does not benefit from a larger cache. We picked a typical cache block size of 32 bytes, which takes advantage of the spatial locality present in the ciphers. Our simulator does model a split transaction bus and cache-block interleaved memory, including contention at the bus and memory.

Our simulator models an in-order-issue processor, which is representative of architectures such as the SUN UltraSPARC and network processors such as the Intel IXP1200. Our results are applicable to SMPs using UltraSPARC processors, and the IXP1200 which has multiple in-order cores on a single chip.

We chose eight cipher algorithms for our analysis. The eight cipher algorithms are *3DES* [9], *RC6* [21], *IDEA [*15], *RC4* [22], *Mars* [6], *Rijndael* 8], *Blowfish* [7], and *Twofish* [3]. All the eight cipher algorithms use at least 128 bits of key data, and each of these is considered strong having undergone extensive review and aggressive cryptanalysis. 3DES, Blowfish, IDEA, and RC4 are used in popular software packages. Mars, RC6, Rijndael, and Twofish were second round candidates for the Advanced Encryption Standard (AES) [28]. Rijndael has been selected as the new US encryption standard (to replace DES) by the National Institute of Standards and Technologies which coordinated a multi-year AES competition. We put extra effort in analyzing the Rijndael cipher and we obtain one of the best encryption rates for this cipher. We use the most efficient algorithms available as our sequential implementation of the ciphers.

To make our multithreaded implementation of the ciphers most efficient, we consider parallelization of the initialization part of the cipher wherever possible, as mentioned in Section 3.3. We semi-parallelize the initialization code of *Rijndael, Twofish,* and *IDEA*. These ciphers have high fractions of serial initialization. Blowfish has the highest fraction of initialization serial code among the eight ciphers. Unfortunately, Blowfish's initialization uses CBC chaining of the initial array and thread level parallelism could not be extracted. For all the ciphers, we use an Iblock size equal to the cache line size of 32 bytes. RC4 is a stream-based cipher which doesn't use chaining. We multithreaded RC4 by using different key-based random-number generators on separate threads which encrypt separate streams of data.

4.2 Multithreading speedups on SMPs

In Figure 3, we show the speedups by our multithreaded implementation of the eight cipher algorithms, varying the number of processors as 2, 3, 8, and 16. The Y-axis shows the execution speedups achieved over a uniprocessor running sequential code (without parallelization overheads) and the X axis shows the input sizes. In this experiment, *both* sequential and parallel codes use prefetching for Iblocks. On an average over all the eight ciphers, the uniprocessor encrypts at a rate of 8.79 Mbytes/s assuming a 1GHz clock speed. For the 128-byte input size, we show only 2- and 4-processor configurations because the Iblock size is 32 bytes and there can only be at most 4 threads for this input size.

As is evident from the plots, we get good speedups (hence encryption rates) by multithreading the ciphers. For example, on a 16-processor SMP, we get a speedup of about 10.64, on average, for 16 Kbytes input and a speedup of about 12.97 for 64 Kbytes input. Assuming a 1 GHz clock speed, these speedups correspond to encryption rates of 91.6 Mbytes/s and 113.3 Mbytes/s for 16 and 64 Kbytes, respectively. We get the best speedups for IDEA. The 16-processor SMP achieves a speedup of 14.5 with 16 Kbytes input and a speedup of 15.5 with 64 Kbytes input. We get the worst speedups for Twofish. The 16-processor SMP, achieves a speedup of 7.5 with 16 Kbytes input and a speedup of 11 with 64 Kbytes input.

To understand the variation in speedups across the ciphers, we analyze the maximum achievable speedup for each cipher. The maximum speedup that could be achieved on a n-processor multiprocessor can be expressed as follows:

$$MaxSpeedup = \frac{Time_{uniprocessor}}{Time_{multiprocessor}}$$

$$MaxSpeedup = \frac{T}{\frac{(1-f)T}{n} + fT + o_n} = \frac{1}{\frac{(1-f)}{n} + f + \frac{o_n}{T}}$$

(EQ 1)

where f is the serial portion of the code (mainly serial initialization) expressed as a fraction of uniprocessor execution time, o_n is the barrier synchronization overhead for n processors, and T is the time to execute on a uniprocessor. The equation implies that the maximum speedup that could be achieved is limited by the serial portion of the code and the barrier overhead, which is consistent with Amdahl's law. Increase in the fraction of serial portion decreases the speedups. In general, the fraction of serial initialization code decreases as the input size increases and the cost of (software) global barriers increase with increase in number of processors in the multiprocessor system.

Table 2 shows the fraction of serial fraction, barrier overhead, and the maximum achievable speedups in all the eight ciphers. It also lists the actual speedups obtained through simulation. These values are for 16 Kbytes input for 4- and 16-processor configurations. We use Table 2 and EQ 1 to explain the graphs in Figure 3.

A basic observation from the graphs in Figure 3 is that speedups increase with increase in the number of processors. This trend is consistent with EQ 1 as long as f + o_n/T, called the *non-parallel portion*, does not increase rapidly with n. For example, for a 16 Kbytes input, the non-parallel portion is 1.34 and 1.91 for 4 and 16 processors, on average. Consequently, these the average speedups improve from 3.57 to 10.64 on going from 4 to16 processors.

On an overall basis, we see that 3DES, RC6, IDEA and RC4 get better speedups than MARS, Rijndael, Blowfish and twofish. This disparity is due to the difference in the non-parallel portion between these two sets of ciphers. For example, for 16-processor, 16K input configurations, the average non-parallel portion for MARS, Rijndael, Blowfish and Twofish at 3.35 is higher than the non-parallel portion for 3DES, RC6, IDEA and RC4 at 1.09. This difference results in average speedup of 13 for 3DES, RC6, IDEA and RC4 on a 16-processor SMP, and 9 for MARS, Rijndael, Blowfish and Twofish.

From Figure 3, we see that the speedups increase with

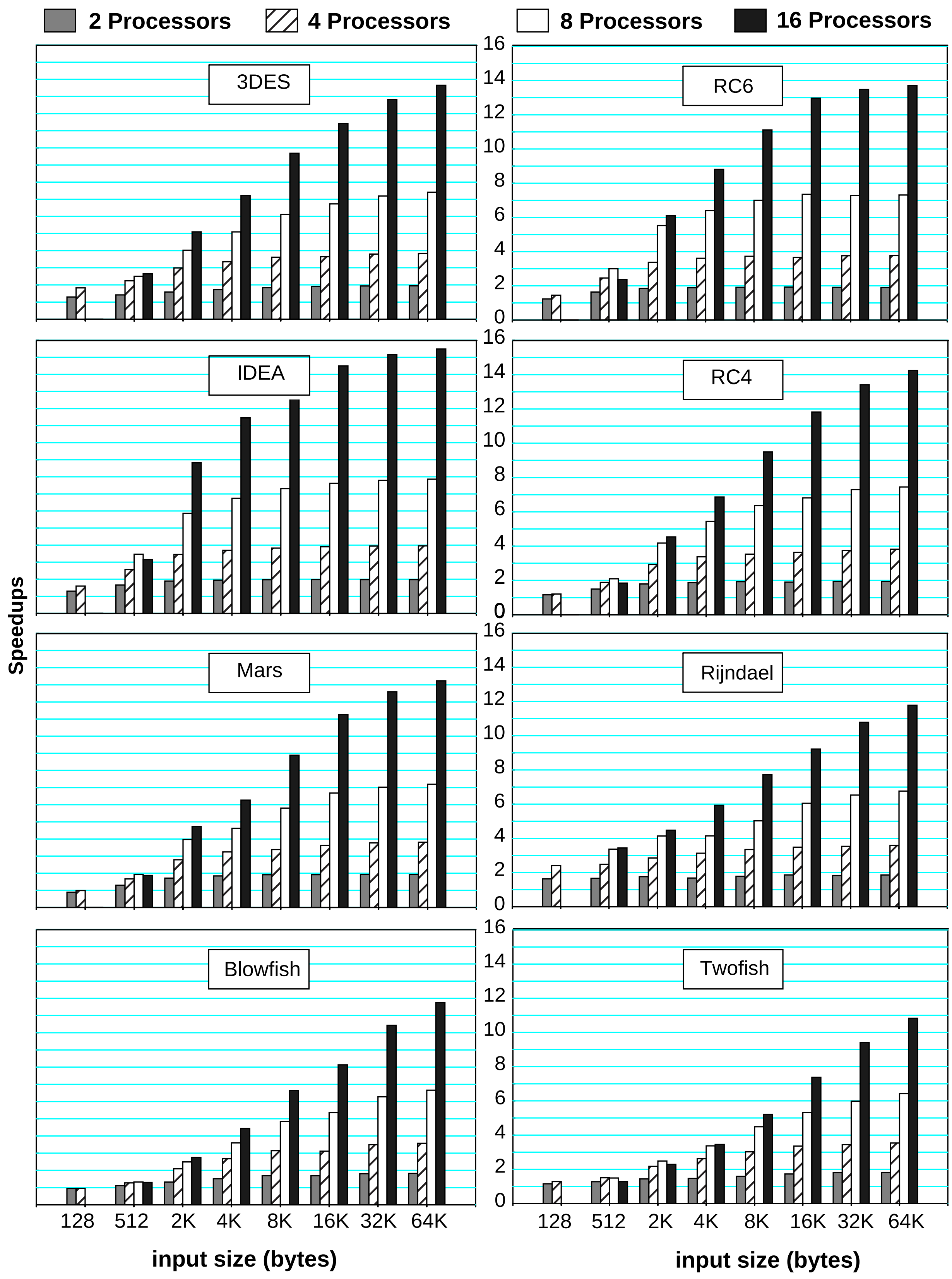

FIGURE 3. Speedups over uniprocessor.

Cipher	fraction of serial code (f * 100)	Barrier overhead (o_n/T * 100)		MaxSpeedup		Actual Speedup	
		(n = 4)	(n = 16)	(n = 4)	(n = 16)	(n = 4)	(n =16)
3DES	1.15	0.20	0.48	3.84	12.79	3.65	11.41
RC6	0.32	0.35	0.84	3.91	13.52	3.66	12.90
IDEA	0.19	0.21	0.50	3.94	14.42	3.90	14.40
RC4	0.77	0.15	0.35	3.89	13.66	3.63	11.82
MARS	0.99	0.37	0.89	3.83	12.39	3.62	11.26
Rijndael	1.60	0.88	2.11	3.69	10.14	3.48	9.22
Blowfish	3.25	0.47	1.11	3.58	9.59	3.16	8.46
Twofish	1.36	1.17	2.79	3.68	9.69	3.35	7.38

TABLE 2. Percent serial code and maxspeedup for 16 Kbytes input.

increase in input size. The length of serial code and the barrier overhead remain constant for a particular cipher and a specific number of processors. So, as input size increases, the overall contribution of serial code and barrier overhead decreases. For example, for 16 processors, the non-parallel portion at 1.91, on average, is higher for 16 Kbytes input compared to that for 64 Kbytes input. As per EQ 1, the average speedup of 10.64 for 16 Kbytes input is correspondingly lower than that of 12.97 for 64 Kbytes input.

From Figure 3, we also see that fewer processors yield closer to linear speedups compared to larger number of processors. Even though the absolute value of the non-parallel portion does not increase much with the number of processors, the relative contribution of the non-parallel portion to parallel execution time increases with the number of processors. For 16 Kbytes input, the non-parallel portion at 1.34 for 4 processors and 1.91 for 16 processors result in speedups of 3.57 for 4 processors, only 10% lower than the linear speedup of 4, and 10.64 for 16 processors, 32% lower than the linear speedup of 16.

In Table 2, we see that for many ciphers the maximum achievable speedups and actual speedups diverge more on going from 4 to 16 processors. The simulator models overhead such as finite bus request table, and finite queuing at the bus and memory, resulting in exposed latency due to queuing delay and queue/table overflow. On the other hand, EQ 1 assumes that there are no overhead other than barriers. Because queuing delays and finite buffering are more significant for 16 processors, the actual speedups seen in our simulations are less than the maximum speedups predicted by EQ 1.

4.3 Memory system effects

We get good speedups with larger number of processors. On initial thought, it might seem that memory system saturation must have set in. To investigate this point, we compare the number of compute cycles to memory latency, as well as bus occupancy.

4.3.1 Prefetching

To compare compute cycles versus memory latency, we count the number of processor cycles required to encrypt a cache line of input, and present the counts in Table 2. We see that for all the ciphers the compute cycles required to encrypt a cache line is large enough to hide typical memory latencies (e.g., 100 processor cycles) by prefetching the next cache line while the current cache line is being encrypted. Although all the processors issue prefetch requests to memory, these requests go to different banks in the interleaved memory (interleaved at cache line granularity) and do not queue up at memory. If memory latency or contention is more, then prefetch would have to be initiated earlier (e.g., two Iblocks ahead). As mentioned in Section 3.2, our prefetching is accurate in that every block prefetched is actually accessed and no unnecessary memory traffic is generated.

Cipher	
3DES	7364
RC6	2804
IDEA	4768
RC4	2208
Mars	2614
Rijndael	1906
Blowfish	1998
Twofish	1926

TABLE 2. CPU Cycles to encrypt a cache line.

CPUs	
1	20
2	40
4	80
8	160
16	320

TABLE 3. Number of CPU cycles occupied on the (unloaded) bus to fetch a cache line per processor from main memory.

Next, we isolate the effect of prefetching by comparing execution times with and without prefetching. In Figure 4, we show improvement as execution time with prefetching relative to execution time without prefetching for the same number of processors. The left graph shows 4 processors and the right graph shows 16 processors. To project the impact of higher processor speeds in the future, we vary processor clock speeds as 1, 2, and 4 GHz, and commensurately decrease memory latency as 100, 80, and 65 ns (i.e., the processor improves by a factor of 2 every step, and memory improves by approximately 20%).

Most improvements are modest because memory cycles are much fewer than compute cycles, so hiding memory cycles is not much opportunity to start with. Going from 1 to 4 GHz decreases compute time, increasing the relative contribution of memory latency and making prefetching increasingly important. Many improvements are so small that they fall within simulation error, and are not reliable to identify any trends. 3DES achieves the least improvements because this cipher has the most compute cycles, as per Table 2.

One would expect that from 4 to 16 processors, improvement will increase because memory is more of a bottleneck. Improvements actually decrease a little, and there are two reasons. First, on going from 4 to 16 processors for the non-prefetching case, memory's contribution to total execution time decreases due to increased overlap among memory requests (all 16 processors simultaneously issue memory requests, as opposed to just 4). Second, the non-parallel portion of execution including serial initialization and barriers does not decrease from 4 to 16 processors, limiting improvements from prefetching. Consequently, improvements due to prefetching decreases from 4 to 16 processors.

4.3.2 Bus occupancy

We consider the effect of increasing number of processors on the bus. Through experimentation, we found that the ciphers incur few cache misses on the internal data structures. These structures fit within our L1 cache, and are only read-shared after initialization. Therefore, the bus traffic is due almost entirely to misses (or prefetches) on the input Iblocks. Although our simulator accounts for queuing at the bus and memory, and accurately models bus occupancy by tracking the timing of bus transactions in the split-transaction bus, we present a simple model of bus occupancy to gain understanding. We compute the number of cycles the bus remains occupied transferring input Iblocks as follows:

$$BusOccupancy = n \times \left(BCPT + \frac{BlockSize}{BusWidth}\right) \times ClkRatio$$

(EQ 2)

where n is the number of processors, BCPT is the number of bus cycles per transaction, block size is the cache block size, and ClkRatio is the bus to CPU clock ratio. For our experiments, we assumed 5 bus cycles per transaction (when there is no queuing), cache block size to bus width ratio is 1, and the bus to processor clock ratio is 4.

Using EQ 2, we list the values for the bus occupancy in Table 3. We see that for all the ciphers there are enough compute cycles per cache line (see Table 2) for the bus to prefetch all the processors' next cache line, even in the 16-processor case. Therefore, bus occupancy does not limit overall performance in any significant manner.

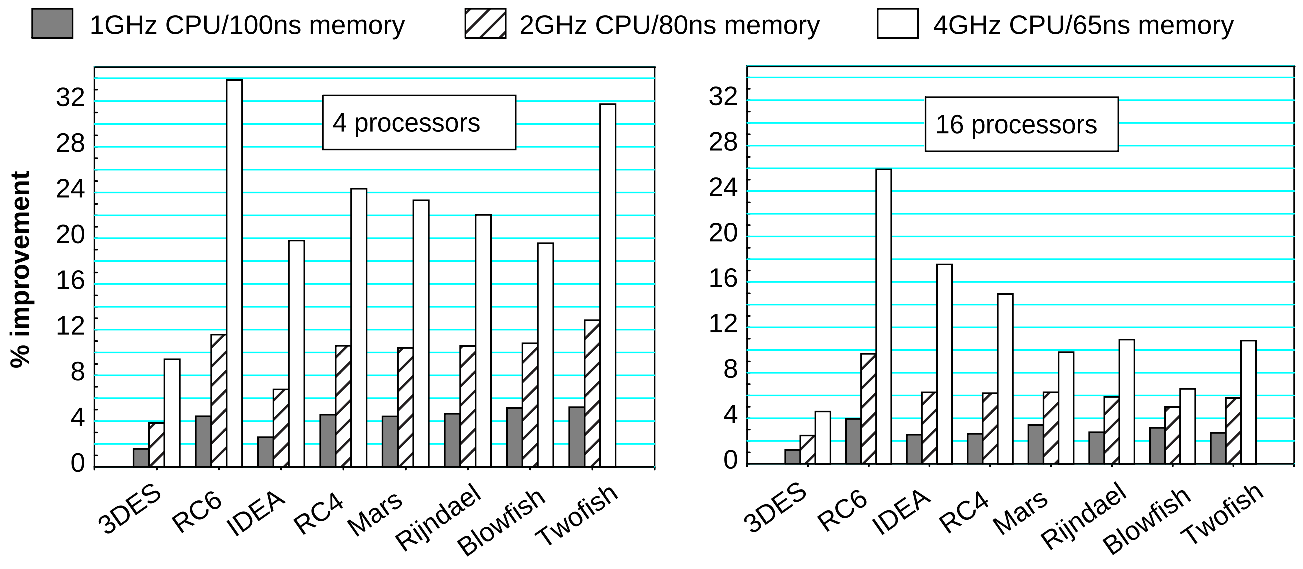

FIGURE 4. Improvement due to prefetch for 4- and 16-processor SMPs.

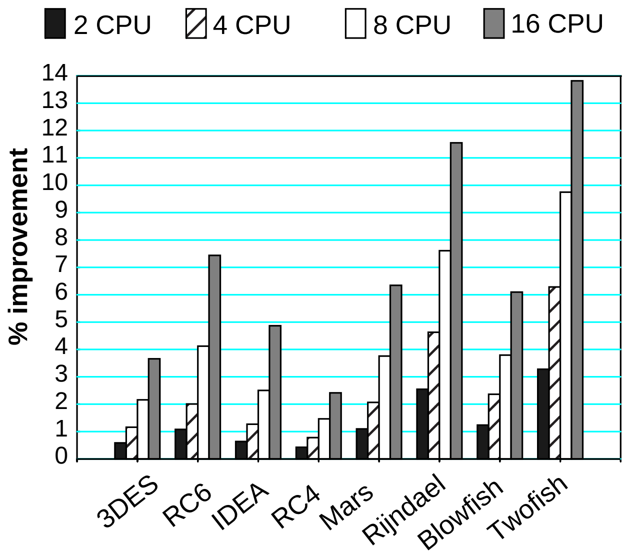

FIGURE 5. Improvement of hardware barrier over software barrier

4.4 Effect of barrier implementation

Because software barriers, despite using efficient MCS locks, impact performance significantly, we experiment with hardware barrier similar to that in the Thinking Machines CM-5. In Figure 4, we measure improvement as execution time using hardware barriers relative to execution time using software barriers for the same number of processors. We use a constant latency of 100 cycles for hardware barriers.

There are two trends apparent in the graph. First, Rijndael and Twofish achieve the most improvements with hardware barriers, in accordance with the fact that these two ciphers have the largest barrier overhead, as shown in Table 2. Second, the 16-processor SMP achieves much more improvement than the 4-processor SMPs because not only does barrier cost increase (Table 2), but also computation time decreases with increasing number of processors. This double effect results in large improvements for 16 processors. On average, 16 processors achieve 6.2% improvement compared to 3.8% and 2.1% achieved by 8 and 4 processors, respectively.

Note that the barrier overhead reported in Table 2 are fraction of uniprocessor execution time, where as improvements in Figure 4 compare the multiprocessor execution times using software and hardware barriers. Therefore, the numbers in Table 2 do not set a bound for the improvements shown in this section.

4.5 Overall encryption rates

We present the overall encryption rates achieved by our multithreaded implementation in Figure 6. In the X axis, we show the ciphers and in the Y axis we show the encryption rates in $\log_2$ scale achieved with a clock speed of 1GHz. The upper graph shows results for 16 Kbytes input and the lower graph shows results for 64 Kbytes input. We use prefetching and software barriers in this experiment because both of these optimizations are software based, and can be implemented in most of today's SMPs.

We see that for the 16 Kbytes inputs, the ciphers achieve about 9 Mbytes/s encryption rates on a uniprocessor. On a 4-processor SMP, the ciphers achieve about 32 Mbytes/s rates, gaining a factor of 3.5 in performance. On a 16-processor SMP, the ciphers achieve around 92 Mbytes/s, reaching a factor of almost 10 improvement over uniprocessor. SMP encryption rates for 3DES is the lowest because of the large number of cycles taken to encrypt each cache block of data, as shown in Table 3. Encryption rates for 64 Kbytes input are better (note the Y axis is logarithmic scale) than those for the 16 Kbytes input due to the amortization of the serial initialization over a larger input. This amortization occurs in both uniprocessor and SMP systems. For this larger input, many of the ciphers achieve over 128 Mbytes/s encryption rates using 16 processors.

5 Related Work

In previous work [5, 32, 24], the authors propose instruction-set support for cryptographic processing. They suggest extending the instruction set as well as hardware implementation to provide instruction-level support to accelerate encryption. Our proposal is orthogonal to these previous proposals and, we would benefit from ISA support because our multithreaded code would naturally exploit any hardware support in the processor to execute individual threads faster.

There have been several proposals for hardware public-key encryption [17,26,25,11,4]. There are several high speed hardware implementations of DES and 3DES algorithms [9,12], IDEA [15], Twofish [3], and RC6 [21]. These are algorithm-specific hardware solutions, whereas our proposal involves changing the software to a multithreaded implementation.

6 Conclusion

To deliver high performance while maintaining high level of security assurance in real systems, the cryptography community has proposed *Interleaved Cipher Block Chaining (ICBC) mode*. In four-way interleaved chaining, the first, fifth, and every fourth block thereafter are encrypted in CBC mode; the second, sixth, and every fourth block thereafter are encrypted as another stream, and so on. Thus, interleaved chaining loosens the recurrence imposed by CBC, enabling the multiple encryption streams to be overlapped. The number of interleaved chains can be chosen to balance performance and adequate chaining to get good data diffusion.

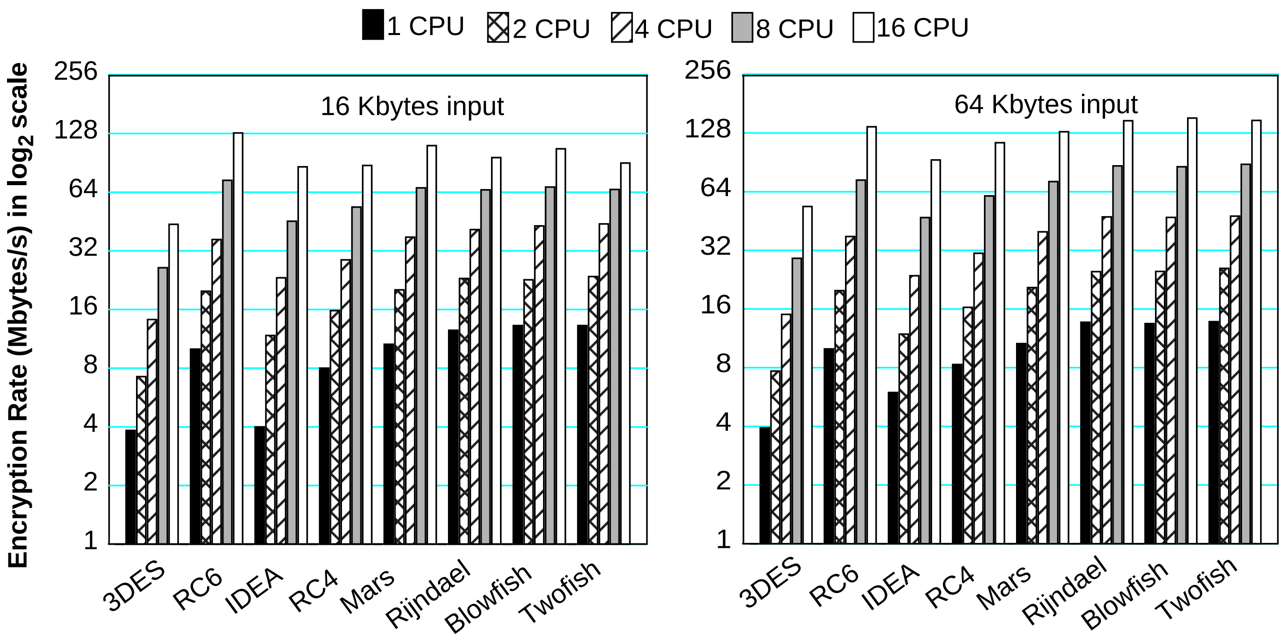

FIGURE 1. Encryption rates using a 1GHz processor with prefetching, and software barrier.

While ICBC was originally proposed to improve hardware encryption rates by employing multiple encryption chips in parallel, this is the first paper to evaluate ICBC via multithreading commonly-used ciphers on a symmetric multiprocessor (SMP). Multithreading ciphers using ICBC allows exploiting the full processing power of SMPs, which spend many cycles in cryptographic processing as medium-scale servers today, and will do so as chip-multiprocessor clients in the future.

Using the Wisconsin Wind Tunnel II, we showed that our multithreaded ciphers achieve encryption rates of 92 Mbytes/s on a 16-processor SMP at 1 GHz, reaching a factor of almost 10 improvement over a uniprocessor, which achieves 9 Mbytes/s. We found that multithreading the ciphers using ICBC makes them a good match for SMPs because there is no data sharing and no communication among the threads. The threads are naturally load balanced and computationally intensive, spending hundreds of cycles per cache line of input data brought from main memory. We showed that the number of compute cycles per cache line of input data is large enough both to hide prefetch of the next cache line, and to allow the bus to service prefetch requests from up to 16 processors. Serial initialization code, software barrier costs, and bus utilization are the factors preventing multithreaded ciphers from achieving perfect speedups on SMPs.

Acknowledgements

This work was supported in part by funding from the Center for Education and Research in Information Assurance and Security (CERIAS) at Purdue University.

References

[1] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Apr. 1996.

[2] R. Atkinson. Security architecture for the internet protocol. In *IETF Draft Architecture ipsec-arch-sec00*, pages 178–189, 1996.

[3] B. Schneier et al. *Twofish: A 128-Bit Block Cipher*. http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish, 1998.

[4] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings. 14th IEEE Symposium on Computer Arithmetic*, pages 70–79, 1999.

[5] J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. In *Proceedings of the 9th International Conference in Architectural Support for Programming Languages and Operating Systems*, pages 178–189, Nov. 2000.

[6] C. Burnwick et al. *The Mars Encryption Algorithm*. http://www.csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS, 1999.

[7] Counterpane systems. http://www.counterpane.com.

[8] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. http://www.csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael, 1999.

[9] D. W. Davies and W. L. Price. *Security for Computer Networks*. Wiley, 1989.

[10] P. Ferguson and G. Huston. *What is a VPN*. http://www.employees.org/ferguson/vpn.pdf, 1998.

[11] J.-H. Hong and C.-W. Wu. Radix-4 modular multiplication and exponentiation algorithms for the RSA

public-key cryptosystem. In *Design Automation Conference (ASP-DAC 2000)*, pages 565–570, 2000.

[12] IBM. *S/390 and OS/390 Cryptography*. http://www.s390.ibm.com/security/cryptography.html.

[13] J. Kase. *Cybernetica's 4-way IDEA*. http://www.cyber.ee/research/fastidea.html, 1999.

[14] S. S. Keller. *NIST Special Publication 800-20*. http://csrc.nist.gov/publications/histpubs/800-20/800-20.pdf.

[15] X. Lai. *On the Design and Security of Block Ciphers*. Hartung-Gorre Verlag, 1992.

[16] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the 4th International Conference in Architectura l Support for Programming Languages and Operating Systems*, pages 269–278, Apr. 1991.

[17] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[18] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Fast and portable parallel architecture simulators: Wisconsin Wind Tunnel II. *IEEE Concurrency*, 2000.

[19] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Oct. 1996.

[20] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, Feb. 1997.

[21] R. L. Rivest et al. *The RC6 Block Cipher*. http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6.

[22] RSA Security. http://www.rsa.com.

[23] B. Schneier. *Applied Cryptography*. Wiley, 1996.

[24] Z. Shi and R. B. Lee. Bit permutation instructions for accelerating software cryptography. In *Proceedings of the IEEE International conference on Application-specific Systems, Architectures and Processors*, pages 138–148, 2000.

[25] C.-Y. Su, S.-A. Hwang, P.-S. Chen, and C.-W. Wu. An improved montgomery's algorithm for high-speed RSA public-key cryptosystem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):280–284, June 1999.

[26] W.-C. Tsai, C. B. Shung, and S.-J. Wang. Two systolic architectures for modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):103–107, Feb. 2000.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[28] US Government. *Advanced Encryption Standard (AES) Development Effort*. http://csrc.nist.gov/encryption/aes.

[29] US Government. *Cryptographic Module Validation Program*. http://csrc.nist.gov/cryptval/.

[30] US Government. *Data Encryption Standard (DES), Triple DES, and Skipjack Algorithms*. http://csrc.nist.gov/cryptval/des.htm.

[31] US Government. *Security requirements for Cryptographic modules*. http://csrc.nist.gov/cryptval/140-2.htm, May 2001.

[32] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A fast, flexible architecture for secure communication. In *Proceedings of the 28thth International Symposium on Computer Architecture*, pages 110–119, May 2001.