Optimization and Benchmark of Cryptographic Algorithms on Network Processors*

Zhangxi Tan, Chuang Lin, Yanxi Li, Yixin Jiang
Computer Science Department
Tsinghua University
Beijing, China
xtan@csnet1.cs.tsinghua.edu.cn

Abstract – *With the increasing needs of security, cryptographic functions have been exploited in network devices. Besides time consuming, security protocols are flexible in algorithm selections. Fortunately, network processors, which serve as the backbone of intelligent network devices, hold performance and flexibility at the same time. In this article, we investigate several principles that can be used with implementing and optimizing cryptographic algorithms on network processors. Also, these principles are applied in real life algorithms, including stream ciphers, block ciphers and digital signatures. Related experiments and benchmark results on Intel IXP1200 network processor are provided.*

Keywords: Cryptographic algorithms, network processor, optimization.

1 Introduction

Salient trend has it that many network devices integrate cryptographic functions to gratify the increasing needs of security. Especially, encryption/decryption and digital signature algorithms are expensive and dramatically affect the all overall performance. Hardware solutions [1] suffer from the cost and flexibility, while software approaches compensate the drawbacks at the cost of performance. Network processor, a device between GPP (General Purpose Processor) and ASIC (Application Specific Integrated Circuit) caters for the requirements of performance and flexibility simultaneously. Unfortunately, several difficulties do exist to hinder the elevation. First, most cryptographic algorithms are designed without a full appreciation of network processors. Second, several preliminary results [2], [12] show encryption algorithms have much more complexity than other header processing applications. Third, many commercial network processors are RISC based and have limited MIPS. Besides, no cache or small cache is built on chip. Hence, optimizing cryptographic algorithms on network processor is not only close to software manners but also a challenging job.

Some related works on GPPs have been done recent years. Bruce Schneier [11] has presented several general optimization principles on Intel Pentium processors. It mainly focuses on "register poor" processors and takes more concern on instruction pairing and superscalar processing, which are not possessed by network processors. However, latency hiding and parallel processing are conspicuous means. Erich Nahum [4] has involved parallel processing on share memory multiprocessor systems and discussed three types of parallelism: per-connection, per-packet, and intra-packet parallelism. Thread parallel techniques will be emphasized in conjunction with latency hiding in our discussions.

In this article, we analyze the general architecture of prevailing commercial network processors, using Intel IXP series as an example. Then, we propose three catalogs of optimization principles. Due to some hardware limits, implementing expensive algorithms (e.g., RSA [7]) on fast path of network processors is not feasible. Thus, we implement and benchmark some compact encryption algorithms and MD5 [6] digital signature algorithm on Intel IXP1200 network processor.

2 Architecture of network processors

To satisfy the requirements of intelligent processing, most commercial network processors are designed with following technologies:

- Pipeline and Parallel mechanism. Network processors contain multiple processing elements (PEs) which organized either in pipeline or parallel manners.

- Optimized memory units. Memory access is an expensive task. This feature provides the feasibility of latency hiding and reimburses the drawback of small cache.

- Special ALU instructions. This is originally used to accelerate route applications, while still suits cryptographic applications' needs.

* 0-7803-7952-7/03/$17.00 © 2003 IEEE.

- Hardware multithread. Many network processors apply several "zero switching overhead" hardware threads to increase utilization rate.

To be more specific, in this article we use Intel IXP1200 as the platform. It is a wide used network processor, which typically incarnates the characteristics mentioned above, shown in Figure 1.

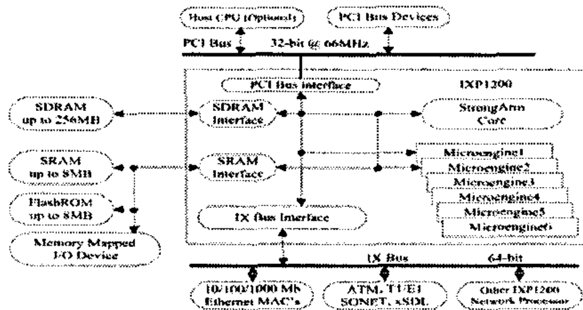

Figure 1. Hardware architecture of IXP1200

IXP1200 is mainly made up of 6 high speed Microengines and one StrongArm management core. Each Microengine owns 4 hardware threads and only one thread can be activated at anytime. Microengines and StrongArm are all RISC based processors sharing memory, bus and other off chip resources, while Microengines carry fast path applications and StrongArm does slow path jobs. Our optimizations will mainly focus on fast path, especially the four threads of one Microengine. Without any special indicate, in our benchmark parallel processing is refer to four threads of one Microengine.

3 General optimization principles

Firstly, we propose two goals for optimizations: one is to increase the utilization rate of Microengines. The other is to minimize memory accesses and communication latency. The former is to stretch the limited computation capacity to the outmost. The later is to downcast expensive memory operations. Regarding these, we summarize some principles in the following three catalogs.

3.1 Computation oriented

On RISC based network processors, most instructions could be executed in one cycle and no instruction patterns are required. However, network processors exploit instruction pipeline techniques and their rich register resources will be highlighted.

- Avoid using complex instructions, which occupy more than one cycle of time.

- Unroll loops, avoid conditional jumps and thread swapping. These can stunt the flush of pipeline and reduce calculations of iteration variables and array indexes.

- Take full advantage of rich register resources which could serve as cache, temporary storage and constant tables.

- Pre-calculate part of algorithms. For instance, setup S-Box entries with keys, compute array indexes and load immediate data into registers.

3.2 Latency hiding

Usually off-chip memory accesses generate long communication latency. During the access waiting time processors are in idle state, so the utilization rate of processors will be degraded. Network processors could improve this situation by controlling the waiting procedure manually. The core idea of latency hiding is to do something useful when waiting memory references. As a whole, the latency seems to be hided.

3.3 Parallel processing

On network processors, parallel processing can be exploited at Microengine level and hardware thread level. Microengine level is somewhat like the parallelism of SMP systems and related approaches have been studied in [4], [5]. Our methods carry more weights on hardware thread level parallelism in conjunction with the latency hiding technique. Since many multithread processors use non-preemptive context arbiters, thread switching can work in either of the two modes:

- Swap out immediately after issuing memory references. In this case, current thread will be deactivated waiting for the arbiter to wake up and next valid thread executes immediately.

- After issuing memory references finish some operations and then swap out. This approach combines the latency hiding technique.

The second mode takes the latency hiding into account and seems perfect, but the latency hiding prevents other threads from execution until it swaps out. Functionally, thread level parallelism we encounter in this article mainly include the following forms:

- All threads serve as homogeneous "processors" and process similar jobs. This method is especially useful to some block ciphers which can be paralleled at data level. Also it is commonly used with connection level parallelism.

- Threads serve different functions and are organized as a functional pipeline. Similar work

can be found in researches of Simultaneous Multithreaded processor (SMT). One early topic by Zilles [13] introduced the concept of helper thread prefetching memory to increase cache hit rate. Although cache behavior is not the major problem on network processors, memory access is still the bottleneck. Thus, we develop the original helper thread idea into a more complex one, by adding some non-critical operations to the helper thread.

4 Analyze and benchmark existing algorithms

In this section, we will apply our principles to some existing cryptographic algorithms. Up to now, most cryptographic algorithms are based on several time-consuming inner loops. Other portions either performed at startup time or sporadically, contribute little to the overall execution time Thus, the inner loops will be our major concerns and their performance will be evaluated on Intel IXP1200 network processor, which is configured at 200 MHz. All raw and unprocessed data are deposited in SRAM.

4.1 Block ciphers: Blowfish and Khufu/Khafre

Blowfish [10] and Khufu/Khafre [3] are concise and their instruction storage requirements are amenable to network processors.

- Precomputation.

As these algorithms, S-Box and some arrays of constant are initialized with keys. During the run time, they will not be changed frequently.

Blowfish: DWORD P[18], S[4][256]

P and S are all initialized according to keys. P contains only 18 entries and can be stored in register, while S has to be placed in SRAM.

Khufu/Khafre: DWORD S[ROUNDS/8][256]

The S-box of Khufu is key depended while Khafre is not. Like Blowfish, the S-box of Khufu/Khafre is also placed in SRAM.

- Optimizing inner loops.

Blowfish:

```
for (j=0;j<16;j++)  {        //16 rounds
    L ^= P[j];
    R ^= ((S[0][ L & 0xFF] + S[1][(L >> 8) & 0xFF])^
      S[2][(L >> 16) & 0xFF])+ S[3][(L >> 24) & 0xFF];
    swap(L,R);               //swap L, R
  }
```

L, R are two long words to be encrypted. The inner loop of Blowfish can be fully unrolled. Also it has some memory accesses and needs complex address calculations. For this reason, we hide these address calculations within the latency of memory references, shown in Figure 2. At cycle 5460, the first S-box read request is issued. Then we continue to calculate S-box indexes and request three more SRAM operations without swapping out. Consequently, 8 cycle computations have been hided.

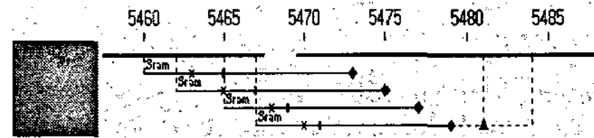

Figure 2. The latency hiding effect of Blowfish

The throughput is more than three times of the raw one, illustrated in Figure 3. Owing to the data independence of block ciphers, each thread can serve a portion of sources and parallel at connection or packet level. In our experiments, we benchmark the multi-thread performance in two modes mentioned in section 3.3. Here, we name parallel without latency hiding "M1" and parallel with latency hiding "M2". As Blowfish, M2 is better than M1. But in Khufu/Khafre, we draw the opposite conclusion.

Khufu/Khafre:

```
for (j=0;j<ROUNDS;j++)  {
    tmp = R ^ S[ROUNDS/8][L & 0xFF];
    R = L <<< C; L = tmp;
  }
```

L, R are two 32 bit blocks to be encrypted. C is a constant and determined according to j. As the security is concerned, ROUNDS must exceed 16. In our experiments we choose 32. Obviously, it is perfect to unroll all the loops, thus the calculation of S-box indexes and rotation constants will be omitted. Besides, loop unrolling can help latency hiding among different rounds. Like Blowfish, we take the advantage of data independence and evaluate the parallel performance in two modes. The results are given in Figure 3.

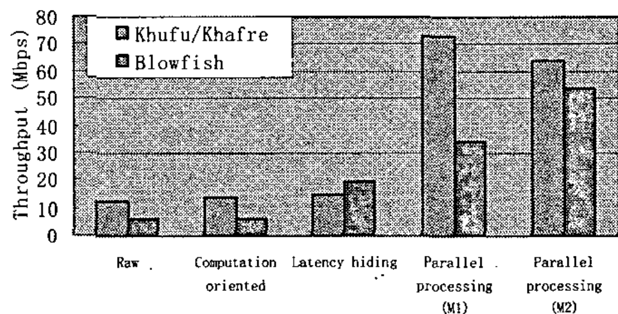

Figure 3. Throughputs of Blowfish and Khufu/Khafre with different optimization techniques

4.2 Stream ciphers: SEAL, RC4

In this section, SEAL [8], RC4 [9] will be analyzed. Disparate from block ciphers, most stream ciphers can not be paralleled at data stream level, because of the strong dependence of encryption operations. As a result, connection level parallelism could not improve the

throughput of single connection. At this circumstance, the helper thread method will be the suitable solution for parallel processing.

• Precomputation.

SEAL: DWORD T[512], S[256], N[4]
RC4: DWORD S[256]

These arrays are initialized based on keys and have to be placed in SRAM

• Optimizing inner loops.

SEAL:

```
Initialize a,b,c,d, N[4] according to keys
for (j=0;j<64;j++) {
    p = a & 0x7FC; b += T[p/4]; a=a >>> 9; b^=a;
    q = b & 0x7FC; c ^= T[q/4]; b=b >>> 9; c+=b;
    p =(p+c) & 0x7FC; d += T[p/4]; c=c >>> 9; d^=c;
    q =(q+d) & 0x7FC; a ^= T[q/4]; d=d >>> 9; a+=d;
    p =(p+a) & 0x7FC; b ^= T[p/4]; a=a >>> 9;
    q =(q+b) & 0x7FC; c += T[q/4]; b=b >>> 9;
    p =(p+c) & 0x7FC; d ^= T[p/4]; c=c >>> 9;
    q =(q+d) & 0x7FC; a += T[q/4]; d=d >>> 9;
    /* output 128bit data */
    a+=N[2*(j&1)];   c+=N[2*(j&1) + 1];
}
```

The inner loop of SEAL consists of initialization works and a 64-round loop. Apparently, the 64-round loop is the bottleneck. For the sake of simplicity, our benchmark only takes the 64-round loop into account. Owing to the large tables, SEAL has the most memory references among the algorithms we have benchmarked in this article.

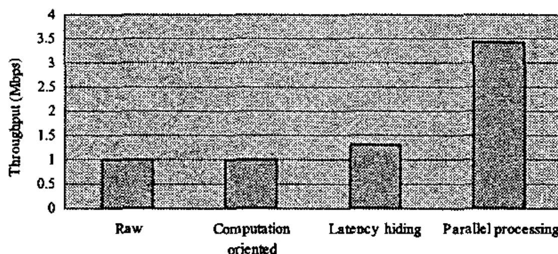

Figure 4. Throughputs of SEAL with different optimization techniques

Loop unrolling can only help to calculate the last two operations of the loop and contribute nothing to the T table index calculations. Also, the strong dependence of consecutive operations bogs down the effect of latency hiding. Only one circular rotation instruction can be hided with each T table reference, as Figure 4 shows the result. Although SEAL is a stream cipher, the generation of variable length outputs can still benefit from the simple parallelism used with block ciphers.

RC4:

```
for (i=j=0;cnt;cnt--,p++) {
  i=(i+1) & 0xFF;     //update index i
  tmpI=S[i];
  j=(j+tmpI) & 0xFF;  //update index j
  tmpJ=S[j];S[j]=tmpI; S[i]=tmpJ; //swap S[i], S[j]
  // calculate a random index of S-box
  t=(tmpI+tmpJ) & 0xFF;
  *p ^= S[t];            // XOR the raw data
}
```

The computation granularity of RC4 is 8 bit, so there are data align problems on 32 bit network processors. In our optimization, we unroll the loop to the multiple of 4 and align S-box entries to 32 bit. Additionally, the increment of i is only calculated every 4 or 8 rounds, thus raw or encrypted data can be easily cached in 32 bit registers. From the view point of latency hiding, we adjust the execution sequence of first two operations, that is to use the memory reference of S[i] to hide "i = (i + 1) & 0xFF".

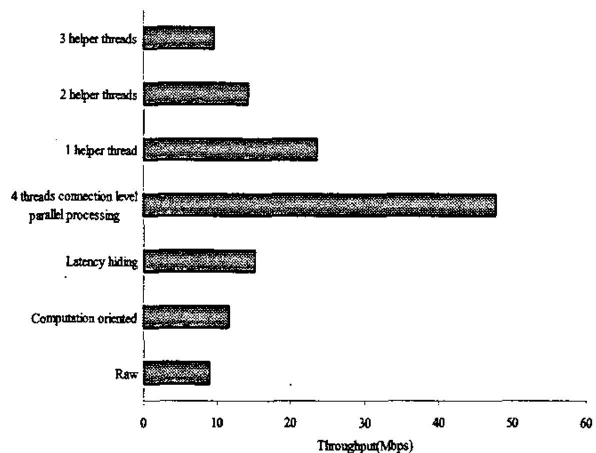

Figure 5. Throughputs of RC4 with different optimization techniques

On the parallel side, the helper thread idea has been introduced. We exploit one main thread to finish some serialized operations (first four lines of the loop) and use one or more threads to accomplish the random index calculation, an XOR operation (the last two lines of the loop) and prefetch operations (e.g. raw data read or write). Moreover, memory operations have been classified to achieve optimized performance, utilizing memory priority features of IXP1200. That is to assign higher priority to the S-box swap operation which is crucial to the following accesses and lower priority to the encrypted data write back. Here we benchmark their performances with different numbers of helper threads in Figure 5. The results show that the more the helper threads the less the performance. This is because the workload of helper thread is relative lighter than that of the main thread. More helper threads compete more with the main thread and adversely degrade the performance.

4.3 Digital signatures: MD5

MD5 is a 128 bit hash function, which has enjoyed widespread use in practice. Like stream ciphers, strong dependence exists among consecutive operations of MD5.

Further, it requires little memory references except reading raw data.

- Precomputation.

Unlike other encryption algorithms, MD5 does not have S-box or array initializations. Only 4 registers should be loaded with immediate data. On the other hand, MD5 employs many 32 bit constants during the main loop. Whereas, calculations with 32 bit immediate data on many 32 bit RISC based network processors require at least 3 instructions (two for loading data into registers, one for ALU operation). Hence, in the precomputation stage we load these immediate data in advance.

- Optimizing inner loops.

Reading raw data;
(Round 1) For j from 0 to 15 do the following:
 t = (A + f(B;C;D) + X[z[j]] + y[j]),
 (A;B;C;D)=(D; B+(t - s[j]);B; C).
(Round 2) For j from 16 to 31 do the following:
 t = (A + g(B;C;D) + X[z[j]] + y[j]),
 (A;B;C;D)=(D; B+(t - s[j]);B; C).
(Round 3) For j from 32 to 47 do the following:
 t = (A + h(B;C;D) + X[z[j]] + y[j]),
 (A;B;C;D)=(D; B+(t - s[j]);B; C).
(Round 4) For j from 48 to 63 do the following:
 t = (A+k(B;C;D)+X[z[j]]+y[j]),
 (A;B;C;D)=(D;B+(t - s[j]);B; C).
(update chaining values)
 (H1;H2;H3;H4) = (H1+A;H2+B;H3+C;H4+D).

A,B,C,D are 32 bit variable. f,g,h,k are functions made up of basic bit operations. X are raw data arrays. s, z ,y are constant arrays. Clearly, loop unrolling can avoid many array indexes. Noticed that the majority of the main loop is composed of register calculations, latency hiding technology contribute little to the optimization. Moreover, MD5 is also a serial algorithm and allows the minimum inner connection parallelism. Therefore, helper thread method has been involved like RC4. This time, the helper thread only performs raw data read operations and uses thread signals to communicate with the main thread.

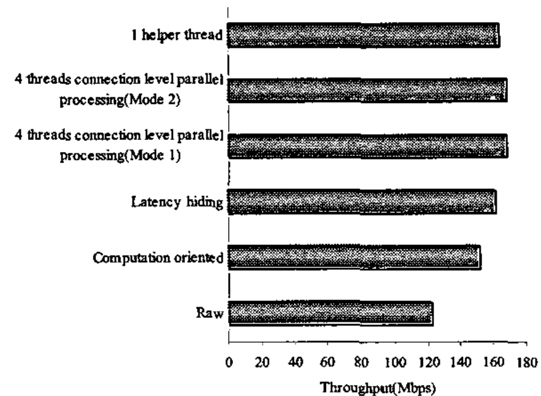

Figure 6. Throughputs of MD5 with different optimization techniques

Results in Figure 6 show that the latency hiding, helper thread and connection level parallelism all achieve throughputs at approximate 160 Mbps. The reason is that MD5 is a computation-intensive algorithm and memory references only occupy a small amount of time. Hence, optimizations towards memory access only make a little improvement.

5 Conclusions

Above, we have optimized and benchmarked 5 fast path cryptographic algorithms, which hold different characteristics. Finally, we compare their intrinsic properties on IXP1200 in Table 1. To show the efficiency of different optimization principles, SRAM throughput after applying different principles and Microengine utilization rate have been given in Table 2.

Table 1. Characteristics of different cryptographic algorithms

Algorithm	Type	Instruction Space	Rounds	Memory accesses / Round
Blowfish	Block Cipher	39	16	4
Khufu/Khafre	Block Cipher	20	32	1
RC4	Stream Cipher	39	/	7
SEAL	Stream Cipher	100	/	15
MD5	Digital Signature	744	/	2

Table 2. SRAM throughput and Microengine utilization rate under different optimizations

Algorithm	SRAM throughput T (Mbps) and Microengine utilization rate ρ (%)									
	Unoptimized		Computation oriented		Latency hiding		Thread parallel		Helper thread	
	T	ρ	T	ρ	T	ρ	T	ρ	T	P
Blowfish	268.1	20.5	281.4	17.4	219.06	39.4	775.87	86.1	/	/
Khufu/Khafre	242.3	31.5	261.0	22.8	280.8	24.3	1103.7	80.6	/	/
RC4	263.5	21.2	270.7	19.6	352.0	22.0	1104.0	69.0	545.2	69.0
SEAL	298.4	22.8	301.5	22.0	390.4	24.9	1292.7	79.2	/	/
MD5	127.7	91.5	157.5	89.2	167.8	95.5	176.2	99.2	171.2	97.5

In Table 2, column "Thread parallel" is refer to thread parallelism with or without the latency hiding technique and the best method is sampled here.

From the tables above, we may draw the following conclusions:

- Algorithms with too many memory references often get poor performance. For examples, SEAL has 15 memory references per round and gets the lowest performance among the algorithms we have benchmarked. In contrast, Khufu/Khafre, MD5 and Blowfish enjoy higher throughput.

- Computation oriented optimizations deliver very limited performance boost unless the algorithm relies seriously on the MIPS of processors (e.g. MD5). Due to the RISC architecture, this can be estimated by the instruction space of Table 1.

- Thread level parallelism and latency hiding are all used to increase the utilization rate of processors. In our experiments most algorithms improve a great deal with these optimizations. If the original algorithm already has a high processor utilization rate (like MD5), these methods will not work perfectly.

- The helper thread is a good method to involve parallelism with "serial" algorithms. In the tests of RC4 and MD5, helper threads really improve the performance and get similar throughput (per thread) as connection level parallelism.

In real world applications, IXP1200 is designed for access or edge devices and mainly supports 100Mbps links. According to our results, to keep up with the link speed, only one Microengine is not enough. Nevertheless, we do believe implementing some lightweight cryptographic algorithms with network processors is still a feasible solution. Although our work is preliminary and many factors remain to be explored, it still can be used with algorithm selections and designs on the platform.

References

[1] W. Feghali, B. Burres, and G. Wolrich, "Security: Adding Protection to the Network via the Network Processor", *Intel Technology Journal,* Vol 6, No. 3, pp. 40-49, Aug. 2002.

[2] G. Memik, B. M. Smith, and W. Hu, "NetBench: A Benchmarking Suite for Network Processors", Proc. IEEE/ACM International Conference on Computer-Aided Design, San Jose CA, pp. 39-42, November 2001.

[3] R. C. Merkle, "Fast Software Encryption Functions", Proc. 10th Annual International Cryptology Conference, pp. 476-501, Santa Barbara CA, August 1990.

[4] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel, "Towards High Performance Cryptographic Software", Proc. 3th IEEE Workshop on the Architecture and Implementation of High Performance and Communications Subsystems (HPCS), Mystic CT, August 1995.

[5] E. Nahum, D. Yates, and S. O'Malley, H. Orman, and R. Schroeppel, "Parallelized network security protocols", Proc. Symp. on Network and Distributed System Security, pp. 145-154, San Diego CA, February 1996.

[6] R. Rivest, "The MD5 message-digest algorithm", *RFC 1321*, Network Working Group, April 1992.

[7] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, page 120-126, Feb. 1978.

[8] P. Rogaway, and D. Coppersmith, "A Software-Optimized Encryption Algorithm", *Journal of Cryptology*, Vol. 11, No. 4, pp. 273-287, Sep. 1998.

[9] B. Schneir, *Applied Crytography, 2nd Edition*, John Wiley & Sons, New York NY, 1996.

[10] B. Schneir, "Description of a new variable-length key, 64-bit block cipher (blowfish)", Proc. the Cambridge Security Workshop on Fast Software Encryption, pp. 191-204, Cambridge UK, December 1993.

[11] B. Schneier, and D. Whiting, "Fast software encryption:Designing encryption algorithms for optimal software speed on the Intel Pentium Processor" , Proc. Fast Software Encryption: 4th International Workshop, Haifa Israel, pp. 240-259, January 1997.

[12] T. Wolf, and M. Franklin, "CommBench-A Telecommunications Benchmark for Network Processors", Proc. IEEE International Symposium on Performance Analysis of Systems and Software, Austin TX, pp. 154-162, April 2000.

[13] C. Zilles, G. Sobi, "Ececution-based Prediction Using Speculative Slices", Proc. 28th Annual International Symposium on Computer Architecture, pp. 2-13, Göteborg Sweden, June 2001.