Workload Characterization of Cryptography Algorithms for Hardware Acceleration

Jed Kao-Tung Chang
Dept. of Electrical Engineering and Computer Science
University of California
Irvine CA 92697
1-949-6789643
jedc@uci.edu

Chen Liu
Dept. of Electrical and Computer Engineering
Florida International University
Miami, FL 33174
cliu@fiu.edu

Shaoshan Liu
Microsoft Corp.
308 Occidental Avenue South
Seattle, WA 98104-3884
shaoliu@microsoft.com

Jean-Luc Gaudiot
Dept. of Electrical Engineering and Computer Science
University of California
Irvine CA 92697
gaudiot@uci.edu

ABSTRACT
Data encryption/decryption has become an essential component for modern information exchange. However, executing these cryptographic algorithms is often associated with huge overhead and the need to reduce this overhead arises correspondingly. In this paper, we select nine widely adopted cryptography algorithms and study their workload characteristics. Different from many previous works, we consider the overhead not only from the perspective of computation but also focusing on the memory access pattern. We break down the function execution time to identify the software bottleneck suitable for hardware acceleration. Then we categorize the operations needed by these algorithms. In particular, we introduce a concept called "Load-Store Block" (LSB) and perform LSB identification of various algorithms. Our results illustrate that for cryptographic algorithms, the execution rate of most hotspot functions is more than 60%; memory access instruction ratio is mostly more than 60%; and LSB instructions account for more than 30% for selected benchmarks. Based on our findings, we suggest future directions in designing either the hardware accelerator associated with microprocessor or specific microprocessor for cryptography applications.

Categories and Subject Descriptors
E.3 [**Data Encryption**]: Standards (e.g., DES, PGP, RSA)

D.4.6 [**Operating Systems**]: Security and Protection – *Security Kernels, Cryptographic controls.*

ICPE/WOSP/STPEW'11, Month 14–16, 2011, Karlsruhe, Germany.

General Terms
Algorithms, Performance, Security

Keywords
Cryptographic algorithms, Load-Store Block (LSB), Overhead

1. INTRODUCTION
Security is of increasing importance in everyday life. With the rapid development in computer and communication technology, most information is stored in the form of electronic data. Without a good security protection scheme, confidential data can be easily stolen. For instance, credit card numbers may be stolen during online payments and/or Internet transactions. Confidential business information, such as intellectual property, customer rosters, employee information, *etc.,* may be accessed with a fake digital signature. People constantly try to hack into the mainframe in Pentagon in searching for classified data. To solve these problems and protect the secrecy and integrity of data, cryptography provides some highly useful property: for example, even an intruder is able to somehow get access to some sensitive data, if the data is encoded, then it cannot be easily decrypted. Therefore, many organizations have been investing heavily on cryptography and information security research. As a result, many cryptography algorithms were released by academia institutions and the industrial world.

However, there are two concerns regarding cryptography algorithms: performance and security. From the security perspective, if strictly implemented in software, this security program can be altered by another piece of software, such as a computer virus. What's more, with the emergence of hardware-based attacks, the software approach would not offer protection against those. Hackers can observe the hardware changes and guess the transmitted data by using appropriate tools. For example, as Huang [20] indicates, hackers can physically observe or interfere

with the operation of the system by putting a device onto the communication path between the processor and the main memory. One can attach the snooping devices to various buses and get unencrypted sensitive data by observing the voltage or current changes from oscilloscopes and multi-meters. From the performance point of view, cryptography algorithms are extremely expensive in terms of execution time, especially for asymmetric encryption algorithms, such as ECC (Elliptic Curve Cryptography), Diffie-Hellman Key Exchange algorithm or other public key algorithms. Encryption/decryption are quite computation-intensive since transmitted data needs to be enciphered via many arithmetic operations so that cipher text will not easily be cracked; and this process is quite memory-intensive too if the amount of data need to be encrypted/decrypted is huge. Using a general-purpose processor for such scenario will be very time-consuming, impact system performance negatively and increase the power consumption largely.

The performance degradation when running the security application may be due to the computation or the memory access overhead. Many previous work focused effort on improving the efficiency of computation. They used software and hardware approaches to improve the efficiency of computation, the "arithmetic and logical instructions" of the cryptographic applications. However, memory access part is often overlooked, which is actually the real performance bottleneck nowadays as the "Memory Wall" problem is worsened. Based on our observation, the percentages of memory read and write instructions reaches at least 52% and up to 88% of the total instruction mix for cryptographic algorithms, as will be discussed in detail in later sections. This shows the intensity of data movement for encryption/ decryption that goes between the CPU and memory, during which much time has been wasted. That is our motivation to study the memory access behavior in order to enhance the performance of cryptographic algorithms. If the time of memory access can be reduced, not only the overall execution time will be saved but also the security strength will be enhanced because it will leave little space for hackers to detect or tamper the data transmitted off-chip. In this paper, we investigate the dynamic behavior of the security applications from the perspective of both computation and memory access pattern.

One thing we want to specify is that we are not looking into the characteristics of these algorithms from a compiler designer's point of view, trying to come out with some application-specific flag design for the compiler; actually we are looking at the problems from a computer architect point of view, hoping our analysis could aid the microarchitecture design to improve the performance of these algorithms. During the process, we employ a performance analyzer to observe the kernel part of the cryptographic benchmarks, as well as a binary instrumentation tool to observe the ratio of the categorized dynamic instructions in each benchmark. Besides, we are especially interested in the behavior of the load and store instructions due to the data intensive characteristic of the cryptographic applications.

The contributions of this paper are mainly three-fold:

- Firstly, we identify that many cryptographic algorithms are mainly consumed by the execution of "hotspot functions", which is suitable for hardware acceleration;
- Secondly, we study the instruction mix of various algorithms and observe that majority of the instructions fall into very limited categories, which provides the aid for the instruction set extension design;
- Thirdly, we introduce the concept of Load-Store Block (LSB) and identify the LSB distribution for all algorithms. Our analysis suggests for algorithms with high LSB distribution, they provide us an opportunity to perform data parallel operations in a SIMD fashion.

The remainder of the paper is organized as follows. We present background research on different algorithms in Section 2. Then we describe our experiment environment and show the methodology to conduct the research in Section 3. In Section 4 we present our simulation results. Lastly, we conclude in Section 5.

2. BACKGROUND RESEARCH

In this section we introduce the algorithms we are going to study in this work and briefly present some related work in cryptographic algorithm analysis.

2.1 Algorithms

In this study, we choose nine cryptographic algorithms: AES, 3DES, RC5, MD5, IDEA, SHA1, Blowfish, ECC and RSA, as our benchmarks.

Rijndael's algorithm [23] was selected from 15 candidates as AES and became the new U.S. federal standard. Compared with other algorithms, it has a good balance in terms of speed, security, and flexibility. It can run on a wide range of processors with high performance and resist against known attacks then. It can be efficiently implemented on general purpose hardware such as 32-bit CPUs, even as low as 8-bit microprocessors. Please note in this study we use the key size of 256-bit for AES. 3DES [22] applied the DES [1] three times to each data block. Although executing it is time-consuming, 3DES is widely adopted in banking information system and electronic payment industry. IDEA [27] is a symmetric key algorithm used by PGP (Pretty Good Privacy) v2.0 to transmit message bodies [21]. Blowfish [24] provides a good encryption rate. It takes only 18 cycles to encrypt one byte on a 32-bit processor with a memory requirement of only 5KB. RC5 [25] is notable for its simplicity. What's more, the length of its secret key, word size, and number of rounds of computation can be configurable. It is used in devices with restricted memory size such as smart cards.

MD5 and SHA1 are both hash algorithms, which are used to verify the integrity of data blocks. MD5 [26] is widely used to assure if the transmitted file has arrived intact and to store passwords. SHA1 [28] is often used in firewall, VPN, and IP-security.

ECC [29-30] and RSA [31] are public key algorithms. Public key cryptography is also called asymmetric key cryptography. Contrary to the symmetric key algorithm which uses the same key to encrypt and decrypt transferred data, asymmetric algorithm has two keys: public key and private key. The sender uses a published public key to encrypt the data and receiver uses the private key to decrypt the data. ECC is based on the algebraic structure of elliptic curves over finite fields. It is now popular due to the fact that it offers the same security level as offered by other contemporary algorithms at a shorter key length. RSA is suitable for encryption and digital signature and used in E-Commerce protocols. Although the execution of ECC and RSA are time-consuming, in these asymmetric algorithms each data can be encrypted or decrypted independently and the operations on these data can be performed in parallel. Table 1 lists the algorithms we use, the category each of them belongs to, and the programming language of their source code.

Table 1. The Benchmarks

Algorithm	Category	Language
AES	Symmetric Key	C
3DES	Symmetric Key	C
Blowfish	Symmetric Key	C
IDEA	Symmetric Key	C
RC5	Symmetric Key	C
MD5	Hash	C
SHA1	Hash	C
ECC	Public Key	C++
RSA	Public Key	C

The reason we choose these nine algorithms is not only due to their popularity but also because their program structures being representative of the contemporary cryptography works. Some of these algorithms, like 3DES, Blowfish, and RC5, use Feistel cipher [16], where the text is split into two halves. The first half is applied round function using a subkey. The output will be XORed with the second half. Then the two halves will be swapped. The following round will have the same pattern. Some other algorithms, such as AES, use the iterative cipher [32]. In each round it operates on the entire data block. The program structures of these two ciphers are pervasive in the design of modern cryptographic algorithms, so we choose them in an effort to observe their impact on the performance of modern microprocessor in finding ways to optimize the hardware microarchitecture correspondingly.

2.2 Related Work

Many efforts have been made in speeding up the cryptographic applications using both hardware and software approaches.

Bielecki *et al.* [14] and Beletskyy *et al.* [15] used parallel programming as a way to increase the performance of the cryptographic algorithm, targeting at a series of algorithms like DES, 3DES, AES, IDEA, Blowfish, RC5, LOK191, GOST, and RSA. Focusing on the loop structures, they performed data dependency analysis on loops and used loop parallelization technology with OpenMP. They observed that the execution time can be decreased significantly with the usage of symmetric multiprocessing (SMP).

The research in [2] and [3] used a dedicated cryptographic coprocessor to alleviate the CPU from cryptographic workload. Although this way of implementation is several orders of magnitude faster than the software implementation, coprocessors lack the flexibility to support different parameters such as the key size or the mode of operations. Moreover, the silicon area will be increased and the system bus connecting the CPU and coprocessor forms a performance bottleneck.

With the rapid development and increasing popularity of graphic processing unit (GPU), people tried to implement cryptographic applications on it due to the high-level parallelism this many-core structure provides. Harrison *et al.* [9] implemented AES Encryption ECB mode on GPU, taking advantage of its large number of simple processing units and stream processing. They mapped the AES algorithm onto GPU by implementing XOR using the Raster Operation Unit and fragment processor hardware. They showed that GPU can run AES with high efficiency and alleviate the cryptographic loads from CPU if used as a coprocessor.

Others tried to implement the cryptographic systems on the reconfigurable architecture. In [12-13], AES was implemented on the Xilinx FPGA board. Instead of translating a high-level language into HDL code, they used Handel-C to design the system and mapped to FPGA directly. This methodology is less error-prone and Handel-C has pipelining and parallelism constructs. The results showed enhanced performance and less die area is required.

The recent research trend is to perform instruction set extension (ISE) through designing some customized instructions and adding them to the existing instruction set architecture (ISA). This way the problems associated with coprocessor design will be mitigated with additional benefit as improved power consumption. In [6] and [7] the custom instructions were designed to implement the costly *Subbytes* and *MixColumn* stages of AES algorithm. Moreover, these instructions were enhanced with the ability to implicitly perform *ShiftRows* transformation. Bertoni *et al.* [8] incorporated the *SubByte*, *ShiftRow*, and *MixColumns* stages of AES into one or two instructions.

However, a good profiling is necessary in order to extend instruction set properly. Through the profiling tool we can learn what operations account for most of the execution time and get an insight in how to extend the instruction set architecture for these operations. Burke *et al.* [4] performed profiling on eight benchmarks: Blowfish, 3DES, Mars, RC4, RC6, IDEA, Rijndael, and Twofish. They illustrated possible hardware factors for the performance bottlenecks, which were the issue width and the number of function units. They also identified the computation and memory access intensive parts being the kernel for all encryption application. They proposed instructions for operations such as 16-bit modular multiplication, bit permutation, rotation, and memory table lookup that executed heavily in the kernel. Finally, they designed and implemented the extended instructions and architecture for these heavy kernel operations.

Fiskiran *et al.* [11] investigated ECC, AES, and SHA1 for their usage in constraint environment, such as in PDA. They used the PLX RISC architecture and divided the instructions into seven classes: Store, Load, Arithmetic, Logical, Shift, and Branch (conditional and unconditional) and reported the ratio of these instruction classes. By analyzing the instruction frequencies, they showed that these benchmarks can be implemented by a simple RISC processor.

Clapp [10] analyzed the AES candidates by using *software critical path* as a metric to determine the upper limit performance of each candidate. A software critical path has the largest weighted instruction count, which means in this path the machine spends the longest time to encrypt each block. Given the number of instructions per data block, the *effective parallelism* is defined as the ratio of the total number of instructions per block to the number of cycles in the critical path, which can indicate maximum number of concurrent execution units could be put into usage in order to achieve the highest performance. Then these benchmarks were run on a family of hypothetical VLIW CPUs. The results showed that two candidates, Crypton and Rijndael, had potential to benefit from the improved instruction-level parallelism (ILP) by increasing the instruction issue slots.

3. EXPERIMENT ENVIRONMENT/ METHODOLOGY

In order to properly perform the workload characterization, we use a performance analyzer and a binary instrumentation tool to observe the dynamic behavior and characterize the performance for each

3.1 Experiment Environment

We choose VTune [18] and PIN [19] as our basic tools. Developed by INTEL® as a commercial product, VTune analyzes the software performance on IA-32 and Intel64-based machines. It collects performance data of the application running on the host system, organizes and displays the data in an interactive view. PIN is a binary instrumentation tool that can inject code dynamically while the executable is running. Through instrumentation, PIN is able to generate and collect data such as instruction count, instruction address trace, memory reference trace, load store trace, *etc.*, to help us better understand the program behavior. In this experiment, all the algorithms are running on a base machine with INTEL® Core™ i7 processor. The major experiment parameters are listed in Table 2.

Table 2. The Experiment Environment

CPU	INTEL i7-920 2.66GHz 4-core
I-Cache and D-Cache	32KB each (per core)
L2 Cache	256KB (per core)
L3 Cache	8MB (shared)
Memory Size	12GB
Operating System	Fedora Release 8 (Red Hat 4.1.2-33)
Compiler	gcc version 4.1.2
Binary Instrumentation Tool	PIN 2.8
Performance Analyzer	VTune 9.0

3.2 Methodology

First, we employ VTune to characterize the performance of these nine algorithms. VTune's call graph view provides a tree structure to show the call relationship among all functions. VTune also provides us with the Self Time and Total Time for each function. Total time is the execution time of this function and all the subroutines it calls, while the Self Time is without counting all those subroutines. If we observe that a function's Total Time is high but Self Time is low, there must be some subroutine of it having high workload. Thus, this would help us identify what we call "hotspot functions". The hotspot function normally consumes a substantial amount of the execution time of the specific algorithm, which we can focus on for hardware acceleration.

Another way to inspect the dynamic behavior of these benchmarks is to observe at assembly language level. We utilize the instruction mix to observe the operations needed by these algorithms. We categorize the instructions based on their functionality. In this way we would know what types of instructions account for the most of the instruction stream so that we can design instruction set extension correspondingly.

Since the cryptography algorithm belongs to data intensive application, there are frequent memory accesses represented by load and store instructions which accounts for a major part of the total execution time (as we will see in Section 4). We define a concept called "Load-Store Block" (LSB), which is a block of instructions started with a LOAD and ended with a STORE, with both pointing at the same effective address, within a single basic block. Here we use *LSB(i)* to represent a LSB with *i* instructions in between the pair of load-store instructions. *LSB(0)* means this LSB has no other instruction aside the load-store pair. The size of LSB varies and we use PIN to record the frequency of Load-Store Blocks of different size. We are interested in observing the behavior of LSB, especially in knowing what percentage of total instruction count falls into the Load-Store Blocks. The percentage of LSB instructions can be expressed by the metric, *LSB_Perc*, which is defined as:

$$LSB\_Perc = \sum_{i=0}^{N} \frac{(i+2) \times Occr(i)}{IC} \quad (1)$$

where *i* is the size of LSB, *Occr(i)* is the number of occurrence of the *LSB(i)*, and *IC* would be the total instruction count. Note that we use *(i+2)* is because we need to include the starting Load and ending Store instructions for the instruction count.

Actually, the purpose of introducing LSB metric is to implement cryptographic algorithms on parallel architecture like Processor-in-Memory (PIM). Traditionally, people would like to deal with the cryptographic algorithm from the perspective of arithmetic intensity. They either explore the ILP or use caching effects to enhance the performance. However, the off-chip transmission is actually the real performance bottleneck nowadays as the "Memory Wall" problem is worsened. The memory access latency degrades the performance more than the traditional ILP problem. Thus, instead of considering the caching effects and ILP-centered approach such as out-of-order execution, we focus on implementing the LSBs as functional units and incorporate them into memory modules of the memory-SIMD hybrid machine such as PIM. In this way, data can be accessed in the local module and much time of the memory access can be saved. We believe that if the memory-wall problem can be relieved by a memory-SIMD hybrid machine, the performance will be improved much more than the conventional ILP-centered approach.

4. RESULT ANALYSIS

We have performed experiments on the nine cryptographic algorithms to study their workload characteristics. We will observe the function behavior, the instruction class and memory access pattern of these benchmarks, from which we can enhance their performance by possible hardware solution.

4.1 Function Breakdown

We use VTune to break down the total execution time at the granularity of function level so that we can observe where the hotspot function is. In the example of 3DES, function *main* calls function *des_encrypt*, which in turn calls function *des_crypt*. Table 3 shows the self time and total time of each function, all normalized to the total time of function *main*, which is pretty much the time required to execute the whole algorithm. Note that the *des_encrypt*'s total time accounts for 93.67% of the total execution time, but its self time (excluding the execution time of its subroutines) accounts only 7.24%, as illustrated. It clearly shows that the hotspot function here is function *des_crypt*, the execution time of which accounts for 86.43% of the total execution time.

Table 3. The Self Time and Total Time of the Function *Main, des_encrypt, and des_crypt* in 3DES Benchmark

Function Name	Self Time	Total Time
Main	6.27%	100%
des_encrypt	7.24%	93.67%
des_crypt	86.43%	86.43%

Figure 1 shows the percentage of the hotspot function(s) occupying the total execution time for each benchmark. We only consider the crypto computation (key setup, encryption, and decryption) part of an application, excluding the file I/O or some system call within the dynamic linking library. We can see the execution time of hotspot function account for majority of the execution time for most of the benchmarks. For example, for 3DES, AES, Blowfish, IDEA, MD5, and RSA, the hotspot functions for each algorithm occupy either in exceed of or as near as 70% of the total exaction time of the whole algorithm. However, SHA1 is an exception because most of its execution time is spent in I/O operation called by the crypto computation functions, such as reading from a file, so there is really no hotspot function we can locate. Thus, the function breakdown helps us to identify the software bottleneck of the application. What's more, for all the benchmarks there are only one or two hotspot functions. These hotspot functions are good candidates for hardware acceleration because we only need to target the extracted hotspot functions in order to achieve satisfying speedups, following Amdahl's Law.

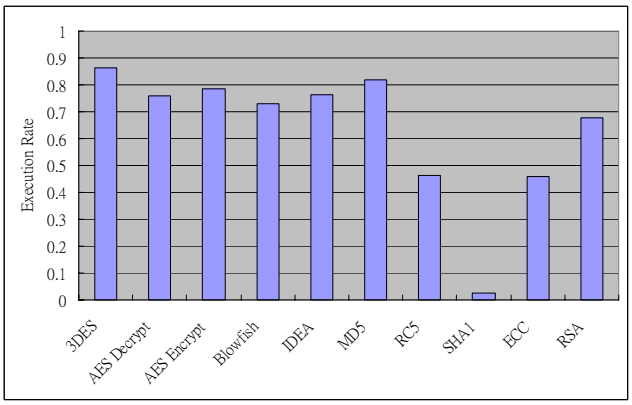

Figure 1. Execution Rate of Hotspot Function(s)

4.2 Instruction Mix

In order to profile the instruction mix, all the benchmarks are compiled into IA-32 assembly instructions. The IA-32 ISA has eight General-Purpose Registers (GPR), six Segment Registers, one Flags Register and an Instruction Pointer. Seventy-five instructions are actually used and are grouped into seven classes based on their functionality: Binary Arithmetic Instructions (BA), Bit and Byte Instructions (BB), Control Transfer Instructions (CT), Data Transfer Instructions (DT), Logical Instructions (L), Shift and Rotate Instructions (SR), and Miscellaneous Instructions (M)[1,2] [17]. These categories and the instructions belonging to them are listed in Table 4.

Table 5 shows the instruction class frequencies for these 9 benchmarks and Figure 2 is the 100% stacked column graph for Table 5. The data are collected by running all the algorithms on the same input file, a 1GB video file in this case. We can see that Data Transfer class accounts for majority of the instruction count in all the benchmarks, ranging from at least 46% for 3DES to as high as 86% for AES Decryption, with the only exception for RSA at 28%. Binary Arithmetic class is the heaviest in RSA at around 37% because RSA itself has a lot of multiplications and modulo arithmetic operations. Binary Arithmetic class is the second heaviest instruction class in AES Decryption, IDEA, MD5, RC5, SHA1, and ECC, ranging between 18% and 30%. Logical class is extremely high for 3DES at 34% and accounts substantially in AES Decryption, Blowfish, and MD5, all at more than 10%. Shift & Rotation class ranks third for 3DES and RC5 while it is Control transfer class for RC5 and SHA1. Note that as the size of input file becomes larger, the distribution of each instruction class will converge to a certain value and remains stable.

Many operations involved in these algorithms, such as bitwise operations, fixed length rotations, non-circular shifts and permutations are simple. Because huge amount of data (for example, video files) need to be encrypted before they are sent to the clients, the asymmetric encryption algorithms contain high data parallelism. Data can be encrypted and decrypted independently. If we perform optimization on those classes of instructions we identified, the performance of these applications can be dramatically enhanced.

Undoubtedly, Data Transfer instructions are heavy in most benchmarks. Binary Arithmetic is also heavy across all benchmarks except 3DES. Shift & Rotation class is only heavy for 3DES and RC5. Logical and Control Transfer classes are heavy in half of the benchmarks, as in 3DES, AES_Decrypt, Blowfish, and MD5; while Control Transfer is heavy the rest of the benchmarks, as in AES_Encrypt, IDEA, SHA1, ECC, and RSA. Overall, the top three instruction classes for each algorithm account for almost 80% of instruction count.

Data transfer instructions include the transfer of data between memory and register and within them. In order to differentiate among those, we need to examine at an even finer degree.

Table 6 shows the percentage of the memory read and memory write instructions over the total instruction count. The first row represents the instructions with memory read, the second row is the instructions with memory write, and the third row with both because IA-32 is of CISC ISA and some instructions include both read and write operations. The fourth row is the sum of the first two rows minus the third row to eliminate the overlapped instructions with both memory read and write. Thus the fourth row would be the ratio of total memory access instructions. Readers may notice that memory read/write percentage is higher than the Data Transfer class percentage shown in Table 5. The reason is that not only Data Transfer class but also other classes such as Binary Arithmetic, Logic, or Shift & Rotation classes include mode of the memory access. For example, for memory read mode we will see mov [addr], Reg, but *add [addr], Reg* and *xor [addr], Reg* belonging to Binary Arithmetic and Logic classes are also included in IA-32. This explains the higher memory read/write percentage than that of the Data Transfer class.

AES_Encrypt has the highest total memory access ratio of 88%. This matches with its highest Data Transfer class instruction of 86% among all algorithms studied. It is followed by Blowfish and RSA with 85% and 76% of the total instruction count being memory access, respectively. The memory access ratio for IDEA, MD5, RC5, SHA1 and ECC are all well over 60%. While for 3DES and AES_Decrypt, the ratios are not that high compared with others, but still count for significant amount of total instructions, at least 52%. The observation demonstrates that memory access operations are indeed heavy in the cryptographic algorithms.

From the instruction mix (Figure 2) and the percentage of memory read and write operations (Table 6), not only can we see the

[1] I/O Instructions and Segmentation Register Instructions do not appear in the applications.

[2] Flag control instructions are merged into the Control Transfer Instruction class and String instructions are merged into Data Transfer instruction class because of their low frequency and similar characters.

instruction mix among all the benchmarks but also we can see that memory access overhead plays an important role in downgrading the performance of security applications. Since the cryptographic algorithm belongs to the data intensive application, there must be frequent memory accesses represented by the Load and Store instructions. These results motivate the introduction of Load-Store Blocks.

4.3 Load-Store Blocks

Given the instruction mix information in the previous subsection, we know that memory access instructions count a significant amount of time in many cryptographic algorithms. It is the part that is overlooked by many previous research works of performance enhancement for the cryptographic applications. Traditionally the computation part receives much attention – arithmetic and logical operations to encrypt/decrypt data have been optimized and corresponding hardware architectures have been designed to improve the performance in an effort to deal with this arithmetic intensity problem. However, the memory part would be the real performance bottleneck: during the encryption / decryption process, data needs to be sent on- and off-chip, which occupies the majority of the execution time. If the performance of this part could be enhanced, total performance would be improve dramatically.

That's the purpose of our study on the LSB behavior. If we build many function units associated with the memory module, we offload the work from CPU and the memory access time will be saved. We can further duplicate the single memory module with function unit so that data could be preloaded and executed in parallel.

Given this insight, we know that the load and store instructions represent the memory access operations and the Load-Store Block is a potential target for us to perform the hardware optimization. Different from the data transfer class in Section 4.2, the target we are interested here is the number of instructions needing to be executed after the data loaded from one memory address and before they are stored back to the same address. That is the number of instructions between the Load and Store instructions with the same effective address. These instructions with the outer Load Store instruction pair form the Load-Store Block. The ratio of the number of instructions belonging to Load-Store Blocks to the total instruction count would be our metric for cryptographic algorithms implementation on parallel architectures, as Section 3.2 indicates. Figure 3 shows the percentage of instructions belonging to the Load-Store Block for all benchmarks, following the Formula (1) from Section 3. We can see that 3DES, IDEA, MD5, and RC5 have a good percentage of Load-Store Block instructions, around 30% to 37%. SHA1 and ECC are about 16%.

One interesting phenomena we observe is that as the input file size increases, the LSB percentage converges to a certain value. Figure 4(a) and (b) illustrates this using the MD5 and RC5 as examples. Mostly after the input file size exceeds 6MB, we can see no fluctuation in LSB percentage with the rate converging to 33.4%.

Figure 5 illustrates the ratio of Load-Store Block of various sizes to the total Load-Store Block instruction count. Following our definition from Section 3, we use *LSB(i)* to represent the Load-Store Block of size *i*. Here the LSBs of size 19 or above (*LSB(i)* with $i \geq 19$) are ignored because their occurrence is negligible. It can be clearly observed that in 3DES large load-store blocks are dominated as LSB(8) and LSB(10) account for 89% of all the load-store blocks. While in IDEA, MD5 and RSA, it is small load-store blocks that account for the most, with LSB(3) and LSB(5) for IDEA at 94%, LSB(1) and LSB(2) for MD5 at 97%, and LSB(2) and LSB(3) for RSA at 99%, respectively. RC5 is very interesting because load-store blocks of various sizes are evenly distributed. SHA1 is a mixture of small and large load-store blocks as LSB(2), LSB(3), and LSB(7) are the prominent. Same for ECC, the LSB(2), LSB(6), and LSB(>10) accounts for 73% of the total load-store blocks.

The data presented above provide us with insight of the computation load in the Load-Store Block and the cost to implement it. If the LSB size is large, it indicates that for a single data loaded, many operations will be performed upon it before it is written back to memory. Hence, we can categorize the application into LSB-computation-bound. On the other hand, if the LSB size is small, this indicates that few operations are performed on the data before write it back to memory. Hence, most probably the memory access will be the bottleneck. So we can categorize the application into LSB-memory-bound. If the LSB is of various sizes, then the application is a mixture of LSB-computation-bound and LSB-memory-bound. If the application is of LSB-computation-bound, then we may consider increase the number of execution unit when designing the hardware; if the application is of LSB-memory-bound with good data parallelism, then it is good candidate for a memory-SIMD hybrid machine, which means grouping multiple elements together and perform the operation using multiple processing units all at once. This can be implemented by exploiting an extra vector processing core inside the microprocessor or implemented in Processor-In-Memory (PIM). More specifically, if there is a small-sized LSB, *e.g.*, Load—Add—Store, a lot of time is spent on Load and Store instructions for the memory accesses while the Add operation only takes one or a few cycles. If we put the data of the Add operation in a memory module with an Adder or a simple ALU, the time for the Load/Store execution will be greatly reduced. Furthermore, if there are a lot of Load-Add-Store LSBs, we can build many memory modules with the same functional unit and load the data in advance. Not only will the time spent on load/store be saved but also multiple additions can be performed simultaneously. Since we only focus on LSB of small size, the function units implementing the instructions will not be complex, which means the hardware overhead will be minimal. That is why the study on LSB can be used to perform data parallel operations in a SIMD fashion. Thus, this approach not only saves the time for data access through memory but also takes the advantage of data parallelism.

To achieve the concept of parallel workloads, small sized LSB is a good candidate for a memory-SIMD hybrid machine, which means grouping multiple elements together and performs the operation using simple processing units all at once. Certainly, another approach would be to construct a multi-core/many-core platform for the LSB speedup. The differences between the two approaches are:

- The load/store in SIMD hybrid machine is just a memory read/write by the function unit in the local memory module and data doesn't need to pass through the memory hierarchy to reach the core.[3]
- The operations in the LSB are rather simple and the hardware overhead needed to implement it is much smaller than a core. Besides, more data parallelism could be explored with our SIMD approach.

[3] We assume the data in PIM and cache structure to be mutual exclusive so they will not have the cache coherence problem.

5. CONCLUSION

Enhancing the performance of cryptographic application is critical to improve the security mechanism in modern enterprise or industry information systems. An effective workload characterization on security algorithms is important before we work on the hardware acceleration for cryptographic applications. In this paper, we observe the dynamic behavior of memory access operations in addition to the computation operations since memory access issue would be a key for the strength and performance breakthrough of the security application. We address this issue by choosing nine cryptographic algorithms of which program structure is pervasive in modern cryptographic algorithm design and performed the following study:

- We profiled each application and identified the "hotspot" functions. In seven out of nine benchmarks the hotspot functions take around 70% of the total execution time, which would be suitable for the hardware acceleration.
- We analyzed the instruction mix and observed Data Transfer class and Binary Arithmetic class count for majority of the instruction mix. And the top three instruction classes account for 80% of the total instruction count for all the benchmarks. What's more, the memory access instruction ratio is more than 50% for all the benchmarks.
- We instrumented the dynamic behavior in running the cryptographic applications and observed that four out of nine applications has a Load-Store Block (LSB) instruction count of more than 30%. If the same LSB appears often with good data parallelism, we could take advantage it by duplicating functional units to implement those operations in an SIMD fashion.

The result of our work provides a solid starting point in building a memory-SIMD hybrid machine to enhance the performance of cryptographic algorithms. Also, previous hardware implementations are difficult to reconfigure to satisfy the requirement of different algorithms like key sizes and data block sizes. Based on this paper's work, our next step is to design a Processor in Memory architecture which satisfies the requirement of high memory bandwidth, parallelism, and reconfigurability. For future research, we will study the program structure of selected algorithms in great detail so that we could extend our works to perform the hardware acceleration. We are also interested in constructing a comprehensive cryptographic benchmark suite if the need arises.

Table 4 Instruction Breakdown Based on their Functionality

BA	BB	CT		DT		L	SR	M
ADC	SETB	CALL_NEAR	JNLE	CMOVB	MOVSD	AND	ROL	LEA
ADD	SETNBE	INT	JNS	CMOVBE	MOVSW	NOT	ROR	NOP
CMP	SETNLE	JB	JNZ	CMOVNB	MOVSX	OR	SAR	RDTSC
DEC	SETNZ	JBE	JP	CMOVNBE	MOVZX	XOR	SHL	
DIV	SETZ	JL	JRCXZ	CMOVNS	POP		SHR	
IDIV	TEST	JLE	JS	CMOVNZ	PUSH		SHRD	
IMUL		JMP	JZ	CMOVZ	XADD			
INC		JNB	LEAVE	CMPXCHG	XCHG			
MUL		JNBE	RET_NEAR	CLD	CMPSB			
NEG		JNL		CWDE	SCASB			
SBB				MOV	STOSB			
SUB				MOVSB	STOSD			

Table 5. Instruction Class Frequencies in the Nine Benchmarks

	3DES	AES_Dec	AES_Enc	Blowfish	IDEA	MD5	RC5	SHA1	ECC	RSA	Average
Binary Arithmetic	4.63%	18.33%	4.89%	7.38%	23.38%	19.91%	30.43%	20.41%	15.75%	36.57%	18.17%
Bit & Byte	0.01%	0.04%	0.06%	0.01%	7.33%	0.04%	0.01%	2.41%	0.96%	0.11%	1.1%
Control Transfer	0.51%	2.63%	4.17%	7.30%	8.80%	0.56%	4.35%	11.78%	10.91%	21.70%	7.27%
Data Transfer	46.25%	54.51%	86.49%	66.50%	52.67%	49.44%	52.15%	52.34%	66.67%	27.99%	55.50%
Logical	34.40%	10.87%	1.62%	11.72%	4.12%	16.81%	0.01%	5.68%	0.28%	0.01%	8.55%
Shift & Rotation	14.05%	5.82%	2.06%	5.15%	3.64%	8.41%	13.03%	4.78%	2.54%	13.61%	7.31%
Miscellaneous	0.15%	7.81%	0.71%	1.95%	0.06%	4.83%	0.01%	2.60%	2.89%	0.01%	2.1%
Total	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	

Table 6. The Percentage of Memory Read and Write Instructions

	3DES	AES Decrypt	AES Encrypt	Blowfish	IDEA	MD5	RC5	SHA1	ECC	RSA
Memory Read	51.47%	49.09%	83.35%	63.62%	53.31%	49.95%	52.52%	49.39%	39.68%	68.84%
Memory Write	14.83%	40.36%	74.66%	35.25%	26.56%	13.62%	13.33%	24.87%	29.06%	36.68%
Memory Read/Write	8.74%	37.22%	70.29%	13.63%	11.06%	1.65%	2.61%	10.37%	0.14%	29.37%
Total Memory Access	57.56%	52.23%	87.71%	85.25%	68.81%	61.92%	63.25%	63.88%	68.60%	76.15%

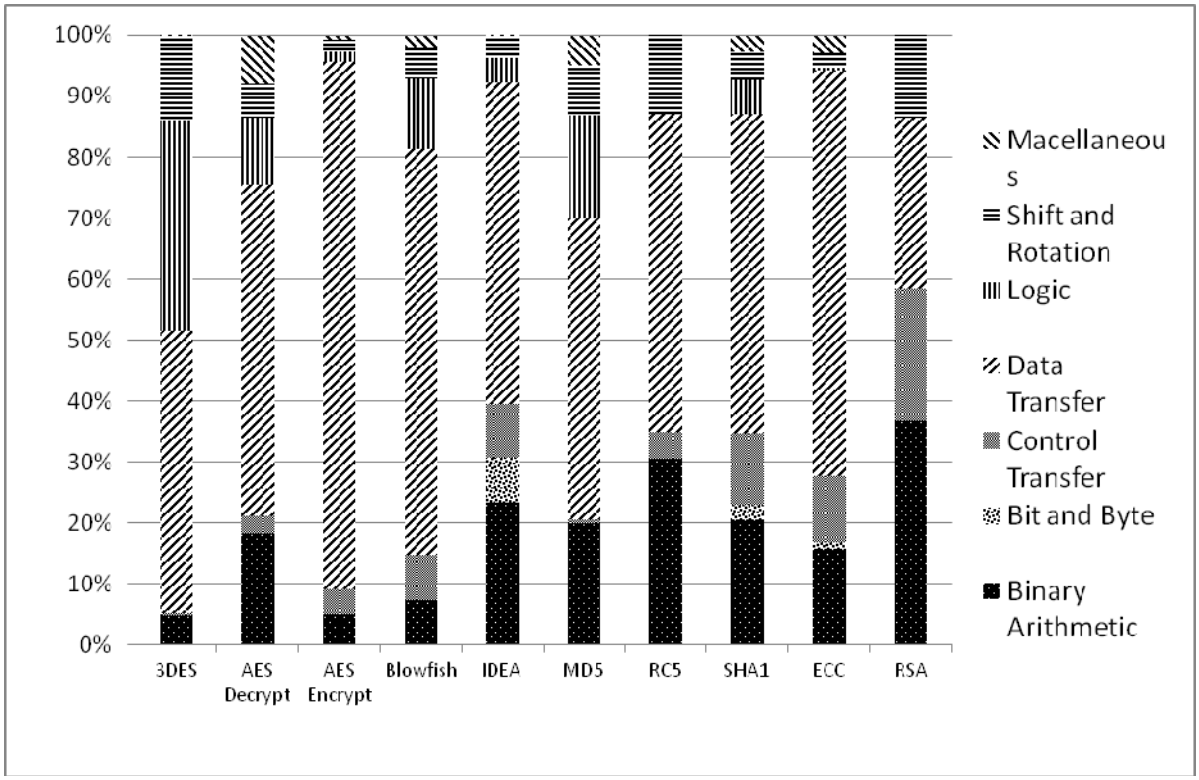

Figure 2. The Instruction Mix for the Benchmark

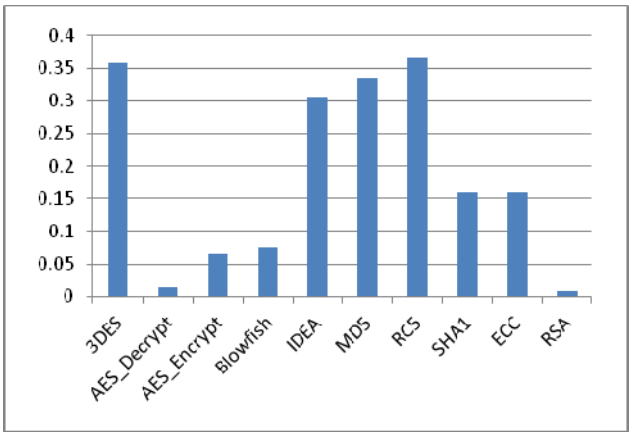

Figure 3. Load-Store Block Percentage

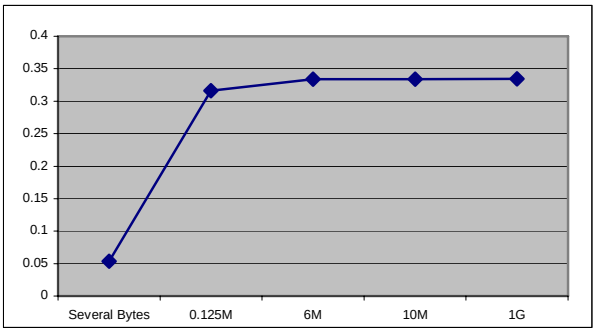

Figure 4(a). The Percentage of LSB for MD5 with Input File of Various Sizes

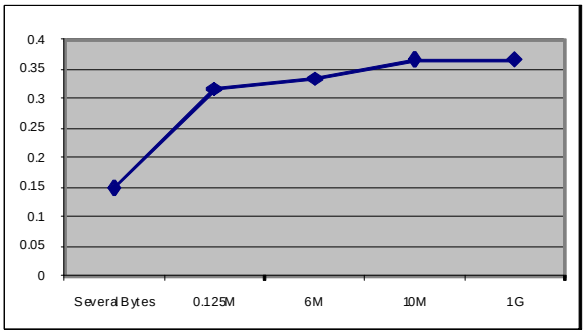

Figure 4(b). The Percentage of LSB for RC5 with Input File of Various Sizes

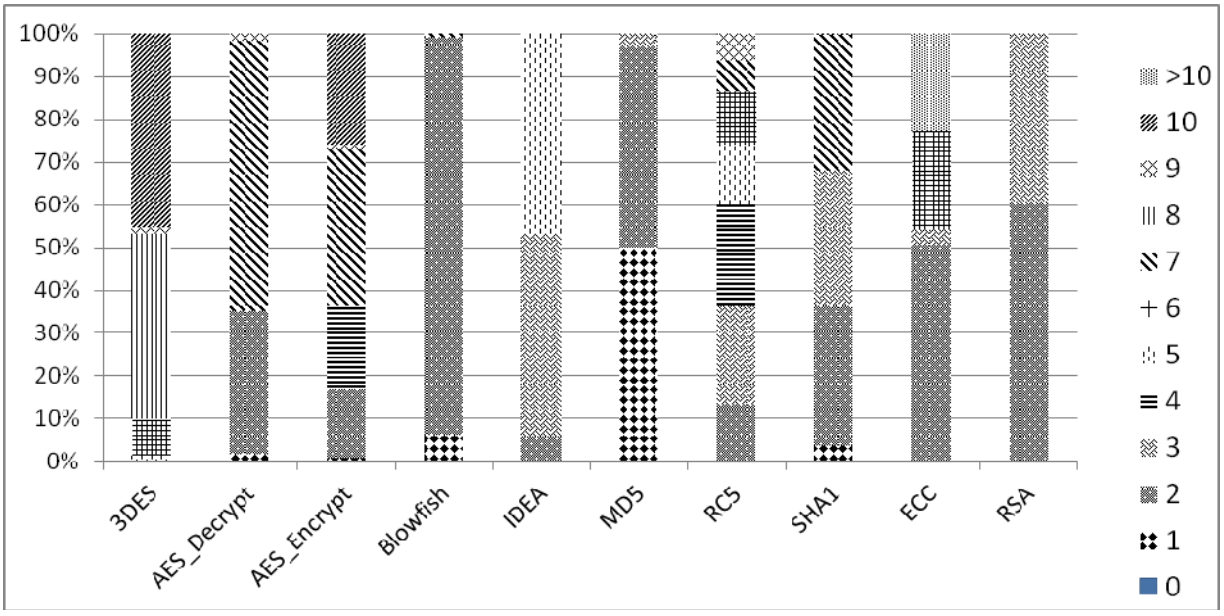

Figure 5. The Ratio of Load Store Block of Various Size in Each Benchmark

6. REFERENCES

[1] NIST, "Data Encryption Standard (DES) - FIPS Pub. 46," January 1977.

[2] Alireza Hodjat, "Interfacing a high speed crypto accelerator to an embedded CPU", *Proceedings of the 38th Asilomar Conference on Signals, Systems, and Computers*, 2004, pp. 488-492

[3] Patrick Schaumont , Kazuo Sakiyama , Alireza Hodjat , Ingrid Verbauwhede , "Embedded software integration for coarse-grain reconfigurable systems", *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (IPDPS 2004), pp. 137-142

[4] Jerome Burke, John Mcdonald, Todd Austin, "Architectural Support for Fast Symmetric –Key Cryptography", in *Proceedings of the 2000 Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2000), pp. 178-189

[5] Arif Irwansyah, Vishnu P. Nambiar, Mohamed Khalil-Hani, "An AES tightly coupled hardware accelerator in an FPGA-based embedded processor core", in *Proceedings of the 2009 International Conference on Computer Engineering and Technology* (ICCET 2009), VOL 02, pp. 521-525

[6] Stefan Tillich, Johann Großschädl, "Instruction set extensions for efficient AES implementation on 32-bit processors", In *Cryptographic Hardware and Embedded Systems* — CHES 2006, pp. 270-284

[7] Stefan Tillich, Johann Großschädl, Alexander Szekely, "An Instruction Set Extension for Fast and Memory-Efficient AES Implementation", *Communications and Multimedia Security* — CMS 2005, pp. 11-21

[8] Guido Marco Bertoni, Luca Breveglieri, Farina Roberto, Francesco Regazzoni , "Speeding up AES by extending a 32 bit processor instruction set", in *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, (ASAP 2006), pp. 275-282

[9] Owen Harrison and John Waldron, “AES Encryption Implementation and Analysis on Commodity Graphics Processing Units”, *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, 2007, pp. 209-226

[10] Craig S. K. Clapp, “Instruction Level Parallelism in AES Candidates”, Second AES Candidate Conference, Rome, March 1999

[11] A. Murat Fiskiran, Ruby B. Lee, “Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments”, in *Proceedings of the 5th annual IEEE workshop on Workload Characterization* (WWC-5), 2002.

[12] Ould-cheikh Mourad, Si-Mohamed Lotfy, Mehallegue Noureddine, Bouridane Ahmed, Tanougast Camel, "AES Embedded Hardware Implementation," ahs, pp.103-109, *Second NASA/ESA Conference on Adaptive Hardware and Systems* (AHS 2007), 2007

[13] Marko Mali, Franc Novak, and Anton Biasizzo, ”Hardware Implementation of AES Algorithm”, *Journal of Electrical Engineering*, VOL. 56, No. 9-10, 2005, pp. 265-269

[14] Wńodzimierz Bielecki and Dariusz Burak, “Parallelization Method of Encryption Algorithms”, *Advances in Information Processing and Protection*, 2008, pp. 191-204

[15] Vladimir Beletskyy and Dariusz Burak, “Parallelization of the IDEA Algorithm”, *Lecture Notes in Computer Science*, VOL 3036, 2004, pp. 635-638

[16] Luby, Michael; Rackoff, Charles (April 1988), "How to Construct Pseudorandom Permutations from Pseudorandom Functions", *SIAM Journal on Computing* 17 (2): 373–386

[17] x86 Assembly Language Reference Manual, Sun Microsystems. http://dlc.sun.com/pdf/817-5477/817-5477.pdf

[18] VTune: Intel performance analyzer http://software.intel.com/en-us/intel-vtune/

[19] PIN: a dynamic binary instrumentation tool. http://software.intel.com/en-us/intel-vtune/

[20] A. Huang. Hacking the Xbox: An introduction to Reverse Engineering. No Starch Press, San Francisco, CA, 2003.

[21] PGP (Pretty Good Privacy), PGP Corp. http://www.pgp.com/

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

[23] NIST (National Institute of Standards and Technology), “Advanced Encryption Standard (AES) – FIPS Pub. 197,” November 2001.

[24] *Counterpane Systems*. http://www.counterpane.com.

[25] R.L. Rivest. The RC5 encryption algorithm. In *Proceedings of the 2nd Workshop on Fast Software Encryption*, pages 86{96, Springer, 1995.

[26] R.L. Rivest. The MD5 message-digest algorithm, Request for Comments (RFC1320), Internet Activities Board, Internet Privacy Task Force, 1992.

[27] Xuejia Lai. On the Design and Security of Block Ciphers. Hartung-Gorre Verlag, 1992.

[28] FIPS 180-1. Secure hash standard, NIST, US Department of Commerce, Washington D.C., Springer-Verlag, 1996.

[29] Miller, V.S.: Use of elliptic curves in cryptography. In Williams, H.C., ed.: Advances in Cryptology — *CRYPTO’85*. Volume 218 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (1986) 417– 426

[30] Koblitz, N.: Elliptic curve cryptosystems. *Mathematics of Computation* 48 (1987)203–209

[31] Rivest, R. L., A. Shamir, L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of ACM*, Vol.21, No. 2, Feb. 1978, pp. 120-126.

[32] V. Rijmen, "Cryptanalysis and design of iterated block ciphers," Doctoral Dissertation, October 1997, K.U.Leuven.