Master of Science in Computer Science and Engineering Thesis:

Fast Flexible Architectures for Secure Communication

Lisa Wu
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
wul@eecs.umich.edu

للاستشارات المنارة

www.manaraa.com

Table of Contents

Acknowledgement 5
Abstract 7
Chapter 1 Introduction 9
Section 1.1 Cryptography 9
Section 1.2 Contribution of This Thesis 11
Chapter 2 The Nature of Cryptography 13
Chapter 3 Cipher Kernel Analysis 17
Section 3.1 Cipher Analysis Tools 17
Section 3.2 Cipher Throughput Analysis 18
Section 3.3 Bottleneck Analysis 19
Section 3.4 Cipher Relative Run Time Cost 21
Section 3.5 Cipher Kernel Characterization 22
Chapter 4 Architectural Extensions 25
Chapter 5 CryptoManiac Architecture 29
Section 5.1 System Architecture 29
Section 5.2 Processing Element Architecture 31
Section 5.3 Instruction Set Architecture 32
Section 5.4 Design Methodology 33
Section 5.5 The Super Optimizer 35
Section 5.6 Physical Design Characteristics 37
Chapter 6 Performance Analysis 39
Section 6.1 Performance Analysis of ISA Extensions 39
Section 6.2 Performance Analysis of CryptoManiac 42
Section 6.3 System Analysis of CryptoManiac 44
Chapter 7 Related Work 47
Chapter 8 Conclusions and Future Work 49
References 51

المنارة للاستشارات
www.manaraa.com

Acknowledgement

Credit for much of the work described in this thesis belongs to my advisor, Professor Todd Austin, for his insight, guidance, and patience. He provided for an excellent research environment, left me enough freedom to do things the way I thought they should be done, and was always available to discuss ideas and problems.

I would also like to thank my committee members Professor Steve Reinhardt and Professor Gary Tyson for reviewing this document and serving on the defense committee.

Other people that have worked on the CryptoManiac project include Chris Weaver for hardware design and synthesis support, Jerome Burke and John McDonald for earlier versions of ISA extensions code modifications.

www.manaraa.com

Abstract

The growth of the Internet as a vehicle for secure communication and electronic commerce has brought cryptographic processing performance to the forefront of high throughput system design. Cryptography provides the mechanisms necessary to implement accountability, accuracy, and confidentiality in communication. This trend will be further underscored with the widespread adoption of secure protocols such as secure IP (IPSEC) and virtual private networks (VPNs). Efficient cryptographic processing, therefore, will become increasingly vital to good system performance.

In this thesis, we explore hardware/software-design techniques to improve the performance of secret-key cipher algorithms. We introduce new instructions that improve the efficiency of the analyzed algorithms, and further introduce the CryptoManiac processor, a fast and flexible co-processor for cryptographic workloads.

Our first approach is to add instruction set support for fast substitutions, general permutations, rotates, and modular arithmetic. Performance analysis of the optimized ciphers shows an overall speedup of 59% over a baseline machine with rotate instructions and 74% speedup over a baseline without rotates. Even higher speedups are demonstrated with optimized substitutions (SBOX's) and additional functional unit resources. Our analyses of the original and optimized algorithms suggest future directions for the design of high-performance programmable cryptographic processors.

To follow up on these suggestions, our second approach is to design an efficient piece of hardware that runs cryptographic algorithms. We present analysis of a 0.25um physical design that runs the standard Rijndael cipher algorithm 2.25 times faster than a 600MHz Alpha 21264 processor. Moreover, our implementation requires 1/100th the area and power in the same technology. We demonstrate that the performance of our design rivals a state-of-the-art dedicated hardware implementation of the 3DES (triple DES) algorithm, while retaining the flexibility to simultaneously support multiple cipher algorithms. Finally, we define a scalable system architecture that combines CryptoManiac processing elements to exploit inter-session and inter-packet parallelism available in many communication protocols. Using I/O traces and detailed timing simulation, we show that chip multiprocessor configurations can effectively service high throughput applications including secure web and disk I/O processing.

المنارة للاستشارات
www.manaraa.com

Chapter 1 Introduction

Cryptography provides the mechanisms necessary to provide accountability, accuracy and confidentiality in inherently public communication mediums such as the Internet. The widespread adoption of the Internet as a trusted medium for communication and commerce has made cryptography an essential component of modern information systems. The trend towards virtual private networks (VPNs) [15] and secure IP (IPSEC) [3] will further emphasize the significance of cryptographic processing among all types of communication. Security-related processing can consume as much as 95 percent of a server's processing capacity [25]. As demands for secure communication bandwidth grow, efficient cryptographic processing will become increasingly critical to good system performance.

Section 1.1 Cryptography

Cryptography is a Greek word that literally means the art of writing secrets [21]. In practice, cryptography is the task of transforming information into a form that is incomprehensible, but at the same time allows the intended recipient to retrieve the original information using a secret key. Cryptographic algorithms (or *ciphers*, as they are often called) are special programs designed to protect sensitive information on public communication networks. During *encryption*, ciphers transform the original *plaintext* message into unintelligible *ciphertext*. *Decryption* is the process of retrieving plaintext from ciphertext. Two forms of cryptography are commonly used in information systems today: *secret-key ciphers* and *public-key ciphers*. Secret-key ciphers (sometimes referred to as symmetric-key ciphers) use a single private key to encrypt and decrypt as illustrated in Figure 1. Public-key ciphers (or asymmetric-key ciphers) use a well-known public key to encrypt and require a different private key to decrypt. The process may also be reversed to produce what is known as a *digital signature*. Digital signatures authenticate the sender. Since only the person holding the private key knows its value, only that person can create a digital signature that others can decrypt with the public key.

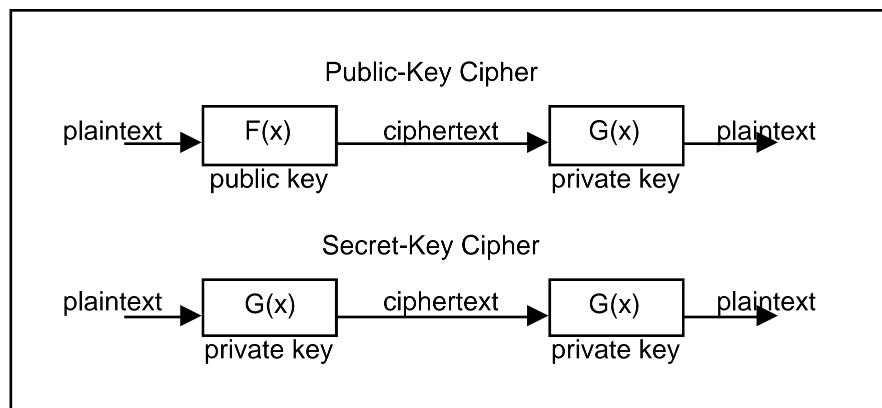

Figure 1. Public-Key and Secret-Key Ciphers.

المنارة للاستشارات
www.manaraa.com

Public-key ciphers have the advantage of being able to establish a secure communication channel without an unsafe exchange of keys. Private-key ciphers, on the other hand, require a shared private key before secure communication can commence. The distribution of the shared private key is the primary obstacle in making secret-key ciphers secure. Strong public-key ciphers are computationally very expensive. Public-key encryption requires exponentiation and modular multiplication of large multi-precision numbers of 1024 bits in length or more. Secret-key ciphers have the advantage of running as much as 1000 times faster than comparable public-key ciphers [30]. To maximize security and performance, most secure protocols use both forms of cryptography. Public key encryption is used at the start of a session to authenticate communicating parties and to securely distribute a shared secret key. The remainder of the session employs efficient secret key algorithms using the private key exchanged during authentication. We refer this type of key management as *public-secret key cryptography*. An example of a system that uses this particular session management strategy is the Secure Sockets Layer (SSL) protocol [39].

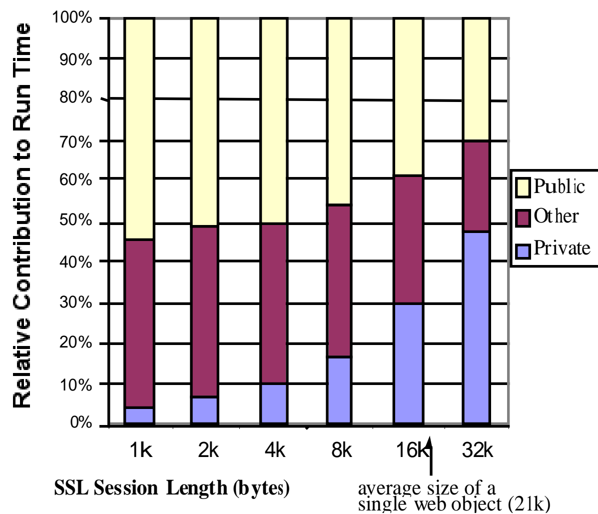

Figure 2. SSL Protocol Relative Contribution to Run Time.

SSL is a standard secure protocol that provides secure communication between web servers and web clients. It is supported by most popular web browsers [25]. SSL extends TCP/IP to support secure encrypted connections with authentication of senders and receivers. The protocol is used by web servers to establish secure HTTP connections. For very short sessions, fast public-key cipher processing is critical for high transaction throughput. For longer sessions, private-key cipher performance becomes more important. Figure 2 illustrates the SSL run-time breakdown by server processing type. Data shown is collected from a heavily loaded web server running on an iA32 platform [29]. Clearly, for very short

www.manaraa.com

sessions fast public-key cipher processing is crucial for high transaction throughput. A recent study [2] found that the average size of a web object was 21k bytes. For a 32k session length, secret-key processing overheads rise to 48% of overall run time. Given that a single session will be composed of many web objects, cryptographic processing will be quickly dominated by secret-key cipher execution. To further reduce the cost of public-key authentication, SSL allows the use of a session cache, where authenticated private keys are held and can later be reused when users reconnect to view other web pages. As such, the focus of our design effort is on improving secret-key cipher performance.

Section 1.2 Contribution of This Thesis

Cryptography can be implemented with software routines, directly in hardware, or a combination of both. A software only approach is the lowest-cost solution but with accordingly lower performance. An example of the hardware-only approach is the IDEA engine [23]. It is targeted specifically at the efficient execution of the IDEA cipher and renders excellent performance. However, its performance is at the expense of flexibility as the hardware cannot be used for other cryptographic processing tasks. In this thesis, we present two hardware/software mixed solutions for efficient cryptographic processing.

The first hardware/software mixed approach is to add architectural extensions that streamline cipher kernel processing. We first examine the execution of eight widely known strong secret-key ciphers. We analyze their performance on detailed microarchitectural models, where we are able to clearly show their performance and the bottlenecks that slow their progress. Armed with these insights, we propose a general set of instructions that from which all these kernels can benefit. The technique improved performance of secret-key ciphers through fast substitutions, general bit permutations, rotates, and modular arithmetic. We re-code the ciphers using these new instructions and then examine their performance on microarchitectural models with varying levels of support for fast cryptography.

We further extended this approach through the design of a fast and flexible cryptographic co-processor. Our design, called the *CryptoManiac*, addresses the primary bottleneck in secret-key ciphers, namely efficiency, through the application of an efficient VLIW architecture with a well-matched instruction set and functional unit resources. The programmable feature supports many secret-key ciphers, in contrast to the IDEA engine. We hand-optimized the eight kernels and then validated the results using a super optimizer as the scheduler with varying design parameters. By combining CryptoManiac processors into parallel configurations, we are able to scale cryptographic performance for applications with inter-session and inter-packet parallelism. We detail the design and implementation of the CryptoManiac processor and analyze its performance using architectural and physical design models.

My research contribution to this work was the design, implementation, and evaluation of the CryptoManaic co-processor; this work was published in [42]. Specifically, I contributed the following: i) design and implementation of the CryptoManiac functional units, ii) area, timing, power, and

المنارة للاستشارات
www.manaraa.com

performance analyses of the CryptoManiac, iii) experiments for design parameter decisions such as the bypass logic of the CryptoManiac, the width of the VLIW co-processor, and how the instruction combining capability affects the design of the hardware, iv) design and implementation of the super optimizer, v) validation and instruction combination studies done by using the super optimizer, and vi) the implications of cryptographic algorithms mixing arithmetic and logical instructions.

We discuss the operations of strong secret-key ciphers by carefully examining the Twofish cipher in Chapter 2. In Chapter 3, we present our experimental framework and analyses of cipher kernel characterizations, operations, and bottlenecks to gain insight as to how to build efficient designs. In Chapter 4, we introduce the architectural extensions to support fast cryptographic processing. In Chapter 5, we present the CryptoManiac design, detailing the processor architecture, the system architecture, and the instruction set. We then examine performance results in Chapter 6. In the final sections, we discuss related work, summarize our findings, and make suggestions for future work.

www.manaraa.com

Figure 3 shows the kernel of the Twofish cipher, developed by Counterpane Systems [35]. It is a candidate for the Advanced Encryption Standard (AES) [1], the US government's effort to develop a new strong encryption standard. Twofish is a particularly good example to look at because it captures many of the operations that ciphers employ. The code shown in Figure 3 is the encryption kernel, run on one 128-bit block of data to encrypt it into a 128-bit ciphertext block using a 128-bit key value. The Twofish decryption kernel is nearly identical except the order of the operations is reversed and inverted (e.g., rotate left becomes rotate right).

```
x[0] = input[0] ˆ key[0] ˆ IV[0];
x[1] = input[1] ˆ key[1] ˆ IV[1];
x[2] = input[2] ˆ key[2] ˆ IV[2];
x[3] = input[3] ˆ key[3] ˆ IV[3];

for (ii=15, jj=0; ii >= 0; ii--, jjˆ=2) {
        t0 =    (sbox1[x[jj][0]] ˆ sbox2[x[jj][1]]
                ˆ sbox3[x[jj][2]] ˆ sbox4[x[jj][3]]);
        t1 =    (sbox1[x[jjˆ1][3]] ˆ sbox2[x[jjˆ1][0]]
                ˆ sbox3[x[jjˆ1][1]] ˆ sbox4[x[jjˆ1][2]]);
        x[jjˆ3] = x[jjˆ3] <<< 1;
        x[jjˆ2] = x[jjˆ2]ˆ(t0+t1+key[ii<<1]);
        x[jjˆ3] = x[jjˆ3]ˆ(t0+(t1<<1)+key[(ii<<1)+1]);
        x[jjˆ2] = x[jjˆ2] >>> 1;
}

output[0] = IV[0] = x[2] ˆ key[0];
output[1] = IV[0] = x[3] ˆ key[1];
output[2] = IV[0] = x[0] ˆ key[2];
output[3] = IV[0] = x[1] ˆ key[3];
```

Figure 3. The Twofish Cipher Kernel. All variables are 32-bit integers. Rotates are indicated by <<< and >>>.

The cipher algorithm first reads the input data, XOR's it with the 128-bit intermediate vector (IV) and key, and then enters the encryption loop. The encryption loop executes 16 iterations (or rounds as they are called in cryptography literature) to produce 128 bits of ciphertext. The ciphertext is then once again XOR'ed with the key and stored to the intermediate vector and the output buffer.

Within the kernel loop, the cipher algorithm employs a series of reversible operations to implement a process called diffusion. Diffusion works to randomly impress upon each of the output bits some information from each of the input bits. The direction of the diffusion process is set by the private key. The more seemingly random and complete the diffusion process is, the more difficult it is to recreate the plaintext without the key value. With large keys and good diffusion, the ciphertext is extremely resistant to attackers. Quantitatively, a strong encryption algorithm is one where any change in the input results in

المنارة للاستشارات
www.manaraa.com

a random perturbation of each output bit with probability 50%. Moreover, any change to the key value should have an equally dramatic effect on the ciphertext produced.

The process of diffusion has two important implications to the underlying machine architecture. First, diffusing input bits is computationally expensive on modern microprocessors. Most algorithms run their kernel loop at least 16 times on each block of data encrypted, successively mixing the data more and more on each round. The second implication is that cipher kernels have little parallelism. Parallelism in the cipher (especially coarse-grained parallelism) would imply that some aspect of the computation does not affect later ciphertext results, which would in turn imply that the cipher algorithm was not a strong one! While we did find a small level of ILP in cipher kernels, the process of making the kernels run fast primarily entails improving their execution efficiency on the underlying microarchitecture. The intermediate vector IV ensures that the diffusion process propagates to all remaining ciphertext in the communication stream. The ciphertext value of encrypted block *i* is first XOR'ed with plaintext block *i*+1 before it is encrypted. The end result is that the cipher kernel execution is a very long recurrence with virtually no parallelism.

The kernel loop also exhibits a mixing of arithmetic and logical operations. Kernels that contain only arithmetic or only logical operations can easily be attacked using linear analysis. The implication of mixing these instructions with sequence as an ADD followed by a XOR or an AND followed by an ADD is to make the cipher resilient to attacks.

To ensure that the ciphertext can be decrypted back to the plaintext, the cipher kernel must employ a series of key-parameterized reversible operations. The Twofish algorithm demonstrates a number of these:

Rotates Rotates are easily reversible (simply rotate the same distance in the opposite direction). Rotates also have good diffusion properties, impressing each bit onto another bit of the output.

Modular Addition Modular arithmetic, if based on a power-of-two base, is cheap, fast, and has relatively good diffusion properties. Moreover, it is easily inverted using modular subtraction or modular addition with the two's-complement of the addend. XOR operations, which are modular additions in base 2, are easily reversible by XOR'ing the same value onto the resulting ciphertext.

Substitutions Table-based substitutions can be used to quickly implement any key-parameterized function. An SBOX is a table of values indexed with plaintext (usually byte) that produces the result of the key-parameterized function. SBOX's are easily reversible by inverting the table, i.e., indices become values and values become indices.

In addition, other algorithms often employ two other mechanisms, modular multiplication and XBOXs.

المنارة للاستشارات
www.manaraa.com

Modular Multiplication Modular multiplication has been shown to have particularly good diffusion properties [23], and the operation can be easily reversed with modular multiplication of the modular inverse of the multiplicand. If the multiplicand is part of the key, all divides (which are typically much more expensive) can be confined to the cipher setup code. If the modulus of the operation is a power-of-two (as in RC6), it can be efficiently implemented using existing multiply instructions. A few algorithms (most notably IDEA) use a modulus of a 2^N+1 prime number. This further improves diffusion properties of the operation at the expense of more computation. Techniques have been developed to efficiently implement 2^N+1 prime modulus operations using only two additional (and parallel) adds plus one multiply [23].

General Permutations General permutations map N bits onto bits with an arbitrary exchange of individual bit values. While trivial to implement in hardware with a wire network (called an XBOX), these permutations are quite expensive to implement in software. Consequently, newer ciphers strictly avoid permutations. We still consider them, however, as they are used in DES [13], the US encryption standard put into practice in the early 1970's and still in wide use today.

Cipher	Key Size	Blk Size	Rnds/Blk	Author	Example Application
3DES	112	64	48	CryptSoft	SSL, SSH
Blowfish	128	64	16	CryptSoft	Norton Utilities
IDEA	128	64	8	Ascom	PGP, SSH
Mars	128	128	16	IBM	AES Candidate
RC4	128	8	1	CryptSoft	SSL
RC6	128	128	18	RSA Security	AES Candidate
Rijndael	128	128	10	Rijmen	AES Standard
Twofish	128	128	16	Counterpane	AES Candidate

Table 1. Secret-Key Symmetric Ciphers Analyzed.

We analyzed the eight secret-key symmetric ciphers listed in Table 1. The table lists for each algorithm the key size used for the experiments, the block size encrypted by each application of the cipher kernel, the number of rounds executed within the cipher kernel, the author of the algorithm, and popular applications that use the cipher. Cipher algorithms typically have three operational parameters: key size, block size, and number of rounds. The key size is the length of the key used to encrypt or decrypt data. The block size is the amount of data processed each time the cipher kernel is invoked. The number of rounds specifies the total number of iterations executed by the cipher kernel loop. 3DES has a key size of 112 bits, block size of 64 bits, and 48 rounds. Rijndael has key size of 128 bits, block size of 128 bits, and

المنارة للاستشارات
www.manaraa.com

10 rounds. The remaining kernels use at least 128 bits of key data. Each algorithm is generally considered a strong algorithm, having undergone review and aggressive cryptanalysis. A strong cipher has high resilience to the efforts of an unrelated party attempting to determine the original content of an encrypted message. Four of the ciphers, i.e., 3DES [13], Blowfish [11], IDEA [23], and RC4 [34], are algorithms used in popular software packages. 3DES runs the US DES standard encryption algorithm [11] serially three times with three 56-bit keys. This is the mode of operation specified in the Secure Sockets Layer (SSL) protocol specification [39]. Rijndael [10] runs the new US AES standard encryption algorithm selected by The National Institute of Standards and Technologies (NIST). The remaining algorithms, i.e., Mars [7], RC6 [33], and Twofish [35], are second round candidates for the Advanced Encryption Standard (AES) [1]. Many of the AES algorithms will likely emerge as high-quality encryption algorithms that will see use in popular applications and protocols.

The baseline implementation of each algorithm is quite efficient. We obtained optimized implementations of each of the AES finalists from the inventors of the algorithms. 3DES, Blowfish, and RC4 were all developed by Eric Young of CryptSoft [11]. CryptSoft's code is quite efficient, as a result, it has found its way into many popular software systems including SSH, OpenSSL, FreeBSD, and the Mozilla web browser. The optimized IDEA implementation was provided by Ascom, inventors of the algorithm. Operation of the algorithms can be tailored significantly, e.g., number of rounds, block size, and key size. We configured each algorithm as suggested by the inventors to maintain good strength and performance. 3DES was configured as per the SSL specification [39]. All ciphers were run in chaining-block-cipher (CBC) mode. In this mode, the value of cipher text block i is XOR'ed with plaintext block i+1 before it is encrypted. Nearly all applications use CBC mode as it produces ciphertext that is more resistant to attacks.

المنارة للاستشارات
www.manaraa.com

Chapter 3 Cipher Kernel Analysis

Successful cryptographic co-processor design requires a thorough understanding of the nature of private key cryptography, its essential operations, and its inherent bottlenecks. In the following sections, we describe the cipher analysis tools used and we analyze cipher throughput and bottlenecks.

Section 3.1 Cipher Analysis Tools

Performance analysis was performed with the SimpleScalar Tool Set version 3.0 for the Alpha architecture [5]. The SimpleScalar tool set [6] includes detailed microarchitecture simulators that can be tailored to reveal the bottlenecks within a program. We used the SimpleView visualization framework to optimize the performance of the cipher kernels. The SimpleView viewer displays graphically the stalls experienced by instruction as they pass though the modeled pipeline, making it fairly straightforward to identify the bottlenecks that slowed cipher kernels.

All baseline codes were compiled with the Compaq Alpha CC compiler (version 5.9) with full optimization and EV6 architecture optimizations (e.g., byte and word loads). All hand-coded versions of the algorithms were based on assembly outputs from the Compaq CC compiler with the same optimizations. All analyzed codes (baseline and optimized) were validated by running the optimized encryption kernel with the original decryption kernel (and vice versa).

We analyzed program performance on the detailed timing simulator (sim-outorder). The timing simulator executes user-level instructions, performing a detailed timing simulation of an aggressive 4-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Our baseline simulation configuration models a future generation out-of-order processor microarchitecture. The processor has a large window of execution; it can fetch and issue up to 4 instructions per cycle. It has a 256 entry re-order buffer with a 64-entry load/store buffer. Loads can only execute when all prior store addresses are known. There is an 8 cycle minimum branch misprediction penalty. The baseline processor has 4 integer ALU units, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 7 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The processor we simulated has 32k 2-way set-associative instruction and data caches. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with 2 ports. The data cache is pipelined to allow up to 2 new requests each cycle. There is a unified second-level 512k 4-way set-associative cache with 32 byte blocks, with a 12 cycle hit latency. If there is a second-level cache miss it takes a total of 120 cycles to make the round trip access to main memory. We

المنارة للاستشارات
www.manaraa.com

model the bus latency to main memory with a 10 cycle bus occupancy per request. Address translation is implemented a 32 en-try 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

Section 3.2 Cipher Throughput Analysis

Figure 4 shows the performance of the cipher kernels executing on a 600MHz Alpha 21264 workstation (Alpha 21264), the baseline microarchitectural model (4W) implemented using SimpleScalar simulators [5], and the upper bound dataflow performance of the algorithm (DF). Dataflow performance was measured on the performance simulator using perfect branch prediction, infinite window size, unlimited fetch bandwidth, perfect memory address disambiguation (e.g., loads never wait for unrelated stores even if their addresses have not yet been computed), and unlimited functional unit resources. The dataflow experiments represent the maximum performance that the original code can achieve. Encryption performance is shown in bytes per 1000 clock cycles executed. This is a convenient metric because it represents the encryption performance in MB/s on a 1GHz machine.

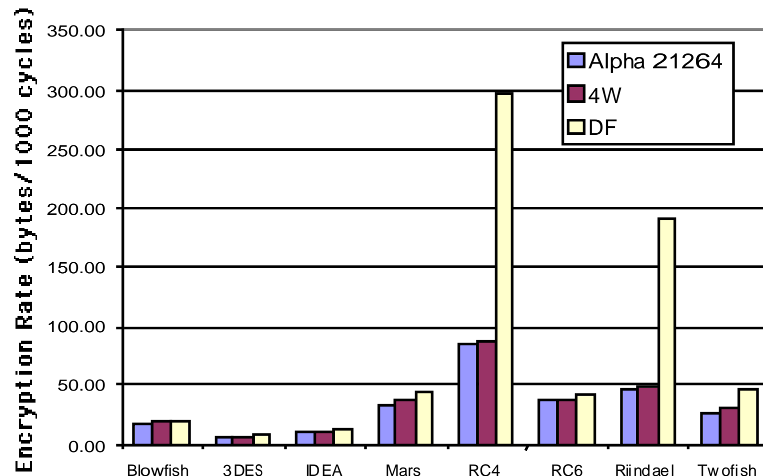

Figure 4. Cipher Performance Comparison Between Alpha 21264, Microarchitecture Model, and the Dataflow Machine.

We felt it was important to validate the performance of the baseline SimpleScalar model so we ran the identical code on a 600 MHz Alpha 21264 workstation (running Tru64 Unix) with a 1GB main memory. We loosely based our simulator baseline on the Alpha 21264 microprocessor, setting the resource configuration and their latencies to values published by Compaq [22]. The correlation in absolute performance was quite close, all except Mars and Twofish were within 10% of the actual machine tests. Mars was 11% faster, Twofish was 15% faster. It's difficult to assess why these discrepancies exist as many of the details of the Alpha 21264 microarchitecture have not been disclosed. We feel they are likely

www.manaraa.com

due to the 21264's clustered microarchitecture, which is not modeled by SimpleScalar. The clustered microarchitecture has slightly higher forwarding latency for some instructions; this would account for the slightly better performance of the algorithms when running on the simulator. However, performance trends were indeed captured, thus we are fairly confident that our performance models are representative of real hardware.

As shown in Figure 4, the baseline 4-wide superscalar processor model performance (4W) varied dramatically. The worst performance was given by 3DES. 3DES is computationally very complex, and it contains operations (e.g., general permutation) that do not map well to a general-purpose microprocessor. It is interesting to note that a 1 GHz processor running the 3DES kernel (a common encryption mode for secure web transports) would produce a maximum throughput of 7.32 MBytes/s, not even enough throughput to saturate a trailing edge 100 MBs Ethernet link, and barely enough to saturate a low-cost T3 communication line. IDEA also turned in a poor performance, the primary bottleneck in this cipher is numerous 7-cycle multiplies.

The AES standard candidates have much better performance with Rijndael leading the pack at 48.51 bytes/1000 cycles. The best performance overall was delivered by RC4 at 88.16 bytes/1000 cycles, more than 10 times the performance of 3DES. RC4 benefits from significantly more parallelism than the other algorithms. The algorithm is essentially a key-based random number generator that XOR's a random sequence onto the input stream. The iterations of the random number generator are (mostly) independent, al-lowing its execution to fully saturate a wide machine, resulting in very high-bandwidth encryption.

The last experiment (DF) shows the scalability of cipher kernel performance. In these experiments, the original code is executed on a dataflow machine. We found these results quite surprising, Blowfish, IDEA, and RC6 are running within 20% of dataflow machine performance. There is slightly more headroom for Mars and Twofish, with potential speedups of 29% and 76%, respectively. RC4 and Rijndael are the outliers, these codes have ample parallelism and could be sped up with more capable hardware.

Section 3.3 Bottleneck Analysis

Figure 5 illustrates the factors that slow down the cipher kernels with performance headroom. Results are only shown for the algorithms that were not running at dataflow performance in Figure 4. The graph shows the performance impact of inserting a single bottleneck into the dataflow machine execution. The resulting performance impact indicates the extent to which the bottleneck is fully exposed during execution, independent of all other bottlenecks. Note that if a bottleneck affects the dataflow machine, it may not help the performance of the baseline machine if it is removed. *All* of the exposed bottlenecks may have to be removed before performance improvements are seen in the baseline machine. The most

www.manaraa.com

important aspect of these analyses is that bottlenecks that do not affect the dataflow machine will not (and cannot) affect the performance of the baseline machine.

Six bottlenecks are analyzed. The *Alias* bar shows the impact of stalling loads in the pipeline until all earlier store addresses have been resolved (i.e., no address aliases). *Branch* shows the effects of mispredictions, *Issue* shows the impact of reducing issue width. *Mem* shows the impact of introducing a realistic memory system, and *Res* gives the impact of limited functional unit resources. *Window* shows the impact of a limited-size instruction window, and *All* shows performance of the machine with all bottlenecks enabled.

It is interesting to note that branch mispredictions and data memory performance do not impair the performance of any ciphers. This observation is in stark contrast to other benchmarks commonly studied in the computer architecture literature. Branch mispredictions are not a problem for these codes as branches are quite predictable, usually found in kernel loops. Cache misses rarely occur as the algorithms read one memory value and then compute with it, often for hundreds of cycles. In addition, our memory system has a next-line prefetch capability which eliminates virtually all data cache misses in the cipher kernel.

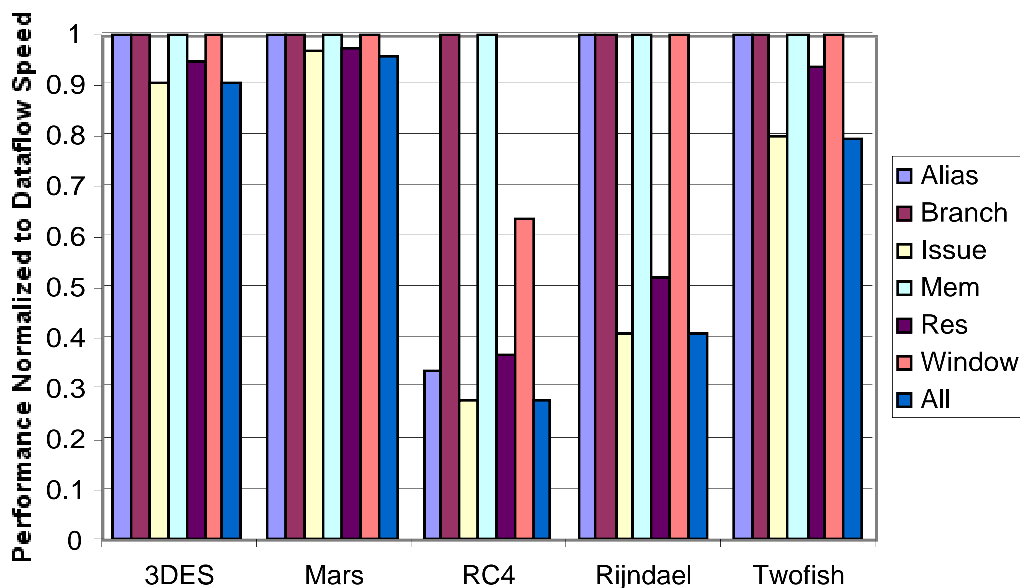

Figure 5. Cipher Kernel Bottleneck Analysis.

A limited window size also has little affect on any of the codes, except RC4. RC4 has significant parallelism between rounds of the kernel cipher. The other algorithms do not, especially when configured in chaining block mode (as they were for these experiments). While there is some parallelism, it is fairly local for all the algorithms except RC4. A similar trend can be seen for address aliases. All the algorithms (except RC4) only access memory to update intermediate vectors and read input data, as a result, all loads

المنارة للاستشارات
www.manaraa.com

are dependent on the previous stores. Having perfect alias detection does little for these codes as they still end up waiting on all previous stores. This is not the case for RC4, however, as this code performs many stores to an internal table used for key-based random number generation. Depending on the input stream, stores in the previous round of the algorithm could be dependent, however, the probability of this is 1/256 (assuming good diffusion). In the dynamically scheduled microarchitecture, these unknown (and mostly independent) stores stall later loads and reduce the overall throughput of the algorithm.

As shown in Figure 5, introducing these stalls has a significant affect on dataflow machine performance. The most prevalent factors leading to poor performance are insufficient issue bandwidth and lack of function unit resources, with Rijndael and RC4 having the largest impacts. Consequently, ciphers with performance headroom will benefit most from additional resources and issue bandwidth. For the benchmarks running at near dataflow speeds, we will have to rely on latency reduction of individual operations to improve their performance.

Section 3.4 Cipher Relative Run Time Cost

When attacking a bottleneck, it is important to accurately discern what part of the code is creating the bottleneck. We can best spend optimization resources by focusing on this part of the algorithm. Each of the ciphers is composed of three major components: setup, encryption kernel, and decryption kernel. The setup code processes the key to create various SBOX's and key-based permutations required by the cipher kernel. In addition, many of the algorithms pre-compute lookup tables (based on the key) that speed processing in the cipher kernel. The encryption and decryption kernels process one block of data.

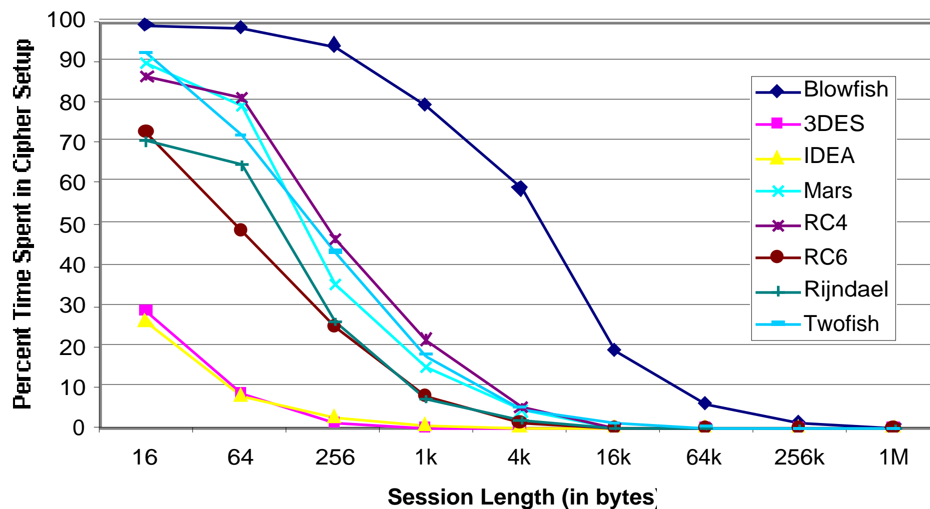

Figure 6. Setup Cost as a Function of Session Length.

المنارة للاستشارات
www.manaraa.com

Figure 6 shows the relative cost of cipher setup compared to the encryption kernels. (The decryption kernels were omitted as their cost is identical to encryption). Costs are measured in run time for varied session lengths. Since setup code is called only once per session, longer sessions better amortize setup cost. As shown in the figure, setup costs for 3DES and IDEA are quite small, even for 16 byte sessions. 3DES's setup times are small due to its costly encryption kernel. IDEA, on the other hand, is designed to have very low-cost startup. The next group, Mars, RC4, RC6, Rijndael, and Twofish all have moderately sized setup costs, with overheads dropping well below 10% at session lengths of 4k or greater. The clear outlier is Blowfish, which only sees setup costs below 10% for sessions longer than 64k bytes. Blowfish runs the encryption kernel on the 128-bit key 520 times before commencing encryption 8k bytes of input data, requiring much longer sessions to amortize setup.

Given the dominance of cipher kernels in overall performance (even for Blowfish), we focused our attention on the cipher kernels for the remainder of our analyses. For all remaining experiments, we use a session length of 4k bytes.

Section 3.5 Cipher Kernel Characterization

To gain a deeper understanding of the cipher kernel operations executed, we profiled each kernel and classified individual operations into nine categories. Their operations are classified into general arithmetic, logical, multiplication, rotate, memory operations, substitutions, permutations, moves, and branches. Figure 7 shows the breakdown by category of all dynamic operations executed. All cipher kernels studied exhibit varied but similar characteristics. Substitution operations implement a key-based transformation function using a byte-indexed array called an "SBOX". Permutation operations rearrange the bit values using a key-parameterized network called an "XBOX". Arithmetic operations are primarily additions and address computations. Multiplication operations include regular multiplication and modular multiplication operations. Logical operations primarily consist of AND and XOR operations. It is interesting to note that very few fundamental operations are used to implement these ciphers. In addition, many of these fundamental operations, e.g. XORs, make inefficient use of the processor clock cycle, creating opportunities for more efficient designs.

As shown in Figure 7, the algorithms can be broadly divided into two categories: those that rely on arithmetic computation, and those that rely on substitutions. IDEA and RC6 are computational algorithms relying heavily on multiplies to diffuse input data bits. Blowfish, 3DES, Rijndael and Twofish, on the other hand, rely heavily on SBOX's to translate input data to ciphertext. The former groups will benefit from more computing resources (especially multiplies) and from faster operations (e.g., rotates). The latter group will benefit from increased memory bandwidth and faster memory accesses for SBOX translations. Besides improving the efficiency of kernel operations, it is possible to speed up an algorithm using value prediction.

المنارة للاستشارات
www.manaraa.com

Value predictors produce values that break dependencies between instructions. As dependencies are removed, instruction level parallelism and program performance increases. For example, if a value predictor could predict the input values to the cipher kernel rounds, it would be possible for kernel rounds to execute in parallel, resulting in significantly more cipher throughput. To test this possibility, we instrumented our microarchitecture model with an infinite-sized last value predictor [24] and used it to predict the results of all instructions in each cipher kernel. The most predictable dependence edge in any of the cipher kernels was predicted correctly only 6.3% of the time! Clearly, diffusion works to transform data in unpredictable ways, eliminating the possibility that value speculation might be useful in improving cipher performance.

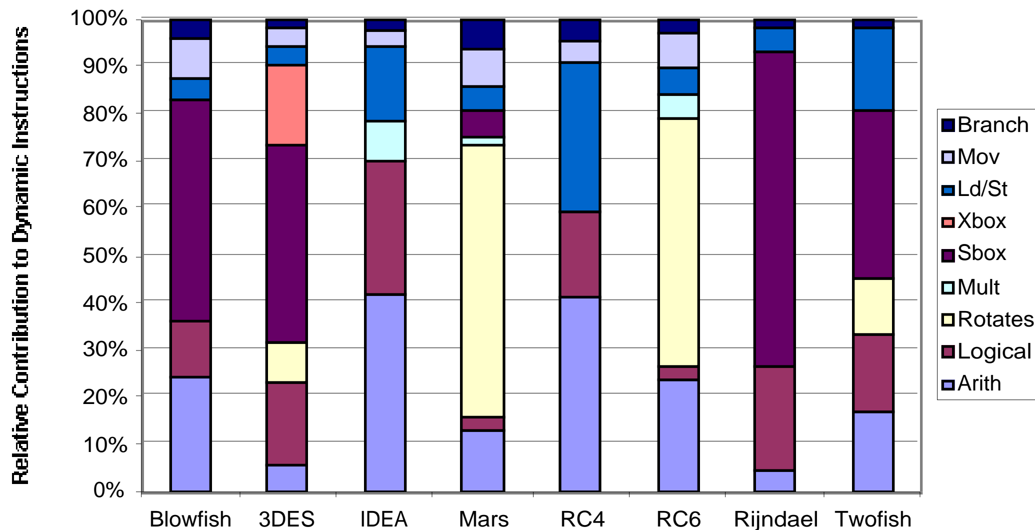

Figure 7. Breakdown of Cipher Operations.

To summarize the characterization of these ciphers, they employ few fundamental operations with varied latency. Many of the algorithms have little or no parallelism. The algorithms that do have performance headroom would require more issue bandwidth and function unit resources to execute faster. None of the programs have branch or memory bottlenecks, and most are not slowed down by memory aliases. Given these characteristics, it seems the ideal architecture for fast cryptographic processing is a scalable design that efficiently executes the fundamental operations of cryptography. We implemented two possible solutions that provide efficient cryptographic processing. The first possible solution is to add architectural extensions for the Alpha processor. The other solution is a co-processor that is designed specifically to run all cryptographic algorithms efficiently. These implementations are presented in detail in Chapter 4 and 5 respectively.

www.manaraa.com

Chapter 4 Architectural Extensions

Figure 8 lists the additions we made to the instruction set to support fast execution of symmetric secret-key ciphers. A number of important considerations drove the design of these instructions. First, all instructions are limited to two register input operands and one register output. While having three register inputs would provide significantly more opportunity to combine low-latency operations, we felt the resulting impact of a third input operand was too great to consider for a simple instruction set enhancement. Adding a third register operand would increase the number of read ports on the register file by 50%, which would subsequently increase its cycle time. In addition, more bandwidth would be required from the register renamer as well; slowing the renamer down could potentially slow the entire pipeline.

Second, we carefully considered the impact on cycle time each new instruction could potentially create. Baseline functional units and modified functional units were specified in structural Verilog, synthesized using Cascade EPOCH synthesis tools, and timing analyses were performed using SPICE for a 0.25u MOSIS TSMC process. Finally, we worked to develop a small set of canonical operations that could be broadly applied to many cipher algorithms.

Rotates (ROL and ROR) are fully supported, for 64 and 32-bit data types. The Alpha architecture does not support rotate instructions, as a result, the addition of rotates saves 4 instructions (and 3 execution cycles). Nearly all the ciphers (except IDEA and Rijndael) have a fairly frequent usage of rotates.

The ROLX and RORX instructions support a constant rotate of a register input, followed by an XOR with another register input, and the result replaces the second register input. This was the only consistent opportunity we found to combine operations; all other combinable operations required at least three input operands. While this instruction does require three inputs, the third is a constant (within the instruction), as a result, there is no impact to the register files or renamers. The ROLX and RORX instructions are useful in speeding up Mars and RC6. Timing analyses indicated that these instructions easily fit in the cycle time of a same-sized ALU.

The MULMOD computes the modular multiplication of two register values modulo the value 0x10001. This is a fairly fast operation, we use the algorithm detailed in [23]. The implementation requires a 16-bit multiply, followed by two 16-bit (parallel) additions and then two levels of MUX'ing. Timing analyses indicate the operation can complete in just over three cycles, we conservatively estimate that the operation can complete in four ALU cycles.

المنارة للاستشارات
www.manaraa.com

```
MULMOD          <srca>,<srcb>,<dest>
          REG[<dest>] <- (REG(<srca>) * REG[<srcb>]) % 0x10001
ROL       <srca>,<srcb>,<dest>
          REG[<dest>] <- REG[<srca>] <<< (REG[<srcb>] & 0x3f)
ROR       <srca>,<srcb>,<dest>
          REG[<dest>] <- REG[<srca>] >>> (REG[<srcb>] & 0x3f)
ROLX      <src>,#<rot>,<dest>
          REG[<dest>] <- (REG(<src>) <<< #<rot>) ^ REG[<dest>]
RORX      <src>,#<rot>,<dest>
          REG[<dest>] <- (REG(<src>) >>> #<rot>) ^ REG[<dest>]
SBOX.#<tt>.#<bb>.<aliased>       <table>, <index>, <dest>
          REG[<dest>] <- MEM32[(REG[<table>]&~0x3ff) | (((REG[<index>]>><bb>*8)&0xff)<<2)]

          The SBOX instruction extracts byte #<bb> (0..3) from register <index> and concatenates
          the resulting 8-bit value with register <table> to produce a 32-bit aligned Sbox address.
          The 32-bit value at the SBOX address is loaded (zero-extended) into <dest>.

          NOTE: stores are not visible within the SBOX until an SBOXSYNC instruction is executed,
          unless the <aliased> flag is indicated. The table designator (#<tt>) may be any value,
          however, performance of the SBOX instruction may be improved if each SBOX table has
          associated with it a different table (#<tt>) value.

SBOXSYNC.<tt>

          SBOX #<tt> is synchronized with memory. Once this instruction is executed, stores to the
          SBOX memory since the last SBOXSYNC will become visible to later SBOX instructions.

XBOX.#<bbb> <srca>,<srcb>,<dest>
          REG[<dest>] = 0
          for (i=#<bbb>*8, j=0; i < #<bbb>+8; i++, j++)
                    REG[<dest>].bit[i] <• (REG[<srca>] >> (REG[<srcb>] >> j*6)) & 1
          The XBOX instruction performs a partial general permutation of register <srca>, given the
          bit permutation map in register <srcb>. The result of the permutation is placed in register
          <dest>.
```

Figure 8. Architectural Support for Secret-Key Ciphers.

The SBOX instruction speeds the accessing of substitution tables. The instruction restricts SBOX's to 256-entry tables with 32-bit contents. As shown in Figure 9, a SBOX is accessed by concatenating the upper bits of the table virtual address with eight bits extracted from the index register. The access returns from memory a 32-bit table value. The cipher algorithms studied all use 256 entry SBOX's, with either 32 or 8 bit entries. We implemented 8 bit entries by zeroing the upper 24 bits of each SBOX table entry. Other SBOX table orientations could be efficiently implemented with these instructions as well. Smaller SBOX's could replicate SBOX entries, thereby creating a don't-care bit in SBOX byte index. Larger

المنارة للاستشارات
www.manaraa.com

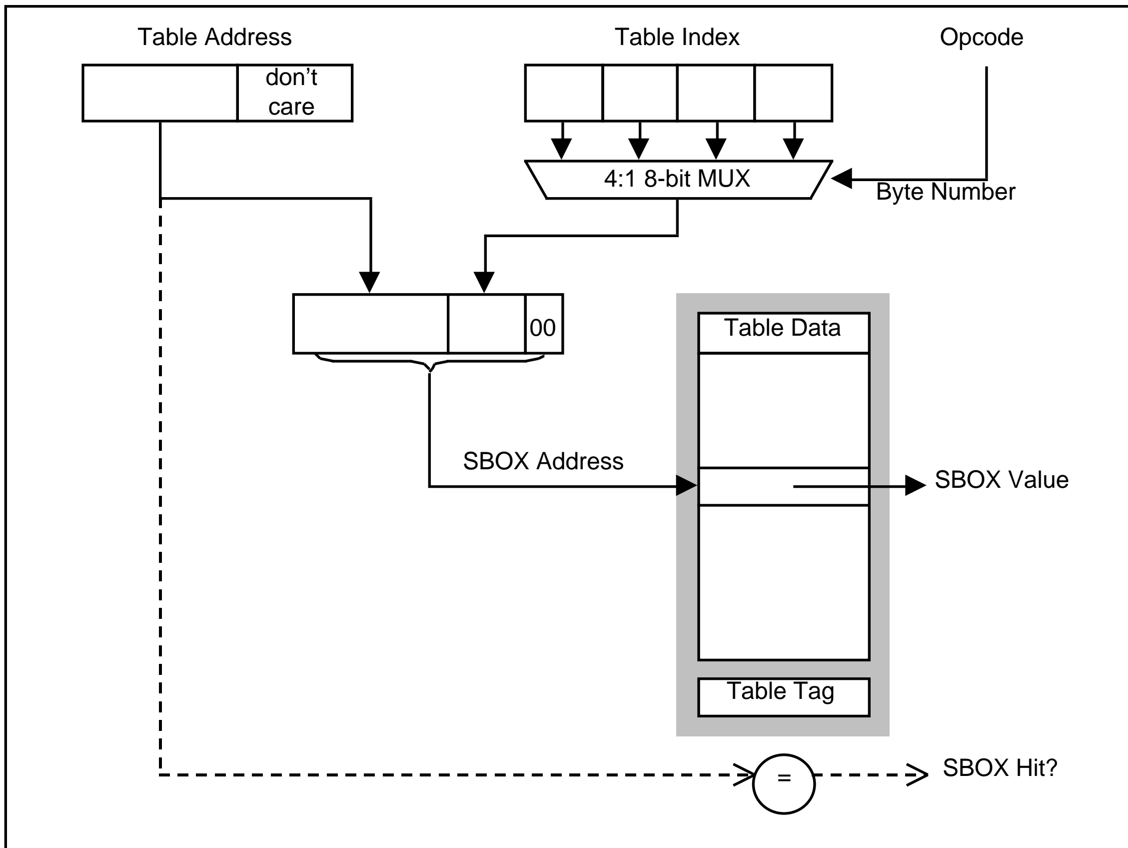

Figure 9. SBOX Instruction Semantics.

SBOX's could be implemented by striping the table across multiple architectural tables and selecting the correct value based on the upper bits of the larger table index.

As shown in Figure 8, the SBOX instruction eliminates address generation (which takes a full cycle on the baseline machine). This is accomplished by restricting that all SBOX tables be aligned to a 1k byte boundary. SBOX address calculation then simplifies to zero-latency bit concatenation. To speed most SBOX operations, stores to SBOX storage are not visible by later SBOX instructions until an SBOXSYNC instruction is executed. This optimization eliminates the need for SBOX instructions to snoop on store values in the processor core. Moreover, SBOX implementations are possible that have separate storage that need not be kept coherent with cache memory. It was fairly straightforward to identify the locations to place SBOXSYNC instructions - always at the end of key setup routines which generated the key-based SBOX entries. Notably, RC4 stores into its SBOX table. To support this algorithm, we added an alias bit to the SBOX instruction, which if set allows later SBOX instructions to observe earlier store values. We implemented this form of the SBOX instruction by treating it as a load with optimized address generation.

المنارة للاستشارات
www.manaraa.com

We explored two SBOX implementations: a simple cache-based implementation and a dedicated SBOX cache. The simple implementation produces the SBOX storage address and then sends the memory request to a data cache memory port. If the SBOX aliased bit is not set, SBOX instructions may execute in any order. As a result, these SBOX instructions need not enter the memory ordering buffer (the device that implements out-of-order load/store execution). The SBOX instructions simply enter the cache pipeline when a free port is available. With this implementation, SBOX instructions complete in 2 cycles, much faster than the 4 cycles required to implement SBOX accesses with load instructions.

Our more aggressive SBOX implementation adds four SBOX caches to the microarchitecture. SBOX caches have a single tag (the table base address), making them a one line sector cache [16]. Each SBOX cache sector is 32-bytes in length (one data cache line). As shown in Figure 9, SBOX addresses are sent to the specified SBOX cache. The table indicator in the SBOX instruction allows the programmer to "schedule" the SBOX caches, specifying which cache contains a particular table. As a result, the underlying implementation need not implement a 4-ported 4k byte cache, but rather four faster single-ported 1k byte SBOX caches. The instruction scheduler directs SBOX instructions to the correct SBOX cache based on the instruction opcode table specifier. The SBOX cache is virtually tagged, thus TLB resources are only required on misses. When the virtual tag does not match, the SBOX cache is flushed and the touched sector is fetched from the data cache. When the SBOXSYNC instruction is executed, all sector valid bits are cleared forcing subsequent SBOX instructions to re-fetch SBOX data from the data cache. On a task switch, the SBOX cache is flushed by invalidating its tag. No writeback is necessary as SBOX caches are read-only.

The XBOX instruction implements a portion of a full 64-bit permutation. The operation takes two input registers. One register is the operand to permute; the other register is a permutation map that describes where each input operand bit is written in the destination. The permutation map contains eight 6-bit indices, each indicates which bit from the input operand will be written in the output. The XBOX instruction opcode indicates which of the eight bytes in the destination register are permuted. The 32-bit permutations in the 3DES algorithm can be completed in 7 instructions (and executed in 3 cycles), a significant improvement over the baseline code which requires 39 instructions.

المنارة للاستشارات
www.manaraa.com

Chapter 5 CryptoManiac Architecture

In order to reach better cipher performance, efficiency becomes the goal of our design. In pursuing our goal, the design will focus on a simple microarchitecture, an efficient implementation of operations, and more efficient use of the clock cycle. CryptoManiac is a 4-wide 32-bit Very Large Instruction Word (VLIW) machine with no cache and a simple branch predictor. Since kernel dependencies through registers and memory are well described, a static VLIW scheduler suffices. The lack of branch bottlenecks eliminates the need for a complex branch predictor. A simple Branch Target Buffer (BTB) can correctly predict nearly all branches. We chose not to include a cache structure because code and data sets fit comfortably in a small static RAM. We employ a triadic (three input operands) ISA that permits combining of most cryptographic operation pairs for better clock cycle utilization. Finally, the CryptoManiac processing elements can be combined into chip multiprocessor configurations for improved performance on workloads with inter-session and inter-packet parallelism.

Section 5.1 System Architecture

Figure 10 details the high level architecture of the CryptoManiac (CM) processor. The host processor interfaces to the CM through the input (InQ) and output (OutQ) request queues. Cryptographic processing requests are inserted into the InQ by a host processor over a connecting bus. The *request scheduler* distributes host processor requests, in the order received, to CM processing elements. The CM processing elements service requests from the InQ, placing any results produced in the OutQ for the host processor. We envision that the input and output queues would be accessible by the host processor via a standard bus interface, such as a PCI bus. It is sufficient to have one input queue for many CM processing elements, as the computationally intensive nature of cryptographic processing limits the bandwidth requirements on this interface.

The CM processing elements block waiting on a request from the host processor. When a request is dispatched to the CM processor, it uses to the request code to initiate the correct handler function. When CM processing is complete, the tagged result of the computation is pushed into the OutQ for reception by the host processor. The host processor requests are tagged with a unique ID that can be used to identify requests as they complete and exit the OutQ. The unique ID permits requests to complete out of order as may be the case with varied processing demands on CM processors. Also contained in the CM request is a session identifier that names a unique communication channel being processed in the CM.

The CM requests specify an operation for the CM to perform on the incoming data. Operations include:

المنارة للاستشارات
www.manaraa.com

- Create a private key session. This request specifies the cryptographic algorithm, operating mode (e.g., electronic codebook *vs.* chaining mode), and the private key of the session. Algorithm setup is performed during this request, creating key-specific substitution tables.
- Delete a private key session. This request releases all storage associated with a session.
- Encrypt/Decrypt data. The requested data is processed and the resulting ciphertext or plaintext is returned in the result packet.

Additional administrative requests are supported that allow the host processor to initialize CM processor memory and redirect execution of individual CM processors.

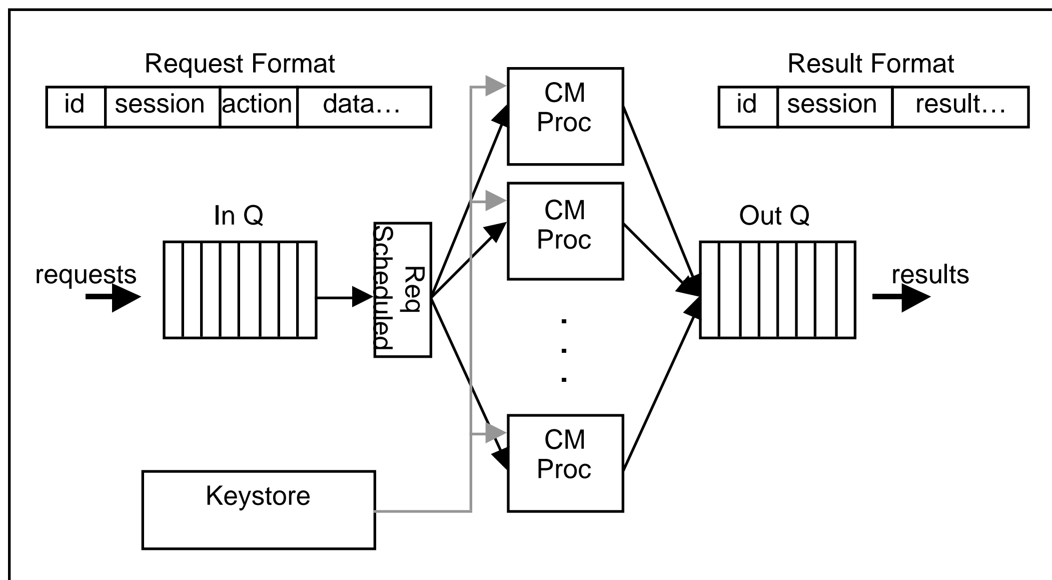

Figure 10. High-level Schematic of the CryptoManiac Architecture.

Requests arriving for CM processing are dispatched from the InQ to CM processing elements by the request scheduler. Requests are distributed first to a free CM processor. If multiple CM processors are free, the request is dispatched to the CM processing element that most recently processed a request in the same session. If there are no free CM processors in the same session, the request is assigned to the least recently used CM processor. Directing requests to CM processors in the same session reduces the setup time to service the request, and when multiple CM processors are free but not in the same session, the least recently used CM processor is likely to contain an unneeded session context.

The *keystore* is a high-density storage element that contains key-specific storage such as key data and substitution tables. The keystore permits the CryptoManiac to process simultaneous sessions on the same CM processor by storing key-specific data in the shared keystore. When a new context is loaded into a CM processor, key specific data is transferred over a high performance interface to internal CM storage. This data includes substitution data, permutation counters, and other internal algorithm state, at most 5k

www.manaraa.com

bytes for any of the algorithms implemented. Key setup is quite expensive for many algorithms, thus performance is greatly improved by having a convenient place to store key-specific data. The keystore is an optional component to the CryptoManiac design, it is only required when multiple sessions must be serviced simultaneously. In single session applications, such as virtual private networks and secure disk processing, the keystore is not required.

Section 5.2 Processing Element Architecture

The CM processing element is a simple 4-wide 4-stage VLIW processor as shown in Figure 11. Each cycle the pipeline fetches a single statically scheduled VLIW instruction word that contains four independent instructions (or NOPs if independent instructions are not available). These four instructions access the register file in parallel while decoding. In the execute stage, instructions perform up to four parallel functional unit operations. In the writeback stage of the pipeline (WB), the results of the CM instruction are written back to the register file.

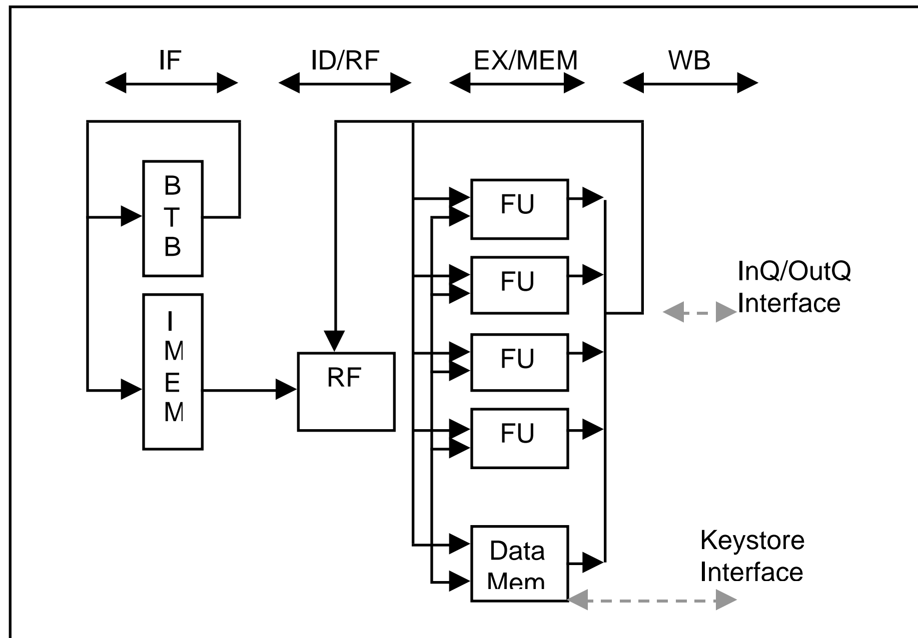

Figure 11. High-level Schematic of CryptoManiac Processing Architecture.

The front end of the CM pipeline contains a simple branch target buffer (BTB) used to predict branch targets. The BTB contains the target address of a branch, any instruction that hits in the BTB is considered a taken branch. When a VLIW word contains multiple branches, the BTB always makes a prediction based on the last taken branch. Most branches in cipher codes are trivial to predict as nearly all branches are taken branches at the end of cipher kernel loops. Instruction memory is accessed in parallel with the BTB, returning a VLIW instruction word at the end of the processor cycle. Due to the small working set size of cipher algorithms, a very small BTB with 16 entries suffices. Moreover, the

المنارة للاستشارات
www.manaraa.com

instruction memory need not be large. 1K bytes is sufficient to hold any of the cipher algorithms. If key setup codes are kept off-chip, for example, by running setup codes on the host processor, many cipher kernels could be stored in a single 1K instruction memory.

During instruction decode, instructions access the register file. To support instruction combining, the register file supports three operand reads and one write per cycle per instruction. In the EX/MEM stage, instruction operations are executed, including loads and stores. Data memory need not be large, since key tables are stored in SBOX caches within the functional units (detailed in Section 5.4); a 4K-byte data SRAM suffices for all the algorithms we implemented.

The execute stage includes four 1K-byte static RAM SBOX caches, used to speed up substitution operations. Each SBOX cache contains a 1K byte page-aligned substitution table. The alignment restriction reduces address generation to a single bit-wise concatenation of the table address and the table index. The details of this design are described in Chapter 4.

Section 5.3 Instruction Set Architecture

Figure 12 gives a brief overview of the CryptoManiac instruction set architecture. Instructions are 32 bits in length. Each instruction contains three input registers and one output register. Three input operands are required to take advantage of the instruction-combining feature. In a conventional microarchitecture, regardless of the latencies of instructions, each instruction takes one or more clock cycles to complete. This is the case for even very low latency instructions such as XORs and ANDs. Earlier analyses revealed a high frequency of these low latency instructions, resulting in inefficient use of the processor clock cycle. Further analyses of the cipher kernels reveal that arithmetic operations are often followed by logical operations. This property is endemic to cipher algorithms because the mixing of linear and non-linear operations prevents attacks using simple linear analysis. We can leverage this property to better utilize the processor clock cycle by combining arithmetic and logical operations within a single cycle.

bundle := <inst><inst><inst><inst>
inst := <op pair><dest><operand1><operand2><operand3>
operation pair := <short><tiny>|<tiny><short>|<tiny><tiny>|<long><nop>
tiny := <xor>|<and>|<inc>|<signext>|<nop>
short := <add>|<sub>|<rot>|<sbox>|<nop>
long := <mul>|<mulmod>

Examples:

Instruction	Expression
Add-Xor R4, R1, R2, R3	R4 <- (R1+R2)⊗R3
And-Rot R4, R1, R2, R3	R4 <- (R1&R2)<<<R3
And-Xor R4, R1, R2, R3	R4 <- (R1&R2)⊗R3

Figure 12. CryptoManiac ISA in CNF form.

المنارة للاستشارات
www.manaraa.com

Our *instruction-combining* architecture divides operations into three classes: *tiny*, *short*, and *long*. Tiny operations include all logical operations and sign extension; short operations include arithmetic operations, rotates, and substitutions; multiplies are classified as long operations. Function units contain datapath networks that allow any pairs of tiny operations or short/tiny operations to execute together in a single cycle. Therefore, we combine general arithmetic instructions with logical instructions, substitutions with logical instructions, and finally rotate operations with logical instructions. Timing analyses indicate a multiplication operation can complete in under three cycles. Modular multiplication operations are implemented using one regular 16-bit multiplication followed by two 16-bit parallel additions and two levels of MUX'ing. This algorithm is derived from the Chinese remainder theorem detailed in [28]. Modular multiplication can be completed in just under three cycles.

Section 5.4 Design Methodology

The main loops of the cipher kernels are hand-optimized for the CryptoManiac instruction set. Hand-optimization of the kernels includes selection of instruction combinations and placement of instructions within VLIW instruction words. Instruction combining was performed by analyzing the known data dependencies of each kernel loop and pairing 3-input short-tiny, tiny-short, or tiny-tiny instruction combinations for minimal cycle counts. Instruction schedules are generated by analyzing kernel dependence graphs such that instructions on the critical path are executed as early as possible. We generated kernel schedules for a variety of processor widths and with/without instruction combining. We used the 4-wide instruction-combining model as our baseline model. Three more designs were evaluated in detail, including a 3-wide VLIW with combining (3WC), a 2-wide VLIW with combining (2WC), and a 4-wide VLIW without combining (4WNC). We validated our hand schedules with a super optimizer that given a kernel dependence graph as an input can generate all possible schedules for a given architecture, keeping only those with the best performance and lowest resource requirements. In a few cases, we were able to improve upon the earlier hand schedules.

Encryption Kernel Cycle Counts (per round)					
	Alpha	4WC	3WC	2WC	4WNC
Blowfish	9.58	4	4	6	5
3DES	23.56	7	8	9	12
IDEA	91.95	14	14	17	15
Mars	28.86	9	9	9	10
RC4	11.49	8	8	8	9
RC6	23.24	7	7	7	9
Rijndael	33.84	9	11	17	10
Twofish	37.36	7	8	11	8

Table 2. Optimization of Kernel Loop Cycle Counts.

www.manaraa.com

Table 2 lists kernel cycles per round for each original (Alpha only) and optimized cipher. The kernel cycles per round for the Alpha experiments are fractional because the code is dynamically scheduled, therefore instructions from different rounds can overlap in the same cycle. The number of cycles per round is equal to the total number of cycles to encrypt a block divided by the number of rounds. An example schedule of the Blowfish kernel for a 4-wide combining architecture is illustrated in Figure 13.

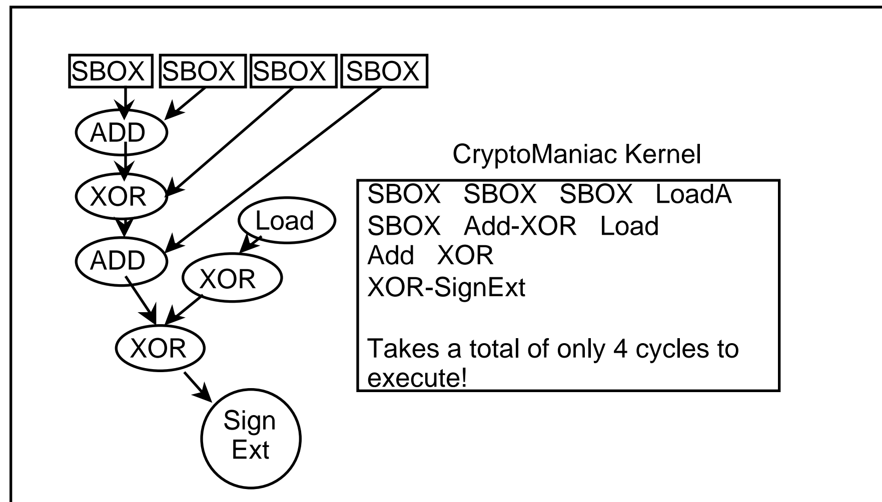

Figure 13. Blowfish Dependence Graph and Scheduled VLIW Code.

As shown in Table 2, all optimized kernels take fewer cycles than the original Alpha to execute. This result does not necessarily mean better performance because we have not yet considered the frequency at which the CryptoManiac can operate. A performance metric that combines both clock cycle time and kernel cycle counts is discussed in Chapter 6.

To gauge the cycle time of the design, a Verilog hardware model was built. The execution stage of the VLIW machine, along with full-crossbar bypass logic, and input/output queues was written in Verilog HDL. We used Synopsys logic synthesis tools [38] to evaluate design timing, area, and power. The synthesis tool accepts Verilog HDL blocks and synthesizes them according to timing constraints given. The design component library uses a 0.25um standard cell library to predict the final timing and area. To assure optimal clock speed, timing constraints were tightened in 0.25ns intervals until the synthesis failed. A 3% clock skew and interconnect wire delays were modeled as well.

Our baseline model requires four functional units to support 4-wide issue with instruction combining. As shown in Figure 14, each functional unit consists of two logical units, one adder, one 1k-byte SBOX cache, and one rotator. Only two of the four functional units contain a multiplier since none of the kernels require more than two multiplies per cycle. The logical unit can perform a XOR or AND operation as specified by the opcode of the combined instruction. Two logical units are available to provide the

www.manaraa.com

flexibility of logical operations at either the beginning or the end of the processor clock cycle. The rotator is implemented using a barrel shifter.

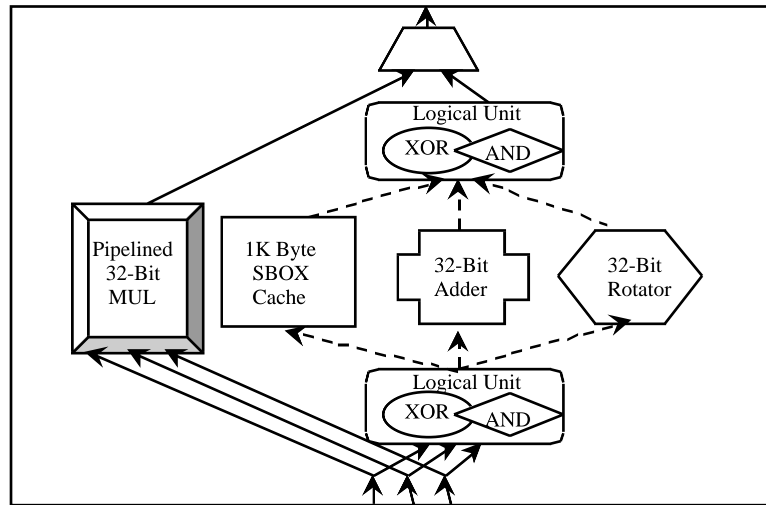

Figure 14. High-level Schematic of a Single Functional Unit.

SBOX cache timing and area analysis was performed with Cacti 2 [9], a tool for estimating memory components. A 1k byte cache is roughly 0.3mm X 0.3mm, which is 0.09 mm^2 in 0.25um technology. Timing analyses indicated that the SBOX cache latency was not on the critical path of the function unit.

We explored both full and half crossbar configurations for data bypass in the execute stage. We quickly discovered that the performance impact of using a half crossbar was too great (up to 40% slower) due to many additional move instructions needed to transfer values between unconnected function units. This is not surprising given the nature of cryptographic kernels, where bit-diffusion operations require communication between all functional units.

Section 5.5 The Super Optimizer

The super optimizer, the VLIW scheduler for CryptoManiac is created for the purpose of verifying hand-optimized cryptography kernel results and to understand the utilization of instruction combining. Detailed understanding of instruction combining gives us insight as to eliminate unnecessary hardware inside each functional unit of the CrytoManiac. It also automated generation of optimized kernels for the various CM architecture studied. The pseudo code for the super optimizer is shown in Figure 15.

The super optimizer takes the disassembly code of the cipher kernels and instruction dependencies as inputs, and produce an output with which instructions are executed at which clock cycle according to the machine width specified. The optimizer first compiles a list of *ready instructions*, instructions that have all their operands available for use, then take the ready list and check for any instruction combinations. After checking for instruction combination, if the number of ready instructions exceeds the machine

المنارة للاستشارات
www.manaraa.com

width specified, the super optimizer tries all combinations exhaustively. Without any pruning mechanism, a kernel such as Rijndael, which has up to 16 instructions ready at once, could take a significant amount of time to run through. As such, we kept track of the minimal cycle count needed to execute the kernel and prune any other combinations that do not meet the minimum.

We performed experiments of the cryptography kernel cycles scheduled with all combinations, partial combinations, or no combinations. All combinations includes short-tiny, tiny-short, and tiny-tiny combinations. The results show that the most important combining instruction type is the short-tiny instructions. This indicates that it might be cost-effective to only implement a logical unit after the arithmetic unit but not before. Encryption rates (performance) vs. area were studied in conjunction with this data, however, the area saved by not having a logical unit before the arithmetic unit did not improve performance significantly. Therefore, we decided to use the logic-arithmetic-logic approach for CryptoManiac functional unit design as described in previous sections.

```
instruction_list = NULL;
instruction_list = Load_Instruction(Blowfish);
Schedule(unscheduled_inst_list, cycle_number) {
        ready_list = NULL;
        expanded_ready_list = NULL;
        if (cycle_number > MIN_cycle_scheduled)
                return;
        Get_ready_list(unscheduled_list, ready_list, expanded_ready_list);
        if (number_of_ready_instructions > width of VLIW) {
                Try_All_Combinations(unscheduled_list, expanded_ready_list);
                Update_Unscheduled_List(unscheduled_inst_list);
        }
        else {
                Update_Unscheduled_List(unscheduled_inst_list);
        }
        cycle_number++;
        if (unschedule_inst_list != NULL)
                Schedule(unscheduled_list, cycle_number);
        else {
                if (cycle_number <= MIN_cycle_number)
                MIN_cycle_number = cycle_number;
                return;
        }
}
for (i = 1 to MIN_cycle_number)
        Print_Inst_Scheduled(i);
```

Figure 15. Pseudo Code for the Super Optimizer.

Another study made possible was to increase the width of the VLIW. Several kernels can take advantage of an 8-wide VLIW because there are more than 4 ready instructions at a particular cycle for scheduling. A good example is Rijndael, it increased performance by 33% from the 4-wide configuration

www.manaraa.com

when run on an 8-wide CryptoManiac. Only special case studies of 8-wide 3DES and Rijndael will be presented in next chapter.

Section 5.6 Physical Design Characteristics

Table 3 shows the timing, area, and power consumption results for various configurations. A 4-wide CryptoManiac with instruction combining has an estimated clock cycle of 2.78ns, yielding a CryptoManiac processor that runs at 360MHz! The same configuration has an area of 1.39mm X 1.39mm (1.93mm^2), which is roughly 1/100th the size of a 600MHz Alpha 21264 processor (200mm^2) in the same technology. The average estimated power consumption of the 4-wide combining model running at 360MHz with a V_{dd} of 2.1V is 606mW. The Alpha processor running at 600MHz dissipates 75W. The power consumption results, although an estimate, seem reasonable because the relative power with CryptoManiac running at 600MHz is 1.01W (1/75th of the Alpha 21264), which is proportional to the area differences.

The chip area of the CryptoManiac co-processor is likely over-estimated. Synthesized designs tend to be larger than their hand-optimized counter parts, and the Cacti 2 area estimates are known to be larger than physical designs we have constructed in the past. An experienced design team could likely produce a smaller and faster design.

Timing and Area Estimates for Various CryptoManiac Design Configurations				
	4W Combining	3W Combining	2W Combining	4W NoComb
Timing Result	2.78 ns	2.66 ns	2.54 ns	2.76 ns
Area Result	1.39mm X 1.39mm	1.33mm X 1.33mm	1.26mm X 1.26mm	1.3mm X 1.3mm
Power Result	606.37 mW	593.51 mW	568.50 mW	586.86 mW
Synthesis	3 ns	3 ns	3 ns	3 ns
Critical Path	byps-lgc-add-lgc	byps-lgc-add-lgc	byps-lgc-add-lgc	adder

Table 3. Timing and Area Results for CryptoManiac.

المنارة للاستشارات
www.manaraa.com

Chapter 6 Performance Analysis

Section 6.1 Performance Analysis of ISA Extensions

We hand coded optimized versions of each cipher kernel, and then examined their performance on four microarchitectures, ranging in cost and performance. Table 4 lists the four microarchitectures studied. The 4W microarchitecture is a typical high-performance four-issue microarchitecture with moderately sized memory system and resources. It is roughly modeled after the Alpha 21264 microarchitecture. In the 4W model, there are four ALUs, two data cache ports, and two Rotate/XBOX units that implement rotates and general permutations. SBOX instructions access the cache memory, thus they must compete with loads and stores for cache access ports. The 4W model also supports optimized multiplication. Word-sized (32-bit) multiplies have an early out after 4 cycles. In addition, modular 16-bit multiplies (modulo 0x10001) are implemented in hardware in 4 cycles. The 4W model can initiate one 64-bit multiply, two 32-bit multiplies, or two 16-bit modular multiplies per cycle. This resource configuration could be implemented inexpensively by mapping the shorter multipliers onto the 64-bit multiplier hardware. For example, a Wallace tree based multiplier can be converted to multiple shorter precision multipliers by simply isolating portions of the Wallace tree with MUXes.

	4W	4W+	8W+	DF
Fetch Speed	1 block/cycle		2 blocks/cycle	infinite
Window Size	128		256	infinite
Issue Width	4		8	infinite
IALU Resources	4 (1 cycle)		8	infinite
IMULT/MULTMOD	1-64 (7 cycles)/2-32 (4 cycles)		2-64/4-32	infinite
D-Cache Ports	2 (2 cycles)		4	infinite
SBOX Caches	0	4 single port (1 cycle)	4 dual ported	infinite
Rotator/XBOX	2 (1 cycle)	4	8	infinite

Table 4. Microarchitecture Models.

The 4W+ microarchitecture improves the performance of SBOX instructions, reducing their latency and providing more bandwidth to SBox values without interfering with cache memory accesses. We added four SBOX caches, each a 1k single-line sector cache. This configuration support four simultaneous accesses to four different SBOX tables in a single cycle (four times the bandwidth of the 4W configuration). When a reference is made to an SBOX the tag is checked to see if it contains the referenced table, and then the valid bit of the sector is checked. If the sector is valid, the 32-bit referenced value is returned, otherwise, the SBOX sector data (one data cache line) is demand fetched from the data cache. SBOX storage does not observe stores to the data cache until an SBOXSYNC instruction is executed (which invalidates the SBOX tag). The 4W+ configuration also adds two more Rotate/XBOX

www.manaraa.com

units, thereby doubling the number of rotates and XBOX instructions that can be executed in a single cycle.

The 8W+ model provides approximately twice the execution bandwidth of the 4W+ model. This model doubles the issue width to eight, doubles the number of execution resources, and adds two more data cache ports. To accommodate the additional execution resources, the front-end performance is scaled to fetch two blocks per cycle, and the instruction window size is doubled to 256 instructions to expose more ILP. We don't claim that it would be wise to construct such a machine, we examine this configuration simply to demonstrate the headroom in performance if more resources were made available for cipher kernel execution. Finally, model DF is our dataflow model, described in the previous section. The maximum achievable performance of the re-coded kernels is given by these experiments.

Figure 16 shows the performance of the optimized codes running on these four processor models. All performance measurements are shown as speedups (in total cycles to process a 4k session) normalized to the performance of the original code with rotates running on the baseline microarchitecture with rotate instructions. Many architectures have fast rotates, so we felt that normalizing to this target was a more fair assessment of our instruction set extensions. For machines without rotate instructions, e.g., Alpha-based processors, speedups are even more impressive!

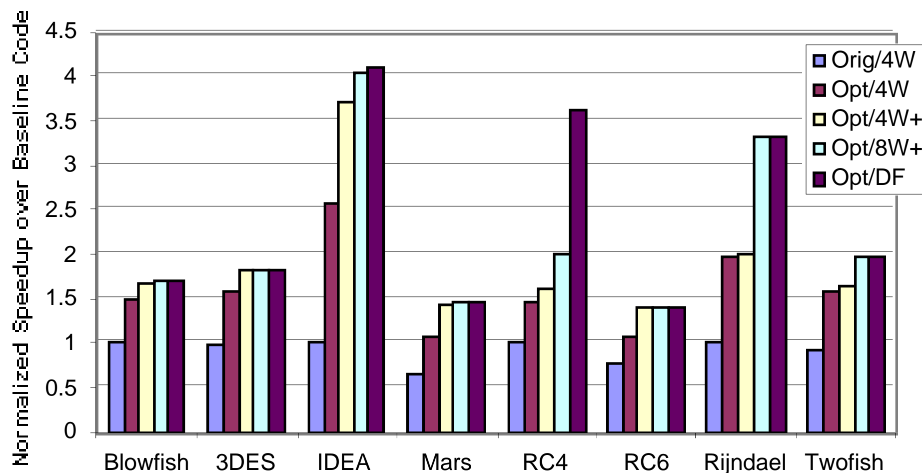

Figure 16. Relative Performance of the Optimized Kernels.

The first bar, labeled Orig/4W, shows the performance of the original code (without rotate instructions) compared to the performance of the cipher kernels with rotate instructions. This experiment shows the impact on performance that an architecture will experience if it does not support rotates. Without a rotate instruction, rotates are synthesized using three instructions for a rotate by a constant amount (2 cycles to execute), and four instructions for a rotate by a variable amount (3 cycles to execute). The algorithms that most heavily use rotates, namely Mars and RC6, saw significant slowdowns of 40%

www.manaraa.com

and 24%, respectively. The "lowest hanging fruit" for architects to gather here are rotates. These simple instructions have little impact on the cost or cycle time of a machine and provide good speedups on three of the ciphers. Intel processors based on the P6 microarchitecture (PII, PIII) have good rotate performance. Measurements on a PIII found that the machine could sustain one rotate per cycle continuously. However, Intel recently announced that shifts and rotates on the Willamette microarchitecture would be at least twice as expensive as addition [31].

The second bar in Figure 16, labeled Opt/4W, give the performance improvement of the fully optimized (hand coded) cipher kernels running on the 4W model. Even on the less expensive microarchitecture, speedups for these new kernels is quite impressive. The kernels saw an average performance improvement of 59%, with IDEA seeing the best overall improvement of 159%. IDEA benefited from the faster and higher bandwidth modular multiplication support, an operation it uses frequently. Rijndael also saw very good speedups, with performance almost doubling. Rijndael benefited mostly from reduced latency for SBOX accesses. With SBox support in hardware, these accesses reduce from three instructions to one, and speedup from five cycles to two.

Blowfish, 3DES, RC4, and Twofish all saw speedups near 50%. Like Rijndael, Blowfish, RC4 and Twofish benefited mostly from improved SBox access latency. 3DES saw benefits from improved SBox access latency and fast XBOX permutations. The outlier in these experiments was RC6. RC6 received most of its benefits with rotates; on the 4W microarchitecture it does benefit from fast modular multiplication, but only slightly.

The third bar in Figure 16, labeled Opt/4W+, gives the performance improvements with additional SBOX resources and rotator/XBOX units. Earlier results suggested that Rijndael and Twofish could benefit from more rotator and SBOX resources, however, in these experiments they have both saturated the machine issue resources and thus cannot leverage more resources. Both ciphers are running at nearly 4 IPC in the 4W machine, additional resources are only useful if the microarchitecture can issue more than 4 instruction per cycle.

Finally, the fourth (Opt/8W+) and fifth (Opt/DF) bars show the optimized program performance with double the execution resources and infinite resources. As with the original code experiments, many of the cipher kernels are running near dataflow speed. Blowfish, 3DES, Mars, RC6 could not be sped up any more without reducing the latency of the individual operations. IDEA could benefit only marginally from addition resources. RC4, Rijndael, and Twofish have plenty of ILP to exploit, more resources improved their performance. In all cases except RC4, doubling the execution bandwidth exposes all available parallelism, permitting the ciphers to run at dataflow speed. RC4, on the•other hand, still has a large supply of untapped ILP.

المنارة للاستشارات
www.manaraa.com

Section 6.2 Performance Analysis of CryptoManiac

In Figure 17, we show the performance of the four models studied, plus the original Alpha, and two versions of the ISA extensions studied. A performance metric of megabytes encrypted per second is used to show the encryption performance of each algorithm. The ISA+ model is a 600MHz Alpha 21264-like processor with cryptographic instruction set enhancements. The ISA++ model has the same micro-architecture as the ISA+ model plus four 1k-byte SBOX caches to optimize substitution performance. In the 4-wide combining (4WC) model, there are four functional units, two multipliers, and support for instruction combining. All other configurations are derived from this model. The 3WC and 2WC models reduce the number of functional units to three and two, respectively. These configurations also have two multipliers each. The 4WNC model has four functional units and two multipliers, and it does not support instruction combining. In this design, an XOR instruction would take one cycle to execute, as would an Add instruction. This design benefits from a more efficient register file, since each instruction has only two input operands.

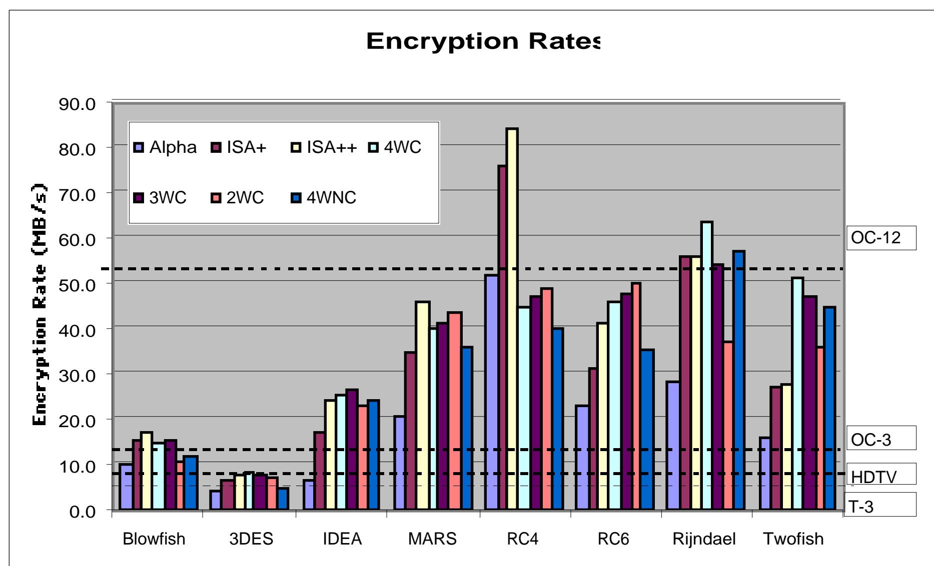

Figure 17. CryptoManiac Encryption Performance.

The encryption rates for the four models we designed are measured using 4K byte sessions with 128 byte blocks. Encryption rates are calculated in MB/s by dividing cycles/byte into the clock period of the hardware model. As shown in Figure 17, we were able to achieve an encryption rate as high as 64 MB/s for Rijndael (the new AES standard) which is 2.25 times faster than the 600MHz Alpha 21264 workstation. All kernels except RC4 gained in performance, ranging from 32% to 290% better than the 600MHz Alpha. RC4 is the only kernel that performs worse than the baseline Alpha processor due to ample aliasing effects described in Section 2. RC4 writes into its key table, creating many ambiguous

المنارة للاستشارات
www.manaraa.com

memory dependencies that lead to poor schedules on the VLIW architecture. Processors that are dynamically scheduled can run RC4 much faster. Nevertheless, the 4-wide combining CryptoManiac configuration ran, on average, 1.2 times faster than the Alpha processor. The Alpha processor with ISA extensions faired much better, out-performing the 4WC CryptoManiac in a few experiments. Keep in mind that the CryptoManiac design is much more cost-effective than a conventional out-of-order processor.

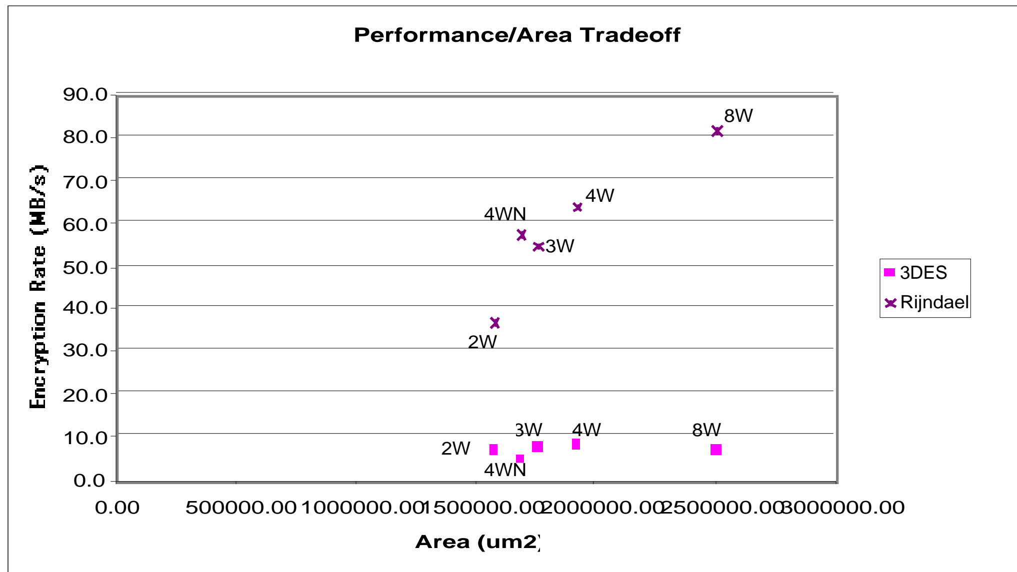

Figure 18. Five Design Models and Their Performance/Area Tradeoff for 3DES and Rijndael.

The dashed lines in Figure 17 represents various real world performance targets. A low-cost T-3 line can be saturated at a speed of 5.375 MB/s. A T-3 line is typically installed as a major networking artery for large corporations and universities with high-volume network traffic. An MPEG-4 HDTV transmits at 8MB/s. HDTV is considered ultra high image compression standard for the telecommunication of digital TV images and is run in one single session. An OC-3 line has the bandwidth of 19.375 MB/s and an OC-12 line has the bandwidth of 77.5 MB/s. All of the kernels running on a CryptoManiac can saturate a T-3 line or an MPEG-4 HDTV line. All but two kernels met the bandwidth requirement for an OC-3 line.

Figure 18 illustrates the tradeoff between performance and area for variety of physical designs. We examined designs from two to eight instructions wide, with and without instruction combining, for the Rijndael and 3DES ciphers. Parallelism exhibited by each algorithm has a direct effect on the encryption rate. MARS, RC4, and RC6 benefited little from additional issue bandwidth. These kernels do, however, run faster with instruction combining. Their performance decreased significantly on the 4-wide non-combining configuration. Rijndael benefits from additional issue bandwidth; an 8-wide configuration is

www.manaraa.com

nearly 30% faster than the 4-wide design. Instruction combining appears to be a beneficial feature, configurations with this capability are more than 10% faster with even smaller increases in area.

Section 6.3 System Analysis of CryptoManiac

There are many potential applications of the CryptoManiac processor. In this section, we examine the performance of the CryptoManiac processor for two applications: secure web server and disk controller. We analyze the performance of general purpose and cryptographic processors using I/O trace-based simulation, measuring the response time for each processor configuration to service requests.

The secure web server experiments were driven by a network traffic trace of the WorldCup 98 official web server (www.worldcup.com), captured during a one-hour period of extremely high traffic [19]. During this one hour period, there was an average of 1971 requests per second with a total transfer rate of 8.75 MB/sec. Thirty web servers were used to service this traffic, we have assembled all the requests into a single trace for this experiment. In addition, this traffic was not secure, so we've transformed it into secure requests by bracketing sessions with an SSL public key authentication [39].

In the network traffic experiments, packets are encrypted and decrypted using Chaining Block Cipher (CBC) mode, as specified by the IPSEC protocol standard [3]. In this mode of operation, the cipher text of the previous encrypted block (128 bits for Rijndael, 64 bits for 3DES) is XOR'ed with the plaintext of the next block before it is encrypted. Chaining blocks increases the strength of cipher algorithms by reducing correlation between the plaintext and ciphertext, at the expense of parallelism. Packet sizes are limited to 1500 bytes, as specified by the IPSEC protocol standard.

The secure disk experiments are driven with a trace of accesses to a 9-disk array of 9.1 GB Quantum Atlas 10K disks [32]. The trace was taken from the DiskSIM disk simulator trace library [14]. During the trace, the disk is heavily loaded, with an average transfer rate of 16.7 MB/sec. The accesses to all the disks are combined into a single trace for the purpose of our analyses. Disk blocks are encrypted using CBC mode encryption in 512 byte encryption units (the minimum disk transfer size).

We examine the performance for single and multiple processor configurations. With multiple processors, network packets can be processed in parallel if they are from different sessions (i.e., different connections from different IP addresses). Within a session, cipher block chaining requires that stream packets be processed serially. For the disk experiments, sector data is processed in chaining block mode, but different disk sector accesses (a sector is 512 bytes) may be processed in parallel with multiple CryptoManiac processors.

For all experiments, we only consider the performance of cryptographic processing, all other processing tasks such as OS, web server, and database operations are assumed to be offloaded to other processor components. We assume that public key authentication (used once at the beginning of each session from a unique IP address) is implemented with two dedicated public key processors. Each public

المنارة للاستشارات
www.manaraa.com

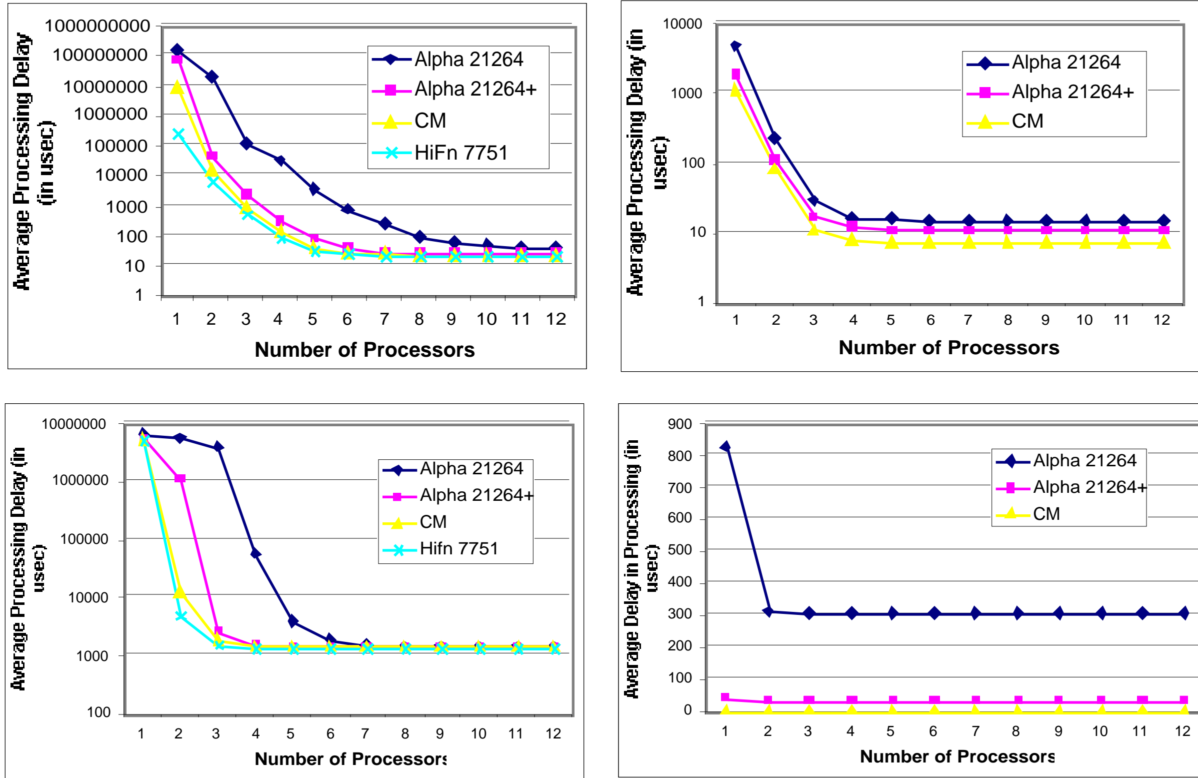

Figure 19. Average Delay in Processing (usec) vs. Number of Processors. Top two graphs are network traces, and the bottom two graphs are Disk I/O traces. Left two graphs are for 3DES and right two graphs are for Rijndael.

key authentication takes 3.2msec, this timing is based on the performance of the HiFn 6500 public key processor [17].

Once authentication completes, the private keys and key tables are stored in the keystore for the entire length of the user session. When a session context is loaded into a processor, for example to process the first packet or to change the session context of a CryptoManiac processor, we assume that the loading of context data takes 720ns. This timing is based on a keystore constructed from a 400 MHz RDRAM [27], which can access key table data in two 40ns accesses (across the rows of 32 banks), plus a 1k byte bus transfer at 16-bits per 10ns plus bus overheads.

Figure 19 graphs the response time of the Rijndael (right) and 3DES (left) ciphers for the secure web server (top) and disk controller (bottom) application. For the 3DES experiments we also show the simulator performance of the HiFn 7751 encryption processor [17]. The 7751 is a high-end encryption processor used in VPN routers and other high bandwidth secure communication applications. The 7751 includes a hardware implementation of the 3DES algorithm capable of encrypting data at 10.375 MB/sec (in IPSEC-compatible CBC mode).

المنارة للاستشارات
www.manaraa.com

As shown in Figure 19, network traffic processing requires more resources than disk I/O processing. This is due to the fact that network I/O processing *a)* has less parallelism due to the chaining of packets within connections, and *b)* switches contexts often necessitating the extra delay of loading key-specific data from the keystore. The disk I/O workload, even though at higher sustained bandwidth, operates within a single context (and thus does not access the keystore). The disk workload also has ample parallelism, since different disk blocks (512 bytes in size) can be processed in parallel on different CM processors.

To keep cryptographic processing overheads for network traffic low, say below 5% for a short 40msec transfer delay, additional transmission delays due to cryptographic processing must be no more than 2msec total, or no more than 1msec (1000usec) on each end of a network transfer. For the 3DES network I/O experiments, an acceptable level of overhead requires at least three CM processor or three 7751 processors. For the Alpha processor experiments, acceptable network delays (with 3DES) require six Alpha 21264 processors or four Alpha 21264 processors with cryptographic ISA extensions (labeled Alpha+). With Rijndael, performance is much better; two processors suffice for all the experiments.

Disk transfers for the Atlas 10K drive average 16msec in length, as a result, overheads can be limited to 5% if the increase in sector transfer latency is no more than 800ns. The disk I/O experiments cannot meet this goal for any 3DES configuration examined due to disk block processing delays. Minimum processing latency was always greater than 800ns. For Rijndael, performance is again much better with all configurations able to service disk I/O with acceptable delay using only a single processor.

المنارة للاستشارات
www.manaraa.com

Chapter 7 Related Work

Most published work on cryptographic hardware has focused on public key ciphers. The most expensive component of these algorithms is modular multiplication of multi-precision (1024-bits or more) operands. Most high performance algorithms are based on the Montgomery method [28]. There have been a number of proposals on how to speed this computation in hardware [41, 18, 40, 4], and Intel has demonstrated that the Merced iA64 processor has particularly good performance for this algorithm [29]. A number of algorithm-specific hardware implementations that have been described are as follows. IBM's original DES proposal described a hardware implementation [13]. Shiva [37], IBM [20], Chrysalis-ITS [8], and Hi/FN [17] all offer high speed hardware implementations of the DES and 3DES algorithms. Published performance numbers for 3DES on these designs range from 8 MB/s to 58 MB/s. Our 3DES algorithm on a 1 GHz processor would achieve a performance of 12 MB/s - clearly there remains value in a hard-ware design for a specific algorithm. The details published on the IBM implementations [43, 44] are particularly interesting as they highlight other challenges that arise when developing a cryptographic processor including random number generation and key protection.

Hardware implementations have also been described for IDEA [23], Twofish [35], and Blowfish [33]. The FPGA research community has also shown that public-key cipher algorithm performance can be improved using FPGA-based implementations [26]. While our approach cannot attain the peak performance that algorithm-specific hardware implementations can attain, our approach does provide the advantage of both performance and flexibility. Using a canonical set of symmetric cipher operations we speed the processing of many algorithms, possibly offering performance improvements for yet-to-be-developed algorithms. Given the wide variety of algorithms in use today, and the need for servers and clients to support different ciphers for different applications (or even connections), we feel that there are benefits to providing instruction set support. We are aware of only one previous proposal to add instruction set support for secret-key symmetric cryptography. Shi and Lee proposed adding an instruction (GRP) that supports efficient software implementations of general bit permutation [36]. Their approach is more efficient than our proposal. For a 32-bit operand they can perform any permutation in 5 instructions, our approach requires 7 instructions. Their approach also scales more favorably to larger operands. We are currently enhancing our tools to use Shi and Lee's GRP instruction, however, we expect the performance impacts of this change to be small as none of our cipher algorithms have general permutation within their kernel loops. 3DES is the only algorithm that uses general permutations (for the initial and final permutations).

www.manaraa.com

Chapter 8 Conclusions and Future Work

The growth of the Internet as the primary vehicle for secure communication and electronic commerce has made efficient cryptographic processing a key factor of good system performance. In this paper, we demonstrated that a hardware-software co-design provides excellent performance while maintaining the flexibility to support new algorithms in the field.

To motivate our design, we analyzed the characteristics of eight secret-key cipher kernels. We showed that they lack branch or memory bottlenecks, have few unknown dependencies, and offer little headroom for performance improvement on traditional architectures. Given these analyses, we proposed new instructions that speed the common operations of symmetric ciphers, and efficient hardware that improve kernel performance. Instruction set support is added for substitutions, permutations, rotates, and modular multiplication. CryptoManiac is an application-specific co-processor that is a 4-wide VLIW machine. We then examine their performance on microarchitecture and hardware models of varying cost and performance. Performance analysis of the optimized benchmarks revealed a 59% speedup over machines with rotate instructions, and a 74% speedup over machines without rotates for the architectural extensions. CryptoManiac was able to run Rijndael 2.5 times faster than the Alpha 21264 workstation with $1/100^{th}$ area and $1/100^{th}$ power of the Alpha processor.

We evaluated different design configurations by building detailed hardware models of varied widths and capabilities. We then calculate encryption rate by synthesizing the models to obtain timing estimates. Our systematic approach allowed us to study the tradeoffs between chip area and performance. We showed that the highest-performing and most cost-efficient design is the 4-wide combining configuration. Rijndael, the new AES standard, runs 2.25 times faster on a 360MHz CryptoManiac. Our analysis of the original and optimized algorithms suggests that there is more opportunity to speed up cryptographic processing. We are considering improved functional unit designs as well as more aggressive circuit implementations.

Our results make a very strong case for the deployment of cryptographic co-processors, however, we believe the results in this paper have stronger implications for the computer architecture community as a whole. With an additional 1% area (for an Alpha 21264 design), we were able to affect a 20% performance improvement over a broad class of cipher algorithms, with individual algorithms benefiting as much as 190%. This is a striking result considering that many commercial design teams use a rule of thumb that any optimization that returns 1% performance improvement for 1% area is a good one. This result is further underscored by the fact that our design is completely synthesized, if the talents of an experienced design team were marshaled to this task, the resulting design would be smaller, faster and lower power.

المنارة للاستشارات
www.manaraa.com

The reason for these striking results is simple - an *application specific processor* design can achieve a level of efficiency that is impossible for general purpose designs to attain. Our application specific design contains none of the baggage necessary to execute non-cryptographic workloads, making the resulting design smaller and lower power. In addition, our limited application domain creates opportunities to optimize the implementation, yielding superior performance results. Going forward, we are working to assess the cost of programmability in the CryptoManiac. A dedicated Rijndael implementation is under development that will be compared to the design presented in this paper. We are going to show the comparison between the cost of hardware programmability (FPGA), software programmability (CryptoManiac), and no programmability (hardware-only version of Rijndael). In addition, we are developing application specific processors for other application domains. Through this work we hope to demonstrate that application specific optimization can be a powerful tool for improving system performance and cost.

المنارة للاستشارات
www.manaraa.com

References

[1] *Advanced EncryptionStandard (AES) Development Effort.* US Government, http://csrc.nist.gov/encryption/aes/.

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

[3] R. Atkinson. Security architecture for the internet protocol. *IETF Draft Architecture ipsec-arch-sec00*, 1996.

[4] T. Blum and C.Paar. Montgomery modular exponentiation on reconfigurable hardware. *Proceedings, 14th IEEE Symposium on Computer Arithmetic*, pages 70-77, 1999.

[5] J. Burke, J. McDonald, and T. Austin. Architectural Support for Fast Symmetric-Key Cryptography. *Proceedings of ASPLOS*, 2000.

[6] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, Unniversity of Wisconsin, Madison, June 1997.

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

[8] *Chrysalis-ITS Corporation.* http://www.chrysalis-its.com

[9] *Compaq Corporation.* http://www.research.compaq.com/wrl/techreports/abstracts/93.5.html.

[10] *Counterpane Systems.* http://www.couterpane.com

[11] *CryptSoft Technologies.* http://www.cryptsoft.com, 2000.

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

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

[14] DiskSIM simulator. http://www.ece.cmu.edu/~ganger/disksim.

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

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Palo Alto, Calif.: Morgan Kaufmann, 1990.

[17] Hi/Fn Corporation. http://www.hifn.com

[18] J.-H. Hong and C.-W. Wu. Radix-4 modular multiplication and exponentiation algorithms for the RSA public-key cryptosystem. *Design Automation Conference (ASP-DAC 2000)*, pages 565-570, 2000.

[19] *Internet Traffic Archive.* http://ita.ee.lbl.gov.

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

[21] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World.* Prentice Hall PTR, 1995.

[22] J. Keller. A superscalar alpha processor with out-of-order exeecution. *9th Annual Microprocessor Forum*, 1996.

[23] X. Laai. *On the Design and Security of Block Ciphers.* Hartung-Gorre Veerlag, 1992.

[24] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[25] M. S. Merkow, CCP, and J. Breithaupt. *The Complete Guide to Internet Security.* AMACOM, 2000.

www.manaraa.com

[26] U. Meyer-Base and R. Watzel. A comparison of DES and LFSR based FPGA implementable cryptography algorithms. *3rd International Symposium on Communication Theory and Applications*, pages 290-298, 1995.

[27] *Micron Corporation*. http://www.micro.com/rdram.

[28] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519-521, April 1985.

[29] Stephen Moore. *Enhancing Security Performance Through IA-64 Architecture.* Intel Corporation, http://developer.intel.com/design/security/rsa2000/itanium.pdf, 1999.

[30] *An Introduction to Cryptography*. Network Associates, Inc., http://www.pgpi.org/doc/pgpintro/, 1999.

[31] *Optimizing Software for the Willamette Architecture.* http://developer.intel.com.

[32] *Quantum Corporation*. http://www.quantum.com.

[33] R. L. Rivest and et al. *The RC6 Block Cipher*. RSA Security, http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6.

[34] *RSA Security*. http://www.rsa.com.

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

[36] Z. Shi and R. B. Lee. Bit permutation instructions for accelerating software cryptography. *Proc. Of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 138-148, 2000.

[37] *Shiva Corporation.* http://www.shiva.com.

[38] *Synopsys*. http://www.synopsys.com.

[39] *The SSL Protocol, version 3.0.* Netscape, Inc., http://home.netscape.com/eng/ssl3/draft302.txt, 1999.

[40] 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.

[41] 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, February 2000.

[42] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A fast flexible architecture for secure communication. *International Symposium on Computer Architecture Conference Proceedings,* July 2001.

[43] P. C. Yeh and R. M. Smith Sr. ESA/390 integrated cryptographic facility: An overview. *IBM Systems Journal,* 30(2), 1991.

[44] P.C. Yeh and R. M. Smith Sr. S/390 CMOS cryptographic coprocessor architecture: Overview and design considerations. *IBM Journal of Research and Development,* 43(5/6), September 1999.

المنارة للاستشارات
www.manaraa.com