A High-Performance Flexible Architecture for Cryptography

R. Reed Taylor[1] and Seth Copen Goldstein[2]

[1] Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, USA
rt2i@ece.cmu.edu,
WWW home page: http://ece.cmu.edu/~rt2i

[2] Computer Science Division, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA
seth@cs.cmu.edu,
WWW home page: http://cs.cmu.edu/~seth

Abstract. Cryptographic algorithms are more efficiently implemented in custom hardware than in software running on general-purpose processors. However, systems which use hardware implementations have significant drawbacks: they are unable to respond to flaws discovered in the implemented algorithm or to changes in standards. In this paper we show how reconfigurable computing offers high performance yet flexible solutions for cryptographic algorithms. We focus on PipeRench, a reconfigurable fabric that supports implementations which can yield better than custom-hardware performance and yet maintains all the flexibility of software based systems. PipeRench is a pipelined reconfigurable fabric which virtualizes hardware, enabling large circuits to be run on limited physical hardware. We present implementations for Crypton, IDEA, RC6, and Twofish on PipeRench and an extension of PipeRench, PipeRench+. We also describe how various proposed AES algorithms could be implemented on PipeRench. PipeRench achieves speedups of between 2x and 12x over conventional processors.

1 Introduction

Most cryptographic algorithms function more efficiently when implemented in hardware than in software. This is largely because customized hardware can take advantage of bit-level and instruction-level parallelism that is not accessible to general-purpose processors. Hardware implementations, lacking flexibility, can only offer a fixed number of algorithms to system designers. In this paper we describe a reconfigurable fabric which delivers high performance hardware implementations with the flexibility of general-purpose processors.

The efficiency of an implementation is directly related to the degree to which it is customized to perform a given task. Hardware implementations are even more efficient when they are customized for a specific instance of an algorithm. For example, a hardware multiplier with one constant operand will generally take much less area than a general-purpose two operand multiplier.

Ç.K. Koç and C. Paar (Eds.): CHES'99, LNCS 1717, pp. 231–245, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Of course implementing circuits with such a high degree of specificity in VLSI is generally infeasible because the cost of development and manufacturing must be offset by the chip's applicability. Furthermore, to be responsive, a system must have some control over its embedded algorithms. For example, if a particular algorithm is discovered to be insecure, the system is rendered useless unless a different algorithm can be implemented. Reconfigurable hardware strikes a balance between customization and performance on the one hand and flexibility and cost on the other hand by permitting any algorithm to be highly customized.

Reconfigurable hardware is a general term that applies to any device which can be configured, at run-time, to implement a function as a hardware circuit. Reconfigurable devices occupy a middle ground between traditional computing devices, e.g., microprocessors, and custom hardware. Microprocessors compute a function over time by multiplexing a limited amount of hardware using instructions and registers. They are thus general-purpose and can compute many different functions. At the other end of the spectrum, custom hardware is used to implement a single function, fixed at chip fabrication time. A reconfigurable device, of which the most common is a Field Programmable Gate Array (FPGA), has sufficient logic and routing resources that it can be configured, or programmed, to compute a large set of functions in space. Later, it can be re-programmed to perform a different set of functions. It shares attributes of microprocessors, in that it can be programmed post-fabrication, and of custom hardware, in that it can implement a circuit directly; avoiding the need to multiplex hardware.

The primary ways in which reconfigurable devices are tailored to an application are by matching application parallelism with as many function units as needed, by sizing function units to the word size of the application, by creating customized instructions, by introducing pipelining, and, by eliminating control overhead associated with the multiplexing of function units as in a microprocessor.

In the next section, we describe how reconfigurable computing devices can achieve the efficiency of highly customized designs while maintaining both cost-effectiveness and security. Section 3 focuses on how the components of typical cryptographic algorithms map to reconfigurable devices. Section 4 describes a pipelined reconfigurable device called PipeRench which overcomes many of the problems of using commercial FPGAs to implement datapaths. In particular PipeRench supports hardware virtualization which, like virtual memory, allows designs that do not fit on the physical device to run. Section 5 describes our implementations of several algorithms on PipeRench and our support of on-the-fly customization even in embedded systems. Related work is covered in Section 6. We conclude in Section 7.

2 Reconfigurable Computing

Functions for which a reconfigurable fabric can provide a significant benefit exhibit one or more of the following features:

1. The function operates on bit-widths that are different from the processor's basic word size.
2. The data dependencies in the function allow multiple function units to operate in parallel.
3. The function is composed of a series of basic operations that can be combined into a single specialized operation.
4. The function can be pipelined.
5. Constant propagation can be performed, reducing the complexity of the operations.
6. The input values are reused many times within the computation.

These functions take two forms. *Stream-based functions* process a large data input stream and produce a large data output stream, while *custom instructions* take a few inputs and produce a few outputs. Notice that cryptographic algorithms possess many of the features described above. They can be implemented as stream-based functions which run completely on a reconfigurable device, or, when impractical to implement completely on the a reconfigurable device, pieces of them can be implemented on the reconfigurable device as custom instructions. After presenting a simple example of a custom instruction to illustrate how a reconfigurable fabric can improve performance, we discuss the ways in which a fabric can be integrated into a complete system.

2.1 Custom Instructions: The q_x Permutation from TwoFish

In Twofish [27], in order to generate the key dependent S-boxes, multiple invocations of the q function are required. This function combines XOR, rotation, bit truncation, and table lookups. One way to accelerate the creation of the key dependent S-boxes is to implement a custom instruction, the q-instruction, on a reconfigurable fabric. This instruction takes an 8-bit operand and produces an 8-bit result. The custom instruction exploits the ability of the reconfigurable fabric to operate on small bit-width operands (4-bits), to execute many operations in parallel, and to combine a sequence of operations into a single operator (through the use of lookup tables).

2.2 A System Architecture

Reconfigurable fabrics enhance performance mainly by providing the computational datapath with more flexibility. Their utility and applicability is thus influenced by the manner in which they are integrated into the datapath. We recognize three basic ways in which a fabric may be integrated into a system: as an attached processor on the I/O or memory bus, as a co-processor, or as a functional unit on the main CPU. They are most widely useful when integrated into the processor as a reconfigurable function unit (RFU). The RFU has access to both the register file and the primary cache. The main reason for this is that the they may be used to implement custom instructions which can operate on data in the processor registers. Furthermore, the bandwidth between the fabric

and the processor (and the data in the processor's cache) is highest when the fabric can directly access the cache. As we will show in the rest of the paper, this organization leads to a system which can significantly enhance the performance of all cryptographic algorithms.

A fourth possible system organization is the system-on-a-chip approach used in embedded computing systems. In such an organization the fabric is closely coupled with a processor, but not so tightly coupled as to be on the processors datapath.

3 Cipher Components

Most ciphers can be specified as dataflow graphs consisting of a few different components. In this section we will enumerate the most common of these components and discuss how they map onto reconfigurable hardware.

- Simple Arithmetic Operations
  Simple operations such as addition and subtraction appear frequently in cryptographic algorithms. These operations map easily to hardware, but due to their simplicity they offer no real gain for reconfigurable systems.
- Narrow and Unusual Bit-widths
  Operations involving narrow bit-widths appear often in stream ciphers, and they are important in highly customized ciphers of any type. Standard microprocessors are notoriously bad at performing narrow bit-width operations, particularly if the values are not multiples of the natural word length of the architecture. Customized hardware supports operations on values of any width, avoiding the computation of unneeded values and the costly masking of undesired bits. Implementing a highly customized design with a constant key allows all datapaths to be reduced to their minimum widths, eliminating the need for paths wide enough to support all possible key combinations.
- Multiplication
  Multiplication is a difficult task to perform in hardware, in that simple hardware multipliers consume a large amount of hardware and compute their results very slowly. Because there are many different ways to improve their performance, multipliers are a prime candidate for optimization and acceleration. Here we consider three different types of multiplier.
  - General-Purpose
    General-purpose multipliers (where both operands may take any value) are costly to implement in hardware. However, in many cryptographic algorithms, the result of nxn multiplies is often only n bits wide. On a reconfigurable device, the size and number of the adders can be reduced accordingly, eliminating the need to compute bits which are later ignored.
  - Multiplication by a Constant
    Implementing highly customized cryptographic hardware, for example when the key has been set to a constant value, can serve to change many (or all) of the general-purpose multipliers in a design into constant multipliers

(multipliers where one operand is a constant). Constant multipliers can be made considerably smaller and faster in hardware than general-purpose multipliers.
Suppose that one operand of a multiplier set to a constant. The multiplier requires only as many partial products as there are 1's in the constant operand. On average, single-operand multipliers of this type are half the size and twice as fast as their general-purpose counterparts.

- Multiplication Using a Redundant Coding Scheme
A great deal of space can be saved when performing constant multiplication through the use of a redundant coding scheme. For example, it is straightforward to transform a constant into canonical signed digit, or CSD, form. CSD vectors reduce the number of partial products needed for multiplication by permitting bits in the constant operand to take on negative values. For example, the number 7 in binary is 0111, or $2^2 + 2^1 + 2^0$. Multiplication by this constant requires three partial products; one for each 1 in the binary representation. The CSD representation of 7, however, is $100(-1)$, or $2^3 - 2^0$. Multiplication by this constant vector requires only two partial products.
As long as addition and subtraction take the same amount of time, no hardware overhead is incurred in implementing this type of multiplier. On average, a constant CSD multiplier will be about 75% smaller than a general-purpose multiplier because the number of partial products in constant CSD multipliers scales with the number of *sequences* of ones in the original constant.

- Parallel Logical Operations
Hardware allows many logical operations to be performed in parallel. This instruction level parallelism is one of the fundamental advantages of hardware over software in computation. Reconfigurable devices can be programmed to perform such complex logical operations in parallel, harnessing all the parallelism available to a hardware implementation. Furthermore, since the number and kind of function units needed at any point in the computation is configured for the application, the parallelism is never artificially constrained by a lack of function units (as might happen in a VLIW architecture, for example).
- Sequences of Logical Operations
Most reconfigurable architectures, including standard commercial Xilinx FPGAs the PipeRench architecture discussed later in this paper, implement function units using lookup tables. Thus a sequence of operators can often be combined into a single operator by setting the lookup-table appropriately.
- Table Lookup
Most block ciphers include a substitution box, or S-box. S-boxes are generally not easily expressible as linear transformations and are therefore implemented as table look-ups. Many reconfigurable architectures can implement tables of this kind, while others may need external scratch memory to store the S-box values.
- Rotation and Shifting
Lastly, two very common operations in cryptography are bitwise shifts and rotations. Microprocessors, particularly if programmed in C, are very inefficient at performing operations of this type.

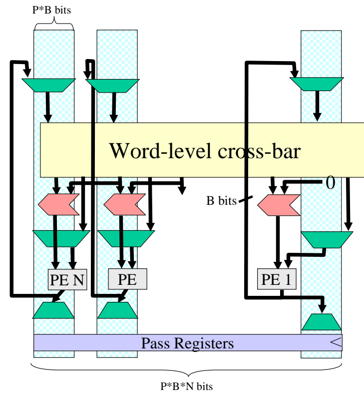
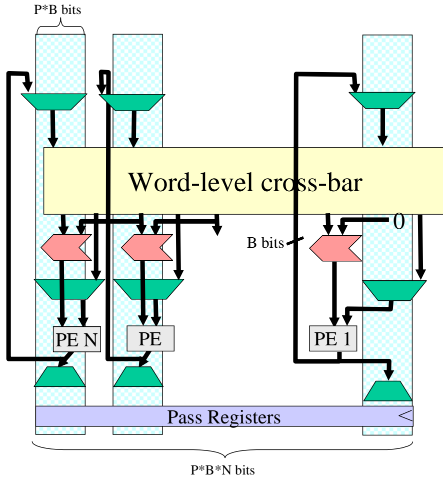


Fig. 1. *Hardware virtualization in PipeRench overlaps computation with reconfiguration and provides the illusion of unlimited hardware resources.*

Hardware, on the other hand, can shift and rotate numbers easily. Variable shifts and rotates can be accomplished with barrel shifters, and constant shifts and rotates do not require any resources at all, as they can be achieved by simply reordering the actual wires.
Reconfigurable hardware can accomplish all of the benefits associated with hardware while providing even more opportunities for optimization. In highly customized designs such as fixed-key implementations, variable shifts and rotations may become fixed, reducing running time and freeing resources.

4 PipeRench

PipeRench is a reconfigurable fabric being developed at CMU. It is an instance of the class of pipelined reconfigurable fabrics [26]. From the point of view of implementing cryptographic algorithms the three most important characteristics of PipeRench are: it supports hardware virtualization, it is optimized to create pipelined datapaths for word-based computations, and it has zero apparent configuration time. Hardware virtualization allows PipeRench to efficiently execute configurations larger than the size of the physical fabric, which relieves the compiler or designer from the onerous task of fitting the configuration into a fixed-size fabric. PipeRench achieves hardware virtualization by structuring the fabric (and configurations) into pipeline stages, or *stripes*. The stripes of an application are time multiplexed onto the physical stripes (see Figure 1). This requires that every physical stripe be identical. It also restricts the computations it can support to those in which the state in any pipeline stage is a function of the current state of that stage and the current state of the previous stage in the pipeline. In other words, the dataflow graph of the computation cannot have long cycles.



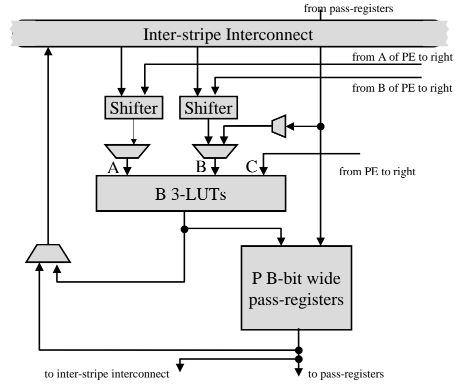


Fig. 2. *The interconnection network between two adjacent stripes. All switching is done at the word level. All thick arrows denote B-bit wide connections.*

Fig. 3. *The structure of a processing element. There are N PEs in each stripe. Details about the zero-detect logic, the fast carry chain and other circuitry are left out.*

Each stripe in PipeRench is composed of N processing elements (PEs). In turn, each PE is composed of B identically configured 3-LUTS, P B-bit pass registers, and some control logic. The three inputs to the LUTS are divided into two data inputs (A and B) and a control input similar to [8]. Each stripe has an associated *inter-stripe interconnect* used to route values to the next stripe and also to route values to other PEs in the same stripe. An additional interconnect, the *pass-register interconnect*, allows the values of all the pass registers to be transferred to the pass registers of the PE in the same column of the next stripe.

The structure of the interconnect is depicted in Figures 2 and 3. Both the inter-stripe interconnect and the pass-register interconnect switch B-bit wide buses, not individual bits. A limited set of bit permutations are supported in the interconnect by barrel shifters, which can left shift any input coming from the inter-stripe interconnect. Currently, the inter-stripe interconnect is implemented as a full crossbar.

From the perspective of cryptographic algorithms, the current version of PipeRench has one significant drawback: It cannot perform large table lookups. Thus S-boxes with more than a few entries cannot be efficiently supported directly in the current version of PipeRench. One proposed extension to PipeRench, PipeRench+, allows the individual stripes to make memory accesses. This would allow PipeRench to efficiently support S-boxes and thus all the operations listed in Section 3. Without the memory extension, algorithms with S-box operations would be best supported by decomposing the algorithm into pieces where

the non-S-box portions are implemented as custom instructions and the S-box lookups are performed in the processor core.

The performance numbers we use in this paper are for an implementation in a 0.25 micron process. After an analysis described in [12] we determined that each stripe will have 16 8-bit PEs, yielding a 128-bit wide stripe. Each PE contains 8 pass registers. The final chip will use 100mm^2 for 28 stripes and an on-chip cache capable of holding more than 512 virtual stripes.[1]

Along with the development of the PipeRench fabric, a fast compilation framework was built [6]. Except where noted, all performance numbers are on simulations of a 28-stripe $N = 16$, $B = 8$, $P = 8$ instance of PipeRench running configurations created automatically by the compiler. For PipeRench+, we consider PipeRench to be augmented by a small scratchpad memory of 1K bytes. We consider two versions, PipeRench+16, which allows up to 16 simultaneous reads, and PipeRench+4, which supports up to 4 simultaneous reads. They increase the total area by 5% to 20%.

5 Applications

In this section we describe how IDEA, Crypton, RC6, and Twofish can be implemented on PipeRench, yielding high performance. We also describe how these algorithms would be aided by PipeRench+. As an example of the flexibility of reconfigurable systems we describe how key-specific instances of IDEA can be easily created on the fly, without a compiler. We then evaluate reconfigurable implementations for the proposed algorithms of AES.

5.1 IDEA

The IDEA block cipher [28] is comprised entirely of three fundamental operations described in Section 3: addition modulo 2^{16}, 16-bit XOR, and 16x16 multiplication modulo $2^{16} + 1$. The 128-bit key is used to generate 52 16-bit subkeys. Throughout the algorithm there are no backwards paths for data. In addition, one operand of every multiplication operation in the algorithm is a subkey, and for a highly-customized implementation it may be treated as a constant.

This means that the algorithm maps exceptionally well onto PipeRench. The forward-only datapath permits the entire application to be constructed as a single, long virtual pipeline. PipeRench is sufficiently wide to receive one complete 64-bit cleartext block and to return one 64-bit ciphertext block per cycle.

Multipliers are the best candidates for optimization in this algorithm. If implemented as general-purpose two-operand shift-and-add multipliers, they require 16 partial products each. The modulo $2^{16}+1$ operation can be packed into one stripe and computed using only three operations [18].

A simple key-specific implementation can be created if the compiler is given the subkeys as constants. The compiler performs constant propagation reducing

[1] First silicon for a prototype of PipeRench implemented in 0.35 micron technology is expected in October 1999. It will have 16 stripes.

Table 1. *Comparison of IDEA implementations.*

Processor	Clock Speed	Clocks per Block	Throughput (MBytes/sec)
PipeRench (template)	100 MHz	6.3	126.6
PipeRench (compiler)	100 MHz	12	66.3
Pentium-II using MMX [21]	450 MHz	358	10.0
Pentium [23]	(scaled) 450 MHz	590	6.1
IDEACrypt Kernel [22]	100 MHz	3	90.0

the number of partial products to an average of 8 per multiplier. Further optimization can be performed by transforming the shift-and-add multiplier into a constant CSD multiplier.

In Table 1 we compare both the template- and compiler-generated IDEA to optimized software implementations running on state-of-the-art processors, and to custom VLSI designs. PipeRench outperforms the processors listed by over 10x.

Somewhat surprisingly, PipeRench outperforms the .25 micron IDEACrypt Kernel from Ascom [22]. This is due to several factors: first, the PipeRench implementation of IDEA does not include the time taken to generate keys. This is because PipeRench targets streaming media applications, in which key generation comprises only a small preprocessing step. Secondly, because of the pipelined nature of PipeRench, IDEA has effectively been pipelined into 177 stages. If a custom silicon implementation were built with such a high degree of pipelining, the circuit would allow a fast clock (at the cost of silicon area.) Lastly, there is a 177-cycle latency through the pipeline. Nonetheless, it is noteworthy that the raw throughput of PipeRench is 40% faster than full-custom silicon.

5.2 IDEA in Embedded Systems

One of the challenges of placing a PipeRench fabric running IDEA in an embedded system is to reduce the time to generate a single-key configuration. While the compiler for PipeRench can compile a complete, single-key optimized IDEA application in less than one minute, this is too long for an embedded system. One method for accomplishing this is the use of precompiled CSD multiplier templates.

An 8-round IDEA pipeline (with the output transformation) contains 34 16x16 bit constant multipliers. The vast majority of the compilation time in generating a single-key IDEA pipeline is spent propagating constants through these multipliers, reducing them to the minimum required number of partial products. This operation is very important since nearly all the efficiency gained by fixing the key is a result of this reduction.

The task of compilation can be separated into two components: optimization and generation of the multipliers, and generation of the rest of the pipeline. If an interface is agreed upon by the multipliers and the rest of the pipeline, the two

tasks can be performed independently. We have developed a system for creation of IDEA with template-based multiplication.

The non-multiplier portions of the pipeline were hand-compiled beforehand to give maximum performance. They interact with the template-generated multipliers according to a pre-defined interface, and they do not place any important data in the registers which are used by the multiplier. The multipliers are thus treated as black boxes, and the non-multiplier operations are wrapped tightly around the multipliers to perform all the necessary computations in the minimum possible number of stripes.

To generate the multipliers themselves, a system is required that rapidly returns configuration bits for the stripes that perform the multiplication. Rather than expending a tremendous amount of effort and silicon area on hardware which actually computes these bitstreams, it is preferable to simply construct a lookup table which converts constant multiplicands into the necessary configuration bits.

Such an implementation would consist of nothing more than a ROM preloaded with the appropriate values. To reduce the size of the ROM, each 16-bit constant is broken into two 8-bit constants. The 8-bit constant is used as an index into a table of stripe configurations which implement that portion of the multiplier. The ROM would need approximately 256 120-bit entries. Although there is some overhead when recombining the two portions of the constant, using CSD representations we can still build the entire multiplier in two or three stripes. (Three stripes are required only for certain "bad" CSD vectors, which occur in only 1/16 of the entries. The multiplier interface is maintained whether the multiplier needs two or three stripes.)

Because the design of the template-generated multipliers and the logic placed between them only needs to be done once, great care can be taken to make the design very efficient. A single round of IDEA generated by this system is generally only 20 or 21 stripes long, resulting in a complete IDEA pipeline of only 177 stripes. This is a tremendous improvement over the 338-stripe compiler-generated pipeline. This improvement is primarily due to the compiler's not using registered feedback within a stripe.

5.3 Crypton

The Crypton [20] cipher can be implemented as a complete stream-function on PipeRench, with reasonable speedup. There are, however, operations within the Crypton cipher which are difficult to accomplish on the PipeRench architecture.

Most parts of the cipher map easily onto PipeRench. The byte transposition τ can be implemented entirely in the interconnect of PipeRench and does not require any computational resources at all. The bit permutations, π_o or π_e, can each be completed in four stripes. The key addition, σ_k, takes only one stripe.

The nonlinear S-box substitution, γ, however, is not easy to implement on PipeRench. Each of the three small 4x4 P_x s-boxes can be implemented either as logic or as a look-up table. In either case, because the PEs on PipeRench operate only on 8-bit quantities, 7/8 of each PE is wasted. Each PE generates

Table 2. *Comparison of different PipeRench systems and the speedups they achieve over the best versions in the cited papers. The "method" column indicates how the code was created: 'C' indicates it was automatically generated by the compiler. 'A' indicates a hand-coded version in CVHASM [15].*

Cipher	System	method	Clocks/block	Throughput (Mbytes/sec)	Speedup
RC6 [25]	PipeRench	A	28	58.8	4.7x
Crypton [20]	PipeRench	A	65	24.8	1.3x
	PipeRench+4	A	50	32.5	1.8x
	PipeRench+6	A	19	86.8	4.7x
Twofish [27]	PipeRench+4	C	51	15.6	2.8x
	PipeRench+16	C	15	54.3	9.7x
	PipeRench+4	A	36	36.0	3.9x
	PipeRench+16	A	9.7	164.7	14.6x

only a single bit, which is later combined by other PEs into a 4-bit quantity. When implemented by hand, this process requires three stripes per P_x.

A single round of Crypton uses 16 S-boxes, with three P_x units in each S-box. As a result, a single round of Crypton requires about 150 stripes, causing the entire 12-round cipher to occupy 1800 virtual stripes. PipeRench may have difficulty handling an application of that size due to limitations on the storage space for virtual stripes. The application can be re-pipelined on the inside in order to re-use the S-box in each round on all four 32-bit words. This cuts the length of the virtual pipeline by a factor of four, but it incurs a considerable amount of overhead in re-pipelining, and so reduces the overall throughput of the application.[2] Even with this large number of stripes, the application still gives 24.8 MByte/sec of throughput (with tremendous latency), compared with 18.46 MByte/sec on a 450 MHz Pentium Pro (coded with in-line assembly).

When implemented on PipeRench+, Crypton is significantly smaller and faster. Each of the S-boxes requires only a single stripe—the stripe that contains the load. Thus, the entire round takes only 24 stripes yielding a total of 288 stripes. Due to the limit on memory accesses per cycle this implementation yields 87 MByte/sec on PipeRench+16.

5.4 RC6

RC6 [25] is easy to implement as a stream function, but is only 5x faster than a 200Mhz Pentium-Pro due to the general purpose 32-bit multiplies (2 in each round). Unlike IDEA, neither operand of the multiplier is a constant. However, because the multiplier result is only 32-bits, the size of the multiplier is reduced by half. The variable rotates also require a significant amount of hardware: six stripes to do both rotates in parallel.

[2] Another solution is to chain multiple PipeRench chips together making a bigger pipeline. This solution also doubles performance.

5.5 Twofish

The Twofish [27] algorithm, like many others makes substantial use of S-boxes which is unsuitable for PipeRench. However, as described in Section 2.1, pieces of the algorithm can be mapped to custom instructions. For example, the q-function can be mapped to ten stripes on PipeRench. The space-consuming part of the function is again the four table lookups. In spite of this, using PipeRench to compute the S-boxes reduces the time for key setup making "full keying" a viable option even when very few blocks are encrypted with a single key.

However, on PipeRench+, the S-box lookups are easily handled. Each round requires 16 loads (from the g_0 and g_1 functions) and some rotating, XORing, and addition all of which are easy to accomplish on PipeRench+. Both the compiler and hand-coded versions achieve a speedup of about 3x on PipeRench+4 over the fastest assembly version running on a 200Mhz Pentium Pro/II. Interestingly, this is in spite of the fact that the compiler version is twice as large. This is because the compiler version spaces out the loads so there are very few stalls. The extra memory bandwidth of PipeRench+16 allows the hand-coded version to get more than 14x speedup.

5.6 Other AES Algorithms

We now discuss the AES algorithms which cannot be fully implemented on PipeRench. However, certain operations in the algorithms can be implemented as custom instructions, resulting in faster overall performance. In addition, most could be implemented on PipeRench+.

Cast-256 [1], like many of the proposed AES algorithms is heavily based on S-boxes which are not amenable to the current version of PipeRench. However, for each of the three keyed-round operations there are key-specific left rotations which can be optimized to constant rotations. Thus custom instructions can be created based on the subkeys which reduce the time to compute the S-box input to one cycle.

Performance of DFC [33] would improve by implementing the necessary multiprecision arithmetic as custom instructions.

The Hasting Pudding Cipher [29] gains significant performance as a series of custom instructions. The basic operations are all easily performed on PipeRench, but the entire cipher cannot be implemented as a stream function due to the large tables needed to hold the key expansion.

Both the key schedule and encryption/decryption in LOKI97 [5] can be sped up with a custom instruction which implements all but the S-box of the g-function and the data routing.

Deal [17], E2 [31], FROG [11], MAGENTA [3] and MARS [7] appear to be unsuitable for implementation on PipeRench due to the use of large table lookups in both key formation and encryption/decryption.

When encrypting or decrypting using Rijndael [10] each round consists of 4 basic functions of which three can be implemented as custom instructions.

SAFER+ [9], like HPC, DFC, Cast-256, Rijndael, and LOKI97, can benefit from using custom instructions, although the entire cipher cannot be implemented on PipeRench due to the large M matrix.

Serpent [2] uses S-boxes for the entire process except for the initial and final permutations which could be custom instructions.

6 Related Work

There is a growing body of work on using reconfigurable devices to implement cryptographic algorithms. Reconfigurable implementations of DES [32,14] and RSA [30] have all achieved significant speedups over general-purpose processors. However, in none of these cases were key-specific hardware implementations generated. The impact on the hardware size and throughput of key-specific implementations of DES using Xilinx FPGAs is discussed in [19].

In [16] FPGA-based implementations of DES are described that make use of many of the helpful attributes of reconfigurable devices which are used on PipeRench, including loop unrolling (which PipeRench requires) and pipelining (which PipeRench does implicitly). The acceleratation of modular multiplication and exponentiation (as used in RSA) using arithmetic architectures which have been optimized for use on FPGAs is described in [4].

More generally, PipeRench is one of several approaches towards making reconfigurable hardware more applicable to computation of the sort needed by cryptographic applications. PRISC [24] is among the earliest work on integrating a reconfigurable function unit with a processor. GARP [14], Chimaera [13], and One-Chip [34] are more recent examples of such work. The main difference between these systems and PipeRench is that PipeRench supports virtual hardware, freeing the application designer from fixed hardware constraints.

7 Conclusions

The use of reconfigurable hardware in cryptographic systems has many advantages. Reconfigurable implementations benefit from the hardware-based performance of custom VLSI while maintaining the flexibility and adaptability of software. Unlike fixed hardware, reconfigurable devices deliver highly customized, efficient solutions that are adaptable and robust to changing system needs.

The PipeRench reconfigurable architecture is well suited to many cryptographic tasks. Because it supports hardware virtualization, it can implement designs which are larger than the amount of physical hardware available. This is important, as many of the algorithms which can be mapped entirely to reconfigurable devices require tremendous amounts of physical hardware. Some algorithms which cannot be mapped completely onto PipeRench can still be accelerated by building custom instructions. PipeRench has two drawbacks; both relating primarily to table lookups. For small tables, such as the P tables in Crypton, the 8-bit PEs are very inefficient. For larger tables, such as S-boxes

found in many of the algorithms, PipeRench does not have any facility for performing memory accesses. We explore an extension to PipeRench, PipeRench+, which overcomes this second drawback.

Finally, key-specific circuits for PipeRench can be generated in embedded systems: with the use of a simple table lookup operation, the full performance of PipeRench can be obtained without waiting for software-based compilation. In the case of IDEA, we are able to exceed the performance of custom hardware.

Acknowledgements

This work was supported by DARPA contract DABT63-96-C-0083. We would like to thank the PipeRench group for tools and ideas, especially Mihai Budiu and Hari Cadambi for their help with the compiler.

References

1. C. Adams. The CAST-256 encryption algorithm. `www.entrust.com/resources/pdf/cast-256.pdf`.
2. R. Anderson, E. Biham, and L. Knudsen. SERPENT. `www.cl.cam.ac.uk/~rja14/serpent.html`.
3. E. Biham, A. Biryukov, N. Ferguson, L. Knudsen, B. Schneier, and A. Shamir. Cryptanalysis of MAGENTA. `www.ii.uib.no/~larsr/papers/magenta.pdf`.
4. T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH-14)*, Adelaide, Australia, April 1999.
5. L. Brown and J. Pieprzyk. Introducing the new LOKI97 Block Cipher. `www.adfa.oz.au/~lpb/research/loki97/`.
6. M. Budiu and S.C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, Feb. 1999.
7. Burwick, Coppersmith, D'Avignon, Gennaro, Halevi, Jutla, Matyas Jr., O'Connor, Peyravian, Safford, and Zunic. MARS - a candidate cipher for AES. `www.research.ibm.com/security/mars.html`.
8. D. Cherepacha and D. Lewis. A datapath oriented architecture for FPGAs. In *Second Int'l ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1994.
9. Cylink Corporation. SAFER+. `www.cylink.com/SAFER`.
10. J. Daemen and V. Rijmen. AES Proposal: Rijndael. `www.esat.kuleuven.ac.be/~rijmen/rijndael/`.
11. D. Georgoudis, D. Leroux, and B.S. Chaves. The "FROG" encryption algorithm. `www.tecapro.com/aesfrog.htm`.
12. S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.
13. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 87–96, April 1997.

14. J.R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 24–33, April 1997.
15. Kevin Jaget. Cvhasm: An assembler for pipelined reconfigurable architectures. Master's thesis, Carnegie Mellon University, 1998.
16. J.-P. Kaps and C. Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. In *Selected Areas in Cryptography '98*, volume 1556 of *Lecture Notes in Computer Science*, Kingston, Ontario, Canada, August 1998. Springer-Verlag.
17. L. Knudsen. DEAL: A 128-bit block cipher. Technical Report 151, Department of Informatics,University of Bergen, Norway, Feb 1998.
18. X. Lai and J.L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology Eurocrypt '90*, pages 389–404, 1991.
19. J. Leonard and W. H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. In *Field-programmable Logic and Applications: 7th International Workshop (FPL'97)*, London, UK, September 1997.
20. C. H. Lim. "CRYPTON". `crypt.future.co.kr/~chlim/crypton.html`.
21. H. Lipmaa. IDEA: A cipher for multimedia architectures? In *Selected Areas in Cryptography '98*, volume 1556 of *Lecture Notes in Computer Science*, pages 248–263, Kingston, Ontario, Canada, August 1998. Springer-Verlag.
22. Ascom Systec Ltd. IDEACrypt Kernel. `www.ascom.ch/infosec/idea/kernel.html`.
23. B. Preneel, V. Rijmen, and A. Bosselaers. Recent developments in the design of conventional cryptographic algorithms. In *Computer Security and Industrial Cryptography*, volume 1528 of *Lecture Notes in Computer Science*, pages 106–131. Springer-Verlag, 1998.
24. R. Razdan and M.D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO-27*, pages 172–180, November 1994.
25. R. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. `theory.lcs.mit.edu/~rivest/rc6.ps`.
26. Herman Schmit. Incremental reconfiguration for pipelined applications. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 47–55, Napa, CA, April 1997.
27. Schneier, Kelsey, Whiting, Wagner, Hall, and Ferguson. Twofish: A 128-bit block cipher. `www.counterpane.com/twofish.html`.
28. Bruce Schneier. *Applied cryptography: Protocols, algorithms, and source code in C*, chapter 13, pages 319–325. John Wiley and Sons, Inc., 1996.
29. R. Schroppel. The Hasty Pudding Cipher. `www.cs.arizona.edu/~rcs/hpc`.
30. M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *11th IEEE Symposium on COMPUTER ARITHMETIC*, 1993.
31. Nippon Telegraph and Telephone Corporation. The 128-bit block cipher E2. `info.isl.ntt.co.jp/e2/`.
32. K. W. Tse, T. I. Yuk, and S. S. Chan. Implementation of the data encryption standard algorithm with FPGAs. In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pages 412–419, Oxford, England, September 1993.
33. S. Vaudenay. DFC. `www.dmi.ens.fr/~vaudenay/dfc.html`.
34. R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.