Divide-and-Concatenate: An Architecture Level Optimization Technique for Universal Hash Functions

Bo Yang Ramesh Karri
ECE Department
Polytechnic University, Brooklyn, NY, 11201
yangbo@photon.poly.edu, ramesh@india.poly.edu

David A. McGrew
Cisco Systems, Inc.
San Jose, CA 95134
mcgrew@cisco.com

ABSTRACT
We present an architecture optimization technique called divide-and-concatenate for universal hash functions. The area of a multiplier increases quadratically and its speed increases gradually with the operand size and two universal hash functions are equivalent if they have the same collision probability property. Based on these observations, the divide-and-concatenate approach **divides** a 2w-bit data path (with collision probability 2^{-2w}) into two w-bit data paths (each with collision probability 2^{-w}), applies one message word to these two w-bit data paths and **concatenates** their results to construct an **equivalent** 2w-bit data path (with collision probability 2^{-2w}). We demonstrate this technique on Linear Congruential Hash (LCH) family. When compared to the 100% overhead associated with duplicating a straightforward 32-bit LCH data path, the divide-and-concatenate approach that uses four equivalent 8-bit data paths yields a 101% increase in throughput with only 52% hardware overhead.

Categories and Subject Descriptors
B.2.4 [**High-Speed Arithmetic**]: Cost/performance

General Terms
Design, Performance, Experimentation

1. INTRODUCTION
A hash function converts an input from a large domain into an output in a small range (the hash value). Hash functions have a wide range of applications in computing communications and networking. In database and web search engines we can speed up the lookup of a record by using hash value as an index. In cryptography, keyed hash functions are used as message authentication codes. MD5 [1] and SHA-1 [1] are two popular hash algorithms. These hash algorithms are iterative; the current computation step depends on the result of the previous step. So they yield moderate throughput [2]. Recently, hash functions such as MMH [2] and TMMH [3] have been designed to exploit architectural support for multiplication in modern processors.

Message authentication on a packet-by-packet basis in the IPsec protocol requires that the implementation of the underlying hash function match the >10 bps wire speed requirements of network traffic. In virus detection and content classification applications[4], the digest of each and every packet has to be computed and compared with pre-defined signatures at these wire speeds. For these and other emerging applications, software implementations of hash functions are not sufficient [5]. Furthermore, implementing MD5 and SHA-1 in hardware only yields moderate improvement in throughput due to their iterative structure. Commercial implementations of MD5 and SHA1 targeting 0.18 micron ASIC library have a throughput of about 1 Gbps [6] [7].

In the rest of this paper we will describe the divide-and-concatenate architecture optimization technique. Specifically, we will introduce LCH in section 2. We will apply the divide-and-concatenate technique to LCH architectures in section 3. In section 4 we will summarize our conclusions.

DAC 2004, June 7-11, 2004, San Diego, California, USA

2. LINEAR CONGRUENTIAL HASH (LCH)
We will investigate the important class of universal hash functions [8] and show how this class can be efficiently implemented in hardware using a novel divide-and-concatenate technique. A universal hash function is defined as follows [8]:

Definition 1: Let A and B be two sets and H be a family of functions from A to B. H is a universal family of hash functions if for every pair x1, x2 $\in$ A with x1$\neq$x2, h$\in$H and h(x1), h(x2) $\in$B, the collision probability of h(x1) =h(x2) = 1/|B|.

|B| is size of set B and 1/|B| is the smallest possible value of the probability. When B is small, the collision probability is large. In many cases, some hash function families can only achieve a collision probability ε which is slightly larger than 1/|B| [8]. Such universal hash functions are called almost universal. In this paper we will focus on LCH, a widely used universal function family which is defined as:

$$h_{m,x}(m) \equiv \sum_{i=1}^{k} (m_i x_i + t) \bmod p$$

where m_i is the i^{th} word in a message block m and x_i is the i^{th} word in key x and $t \in Z_p$. p is a prime number which is close to 2^w. Modular reduction of the accumulated result using p generates either a w-bit or a w+1-bit hash value. Since p is close to 2^w, the collision probability of LCH (1/p) is close to 2^{-w}. The key block and every message block consist of k w-bit words. Since the modular reduction step can be amortized over k multiply-and-accumulate operations, increasing k can increase the speed. However, a large k results in a longer key.

2.1 LCH Architecture
Consider the 32-bit LCH architecture shown in Figure 1. It uses four 32-bit registers (R1, R2, H and L), two 64-bit registers (R3, R4), two 32-bit 2-to-1 multiplexers (MUX1 and MUX2) and two 64-bit 2-to-1 multiplexers (MUX3 and MUX4).

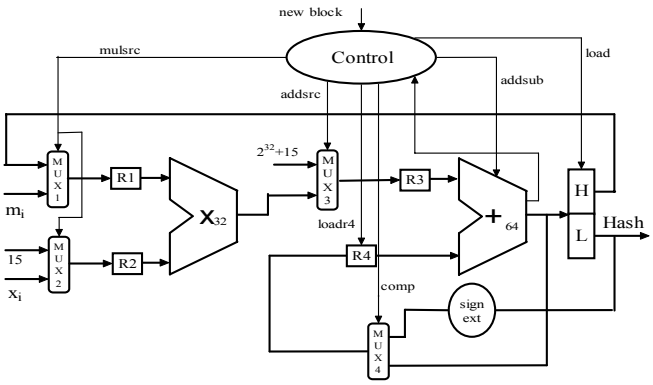

Figure 1: 32-bit hardware architecture for the LCH. The critical path of this architecture is R1->X32->MUX3-> R3.

We can rewrite LCH as follows:

$$h_{m,x}(m) \equiv ((\sum_{i=1}^{k} (m_i x_i)) + kt) \bmod p$$

At the beginning of every message block when newblock is set to active, registers R1, R2, R3, L and H and the counter in the control logic are all set to zero while register R4 is initialized to kt (this is a constant for a specific instance of LCH function).

This is followed by k multiply-and-accumulate steps of the k 32-bit message words with the k 32-bit key words. In each step, mi in register R1 is multiplied with xi in register R2 and the result is stored in register R3. For this purpose, the control signal mulsrc of multiplexers MUX1 and MUX2 are set to select lower input. This is then accumulated into register R4 using the adder/subtractor unit. The 64-bit accumulated result at the end of the k+1 clock cycles is stored in the (H, L) register pair.

2.2 Modular Reduction

Reducing the 64-bit result of the multiply-and accumulate step of the 32-bit LCH function modulo p ($=2^{32}+15$) yields a 33-bit hash value. We will describe a division-less modular reduction algorithm from [2] that uses 2 multiplications and 3 subtractions. Let $x=2^{32}a+b$ be the 64-bit input where a, b are unsigned 32-bit integers.

$$2^{32}a + b \equiv (2^{32}a + b) - a \cdot (2^{32}a + 15) = b - 15a \pmod{2^{32}a + 15}$$

Step 1: Since a, b $\in[0, 2^{32}-1]$, b-15a $\in[-15 \cdot (2^{32}-1), 2^{32}-1]$. If y=b-15a then $y \equiv x \pmod{2^{32}+15}$ can be represented as a signed 64-bit integer $y=2^{32}c+d$, where c $\in\{-15,\ldots, 0\}$ and d is an unsigned 32-bit integer. Step 2: Repeat step 1 to compute z=d-15c. $z \equiv y \equiv x \pmod{2^{32}+15}$ and $z \in \{0,\ldots, 2^{32}+15^2\}$. Step 3: If $z > 2^{32}+15$ return $z-(2^{32}+15)$ else return z.

After the accumulation, the modular reduction is applied to the 64-bit accumulated result in (H,L) to get a 33-bit hash. Step 1 of the reduction algorithm consumes two clock cycles. By setting mulsrc to select the upper input of MUX1 and MUX2 respectively, we will get 15a at the end of cycle 1. By setting comp signal to select the upper input of MUX4 and setting addsub to do subtraction we will get y=b-15a at the end of cycle 2. By repeating these operations in clock cycles 3 and 4 we will get z (step 2 in section 2.1). In the last cycle, we perform z-p. If the result is less than zero, the value in register pair (H, L) is the hash value. Otherwise output of the adder/subtractor is the hash. This is loaded into the output registers (H, L) by setting the load signal to 1. Overall, this architecture generates a 33-bit hash for a message block in k+6 cycles. We used Synopsys Design-Complier to synthesize this and other LCH architectures targeting a 0.13 micron IBM ASIC library. An implementation of this 32-bit LCH architecture using a Brent-Kung (bk) adder and combinational multiplier consumes 9071 gates with the control logic consuming ~1% and the multiplier consuming 72% of the area.

3. DIVIDE-AND-CONCATENATE: AN ARCHITECTURE SPEEDUP TECHNIQUE

3.1 Motivation

A straightforward approach to speeding up the LCH architecture is using fast multipliers and adders. Since the speed of a multiplier or an adder is determined by its carry chain, circuit level [10] and logic level [11] speed-up techniques were proposed to reduce this critical path. Orthogonal to circuit and logic level speed-up techniques are architecture level pipelining [9] and loop unrolling [9]. For example, we can use an additional multiplier and an adder after the (H, L) register pair to perform the modular reduction. This fully pipelined LCH architecture improves the overall throughput by (k+6)/k. We can use two LCH data paths in parallel, each operating on a different message block to process 2× input message bits. Although this method yields a 100% improvement in throughput, it entails 100% area overhead.

3.2 Bottlenecks in Speeding up LCH

Hardware complexity of a multiplier increases quadratically with operand size as shown in Figure 2(a). For example, while an 8-bit combinational multiplier consumes 399 gates a 16-bit combinational multiplier consumes 1767 gates (~4.43x 8-bit combinational multiplier).

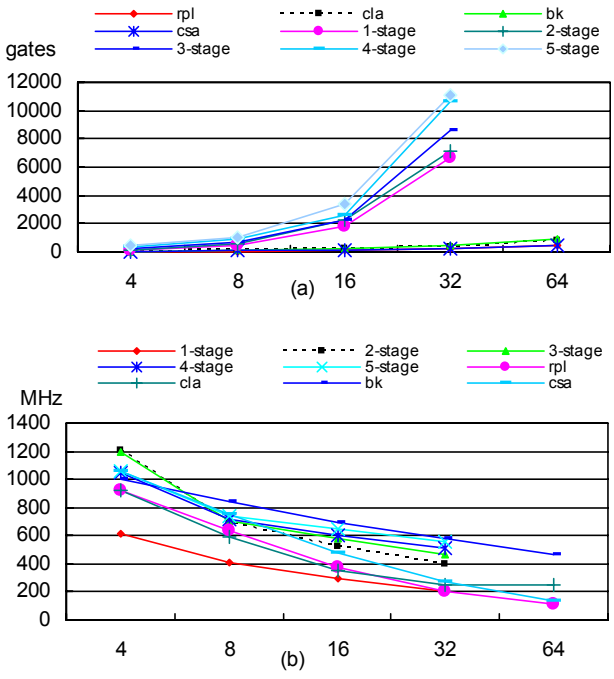

Figure 2: Area and clock rates of adders, multipliers as a function of operand size. (a) the area of a multiplier grows quadratically and (b) their clock rates decrease gradually with operand size increasing

CSA (Carry Save Array) structure is better for small operands, while Wallace Tree structure is better for larger operands. We used CSA for 4 and 8 bit multipliers and Wallace Tree for 16 and 32 bit multipliers. Multipliers with more pipeline stages are larger than those with less pipeline stages. For example, while an 8-bit combinational multiplier consumes 399 gates, an 8-bit 3-stage pipelined multiplier consumes 720 gates. Hardware complexity of an adder increases only linearly with operand size. Figure 2(b) shows clock rates of adders and multipliers. They decrease gradually with operand size. For example, clock rates of 8-bit, 16-bit and 32-bit combinational multipliers are 412, 296 and 205 MHz respectively.

In a nutshell, hash functions that do not use multiplications are superior to those that do and hash implementations that use small sized operands are superior to those that use large operands.

3.3 Reducing the Collision Probability

In software implementations, since the width w of multiplications and additions are constrained by the underlying processor architecture, increasing w from 32-bit to 64-bit or 128-bit is not feasible. A simple way to reduce the collision probability of universal hash functions is to hash the message n times using n independent keys and concatenate the results. This reduces their collision probability from 2^{-w} to 2^{-nw}. This solution requires n× key material. The Toeplitz-extension described in [2] reduces the amount of key material making this approach practical. When we use two copies of the 32-bit LCH data path to construct a 64-bit data path, using Toeplitz-extension the keys for the second LCH data path can be obtained by shifting the original key. This method will yield efficient LCH architectures even for small w.

3.4 The Divide-and-Concatenate Technique

We propose to divide a 2w-bit data path into two w-bit data paths and concatenate their results to construct an equivalent 2w-bit data path. The concept of equivalence is crucial. Obviously, a straightforward data path and the corresponding divide-and-concatenate data path cannot be equivalent in terms of the results that they output. We define two data paths to be equivalent if the results that they output have the same collision probability.

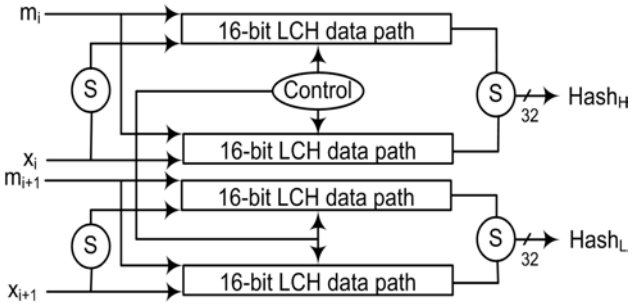

Figure 3: This equivalent divide-and-concatenate architecture composed of four 16-bit LCH data paths has a collision probability of 2^{-32}.

A 32-bit LCH architecture with a collision probability of 2^{-32} can be constructed using two 16-bit LCH data paths, each with collision probability of 2^{-16} (the divide step), and concatenating their 16-bit results to generate a 32-bit hash value (the concatenate step). This is shown in Figure 3. These two 16-bit data paths share control logic. The shift S and the concatenate C do not contribute to the area. While the two data paths at the top process 16-bit messages every cycle, the straightforward LCH architecture processes 32-bit message. In order to match the 32-bit input message length of the straightforward LCH architecture, we propose to use four 16-bit data paths to process a 32-bit message every cycle. (m_{i+1}, m_i) and (x_{i+1}, x_i) are 32-bit message and key word respectively. The 64-bit ($Hash_H$, $Hash_L$) is the hash of a message block.

Using the data from Figure 2, the throughput of the straightforward 32-bit LCH data path is 6.56 Gbps (=205MHz×32bits) (The coefficient k/k+6 is omitted, because all the throughputs are scaled by it.). The throughput of the equivalent 16-bit divide-and-concatenate data path is 9.48 Gbps (=296MHz×32bits). The equivalent 16-bit divide-and-concatenate architecture consumes 11506 gates compared to 9071 gates by the straightforward architecture. The 16-bit divide-and-concatenate architecture achieves 45% throughput improvement with 26% area overhead. Overall, the throughput/area ratio of 0.824 Mbps/gate (=9.48Gbps÷11506 gates) for the 16-bit divide-and-concatenate architecture is 14% more efficient than 0.723 Mbps/gate (=6.56Gbps÷9071gates) for the straightforward 32-bit architecture.

Let us apply this divide-and-concatenate technique once more and construct each 16-bit LCH data path using four 8-bit LCH data paths. This translates into sixteen 8-bit LCH data paths to construct an equivalent 32-bit LCH data path. The area of this equivalent architecture is 13824 and runs at 412 Mhz, yielding a throughput of 13.184 Gbps (=412 MHz×32 bits). Compared to 32-bit straightforward LCH architecture, the 8-bit equivalent divide-and-concatenate architecture achieves 101% throughput improvement with 52% area overhead. The throughput/area ratio of this 8-bit equivalent data path is 0.954Mbps/gate (13.184Gpbs÷13824gates), which is 30% more efficient than that of the straightforward architecture and is better than that of 16-bit equivalent divide-and-concatenate architecture.

Can we apply this divide-and-concatenate technique using 4-bit unit? Sixty four 4-bit data paths are required, but the key word length is only 4 and we can not shift it 8 times using Toeplitz-extension. However, from the results of 16-bit and 8-bit equivalent data path, it seems that the more components that the basic data path is divided into, the better the resulting divide-and-concatenate data path. Suppose we do not use Toeplitz-extension and just use 8× times key material when constructing a 32-bit LCH data path using eight 4-bit data paths, the throughput of this data path is 19.616Gbps (=613MHz×32bits) and area is 20830 gates yielding a throughput/area ratio of 0.941 Mbps/gate (19.616Gpbs÷20830gates), which is less than that of the 8-bit equivalent divide-and-concatenate data path.

The 4-bit divide-and-concatenate equivalent architecture is not as efficient as an 8-bit divide-and-concatenate equivalent architecture. In 8-bit and larger designs the area is dominated by large multipliers. On the other hand, in 4-bit designs the area is dominated by linear components, like adders, multiplexers and registers. Overall, the 8-bit divide-and-concatenate equivalent architecture has the best throughput/area ratio. When compared to the straightforward 32-bit LCH data path yielding a 101%

increase in throughput with only 52% hardware overhead. The results of these discussions are summarized in Table 1.

Table 1: Number of data paths, throughput, area and throughput/area ratio for the straightforward 32-bit LCH architecture and equivalent divide-and-concatenate architectures with collision probability of 2^{-32}

Architecture	Straight forward	Divide-and-Concatenate		
Data path width (bits)	32	16	8	4
# of data paths	1	4	16	64
Throughput (Gbps)	6.56	9.48	13.18	19.61
Area(Gates)	9071	11506	13824	20830
Throughput/Area (Mbps/Gate)	0.723	0.824	0.954	0.941

3.5 Relation to Other Techniques

The divide-and-concatenate technique is complementary to circuit [10], logic [11] and architectural [9] level speedup techniques. One can speed up any universal hash data path using any of these optimization techniques. If multipliers still dominate the area, the divide-and-concatenate technique can be applied. For example, we can speed up the straightforward architecture by pipelining multipliers. We can then apply the divide-and-concatenate technique to this pipelined LCH data path. Figure 4 summarizes throughput/area ratios of equivalent LCH architectures with combinational and pipelined multipliers. It is shown that for 32-bit LCH architecture with 2-stage pipelined multiplier, the equivalent divide-and-concatenate architecture using 8-bit unit is 12% better in throughput/ratio than the straightforward 32-bit architecture with 2-stage pipelined multiplier.

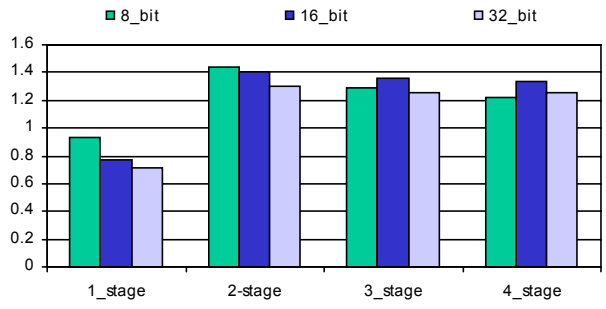

Figure 4: Throughput/area ratios of three equivalent divide-and-concatenate data paths with different pipeline stages with collision probability of 2^{-32}.

4. CONCLUSIONS

A drawback of the divide-and-concatenate technique is that the length of the output hash doubles as you go from the straightforward 32-bit data path to an equivalent 16-bit data path to an equivalent 8-bit data path. In fact, the length of hash value of an equivalent 8-bit data path is 4 times longer that of the straightforward data path. If the throughput of two 16-bit data paths as shown at the top of Figure 3, that can provide the same collision probability as a 32-bit data path, is enough, we do not need four 16-bit data paths to take 32-bit input every cycle. A 32-bit text can be serialized into 2 16-bit words. In this case, the 32×32 block is organized as 64×16. This two 16-bit equivalent architecture provides the same collision probability, uses same key material and has the same output length, but has a better throughput/area ratio.

The divide-and-concatenate technique cannot speed-up software implementations but can only improve the collision probability beyond that provided by the processor architecture. This is because, if a processor supports w-bit additions and multiplications in one or two cycles, then w/2-bit operations will also consume the same number of cycles as w-bit operations.

MMH [2] and TMMH [3] are special cases of LCH with t=0. 32-bit MMH is defined as:

$$h_{m,x} \equiv [[[\sum_{i=1}^{k} m_i x_i] \bmod 2^{64}] \bmod (2^{32} + 15)] \bmod 2^{32}$$

and m_i, x_i are 32-bit. MMH is an almost universal hash function with a collision probability of 1.5×2^{-32}. 16-bit TMMH is defined as:

$$h_{m,x} \equiv [[[\sum_{i=1}^{k} m_i x_i] \bmod 2^{32}] \bmod (2^{16} - 1)] \bmod 2^{16}$$

In MMH and TMMH, b is zero and we just need to set register R4 to zero at the beginning of every block.

5. REFERENCES

[1] B. Schneier, "Applied Cryptography," Second Edition, John Wiley & Sons, Inc. New York, 1996.

[2] S. Halevi, and H. Krawczyk, "MMH: Software message authentication in the Gbit/second rates," Workshop on Fast Software Encryption, pp. 172-189, 1997.

[3] D. A. McGrew, "The Truncated Multi-Modular Hash Function (TMMH)," IETF Internet Draft, 2001. http://www.mindspring.com/~dmcgrew/draft-mcgrew-saag-tmmh-01.txt

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull and J. Lockwood. "Deep Packet Inspection Using Parallel Bloom Filters." Symposium on High Performance Interconnects, pp.44-52, 2003.

[5] http://www.cs.ucdavis.edu/~rogaway/umac/perf00bis.html

[6] Helion Technology. Datasheet-High Performance SHA1 Hash Core for ASIC, 2003. http://www.heliontech.com/downloads/sha1_asic_helioncore.pdf

[7] Amphion. Datasheet-High Performance MD5 Core, 2003. http://www.amphion.com/acrobat/DS5315.pdf

[8] L. Carter, and M. Wegman, "Universal hash functions," Journal of Computer and System Sciences, vol 18, pp.143-154, 1979.

[9] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," IEEE Trans on VLSI Systems, 9(4), pp. 545-557, 2001.

[10] P. Stelling, C.Martel, V.Oklobdzija, and R. Ravi, "Optimal Circuits for Parallel Multipliers," IEEE Trans. Computers, vol. 47, no. 3, pp. 273-285, 1998.

[11] G. Goto et al., "A 54×54-b Regularly Structured Tree Multiplier," IEEE J. Solid-State Circuits, vol. 27, no. 9, pp.1229-1236, 1992.