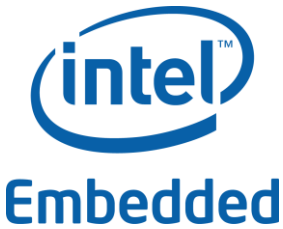


White Paper

**Vinodh Gopal**
**Wajdi Feghali**
**Jim Guilford**
**Erdinc Ozturk**
**Gil Wolrich**
**Martin Dixon**
IA Architects
**Max Locktyukhin**
**Maxim Perminov**
Software Architects
Intel Corporation

# Fast Cryptographic Computation on Intel® Architecture Processors Via Function Stitching

April, 2010

323686



# Executive Summary

Cryptographic applications often run more than one independent algorithm such as encryption and authentication. This fact provides a high level of parallelism which can be exploited by software and converted into instruction level parallelism to improve overall performance on modern super-scalar processors. We present fast and efficient methods of computing such pairs of functions on IA processors using a method called "function stitching". Instead of computing pairs of functions sequentially as is done today in applications/libraries, we replace the function calls by a single call to a composite function that implements both algorithms. The execution time of this composite function can be made significantly shorter than the sums of the execution times for the individual functions and, in many cases, close to the execution time of the slower function.

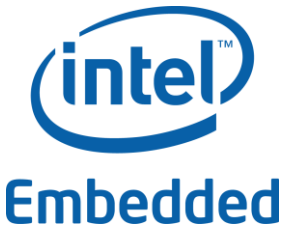
Function stitching is best done at a very fine grain, interleaving the code for the individual algorithms at an instruction-level granularity. This results in excellent utilization of the execution resources in the processor core with a single thread.

We show how stitching pairs of functions together in a fine-grained manner results in excellent performance on IA processors. Currently, applications perform the functions sequentially. We demonstrate performance gains of 1.4X-1.9X with stitching over the best sequential function performance.

We show performance results achieved by this method on the Intel® processors based on the Westmere architecture.



# Contents



# Overview

Function stitching significantly speeds up pairs of algorithms, e.g., encryption/authentication, on existing Intel® architecture (IA) processors. It applies to algorithms which, due to instruction dependencies or instruction latencies, cannot fully utilize processor core execution resources. In some cases, the type of instructions used by one algorithm may not be able to utilize all the execution units in the core; the other algorithm could have a different instruction mix that permits better overall usage when stitched with the first algorithm. In this paper, we focus on cryptographic applications and restrict the discussion to pairs of functions; however, function stitching can be broadly applied to any set of multiple functions that are called together. Stitching is a transformation of high-level algorithm parallelism into instruction level parallelism (ILP) exploitable by super-scalar processors.

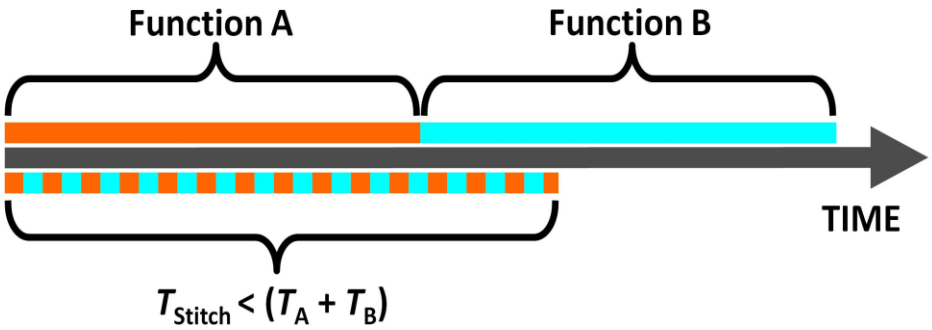
***Note:*** Note that function stitching is not the same as software pipelining [3] which is a method used to optimize loops. These methods are complementary.

# Function Stitching

We define function stitching as a method to interleave instructions from pairs of functions to maximize execution efficiency of the cores. This method allows you to significantly speed up pairs of algorithms that are typically called at or near the same time, for example, encryption and authentication algorithms. Applications or libraries typically call one function to perform one algorithm and then call a second function to perform the other algorithm. However, if these two function calls are replaced by a single call to a composite function that implements both algorithms, the execution time of this composite function can often be made significantly shorter than the sums of the execution times for the two individual functions and, in many cases, very close to the max of the execution times of the individual functions.

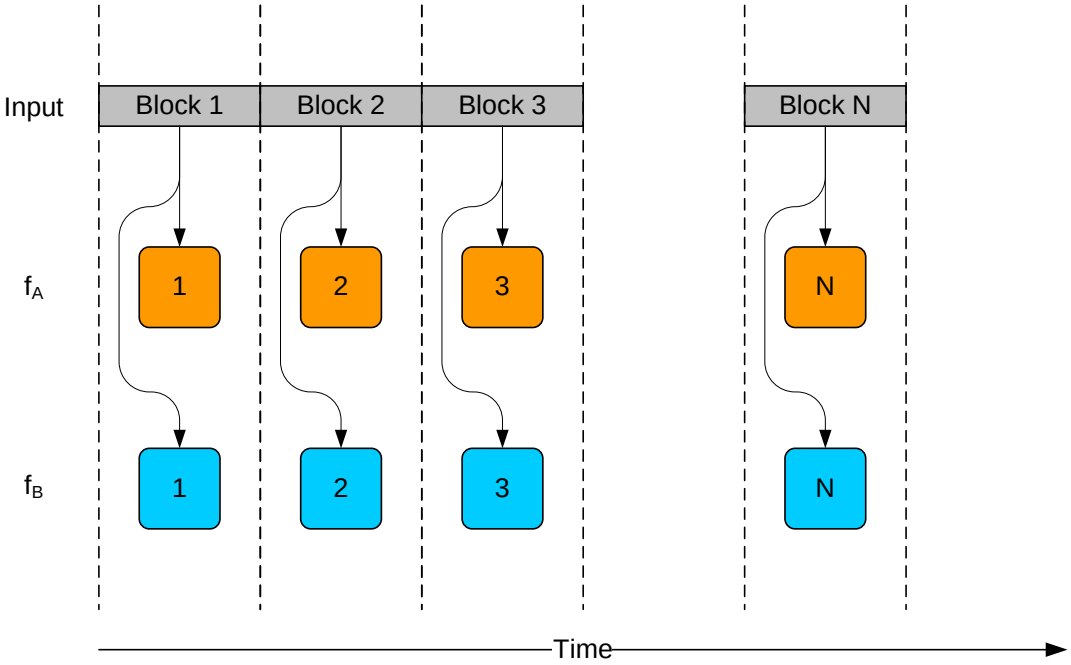
The advantage of having a single composite function is that this function contains code for two different algorithms which are essentially independent of each other. This allows the two code streams to be interleaved at a fine grain, often on an instruction granularity, which we refer to as "stitching" the two algorithms together (see Figure 1). This in turn allows the processor core to execute instructions from both algorithms at the same time, makes better use of the execution resources, and results in a lower total execution time. It may also offer second-order benefits, such as only requiring data to be fetched from memory/caches once rather than twice.

**Figure 1. Stitching Two Functions**




Stitching is straight-forward when both algorithms work on the same input data or on independent data. An example of working on the same data (shown in Figure 2) is if one algorithm is encryption (function A), and the authentication (function B) is applied to the plain-text.

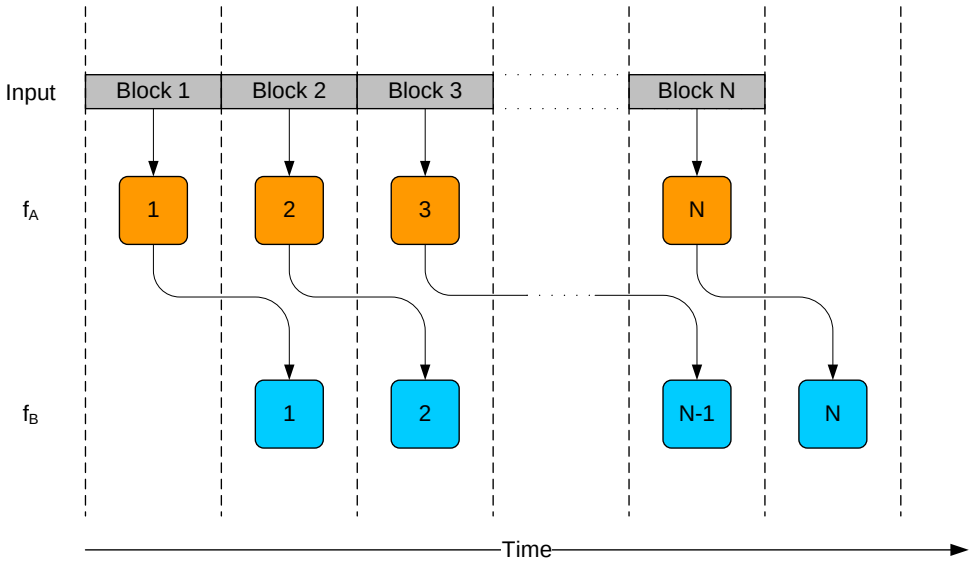
**Figure 2. Stitching Two Functions That Operate on the Same Blocks of Data**




The same technique can be applied when the functions work on independent blocks with different data pointers for the second function. However, it can also be easily used when one algorithm is applied to the output of the other algorithm, e.g., if the authentication is applied to the cipher-text generated by the encryption function when the algorithms are being applied to a buffer containing many basic blocks of data. For example, the AES encryption algorithm operates on 16-byte blocks, and a given input buffer typically

contains some number (N) of blocks. In this case a non-stitched encryption implementation is applied to the first block. Then the stitched code is applied (N-1) times, with the encryption code operating on input block *i*, and the authentication code (such as GHASH) operating on output block (*i*-1). Finally, an unstitched authentication implementation is applied to the last block. This is illustrated in Figure 3:

**Figure 3. Stitching Two Functions When One Operates on a Block of Data Produced by the Other**




Similar techniques can be used to handle the case where the two algorithms are handling input data of different sizes.

Since the code stream from each algorithm is essentially independent of the other stream, they can be interleaved somewhat arbitrarily. This makes it easier to schedule the final code to optimize front-end decoder performance, since on IA processors some instructions can only be decoded on a subset of the decoders, in addition to execution efficiency.

# Types of Architectural Stitches

The Intel® 64 and IA-32 instruction set architectures have two distinct instruction subsets that can be used by cryptographic algorithms: general Purpose instructions and Single Instruction Multiple Data (SIMD) instructions

[1]. SIMD instructions include, and are mostly known as, Intel® SSE, SSE2 etc. extensions. Current SIMD extensions work on 128-bit XMM registers. There are some instructions such as AES and PCLMULQDQ that are defined on the XMM register set but are not in fact SIMD. The general purpose instructions work on 64-bit or 32-bit general purpose registers. At the instruction level, some algorithms could be implemented with the general purpose instructions, some with the SIMD instructions, and in some cases could be implemented with both. There are various benefits/complexities to stitching for these different cases.

In this section, we briefly describe the methods we used for the three different types of stitching. These types of stitches are broadly based on architectural properties of the code found in the functions; it is possible to also exploit microarchitectural properties for stitching, which is beyond the scope of this paper. Note that an algorithm could be implemented with a mix of instructions but we can classify it based on the predominant type of instructions and it can be analyzed for stitching based on the three categories.

Intel® 64 provides 16 vector XMM registers and 16 general purpose registers, while IA-32 provides 8 of each kind. Since stitching requires the registers to be shared among the two algorithms, it could become difficult if the individual algorithms need to use most or all registers. Running out of registers implies spilling to memory which could reduce the efficiency of the stitched code. If the two algorithms do not use lots of registers, then stitching is not particularly complex. It particularly makes 64-bit code better suited for stitched implementations.

Mixing general purpose along with SIMD code is very beneficial for stitched implementations as it extends the effective number of available registers and provides additional opportunities for creating a better balanced mix of instructions using SIMD and general purpose registers.

The underlying superscalar microarchitecture of Intel® processors provides significant execution resources which a single algorithm can rarely utilize in full. A single algorithm can be limited by low ILP, as in the example of MD5 caused by most of the calculations belonging to a single long dependency chain. It can be dominated by either Arithmetic and Logical Unit (ALU) or memory operations, as in the case of RC4, or can consist of a low number of relatively long latency dependent instructions as in the case of AES CBC-Encrypt implementation using Intel® AES New Instructions (Intel® AES-NI). But when stitched the two usually reach much higher overall utilization of available execution slots because of the significantly improved ILP and/or instruction balancing.

## Integer-Integer Stitch

Two functions that are both implemented with the general purpose instructions can be stitched together. Here the biggest potential problem that can prevent stitching could be the usage of the architectural (arithmetic)

flags; they are defined as implicit inputs and/or outputs of some instructions. No flag(s) producing/consuming instruction from one algorithm can be inserted between flag(s) producing and consuming instructions of the other algorithm. Fortunately, the algorithm pairs for symmetric encryption and authentication do not require architectural flags. The main issue with this mode could be the register limitations as a result of sharing the general purpose registers among the stitched functions.

One example for this type of stitching is RC4-MD5, our current best implementation; however, it does use a small mix of SIMD instructions. The MD5 algorithm is defined in a way that has a critical data-dependency chain, severely limiting the amount of ILP that can be exploited by IA processor cores, described in detail in Appendix A. Similarly, RC4 is an algorithm that has a critical dependency chain that limits ILP; RC4 performs lookups into a small table which involve load instructions. Stitching these two functions together allows for more operations to execute in parallel. The execution efficiency of the RC4-MD5 stitch is also helped by the fact that the MD5 algorithm uses a different mix of instructions compared to the RC4 algorithm, leading to a better overall utilization of the execution/load-store units. Another cipher that can benefit from this type of stitch with MD5 is DES/3DES, since the implementation has limited ILP and performs many load operations to tables. There are no examples using DES/3DES in this paper.

## SSE-Integer Stitch

This is the easiest type of stitch - a function implemented with general purpose instructions could be stitched with another function implemented with SIMD instructions. The AES-SHA1 algorithm pair could be interleaved in this manner because SHA1 is an algorithm that can be efficiently implemented with scalar general purpose instructions, whereas AES can be implemented on the Westmere microarchitecture using the Intel® AES-NI new instructions that have been defined as a set of SSE instructions. The performance benefits of this stitch are increased due to the use of different microarchitectural resources in the execution units and hiding latencies of the individual functions. It should be noted that the best SHA1 implementation [2] uses a limited number of SIMD instructions; however for simplicity, we categorize it as a general purpose type in this paper. Stitching applies equally well to the SHA1 implementation that has a small mix of SIMD instructions, since the AES instructions do not use the SIMD execution units at a high rate.

## SSE-SSE Stitch

Two functions that are implemented with SIMD instructions could be stitched together. The main issue with this mode could be the register limitations as a result of sharing the XMM registers among the stitched functions. One example of this type is the AES-CBC-XCBC pair of functions that use AES for encryption and also for authentication, which are efficiently implemented using the Intel® AES-NI. Another important example is the AES-GCM that

uses AES in Counter mode for encryption and Galois-Hash for authentication; these can be implemented using Intel® AES-NI and PCLMULQDQ-NI available in the Westmere microarchitecture.

# Performance

The performance results provided in this section were measured on a dual socket system running two 6-core Intel® Xeon® X5670 processors at a frequency of 2.92 Ghz, based on the Intel® 32-nm technology microarchitecture, supporting Intel® AES-NI. Results represent peak crypto bandwidth of a system, measured on common pairs of cipher and hash and comparing stitched algorithms with optimized standalone algorithms running sequentially. The tests were run with Intel® Turbo Boost Technology on, and represent the performance without Intel® Hyper-Threading Technology (Intel® HT Technology).

## Choosing the baseline

We started with the fastest available implementations of the individual crypto algorithms found in libraries such as OpenSSL. We then looked for opportunities to optimize those further and found ways to significantly improve the performance of the standalone algorithms' implementations, over best known baselines for SHA-1 and RC4.

The article [2] describes details of the optimizations to the standalone SHA-1 algorithm. RC4 optimization targeted the improvement of the runtime disambiguation of memory references in the algorithm, which in most cases are independent, and can be executed with higher throughput.

In the case of AES-128 and AES-256 we achieve large performance improvement over any legacy baseline code by using Intel® AES-NI. After developing these highly optimized standalone crypto algorithms, we proceeded with stitching to achieve yet another major leap in performance improvement.

## Methodology

Secure communications is the primary usage model of paired cipher and hash algorithms. Current predominant Ethernet connections are capable of transferring 1 Gb ($10^9$ bits) of data per second. Faster network connections capable of speeds of 10 Gb/second are also widely used today, with a push for faster connections such as 40 and 100 Gb/second. In the next few years, we expect to see greater usage of 40 Gb Ethernet, capable of transferring 40 Gb of data per second.

We therefore represented the performance results as time, in seconds, required to process 40 Gb of data. It represents the system's peak crypto throughput to run each particular pair of a block cipher and a hash. The

baseline code has cipher and hash functions called on an input buffer sequentially, while stitched code calculates both with a single call.

To keep the comparison fair for sequential code we chose the input buffer size to fit into the second level data cache. Larger buffers may give additional advantage to the stitched code as it would have to fetch data from memory only once, while usage of large buffers is much less common and the focus of this paper is primarily on execution efficiency achieved with stitching.

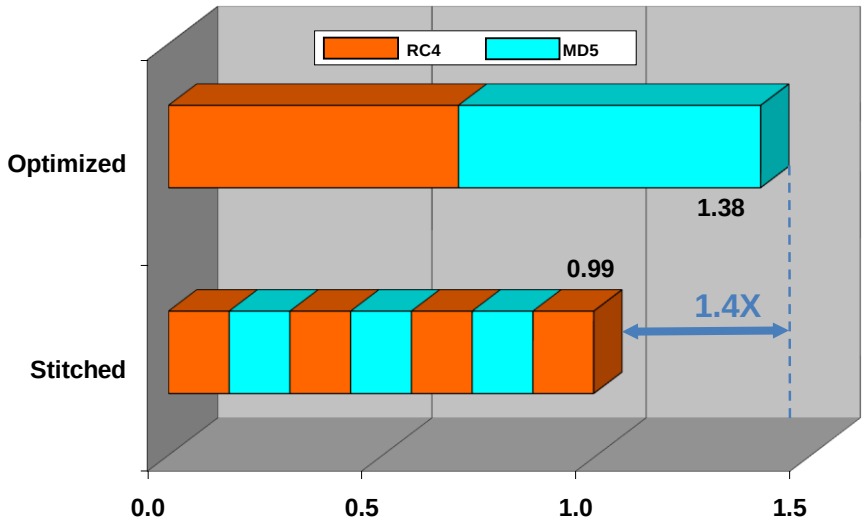
***Note:*** The throughput of the memory subsystem on every level is much higher than peak achievable throughput of computationally intensive crypto algorithms; thus different buffer sizes lead to only very slight drifts to the performance results provided.

Results were achieved running baseline sequential pair of functions and stitched function on all 12 cores available on a dual-socket system running two 6-core Intel® Xeon® 5670 processors.

## Performance Results

The RC4-MD5 pair is part of the SSL protocol, and is still one of the most commonly used pairs for secure communications. Figure 4 shows the performance of sequential and stitched implementations of the RC4-MD5 pair as an example of the integer-integer stitch. Although our baseline optimizations already provide at least 1.2X better performance than broadly available standalone implementations of RC4 and MD5, the stitched implementation results in an even larger performance gain of ~**1.4X**.

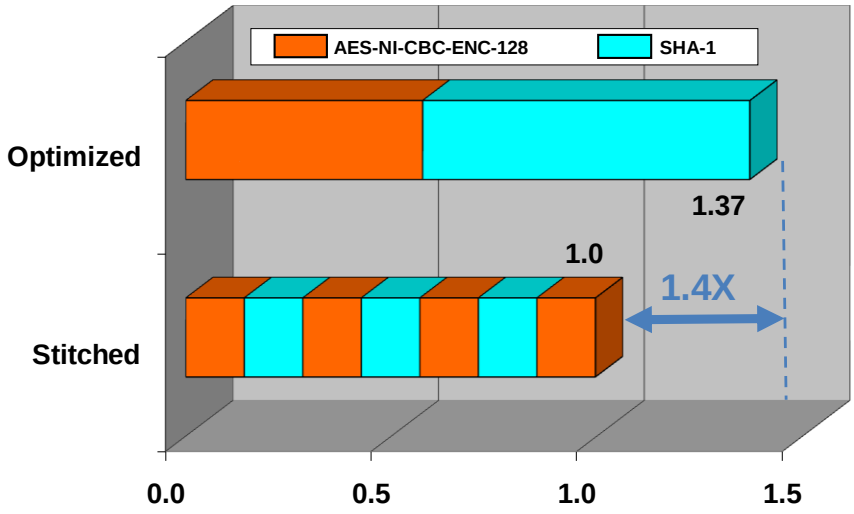
**Figure 4. RC4-MD5: Time, in seconds, to Process 40 Gb of Data**




Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to: http://www.intel.com/performance/resources/benchmark_limitations.htm

The AES128-CBC-Encrypt-SHA1 pair is part of a more modern and broadly used TLS 1.2, very common in secure communications. Figure 5 shows the performance of AES128 in CBC-Encrypt mode with SHA1. With improved SHA-1, and AES-128 implemented with Intel® AES-NI, our baseline is at least 2.6X faster than broadly available non-Intel® AES-NI implementations. However, with SSE-integer stitching, we achieve ~**1.4X** additional performance gain.

**Figure 5. AES128-SHA1: Time, in seconds, to Process 40 Gb of data**




Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to: http://www.intel.com/performance/resources/benchmark_limitations.htm

Figure 6 shows the performance of AES-256 (CBC-Encrypt mode) with SHA1. We achieve ~**1.5X** improvement with SSE-integer stitching over the optimized sequentially-called pair of AES-256 and SHA-1 functions.

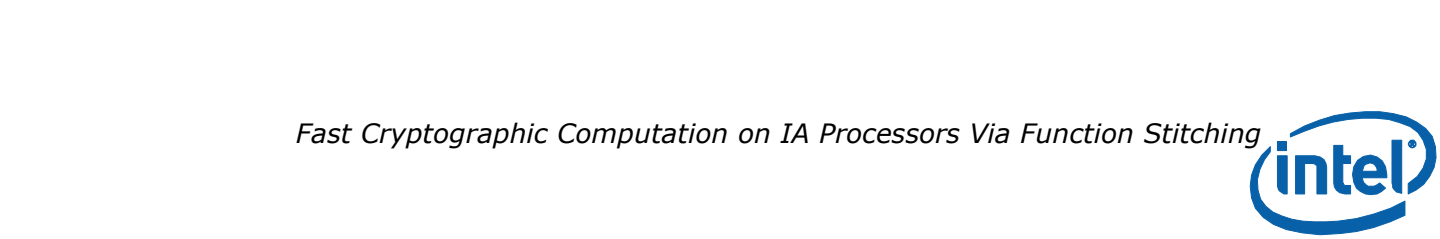
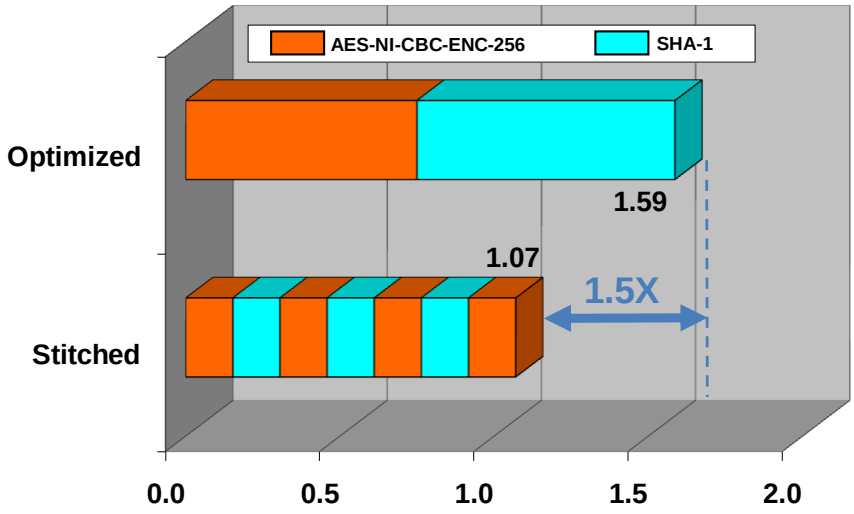


**Figure 6. AES256-SHA1: Time, in Seconds, to Process 40 Gb of Data**



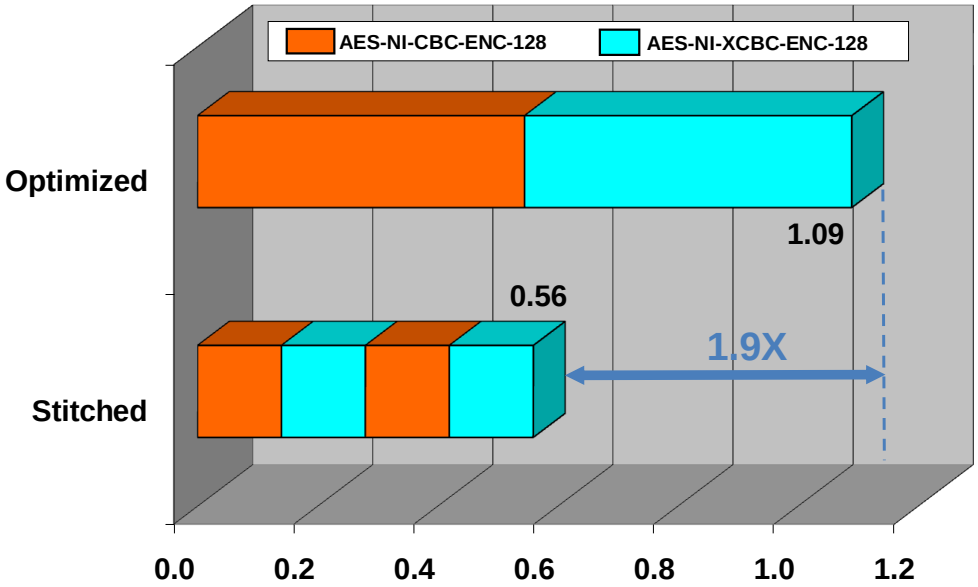

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to: http://www.intel.com/performance/resources/benchmark_limitations.htm

***Note:*** AES-256 performs 14 rounds of encryption compared to AES-128 that performs 10 rounds, and one would expect AES-256 to cost 1.4X as much, as is the case with standalone code. However, due to stitching, we are able to "hide" most of the AES computations inside the SHA-1 function which is the one with the longer compute time. Thus the net performance "penalty" of using more secure AES-256 over AES-128 when stitched with SHA-1 becomes almost negligible ~1.07X. This allows us to remove the performance factor out of the trade-offs for choice of cryptographic strength.

The same applies to the choice of newer AES128-SHA1 over RC4-MD5; with stitching the newer cipher-authentication pair does not come with any performance penalty as both end up having precisely the same performance when stitched.

Figure 7 shows the performance of AES-128 (CBC-Encrypt mode) with AES-128-XCBC for authentication. We created our optimized sequential implementation using the Intel® AES-NI as the baseline. The second bar represents the performance achieved by stitching the two functions, an example of SSE-SSE stitching. We can thus process **AES128-CBC-ENC-XCBC** with a performance gain of **1.9X** over the best sequential code.

**Figure 7. AES128-CBC-ENC-XCBC: Time, in Seconds, to Process 40 Gb of Data**




Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to: http://www.intel.com/performance/resources/benchmark_limitations.htm

# Conclusion

We introduced a novel method to improve computing performance called *function stitching*, and demonstrated its benefits on the example of cryptographic functions. It unleashes the full potential of the modern superscalar microarchitecture of Intel processors. Stitching is the source-level fine grained interleaving of the processing of independent instruction streams.

We stitched the most common pairs of cryptographic functions for increased performance, such as RC4-MD5, AES128-SHA1 and AES256-SHA1, achieving excellent performance improvement with the speedups in the **1.4X-1.9X** range over the optimized individual functions called sequentially. Stitched functions are able to execute very efficiently on IA cores, closely approaching IA core peak execution limits.

Stitching also allowed us to take popular pairs of block ciphers and hash algorithms such as RC4-MD5, AES128-SHA1 and AES256-SHA1 to the same highly improved performance level; now the choice of cryptography strengths comes without any performance trade-off.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. http://intel.com/embedded/edc.

# References

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1

[2] Improving the Performance of the Secure Hash Algorithm (SHA-1) http://software.intel.com/en-us/articles/improving-the-performance-of-the-secure-hash-algorithm-1/

[3] Software pipelining http://en.wikipedia.org/wiki/Software_pipelining

# Appendix A MD5 Data Dependencies

Stitching is motivated by the fact that many predominant cryptographic algorithms used in encryption and authentication have inherent definitions that severely limit parallelism opportunities, causing under-utilization of the execution resources in processor cores. *In addition, microarchitectural considerations may also add latencies or cause sub-optimal resource utilization within a single algorithm*.

As an example, consider the MD5 algorithm used in authentication. MD5 is a block-chained algorithm for computing a digest, working on 512-byte blocks of data. The digest consists of four 32-bit words {A, B, C, D}. Until the digest for a given block is computed, we cannot process the next block. Within each block, the algorithm is defined in a way that limits parallel execution as explained below.

The processing of a block consists of four similar phases consisting of 16 steps each. The basic steps are defined as:

**A = B + rotate(f(B,C,D) + W[i] + const + A)**

**D = A + rotate(f(A, B,C) + W[i+1] + const + D)**

**C = D + rotate(f(D,A, B) + W[i+2] + const + C)**

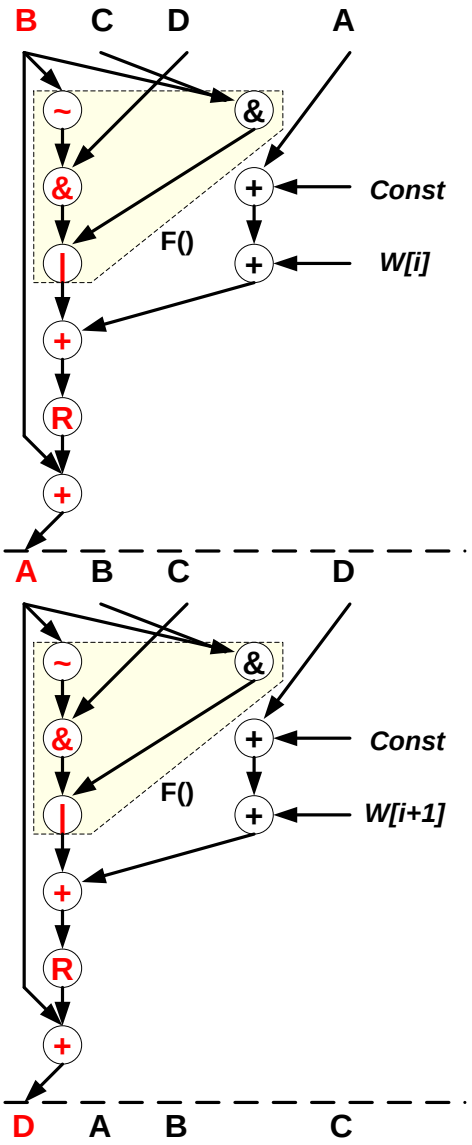
**B = C + rotate(f(C,D,A) + W[i+3] + const + B)**

The MD5 algorithm uses different constants and different functions f(…). We can illustrate the basic dependency with the F function; other functions are also based on logical operations and have similar complexities:

**F(x,y,z) = (x & y) | ((~x) & z)**

The MD5 algorithm uses a mix of rotates, logical operations and additions in a way that prevents reordering in any significant way, e.g., addition does not distribute over rotates/logical. Consider two consecutive steps of the algorithm as shown in Figure 8:

**Figure 8. Two Consecutive Steps of MD5 and the Critical Data Dependency Chain**



In Figure 8, R denotes a rotate operation by a constant. It can be seen that the algorithm has a tight dependency chain highlighted in red that spans across the steps. At most only two operations can execute in parallel; on the average, we can only execute one operation due to the chains.



## Authors

**Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Wajdi Feghali** and **Martin Dixon** are IA Architects with the IAG Group at Intel Corporation.

**Max Locktyukhin** and **Maxim Perminov** are Software architects with the SSG Group at Intel Corporation.

## Acronyms

| | |
|---|---|
| ALU | Arithmetic and Logical Unit |
| IA | Intel® Architecture |
| ILP | Instruction Level Parallelism |
| SIMD | Single Instruction Multiple Data |
| SSE | Streaming SIMD Extensions |