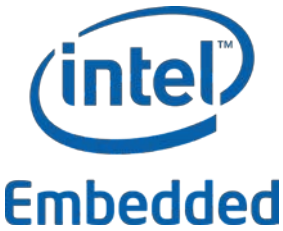


White Paper

Vinodh Gopal
Jim Guilford
Wajdi Feghali
Erdinc Ozturk
Gil Wolrich
Martin Dixon
IA Architects
Intel Corporation

Processing Multiple Buffers in Parallel to Increase Performance on Intel® Architecture Processors

July 2010

324101

Executive Summary

SIMD (Single Instruction Multiple Data) is a well established technique for increasing performance on processors when working on identical workloads. In this case, multiple independent data buffers can be processed simultaneously using SIMD instructions and registers. SIMD can be efficiently implemented on Intel® processors.

A similar “multi-buffer” approach can also speed up certain algorithms where SIMD instructions do not exist, and where data dependencies preclude optimal utilization of the processor’s execution resources. Examples of this include AES-CBC-Encrypt and 3DES.

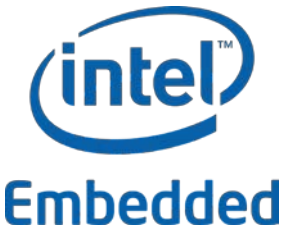
In applications where the workloads are not identical, in particular where the sizes of the buffers being processed vary, we describe a “scheduler” to utilize the multi-buffer routines efficiently. The main challenge is to implement a scheduler with minimal performance overheads, and realize good overall performance gains with the multi-buffer technique despite the presence of significant numbers of small-sized buffers.

This paper describes how processing multiple independent data buffers in parallel can dramatically improve performance, even without SIMD. A scheduler allows this approach to be used even when the size of each individual buffer varies. The net result is a 2X - 3X improvement in performance over the best-known single-buffer methods on Intel® processors.

The paper presents these ideas with pseudo code for the schedulers, along with performance results on cryptographic algorithms measured on an Intel® Core™ i5 processor 650.

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. www.intel.com/embedded/edc.

Contents



Overview

There are many algorithms, such as hashing or encryption, which are applied to a stream of data buffers. This occurs in networking, storage and other applications. Since the amount of data being processed is large, there is an ever-increasing need for very high performance implementations of these algorithms.

In many cases, one way to do this is to process multiple independent buffers in parallel. For example, a networking application might be encrypting each data packet. Each packet is encrypted independently from the other packets. This means that it should be possible to process several packets at the same time.

This may also be done when there is not a stream of buffers in a literal sense. For example, in a data de-duplication application, the first step is usually to partition the input data into a number of chunks, and then to compute the hash digest of each chunk. This is a perfect case where hashing multiple buffers in parallel can speed up the hashing step, as they are independent.

Implementations of the multi-buffer techniques may involve changes at the application level, but can result in speed improvements of **2-3X** on Intel processors. The multi-buffer technique increases the performance of a single thread, similar to the Stitching methods in [4], but is a complementary and different approach.

One of the main challenges is to design a scheduler that can process the multiple data buffers of different sizes with minimal performance overheads. This paper shows how this can be done, illustrates this with pseudo code, and presents the measured performance gains.

Processing Multiple Buffers in Parallel

There are two basic ways that processing multiple buffers in parallel can improve performance: processing the buffers with SIMD instructions or processing multiple buffers in parallel to reduce data dependency limits.

SIMD Approach

The Intel® 64 and IA-32 instruction set architectures have two distinct instruction subsets: general Purpose instructions and Single Instruction Multiple Data (SIMD) instructions [1]. SIMD instructions include, and are mostly known as, Intel® SSE, SSE2 (Streaming SIMD Extensions) etc. extensions.

A straight-forward way to process multiple buffers in parallel is using SIMD instructions. This directly allows one instruction to operate on multiple data items in parallel. Current SIMD extensions work on 128-bit XMM registers.

Each XMM register is 128-bits wide, and can hold, for example, four 32-bit values. This means that a single SIMD instruction can operate on four 32-bit values at the same time rather than just one 32-bit value. This allows the SIMD approach to process 4X the amount of data per instruction. The general purpose instructions are executed in multiple parallel execution units. Some SIMD instructions can also execute on multiple execution units, but the number of units is usually fewer than the units for general purpose instructions. In both cases this results in more than one instruction executing each cycle. The net effect is that the actual gain when going to a 4-way SIMD implementation may be less than 4X.

Non-SIMD Approach

A less obvious case for processing multiple buffers in parallel is where the algorithm or implementation cannot be implemented in SIMD, and where data dependencies prevent multiple-issue cores from taking full advantage of execution unit resources, or where instruction latency exceeds instruction throughput, so that the execution unit is under-utilized.

One example of this is the AES encryption algorithm which can be implemented on Intel's 32-nm technology microarchitecture, using the Intel® AES New Instructions (Intel® AES-NI) that have been defined as a set of SSE instructions, but that are not SIMD. In particular the AESENC instruction, which does one round of AES encryption, has a latency of several cycles. This means that in some modes, such as counter-mode or CBC (Cipher Block Chaining) decrypt, one can implement the algorithm such that multiple blocks of the same buffer are being processed in parallel, but in the case of CBC-encrypt, one cannot start encrypting a block until the previous block has been encrypted. This means that CBC-encrypt requires a serial implementation, where performance is limited by the latency rather than by the throughput.

However, if we can encrypt multiple independent buffers in parallel, we can break the data dependencies and get ideal performance limited only by the throughput.

Another example is 3DES (Triple Data Encryption Standard). In this case, there are no special DES instructions, and the implementation is built on basic logical primitives such as XOR, shifts and table lookup operations. In this case, there are multiple execution units that can execute these instructions, but due to the data dependencies inherent in the 3DES algorithm, many of these execution units end up under-utilized. If we interleave the code to perform two 3DES operations in parallel, we can reduce those data dependencies, make better use of the execution unit resources, and markedly improve the performance.

Multi-buffer Scheduler for Arbitrary Length Buffers

Basic multi-buffer implementations work on identical workloads. In particular, they work on multiple buffers as long as all of the buffers are the same length. But in many applications the buffers are of different lengths. To bridge this gap we use a "scheduler".

The scheduler is an interface layer that presents a single job-oriented interface to the application, and a parallel interface to the underlying algorithm. It takes advantage of the property that these algorithms iterate on the data in a buffer. The processing for a single buffer could be done in one operation, or by doing a series of separate operations on sequential portions of the buffer.

For the sake of the following discussion, assume that the underlying algorithm processes four buffers in parallel. The basic idea behind the scheduler is that it accumulates jobs until there are four of them. It then computes the minimum of the job sizes and then calls the underlying algorithm to process that much data. After this completes, one or more of the jobs are finished. The completed jobs are replaced by new jobs and the process continues.

Basic API

The basic API of the scheduler centers around a "job" object. This is a data structure that completely describes the work item, e.g. the address and size of the buffer, along with other necessary data such as the initial and final hash values, etc.

The basic operation is that through a function call a job is submitted, and it either returns a job or returns NULL. If it is not NULL, then it represents a job whose processing has finished. Note that in general, the returned job will **not** be the same as the submitted job.

There is a related operation, flush, which does not take a new job as input, and which either returns a finished job or returns NULL if there are no remaining jobs in progress.

"In-order" vs. "Out-of-order"

The scheduler can be designed to return jobs in the same order as the order in which they were submitted, or in a different order. Some applications may tolerate jobs being completed in an arbitrary order, whereas other applications might not.

The out-of-order (OOO) scheduler is simpler. It also features more deterministic behavior. For example, consider an out of order scheduler for an algorithm operating on four buffers at a time. When the first three jobs are submitted, there will be no job returned. Thereafter for every job submitted there will be one returned. This implies that there need be no more than four job objects in existence.

If the jobs need to be completed in order, a more involved scheduler can be used. In this case, if a job is finished before an earlier job, it remains within the scheduler and isn't returned until after the earlier jobs are returned. This implies that depending on the sizes of the jobs, an arbitrary number of jobs can reside within the scheduler.

There are many ways one could design such an in-order scheduler. In our prototype design, there is a fixed number of job objects available. If the pool of job objects becomes exhausted because the earliest job is taking a long time and all of the other jobs are waiting for it, then the scheduler will "flush" jobs until that earliest job completes.

Since flushing reduces the efficiency, the size of the job pool should be chosen large enough so that normally it is not exhausted.

Usage

The basic usage of the scheduler can be illustrated by the following pseudo code:

```
while (work to be done) {
        JOB *job;
        job = get_job();
        // fill in job data fields
        job = submit_job(job);
        if (job) {
                // complete application job processing
                return_job(job);
        }
} // end while
while (NULL != (job = flush_job())) {
        // complete application job processing
        return_job(job)
}
```

The **submit_job**() and **flush_job**() functions are provided by the scheduler, while the *get_job*() and *return_job*() functions can be provided by the application or scheduler. For our prototype in-order scheduler, these functions are provided by the scheduler.

In the case of an out of order scheduler, the *get_job()* and *return_job()* functions can be implemented as a simple stack of static objects.

Note that other usages are possible. For example if one were using an out-of-order scheduler that processed 4 buffers in parallel, then one could do something simpler as described below:

```
JOB *job, jobs[4];
for (i=0; i<3; i++) {
        // fill in jobs[i] data fields
        submit_job(&jobs[i]);
}
job = &jobs[3];
while (more work to be done) {
        // fill in job data fields
        job = submit_job(job);
        // job will never be NULL here
        // complete application job processing
}
for (i=0; i<3; i++) {
        job = flush_job();
        // complete application job processing
}
```

This takes advantage of the observation that for this out of order scheduler, the first three jobs submitted will always return NULL, and thereafter submitting a job will always return a completed job. Finally, at the end, there are exactly three jobs to flush.

Starvation

If the arrival of data buffers is bursty in nature, a simple use of the scheduler could be susceptible to starvation, where a particular job is delayed indefinitely. For example, consider the case of a queue of work requests, and a thread processing them through the scheduler. If no new requests arrived and the queue became empty, the last few jobs would be stuck in the scheduler until new requests arrived, which might not occur for a long time.

A variety of approaches could address this. In the above example, the processing thread could, when it finds the work queue empty, start flushing jobs, until either new work items arrived or all of the jobs were flushed. Flushing jobs is less efficient than normal processing, but if the queue is empty then presumably the rate at which work arrives is less than the processing rate, and we can therefore afford the temporary decrease in performance.

In other applications, one might have a watchdog thread that records that no new jobs have arrived in a given time period and starts flushing jobs.

Alternately, the application might have a definitive start and end and therefore not be subject to starvation. An example of this would be hashing the chunks of a larger buffer for data de-duplication.

Performance

The performance results provided in this section were measured on an Intel® Core™ i5 650 processor at a frequency of 3.20 GHz, supporting Intel® AES-NI. The tests were run with Intel® Turbo Boost Technology off, and represent the performance without Intel® Hyper-Threading Technology (Intel® HT Technology) on a single core.

Performance was measured for two code bases. The first implemented HMAC-SHA1. The second implemented AES-128 CBC Encrypt using pre-expanded keys.

For the HMAC case, the multi-buffer code used an out-of-order scheduler and a SIMD-style SSE-based multi-buffer SHA1 kernel. The single-buffer code for comparison used the best-known single-buffer SHA1 kernel [2], which utilizes the SSE hardware for part of the single-buffer calculation, as well as other optimizations.

The HMAC is a keyed message authentication code based on an underlying hash function such as SHA1. It is defined as:

HMAC(K,m) = H((K ⊕ opad) ‖ H((K ⊕ ipad) ‖ m))

where H is the SHA1 hash function, m is the input “message” data buffer and K is the secret key. The ipad and opad are constants defined by the algorithm to be exactly 1 block in length. We assume that the SHA1 digest of (K ⊕ ipad) and (K ⊕ opad) are pre-computed. Details of the HMAC-SHA1 algorithm can be found in [3].

For the AES case, both the multi-buffer and single-buffer code used the AES-NI new instructions. The multi-buffer code demonstrates an in-order scheduler and a non-SIMD multi-buffer kernel. The single-buffer code used the best-known single-buffer AES implementation [5].

These code bases were tested on four different workloads:

Workload	Description	Average Buffer Size
0	Fixed “min sized” buffers: 64 bytes for HMAC 48 bytes for AES	64/48
1	Random Distribution 0: 66% 64 21% 544	330

	13% 1360	
2	Random Distribution 1: 40% Uniformly distributed from 48 – 100 20% Uniformly distributed from 101 – 1023 40% Uniformly distributed from 1024 - 1450	610
3	Fixed “large” buffers: 2048 bytes	2048

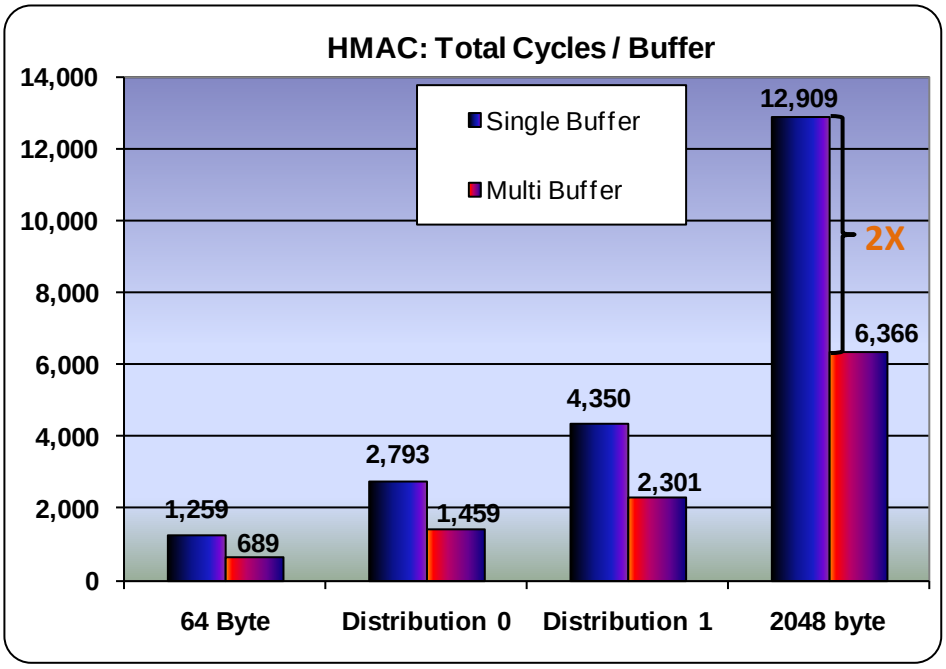
The first and last workload represent “limiting” cases of small and large buffers, and the middle two workloads represent a rough approximation to network traffic sizes. We picked two distributions to show that the performance is not very sensitive to the exact nature of the distribution.

Note: Performance results are based on certain tests measured on specific computer systems. Any difference in system hardware, software or configuration will affect actual performance. Configurations: OS: Windows Server 2008 R2 Enterprise, 64-bit, CPU: Intel® Core™ i5 650 3.20 GHz, Memory 8.00 GB. Testing conducted as described in this section. For more information go to http://www.intel.com/performance

HMAC-SHA1 Performance

The overall HMAC performance is given below:

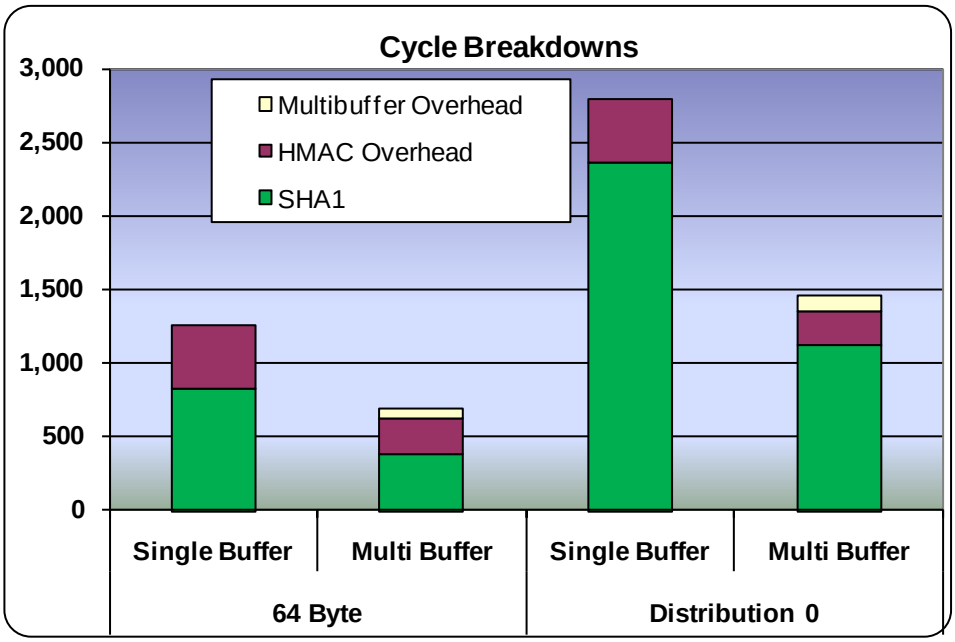
Figure 1. HMAC-SHA1 Performance (Cycles/Buffer)



In all cases, including the multi-buffer scheduler overhead, the multi-buffer version provided about **2X** the performance of the best single-buffer implementation.

Looking at the first two cases in more detail, the cycles break down as:

Figure 2. Breakdown of Cycles for HMAC-SHA1



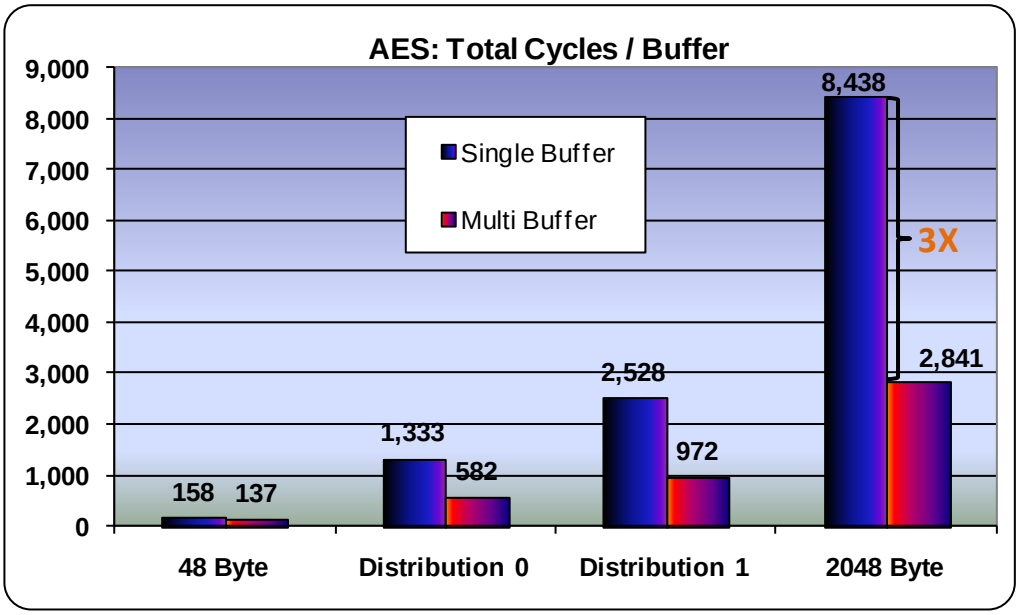
These graphs clearly show that even for small buffers, the overhead of the multi-buffer scheduler is still much smaller than the cost of computing the HMAC digest, and that the improved SHA1 performance causes the overall performance increase.

Note that for the other two workloads, the overheads associated with HMAC and with the multi-buffer scheduler remain essentially the same, while the number of cycles spent hashing the buffer contents increase. For these cases, the overheads become an even smaller percentage of the total time.

AES128 CBC Encrypt Performance

The overall AES results are shown below:

Figure 3. AES128 CBC-Encrypt Performance (Cycles/Buffer)



In this case, the increase in performance due to multi-buffer varies from **1.2X** on min-sized buffers to **3X** on large buffers. This is to be expected as the time to encrypt a min-sized buffer is small enough that the multi-buffer overhead becomes a significant fraction of the encrypt time. The improvement is modest for minimum-sized buffers but becomes much more significant for larger buffers.

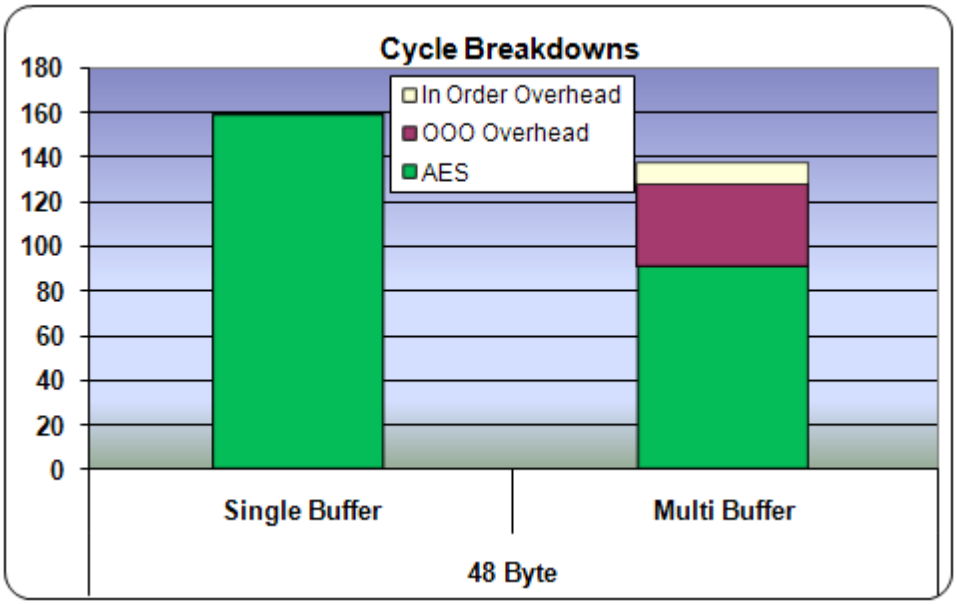
If the entire workload consisted of 48-byte buffers, then the 1.2X improvement may not seem very high. However, in practical applications, we expect a distribution of buffers where the multi-buffer approach has a much larger gain, sufficient to justify the complexity. Furthermore, if all of the buffers were 48 bytes, one could use a simpler scheduler for better performance.

The main point to note is when the buffers have a reasonable mix of sizes (e.g. the other three workloads), then the improvement from using a multi-buffer approach varies from about **2.5X to 3X**.

We should also note that in reality, there will be a mix of encrypt and decrypt jobs, and since CBC Decrypt can be explicitly parallelized without multi-buffer scheduling, the **absolute** performance for CBC on the whole will be better than the worst-case shown here for CBC Encrypt.

The cycle breakdown for the 48-byte worst-case is shown below:

Figure 4. Breakdown of Cycles for AES128 CBC Encrypt (Min size buffer)



From this, we can see that even for small buffers, the out-of-order overhead is still small enough that the multi-buffer version performs better than the single-buffer version, and also that the overhead for returning buffers in order is yet smaller. As the average buffer size increases, the relative contribution of the scheduler overhead becomes much smaller.

Implementation Details

Scheduler API

HMAC-SHA1 (Out of Order Scheduler)

The scheduler API could be designed a number of different ways. For the HMAC (out of order) scheduler, we implemented:

```
void init_mb_mgr_state(MB_MGR_STATE *mb_mgr_state);
JOB* submit_job(MB_MGR_STATE *mb_mgr_state, JOB *job);
JOB* flush_job (MB_MGR_STATE *mb_mgr_state);
```

The basic paradigm is that the application fills in a “JOB” data structure with enough information to fully specify the work to be done (e.g. a pointer to the buffer, the buffer length, etc.) and submits it to the multi-buffer manager/scheduler. This then returns a completed job (in an arbitrary order) or NULL. At the end, the application calls **flush_job**() to get back the remaining jobs without submitting new ones. The basic application flow is:

```
job = NULL;
while (work to be done) {
   if (job == NULL) job = get_new_job;
   fill in job object with new data
   job = submit_job(mb_mgr, job);
   if (job) use completed HMAC hash value
}
while (job = flush_job(mb_mgr)) {
   use completed HMAC hash value
}
```

In this case, there never needs to be more than 4 jobs at any time, so the application could use a fixed pool of 4 jobs and avoid dynamic memory allocation overhead.

AES (In Order Scheduler)

An “in-order” scheduler API could also be designed a number of ways.

The in-order scheduler can have a larger number of buffers currently being processed. For example, if the first buffer submitted is very large, the application might need to submit a large number of smaller buffers before the first buffer is returned. And since the buffers need to be returned in the order in which they were submitted, all of these smaller buffers can’t be returned until the initial large buffer is returned.

So in this case, the buffer management is also handled by the multi-buffer manager/scheduler. The API implemented was:

```
void init_mb_mgr(MB_MGR_AES *state);
JOB_AES* get_next_job(MB_MGR_AES *state);
JOB_AES* submit_job(MB_MGR_AES *state);
JOB_AES* flush_job(MB_MGR_AES *state);
```

The paradigm here is that **get_next_job**() returns a pointer to a job object, which is filled in similar to the HMAC case. The **submit_job**() function submits that job (which is an “implicit argument”). It returns NULL or a completed job. If it returns a job, then the application needs to complete processing on that buffer before it next calls **get_next_job**().

The **flush_job**() function takes no “implicit job” in as an argument, but it returns completed jobs until there are no more jobs to return, in which case it returns NULL.

In this case, the application logic can be described as:

```
while (work to be done) {
   job = get_next_job(mb_mgr);
   fill in job object with new data
   job = submit_job(mb_mgr);
   if (job) process completed AES job
}
while (job = flush_job(mb_mgr)) {
   process completed AES job
}
```

Note that this application code outlined is almost identical to that for the HMAC-SHA1 out-of-order case.

Scheduler Internals

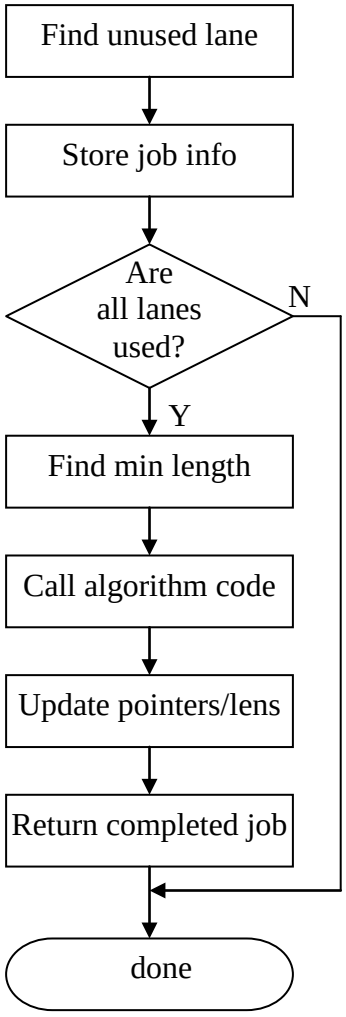
Generic Out-of-Order Scheduler

A generic out-of-order scheduler has a fairly simple implementation. Assume that the underlying algorithm code operates on N buffers in parallel for an arbitrary number of blocks (where the block size is determined by the algorithm). The scheduler accumulates jobs until it has N of them. It then computes the minimum of the lengths of the N jobs, calls the algorithm function, and then updates the buffer pointers and lengths.

At this point, at least one of the jobs is completed and can be returned.

The only complexity is handling the case where the minimum length before the algorithm code is called, is zero. This happens when two or more buffers are completed at the same time.

Figure 5. Generic Out of order Scheduler



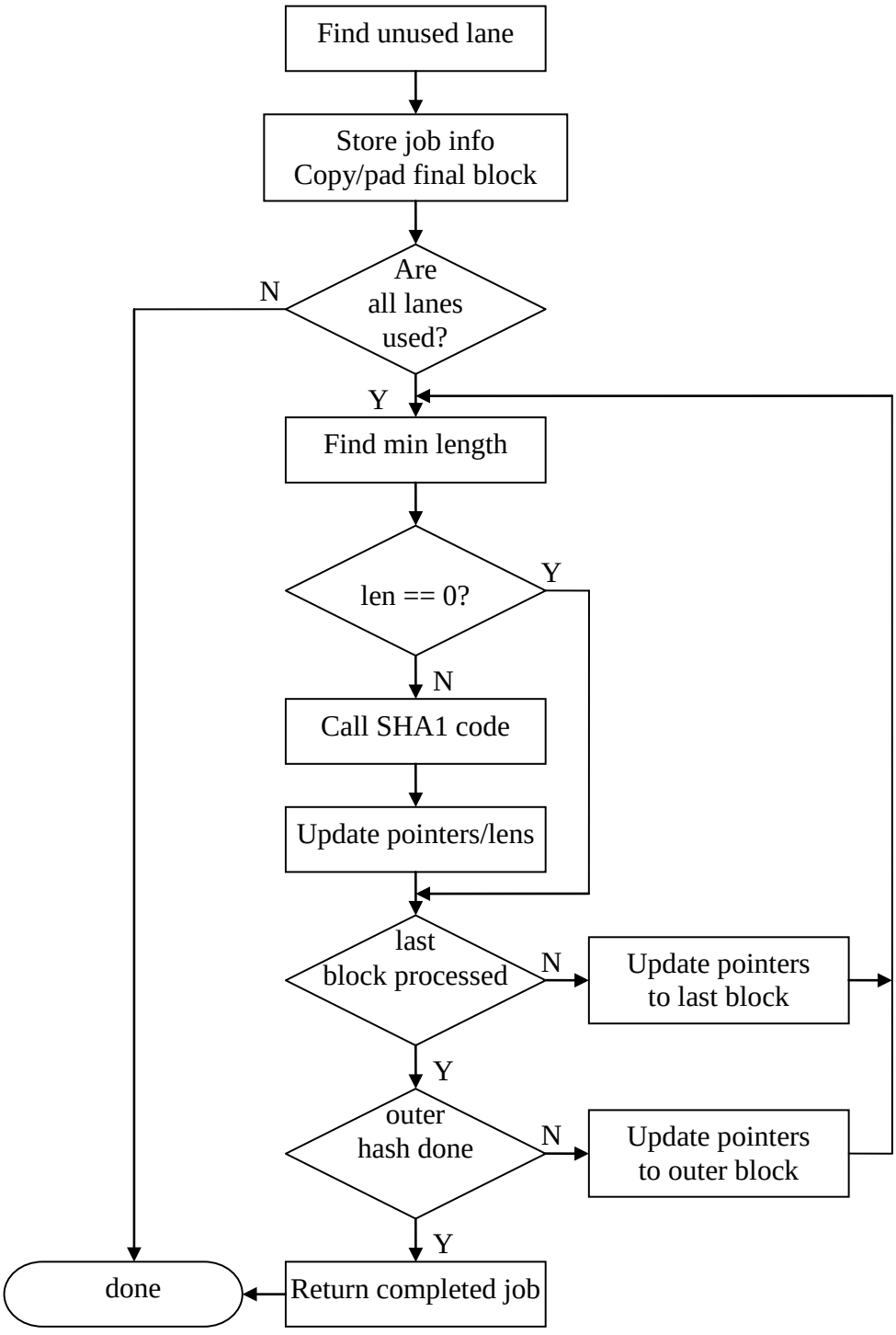
HMAC-SHA1 (Out of Order Scheduler)

The HMAC-SHA1 scheduler as implemented for this paper is more complicated, as it also needs to deal with the HMAC processing. Since the length of the buffer is arbitrary, the SHA1 algorithm requires padding to make the effective buffer length a multiple of 64 bytes. In particular this includes copying the last block to a temporary buffer and adding the padding. For convenience, we have implemented the SHA1 processing required for the last block as part of the HMAC code, which also is responsible for creating a buffer containing the inner hash that is hashed to generate the final outer hash.

This means that in general, each job is sent to the underlying SHA1 code three times: once for all but the last block, once for the last block (with

padding added to handle lengths that are non-multiples of 64 bytes), and once for the final outer HMAC hash. One consequence of this is that after performing a SHA1 hash, there may be no jobs which are actually completed. Therefore the logic that finds the minimum size job, calls the SHA1 code, and processes the results needs to be done iteratively in a loop.

Figure 6. HMAC-SHA1 out of order Scheduler



AES (In Order Scheduler)

The in-order scheduler is layered on top of an out-of-order scheduler. In this implementation, it has a fixed-size array of jobs arranged as a circular buffer. At least one of these is always unused, so it can be returned to the application when the application requests a new job.

When a job is submitted, it is submitted to the underlying out-of-order scheduler. Then there are three cases:

- If the scheduler was previously empty, then the current job is remembered as the “earliest job”.
- If the scheduler is not full, then if the earliest job is completed, that job is returned, otherwise NULL is returned.
- If the scheduler is full and the earliest job is not completed, the underlying out-of-order scheduler is flushed until the earliest job completes, and that job is returned.

This approach removes the need for dynamic memory allocation and bounds the worst-case latency in the sense that the number of “in-flight” jobs can never exceed some threshold.

Coding Considerations

Since the time to encrypt one block with AES using the AES-NI instructions is rather short, one must code the scheduler to be very efficient; otherwise the multi-buffer overhead may exceed the multi-buffer savings for small buffers.

The sizes of buffers in these applications are less than 2^{16}, and therefore we use the PHMINPOSUW SSE instruction to compute the minimum length in one operation, and avoid conditional logic and potential branch mis-predicts.

Conclusion

Processing multiple buffers at the same time can result in significant performance improvements—both for the case where the code can take advantage of SIMD (SSE) instructions (e.g. SHA1), and even in some cases where it can’t (e.g. AES CBC Encrypt).

An efficient scheduler or multi-buffer manager can be used to extend this approach to streams of buffers where in general each buffer will be of a different size. This scheduler can be designed to return completed buffers in an arbitrary order or in the same order in which jobs were submitted.

Even in the case of an in-order scheduler layered on top of AES for small buffers (the worst case), the multi-buffer code with scheduler still performs

better than the best single-buffer version. For larger buffers and for HMAC-SHA1, the performance increase is more dramatic.

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] HMAC http://en.wikipedia.org/wiki/HMAC

[4] Fast Cryptographic Computation on Intel® Architecture Processors Via Function Stitching http://download.intel.com/design/intarch/PAPERS/323686.pdf

[5] Breakthrough AES Performance with Intel® AES New Instructions http://software.intel.com/file/26898

Authors

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

Acronyms

IA	Intel® Architecture
API	Application Programming Interface
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions