NPS-CS-10-011

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

PARALLELIZING SHA-256, SHA-1 AND MD5 AND AES ON THE CELL BROADBAND ENGINE

By

George Dinolt
Bruce Allen
David Canright
Simson Garfinkel

October 25, 2010

Approved for public release; distribution is unlimited

Prepared for: Defense Intelligence Agency
Washington, D.C.

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

This report was prepared for and funded by the Defense Intelligence Agency (DIA), Washington, DC.

George W. Dinolt
George Dinolt
Associate Professor

Bruce Allen
Bruce Allen
Research Associate

David Canright
David Canright
Associate Professor

Simson L. Garfinkel
Simson Garfinkel
Associate Professor

Reviewed by:

Peter J Denning
Peter Denning
Chairman
Department of Computer Science

Released by:

Karl A van Bibber
Karl A. van Bibber
Vice President and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
25–10–2010

2. REPORT TYPE
Technical Report

3. DATES COVERED *(From — To)*
2009-10-01—2010-09-30

4. TITLE AND SUBTITLE
Parallelizing SHA-256, SHA-1 and MD5 and AES on the Cell Broadband Engine

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
George Dinolt, Bruce Allen, David Canright, Simson Garfinkel

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-10-011

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DIA

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

The Cell BE Architecture connects a Power processor with several "synergistic processing units" via a high-speed bus, allowing parallel processing on a chip. Architectural features enabling high speed performance include SIMD, many wide registers, DMA provisions, and dual-issue instructions. We have developed extraordinarily high performance implementatioins of SHA-256, SHA-1 and MD5 for this architecture. We have also developed parallelized implementations of AES Encryption.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	55	

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

Abstract

The Cell BE Architecture connects a Power processor with several "synergistic processing units" via a high-speed bus, allowing parallel processing on a chip. Architectural features enabling high speed performance include SIMD, many wide registers, DMA provisions, and dual-issue instructions. We have developed extraordinarily high performance implementatioins of SHA-256, SHA-1 and MD5 for this architecture. We have also developed parallelized implementations of AES Encryption.

1 Introduction

The Cell BE is a general purpose microprocessor architecture designed by Sony, Toshiba and IBM that features a heterogenous processing architecture and a high-speed interconnect between the processing elements and the system memory. The Cell Broadband Engine (Cell BE) is a 3.2 Ghz microprocessor available on the Sony Playstation 3, on the dual-processor configuration on an IBM QS22 card for the IBM BladeCenter, and as a PCI co-processor card available from IBM and Mercury Computer Systems.

This paper presents our results in developing high-speed implementations of SHA-256, SHA-1 and MD5 hashing algorithms. Although the Cell has a reputation as being hard to program, we show that exercising direct control over a small number of independent processing elements makes it possible to achieve extraordinarily high performance on real-world problems.

In November 2009 IBM confirmed rumors that there would be no further development on the existing Cell processor line[7]. Nevertheless, the results of this paper are important for several reasons:

1. These results show that architectural provisions on the Cell such as SIMD lanes and DMA can bring significant computational power to real-world problems on commodity hardware.
2. Although the current generation of Cell architecture is being abandoned, IBM will be including the Cell VLIW cores in upcoming versions of its Power 7 microprocessors targeted at supercomputers. IBM's QS22 blades will remain in production for more than two years. Organizations needing high-performance functions today with the flexibility of a general purpose CPU and enterprise-class hardware can use our code today.
3. Many of the lessons learned from developing high-speed implementations of these algorithms on Cell will be applicable to future heterogeneous architectures.

We present the following:

- Implementation strategies used to optimize hashing.
- Timing results for hashing operations.
- Additional timing results for AES Encryption.

1.1 Prior Work

IBM has reported the speed of AES encryption on the Cell BE as one of its standard benchmarks from the first of its published Cell papers. The code that produces those timing results has never been publicly released.

Yang and Goodman demonstrated AES running on an AMD HD 2900 XT GPU with speeds “16 times faster than high end CPUs.[8]”

Harrison and Waldron implemented AES on the GeForce 6600GT and GeForce 7900GT graphic coprocessor cards and found that the performance of the GPUs was limited by the speed that data could be transferred to and from the GPU[4].

Bhaskar *et al.* developed a strategy for efficient Galois Field Arithmetic implementation on SIMD Architectures[1].

Costigan and Scott developed a version of SSL that used the Cell BE for cryptographic support[3].

2 Implementation

The implementation of our hashing algorithms utilize the architecture of the Cell BE:

- SIMD lanes, instruction pipelining, and many registers
- Parallel Processing
- SPU initialization
- PPU task delegation
- Shared data structures
- DMA
- Synchronization

2.1 SIMD lanes, instruction pipelining, and many registers

In implementing hashing and encryption on the Cell BE, we have exploited several distinct levels of parallelism. (Here we use a broad meaning of parallelism: utilizing different hardware resources simultaneously in achieving a task.)

1. SIMD instructions can operate on several parts of registers in parallel.
2. Each instruction pipeline can perform (different stages of) several instructions at once.
3. The two pipelines in an SPU can both execute instructions in parallel.
4. Multiple SPUs can work on a task simultaneously, in a pipelined fashion and/or performing the same steps on parallel data.
5. In a dual-processor configuration, both Cells can work in parallel.

By optimizing all these levels of parallelism, our implementations of hashing and encryption set new speed records for this hardware.

To control the three lowest levels of parallelism, we wrote functions to hash or encrypt a bufferful of data in SPU assembly code, and optimized these by hand for speed. Careful scheduling of instructions minimized data latency, where one instruction waits for the result of a previous instruction; this involved finding steps in the algorithm that could be done in parallel over a few clock cycles, for parallelism in a single pipeline (level 2 above).

Another goal was balancing instructions between the two pipelines (level 3), which is complicated by the fact that the instruction sets for the two pipelines are completely disjoint. However, sometimes it was possible and advantageous to replace an instruction in one pipeline by a different one or even several instructions in the other pipeline to effect the same result with better parallelism. (The instruction scheduling also needed to satisfy memory alignment constraints to get dual issue: instructions going to both pipelines at the same clock cycle.)

For all the hashing implementations, we fully unrolled the round loop. For encryption (CTR and ECB modes), we did not unroll the round loop, but partially unrolled the outer block loop, to encrypt four blocks together. Other techniques for increasing speed include:

- Instead of conditional jumps, we used the SPU's predicate operations, allowing execution

to progress at full speed without cache misses.
- Dynamic branch hinting allowed cache pre-fetches for looping without penalty.

The SIMD parallelism (level 1 above) varied depending on the algorithm. For AES, the 128-bit block size matches the 128-bit vector register size of the SPU. The SPU SIMD instructions made it possible to operate on the 128-bit register as a whole (ShiftRows, AddRoundKey), as four 32-bit words (MixColumns), or as sixteen bytes (SubBytes table lookups), allowing us to perform the AES round operations on the whole block at the same time. (We also tried the standard T-table approach that combines SubBytes, ShiftRows, and MixColumns into simple table look-ups, but that was less efficient than the SIMD parallelized approach.)

All three hashing algorithms are defined in terms of a sequence of operations on 32-bit words, which fits nicely with the word size of the SPU. However, all these operations correspond directly to "even"-pipeline instructions; none are available directly in the "odd"-pipeline. While this gives unbalanced pipelines, it does mean the "odd" pipeline is free to move words around or combine them for later SIMD operations. Also, when hashing a single file using SHA-256, it was possible to replace some of the word rotations ("even" instructions) with quad-word rotations ("odd" instructions), by keeping all four words of each register identical.

For hashing a single file, opportunities for SIMD acceleration are not so obvious, due to the sequential nature of the algorithms. The SHA algorithms involve "message scheduling" to extend the 16-word message block to a new word for each round. There, SIMD can do most of the message schedule for four rounds at once; the limitation is where results are needed that are less than four rounds old. In the hashing itself, SIMD was used for certain additions done four at a time, and some Boolean functions two words at a time (on hash words two or more rounds old).

A completely different approach to using SIMD is to do four hashes in parallel, for four different files (messages); we call this the 4-lane approach. (This use of SIMD precludes the others mentioned above.) When processing multiple files, this is by far the most efficient way. And this obviously extends to hashing more files using more SPUs (level 4 parallelism), up to 32 files simultaneously on the 8 SPUs of a Cell, for tremendous throughput.

If only a single source is to be hashed, as fast as possible, than that task can be pipelined between two SPUs (also level 4 parallelism). The message schedule can be preprocessed, since it depends only on the message data, completely independent of the current hash value. Similarly, the addition of round constants can be included in the preprocessing. (This approach does not apply to MD5, since it has no message scheduling, and moreover, we got the rounds to run in the minimum possible number of clock cycles.) Then the second SPU does all the actual hashing, relieved of the burden of the message schedule and round constants. This pipelining increases throughput of SHA-256 by about 16%.

The speed advantages of these two different approaches to parallelizing hashing, either using two SPUs in hashing one file, or using one SPU to hash four files at once, are shown in Table 3.

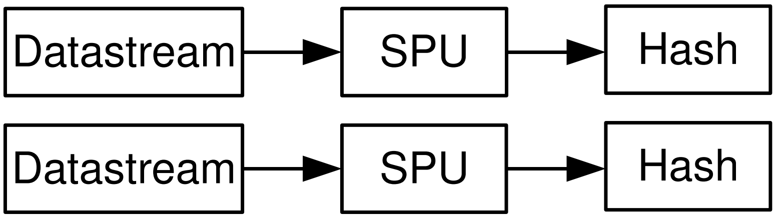


Figure 1: Datastreams are hashed independently on multiple SPUs.

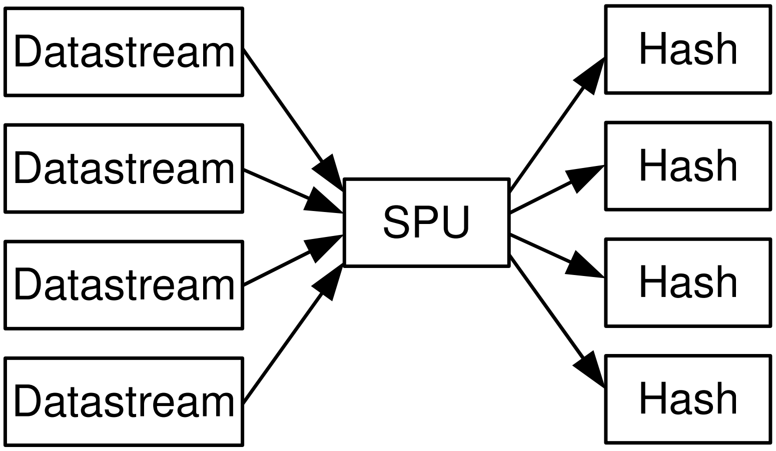


Figure 2: Four separate datastreams are hashed simultaneously on one SPU using SIMD lanes.

2.2 Parallel Processing

The calculation of hash values is a fairly linear approach because the computation of a block of data depends on the result of the block before it. Regardless, we are able to achieve parallelization using the following approaches:

1. Hashing irrelated data streams on separate SPUs as shown in Figure 1. This approach improves performance linearly as the number of SPUs are utilized. Hashing concurrently on all eight SPUs of a Cell BE improves performance by eight.
2. Hashing four irrelated data streams concurrently on one SPU as shown in Figure 2 by using the four SIMD lanes of the 128-bit registers and instructions of the SPU. We use two approaches for parallel processing.
3. Hashing one data stream and pipelining block processing across two SPUs as shown in Figure 3. The first SPU creates message digests from input text while the second SPU concurrently hashes message digests. Although this approach does not provide the greatest throughput for multiple streams, it does provide the minimum latency for one data stream.

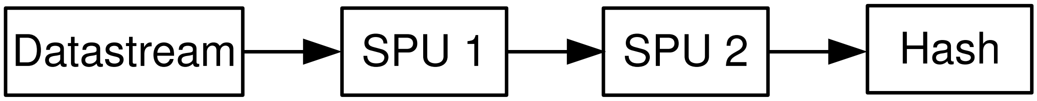


Figure 3: Blocks of a datastream are pipelined across two SPUs. In SPU 1, a message digest is produced. In SPU 2 the message digest is hashed.

2.3 SPU Initialization

The PPU starts one or more SPUs. During initialization, the PPU and each SPU shares with eachother the EA of their shared data structures, enabling data transfers and synchronization between processors.

Once initialized, the system is ready to perform hashing jobs.

Initialization steps are as follows:

1. The PPU clears all synchronization variables to ensure that they do not inadvertently indicate a signal event.
2. The PPU starts SPU(s), providing the EA of the PPU's shared data space in the argp value passed to the SPU when the SPU starts running. The SPUs use their assigned slot of this shared space to signal job completion.
3. The PPU begins to wait to obtain the data spaces of the SPUs. The PPU waits using the 32-bit hardware mailbox synchronization mechanism. DMA communication could have been used instead of the mailbox mechanism, but mailbox is simpler to implement, considering that the synchronization consists of a 32-bit blocking transfer.
4. As each SPU starts, it performs the following initialization sequence:
   (a) It clears all synchronization variables to ensure that they do not inadvertently indicate a signal event.
   (b) It transmits the LS address of its shared data space to the PPU.
   (c) It becomes ready to accept jobs.

For pipelined hashing, which requires two SPUs, the following additional processing is performed before the SPUs are ready to accept jobs:

1. The two SPUs wait for a second mailbox signal providing the EA of the shared data space of the other SPU.
2. The PPU calculates the EA of the shared data spaces of the SPUs by adding the LS of the shared data to the EA offset of the LS space for the given SPU.
3. The PPU sends the EA shared data space addresses of SPU0 to SPU1 and of SPU1 to SPU0.

```
while files do
    read file
    set job for file
    signal job to SPU
    wait for DONE signal from SPU
end while
send DONE signal to SPU
```

Figure 4: The PPU repeatedly tasks jobs to the SPU and consumes results until all jobs are done, then the PPU tasks DONE to terminate.

```
while true do
    wait for job
    break if job == DONE
    process job
    wait for DONE signal from SPU
    send calculated hash back to EA
end while
```

Figure 5: The SPU repeatedly accepts and process jobs until the job indicates DONE.

2.4 PPU Task Delegation

The SPUs repeatedly run job requests in a loop until the PPU says to stop. The PPU algorithm is presented in Figure 4. The SPU algorithm is presented in Figure 5.

For single-SPU jobs, the job request consists of the following:

- job ID
- address of the data
- count of data such as number of blocks or number of files
- address of the returned hash or hashes

For pipelined jobs, the job request is similar, but SPU0 does not receive the address of the returned hash or hashes, and SPU1 does not receive the address of the data.

2.5 Shared Data Structures

The PPU and each SPU contains shared data structures. Shared data structures are spaces where other processes may read or write data. We use these shared data structures in two ways:

- For synchronization. A processor signals another processor via a DMA Put operation.
  For example, in pipelined hashing, SPU0 signals to SPU1 that data is available by performing a DMA Put into SPU1's shared data space.
- For data transfer between SPUs. An SPU performs a DMA Get to obtain data from the shared space of another SPU.
  For example, in pipelined hashing, SPU1 issues a DMA Get to pull Schedule Data from the shared data space of SPU0.

Input data and Hash values are not part of shared data space. These values are provided at runtime during Task Delegation.

2.6 DMA

Data flows between SPUs and global memory via DMA Get and DMA Put operations.

We use DMA for signaling and for data transfer.

DMA is a valuable hardware resource because it is performed by the MMU asynchronously of the processor. We have the SPU issue a DMA Get, then perform some compute-intensive work. By the time the compute-intensive work is done, the DMA Get should be complete.

An SPU DMAs with a second SPU by specifying the LS address of its own data and the EA of the shared data space on the second SPU.

An SPU DMAs with global memory by specifying the LS address of the SPU and the EA of the global memory.

The PPU DMAs with an SPU by placing the LS address of the SPU and the EA of the global memory onto the SPU's MMU.

The DMA bus width is 16 bytes (128 bits), aligned. Although DMA supports transfers of less than 16 bytes, we never issue them because they introduce complexity and offer no gain. When performing a transfer of less than 16 bytes, the transfer size must be a power of two and the source and destination bytes must be aligned. Also, multiple signal channels must not be used within the same Vector.

The DMA Get operation is faster than the DMA Put operation. Also, tests have shown large intermittent pauses (perhaps 1,000 SPU clock ticks) on large DMA Put operations (over 2 KBytes and most evident on 16 KBytes). For these reasons and for the simple clarity of sticking to a "pull-only" data model, we always DMA Get data. Recall that for signaling, we always DMA Put 1 Vector of data.

DMA operations support Barriers and Tag Groups. A Barrier ensures that prior DMA requests that are a member of a tag group finish before future DMA requests that are a member of the same tag group begin. Used together, Barriers and tag groups permit synchronized DMA flow.

2.7 Process Synchronization

Synchronization between SPUs and the PPU is performed using DMA Put operations of 128 bit Vectors. Signaling can be boolean or can contain some additional information. For signaling availability of a hash job, the PPU sends job information along with a job ID signal when issuing hash jobs. For example to dispatch a job to hash ten blocks of data via SHA256, the PPU signals the SHA256 hash job ID, the EA of the start of the data, ten as the number of blocks, and the EA of where to place the computed hash value. For signaling a boolean indicator that data is

```
index i = 0
if more data at index 0 then
    start reading into buffer 0 tag group 0
    while more data at index i do
        if more data at index i+1 then
            start reading into buffer (i+1)%2 tag group (i+1)%2
        end if
        wait for DMA tag group index i%2
        process buffer index i%2
        index i ++
    end while
    send calculated hash to PPU
end if
send DONE signal to SPU
```

Figure 6: Double-buffering is used so that the MMU is reading the next buffer while the current bubffer is being processed.

ready, the SPU sends SET.

2.8 Dataflow Synchronization

When synchronization signaling is predicated on completion of DMA, synchronization actions may be simplified using the DMA Barrier feature. Specifically, a signal request is placed on the DMA queue immediately after a data transfer request is enqueued. Both requests are enqueued in the same tag group and are separated by a Barrier so that the signal goes out immediately after the data transfer has completed. For example when an SPU has completed calculating a Hash value, the SPU enques to DMA the Hash value and then the signal, separated by a Barrier.

We also perform dataflow synchronization by blocking on DMA completion when double-buffering. Specifically, we issue a DMA Get request to read buffer n+1 then begin processing buffer n. To be sure that buffer n is in, we block for buffer n. Unless there is an unexpected pause in processing, the SPU never blocks. This is because the time required for DMA is shorter than the time required for processing. We block on buffer n and not on buffer n+1 by using separate tag groups for each buffer.

2.9 Double Buffering

We use double-buffering whenever we use DMA Get to obtain data to be processed. If more data is available, we start reading buffer n+1 before processing buffer n. This flow is shown in Figure 6.

Hash Algorithm	**CellBE**	**2.27 GHz** **Xeon**
SHA256	205.6	101.8
MD5	317.7	371.3
SHA1	413.3	129.8

Table 1: Measured Best Hash Throughput for a Single File in MBytes/Sec

Hash Algorithm	**CellBE**	**4 Core** **2.27 GHz** **Xeon**
SHA256	3,398.4	407.2
MD5	10,032.0	1,485.2
SHA1	7,296.0	519.2

Table 2: Aggregate Throughput per chip in MBytes/Sec.

3 Timing Results

Timing results are compared between the CellBE and the Xeon Processors on the IBM blades that we have.

Here “Aggregate Throughput” is the measure of the rate number of bytes that can be hashed at one time using multiple files. For example, in the “SHA256” case, a single CellBE can hash 32 different files at one time.

The file used for all measurements was approx. 262M Byes in length. The programs used on the CellBE were built to run on the Cell’s Synergistic Processing Elements. Different programs on the cell were used to obtain the “Best Throughput for a Single File” and for the “Aggregate Throughput”.

The Xeon measurements use the “user time” of the “sha256sum”, “md5sum” and “sha1sum” programs. We did not write special programs. Various people have reported rates of 15-27 cycles/byte for sha256. This is consistent with our speed measurements.

The aggregate is measured on the CellBE but is only Projected on the Xeon since we were unable to get all 8 cores to run simultaneously without interference. One or more are used to support the OS and process tasks. We think all the measurements only include a limited portion of the file system access. We did not use the “openssl” versions on the Xeon. A quick test showed that the “openssl” versions of MD4 and SHA1 are marginally faster than the versions we tested.

Hashing Speed (Gb/s)

implementation	hash type		
who: SPU, file	MD5	SHA-1	SHA-256
IBM: 1 , 1	2.448	2.116	0.854
ours: 1 , 1	2.544	2.604	1.392
ours: 2 , 1	N/A	3.352	1.653
ours: 1 , 4	10.116	7.335	3.403

Encryption Speed (Gb/s, 1 SPU)

	keysize		
AES mode	128	192	256
our CTR	2.071	1.722	1.474
IBM's ECB (encr)	2.059	1.710	1.462
our ECB (encr)	2.092	1.737	1.484
IBM's CBC (encr)	0.795	0.664	0.570
our CBC (encr)	1.191	0.989	0.846

Table 3: Our measured speeds for hashing and AES encryption, compared with IBM's published results[2]. Our hashing results show speedups from two different types of parallelism: using 2 SPUs for a single file, or hashing four files on one SPU.

4 AES on the SPU

The implementation of AES GCM on the SPU consists of an AES implementation, a GCM implementation, and a data pump state machine that schedules data transfer from main memory to SDRAM, performs encryption, and then schedules transfer back from SDRAM to main memory.

While our main interest is in the Counter (CTR) mode of AES, as part of the Galois Counter Mode, we also implemented the Electronic CodeBook (ECB) mode and Cipher Block Chaining (CBC) feedback mode for comparison; the data throughput speeds for a single SPU (encryption only) are summarized in Table 3. Note that our speeds for ECB are slightly faster than IBM's, and our speeds for CBC are significantly better.

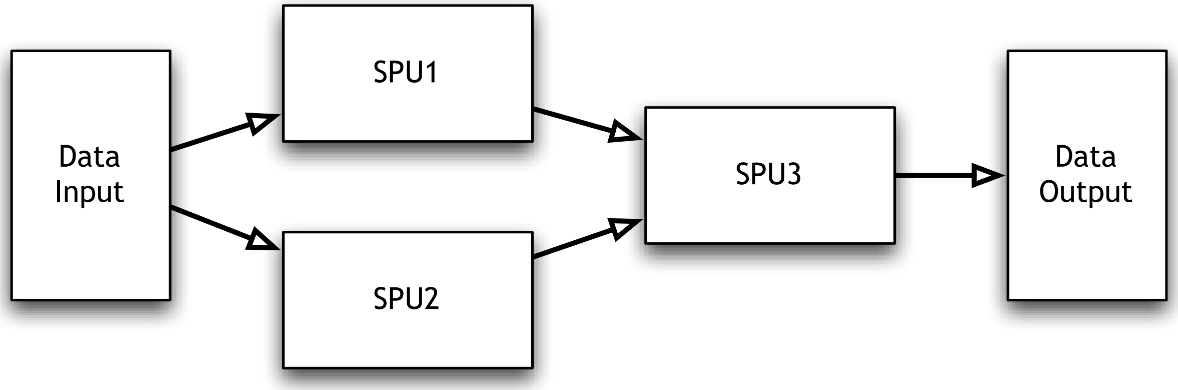


Figure 7: Our test architecture for failsafe AES encryption.

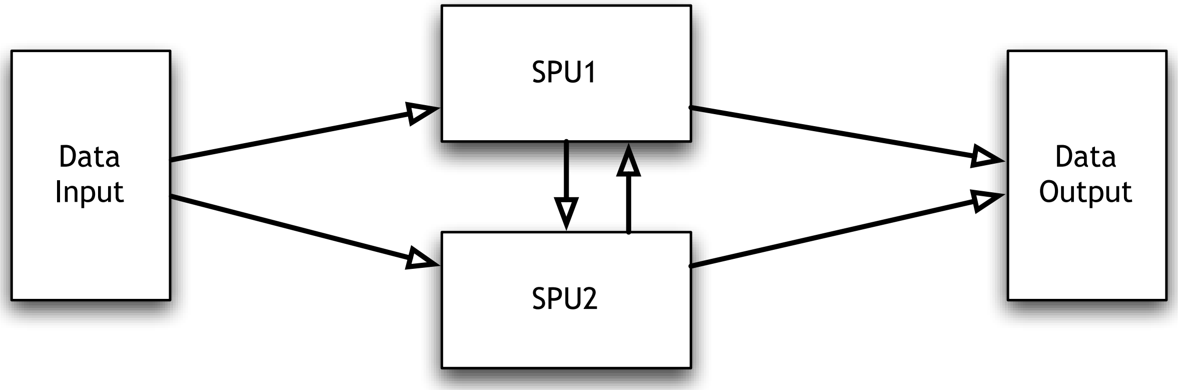


Figure 8: An alternative architecture for failsafe AES encryption. This architecture provides additional reliability against incorrect operation but does not place hard limits between the movement of unencrypted data through the SPU to the output buffer.

5 Parallel Architectures with AES

In our work with AES Encryption, we developed two approaches for utilizing on-chip parallelism provided by multiple SPUs for detecting Single Event Upsets (SEU). Our motivation for detecting SEUs is to ensure that unencrypted data never flows to the output buffer.

Figure 7 shows our first proposal for creating a network for SPUs to improve reliability. Data is transferred from main memory to both SPU 1 and SPU 2. Encrypted data is sent from the two SPUs to SPU 3 where it is compared and, if validated, sent to the output buffer. The advantage of this approach is that no SEU in SPU1, SPU2 or SPU3 can result in cleartext data being copied to the output buffer. The disadvantage is that a SEU in SPU3 can nevertheless result in incorrect data being output, and there is an equal chance of a failure in SPU1, SPU2 and SPU3.

An alternative architecture is shown in Figure 8.

6 Conclusion

In this report we have shown how the CellBE architecture can deliver impressive performance in hashing and encryption through the use of custom-written VLIW assembly code and through careful attention to buffer management. Although the Cell has a reputation as being hard to program, we did not find this to be the case. Certainly the Cell had a learning curve, but we were able to master the complexities of this architecture in a relatively short amount of time. It is unfortunate that IBM halted development on the Cell architecture, as it clearly represented a high-performance, cost-effective computing platform.

A Introduction

We provide two optimizations of high speed SHA-256[5], MD5[6], and SHA-1[5] hashing on the Cell BE:

1. Maximum throughput, achieved by hashing four separate datasets concurrently in lanes on one or more SPUs, and
2. Maximum single-file throughput with minimal latency, where multiple SPUs participate in calculating various hashes using pipelined processing.

Six hash outputs are available: SHA-256, MD5, and SHA-1 file hashes and SHA-256, MD5, and SHA-1 4 KByte Sector hashes of files.

Both programs run from the shell and use SPU resources.

This document describes the following:

- This introduction to hashing.
- An overview of `hash_files` and `hash_file` hashing programs.
- A description of the NPS Cell PPU Hash Utilities.
- A description of the NPS Cell SPU Hash Utilities.
- Details of the `hash_files` application.
- Details of the `hash_file` application.

B Application Overview

B.1 `hash_files`

Application `hash_files` provides *maximum total throughput* using lane processing. Lane processing hashes four separate datasets in four lanes concurrently on an SPU. Multiple SPUs may be active hashing files simultaneously. Application `hash_files` requires the following hashing utilities:

- From NPS Cell PPU Hash Utilities:
  - Lane context modules.
  - File context modules.
  - Data context modules.
- From NPS Cell SPU Hash Utilities:
  - Modules that hash files and Sectors in lanes, and, for finalization, modules that perform a single optimized hash on one SPU.

B.2 `hash_file`

Application `hash_file` provides *maximum single-file throughput* by pipelining hashing tasks across multiple SPUs. SHA-256 and SHA-1 hashing is split into parts and is distributed across multiple pipelining SPUs. MD5 and Sector hashing tasks are split across multiple SPUs. Application `hash_file` requires the following hashing utilities:

- From NPS Cell PPU Hash Utilities:
  - Pipeline context modules.
  - File context modules.
  - Data context modules.
- From NPS Cell SPU Hash Utilities:
  - Modules that hash SHA-256 and SHA-1 in pipelined fashion, modules that hash MD5 as a single thread, and modules that hash sectors in lanes.

C NPS Cell PPU Hash Utilities

PPU hash utilities provide interfaces for hashing and interfaces for managing files and data to be hashed. The NPS Cell PPU Hash Utilities are described in MAN pages.

PPU hash utilities are defined in the following interface files:

- `lanes_hash.h` defines interfaces and type declarations for hashing with maximum throughput on SPUs using lane processing.
- `pipe_hash.h` defines interfaces and type declarations for hashing with maximum single-file throughput on two SPUs using pipelined processing.
- `single_md5_hash.h` defines interfaces and type declarations for hashing MD5 with maximum single-file throughput on one SPU.
- `shared_context.h` defines general interfaces, type declarations, and constants for supporting data and file management involved in hashing.

C.1 `lanes_hash.h` Lane Hash Interfaces

These interfaces provide SPU hashing resources which use lane processing to achieve maximum total throughput.

C.1.1 Lane Interfaces

- `lanes_open_spu()` opens an SPU resource for hashing lanes. Specifically: memory is allocated for the lanes SPU context, the SPE context is created, the `lanes_spu` program is loaded, the SPU thread is started, and the SPU's local store address `ls_lanes_shared_data` is obtained from the started SPU so that the PPU can communicate to the SPU.
- `lanes_close_spu(lanes_spu_ctx_ptr)` closes an SPU resource for hashing lanes. Specifically: the SPU is signalled to stop, the SPU thread is closed, the SPE context is closed, and memory for the lanes SPU context is deallocated.
- `lanes_hash(lanes_spu_ctx_ptr, data_ctx_ptr1, data_ctx_ptr2, data_ctx_ptr3, data_ctx_ptr4, hash_flags)` hashes four datasets residing in four data contexts using the requested hash algorithms.
- `lanes_poll_spu(lanes_spu_ctx_ptr)` identifies when hashing is done and transitions data state when hashing is done.

C.1.2 PPU to SPU Signal Vector

The PPU signals to the SPU that a hashing job is ready using the following signal Vector:

```
typedef struct lanes_ppu_to_spu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile unsigned int hash_flags;
    unsigned int dummy[2]; // alignment to 16-bytes for block DMA transfer
} lanes_ppu_to_spu_t;
```

- Field `signal` signals the SPU that a hash job is available. The PPU sends either SPU_SIGNAL to indicate that a job is available or SPU_DONE to indicate that there are no more jobs and that the SPU should exit. The SPU resets its own signal value to SPU_NO_SIGNAL to indicate that the job has been serviced.
- Field `hash_flags` provides hash job bit fields for SHA-256, MD5, and SHA-1 file and Sector hashes.
- Field `dummy[2]` fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

C.1.3 SPU to PPU signal Vector

When the SPU has completed hash processing, it signals the PPU that the hashing job is completed using the SPU to PPU signal Vector.

```
typedef struct lanes_spu_to_ppu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile unsigned int tick_start; // timer
    volatile unsigned int tick_finish; // timer
    unsigned int dummy[1]; // alignment to 16-bytes for block DMA transfer
} lanes_spu_to_ppu_t;
```

- Field `signal` signals the PPU that a hash job has completed. The SPU sends SPU_AVAILABLE to indicate that it is ready for a hash job. The PPU presets its own signal value to SPU_NOT_AVAILABLE to await the upcoming completion signal from the SPU.
- If the SPU code is compiled with the `__TIMING__` cflag switch set, then the `tick_start` and `tick_finish` fields will contain SPU tick start and stop timing information.
- Field `dummy[1]` fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

C.1.4 PPU Shared Data Structure

The PPU reserves the following data space for communicating with the SPU. The SPU receives a pointer to this structure when it is initialized so that it can read to and write from it.

```
typedef struct ppu_lanes_shared_data_t {
    lanes_ppu_to_spu_t signal_to_spu;
    lanes_spu_to_ppu_t signal_to_ppu;
    data_ctx_ptr_t data_ctx_ptr[4] __attribute__ ((aligned (16)));
} ppu_lanes_shared_data_t;
```

- Field `signal_to_spu` contains the Vector for signaling to the SPU.
- Field `signal_to_ppu` contains the Vector for receiving signaling from the PPU.
- Field `data_ctx_ptr[4]` contains the four pointers to the four data contexts containing data to be hashed. The SPU reads these four pointers.

C.1.5 SPU Shared Data Structure

The SPU Shared Data Structure is the SPU's data space for communicating with the PPU. When the SPU initializes, it sends a pointer to this structure to the PPU so that the PPU knows where to signal to the SPU.

```
typedef struct spu_lanes_shared_data_t {
    lanes_ppu_to_spu_t signal_to_spu;
    lanes_spu_to_ppu_t signal_to_ppu;
} spu_lanes_shared_data_t;
```

- Field signal_to_spu contains the Vector for receiving signaling from the PPU.
- Field signal_to_ppu contains the Vector for signaling to the PPU.

C.1.6 Lanes SPU Context Structure

The Lanes SPU Context tracks the SPU and the process thread that is associated with it, and holds information for communicating between the PPU and SPU.

```
typedef struct lanes_spu_ctx_t {
    spe_context_ptr_t spe_ctx; // pointer to SPU's SPE context
    pthread_t spu_thread; // pthread
    unsigned int spu_index; // used for SPU tracking in printf
    ppu_lanes_shared_data_t ppu_shared_data; // shareable data on PPU
    spu_lanes_shared_data_t *ls_spu_shared_data; // LS of shareable data o
} lanes_spu_ctx_t;
```

- Field spe_ctx points to the SPE context which identifies the associated SPU.
- Field spu_thread identifies the PPU thread under which the associated SPU runs.
- Field spu_index indexes the SPU for diagnostics purposes and is otherwise unused.
- Field ppu_shared_data contains the shared data structure on the PPU side that is associated with the lanes SPU. Data within this structure is shared with the SPU.
- Field ls_spu_shared_data contains the local store address of the shared data structure on the SPU side that is associated with the lanes SPU. The PPU uses this pointer to communicate to the SPU.

C.2 pipe_hash.h Pipelined Hash Interfaces

These interfaces provide SPU hashing resources which use pipelined processing of SHA-256 or SHA-1 to achieve maximum single-file throughput.

C.2.1 Pipeline Interfaces

- pipe_open_spu() opens an SPU resource for hashing SHA-256 or SHA-1 using pipelined processing across two SPUs. Specifically: memory is allocated for the pipeline SPU context, SPE contexts are created, SPU programs pipe1_spu and pipe2_spu are loaded, the two SPU threads are started, the SPU's local store addresses are obtained from the started SPU so that the PPU can communicate to the SPU, and the SPU's local store addresses are exchanged between the SPUs so that the SPUs can communicate with each other.

- pipe_close_spu(pipe_spu_ctx_ptr) closes an SPU resource used for pipelined hashing. Specifically: the SPUs are signalled to stop, the SPU threads are closed, the SPE contexts are closed, and memory for the pipeline SPU contexts are deallocated.
- pipe_hash(pipe_spu_ctx_ptr, data_ctx_ptr, hash_flags) hashes a dataset using the requested SHA-256 or SHA-1 algorithm.
- pipe_poll_spu(pipe_spu_ctx_ptr) identifies when hashing is done and transitions data state when hashing is done.

C.2.2 PPU to Pipe Signal Vectors

The PPU uses the following signal Vector to signal both pipeline SPUs that a hashing job is ready:

```
typedef struct ppu_to_pipe_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile unsigned int hash_flags;
    volatile data_ctx_ptr_t data_ctx_ptr;
    unsigned int dummy[1]; // alignment to 16-bytes for block DMA transfer
} ppu_to_pipe_t;
```

- Field signal signals the SPU that a hash job is available. The PPU sends either SPU_SIGNAL to indicate that a job is available or SPU_DONE to indicate that there are no more jobs and that the SPU should exit. The SPU resets its own signal value to SPU_NO_SIGNAL to indicate that the job has been serviced.
- Field hash_flags provides a hash job code for SHA-256 or SHA-1, and is FILE_SHA256 or FILE_SHA1.
- Field data_ctx_ptr points to the data context for the data being hashed.
- Field dummy[1] fills out the signal Vector to 16 bytes to allow a full 16-byte Vector DMA transfer.

C.2.3 SPU to PPU signal Vector

When the SPU has completed hash processing, it signals the PPU that the hashing job is completed using the SPU to PPU signal Vector. Both SPUs provide this signal when they have completed their respective hashing role, specifically, when pipe1 finishes providing message schedule data to pipe2, and when pipe2 finishes hashing message schedule data.

```
typedef struct pipe_to_ppu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile unsigned int tick_start; // timer
    volatile unsigned int tick_finish; // timer
    unsigned int dummy[1]; // alignment to 16-bytes for block DMA transfer
} pipe_to_ppu_t;
```

- Field signal signals the PPU that a hash job has completed. The SPU sends SPU_AVAILABLE to indicate that it is ready for a hash job. The PPU presets its own signal value to SPU_NOT_AVAILABLE to await the upcoming completion signal from the SPU.
- If the SPU code is compiled with the __TIMING__ cflag switch set, then the tick_start and tick_finish fields will contain SPU tick start and stop timing information.
- Field dummy[1] fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

C.2.4 SPU to SPU signal Vector

The pipe1 SPU signals the pipe2 SPU when the pipe1 SPU has completed preparing schedule data that is available for the pipe2 SPU to consume. Two signal channels are used because two buffers of data are prepared. By using double-buffering, one buffer may be consumed while the other is being prepared.

The pipe2 SPU signals the pipe1 SPU when it has completed pulling schedule data from the pipe1 SPU. When the pipe1 SPU receives this signal, it is free to replace the buffer with the next bufferfull of schedule data. Two signal channels are used because schedule data is double-buffered.

The SPU to SPU signal data structure follows.

```
typedef struct pipe_spu_to_spu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    unsigned int dummy[3]; // alignment to 16-bytes for block DMA transfer
} pipe_spu_to_spu_t;
```

- Field signal signals the other SPU that the resource is available (on pipe1) or has been consumed (by pipe2).
- Field dummy[3] fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

The SPU to SPU signal values follow:

```
// signal between SPUs
#define CLEARED 0
#define SET 1
```

- SET indicates an active signal.
- CLEARED is written on the local SPU to clear the SET signal sent by the other SPU.

C.2.5 PPU Shared Data Structure

The PPU reserves the following data space for communicating with the pipe1 and pipe2 SPUs. The SPU receives a pointer to the two SPU shared structures when it is initialized so that it can read to and write from them.

```
typedef struct ppu_pipe_shared_data_t {
    ppu_to_pipe_t ppu_to_pipe;
    pipe_to_ppu_t pipe1_to_ppu;
    pipe_to_ppu_t pipe2_to_ppu;
} ppu_pipe_shared_data_t;
```

- Field `ppu_to_pipe` contains the Vector for signaling to the SPUs.
- Field `pipe1_to_ppu` contains the Vector for receiving signaling from the pipe1 SPU.
- Field `pipe2_to_ppu` contains the Vector for receiving signaling from the pipe2 SPU.

C.2.6 Pipe1 Shared Data Structure

The pipe1 Shared Data Structure is pipeline 1's SPU data space for communicating with the PPU and with the other SPU. When the SPU initializes, it shares a pointer to this structure to the PPU and the other SPU so that they know where to signal to it.

```
typedef struct spu_pipe1_shared_data_t {
    SPU_PIPE2_SHARED_DATA_T *ea_pipe2_shared_data;
    ppu_to_pipe_t ppu_to_pipe1;
    pipe_to_ppu_t pipe1_to_ppu;
    pipe_spu_to_spu_t pipe1_to_pipe2;
    pipe_spu_to_spu_t pipe2_to_pipe1[2];
    unsigned char data[2][BUFFER_SIZE] __attribute__ ((aligned (16)));
    unsigned char schedule_data[2][SCHEDULE_BUFFER_SIZE] __attribute__ ((a
} spu_pipe1_shared_data_t;
```

- Field `ea_pipe2_shared_data` contains the Effective Address (EA) pointer to the pipe2 shared data structure so that the pipe1 SPU can DMA Put signaling to pipe2.
- Field `ppu_to_pipe1` contains the signaling Vector the PPU uses to signal to pipe1.
- Field `pipe1_to_pipe2` contains the signal Vector pipe1 uses to communicate to pipe2, and is constant.
- Field `pipe2_to_pipe1[2]` contains the two signal Vectors pipe2 uses to communicate to pipe1.
- Field `data[2][BUFFER_SIZE]` contains data to be hashed, and is double-buffered. Pipe1 uses DMA Get to prefetch this data. Pipe1 does not preprocess this data into schedule data until cleared by a signal from pipe2 to do so.
- Field `schedule_data[2][SCHEDULE_BUFFER_SIZE]` contains double-buffered schedule data. Pipe2 uses DMA Get to read this schedule data after pipe1 signals that this schedule data is available.

C.2.7 Pipe2 Shared Data Structure

The pipe2 Shared Data Structure is pipeline 2's SPU data space for communicating with the PPU and with the other SPU. When the SPU initializes, it shares a pointer to this structure to the PPU and the other SPU so that they know where to signal to it.

```
typedef struct spu_pipe2_shared_data_t {
    SPU_PIPE1_SHARED_DATA_T *ea_pipe1_shared_data;
    ppu_to_pipe_t ppu_to_pipe2;
    pipe_to_ppu_t pipe2_to_ppu;
    pipe_spu_to_spu_t pipe2_to_pipe1;
    pipe_spu_to_spu_t pipe1_to_pipe2[2];
    sha256_hash_t sha256_hash __attribute__ ((aligned (16)));
    sha1_hash_t sha1_hash __attribute__ ((aligned (16)));
    unsigned char schedule_data[2][SCHEDULE_BUFFER_SIZE] __attribute__ ((a
} spu_pipe2_shared_data_t;
```

- Field `ea_pipe1_shared_data` contains the Effective Address (EA) pointer to the pipe1 shared data structure so that the pipe2 SPU can DMA Put signaling to pipe1.
- Field `ppu_to_pipe2` contains the signaling Vector the PPU uses to signal to pipe2.
- Field `pipe2_to_pipe1` contains the signal Vector pipe2 uses to communicate to pipe1, and is constant.
- Field `pipe1_to_pipe2[2]` contains the two signal Vectors pipe1 uses to communicate to pipe2.
- Field `schedule_data[2][SCHEDULE_BUFFER_SIZE]` contains double-buffered schedule data. Pipe2 uses DMA Get to read this schedule data from pipe1 after pipe1 signals that this schedule data is available.

C.2.8 Pipe SPU Context Structure

The Pipe SPU Context tracks the SPUs and process threads that are associated with them, and holds information for communicating between the PPU and SPU.

```
typedef struct pipe_spu_ctx_t {
    spe_context_ptr_t spe_pipe1_ctx; // pointer to pipe1 SPU's SPE context
    spe_context_ptr_t spe_pipe2_ctx; // pointer to pipe2 SPU's SPE context
    pthread_t spu_pipe1_thread; // pthread
    pthread_t spu_pipe2_thread; // pthread
    ppu_pipe_shared_data_t ppu_shared_data; // shareable data on PPU
    spu_pipe1_shared_data_t *ls_pipe1_shared_data; // LS of shareable data
    spu_pipe2_shared_data_t *ls_pipe2_shared_data; // LS of shareable data
} pipe_spu_ctx_t;
```

- Field `spe_pipe1_ctx` points to the pipe1 SPE context.
- Field `spe_pipe2_ctx` points to the pipe2 SPE context.

- Field `spu_pipe1_thread` identifies the PPU thread under which the pipe1 SPU runs.
- Field `spu_pipe2_thread` identifies the PPU thread under which the pipe2 SPU runs.
- Field `ppu_shared_data` contains the shared data structure on the PPU side that is associated with the pipeline SPUs.
- Field `ls_pipe1_shared_data` contains the local store address of the pipe1 SPU shared data structure.
- Field `ls_pipe2_shared_data` contains the local store address of the pipe2 SPU shared data structure. The PPU uses this pointer to communicate to the SPU.

C.3 `single_md5_hash.h` Lane Hash Interfaces

These interfaces provide SPU hashing resources which use lane processing to achieve maximum total throughput.

C.3.1 Single MD5 Hash Interfaces

- `md5_open_spu()` opens an SPU resource for hashing MD5. Specifically: memory is allocated for the SPU context, the SPE context is created, the `md5_spu` program is loaded, the SPU thread is started, and the SPU's local store address `ls_spu_shared_data` is obtained from the started SPU so that the PPU can communicate to the SPU.
- `md5_close_spu(md5_spu_ctx_ptr)` closes an SPU resource for MD5 hashing. Specifically: the SPU is signalled to stop, the SPU thread is closed, the SPE context is closed, and memory for the SPU context is deallocated.
- `md5_hash(md5_spu_ctx_ptr, data_ctx_ptr)` performs MD5 hashing.
- `md5_poll_spu(md5_spu_ctx_ptr)` identifies when hashing is done and transitions data state when hashing is done.

C.3.2 PPU to SPU Signal Vector

The PPU signals to the SPU that a hashing job is ready using the following signal Vector:

```
typedef struct md5_ppu_to_spu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile data_ctx_ptr_t data_ctx_ptr;
    unsigned int dummy[2]; // alignment to 16-bytes for block DMA transfer
} md5_ppu_to_spu_t;
```

- Field `signal` signals the SPU that a hash job is available. The PPU sends either SPU_SIGNAL to indicate that a job is available or SPU_DONE to indicate that there are no more jobs and that the SPU should exit. The SPU resets its own signal value to SPU_NO_SIGNAL to indicate that the job has been serviced.
- Field `data_ctx_ptr` points to the data context for the data being hashed.
- Field `dummy[2]` fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

C.3.3 SPU to PPU signal Vector

When the SPU has completed hash processing, it signals the PPU that the hashing job is completed using the SPU to PPU signal Vector.

```
typedef struct md5_spu_to_ppu_t {
    volatile unsigned int signal __attribute__ ((aligned (16)));
    volatile unsigned int tick_start; // timer
    volatile unsigned int tick_finish; // timer
    unsigned int dummy[1]; // alignment to 16-bytes for block DMA transfer
} md5_spu_to_ppu_t;
```

- Field `signal` signals the PPU that a hash job has completed. The SPU sends SPU_AVAILABLE to indicate that it is ready for a hash job. The PPU presets its own signal value to SPU_NOT_AVAILABLE to await the upcoming completion signal from the SPU.
- If the SPU code is compiled with the `__TIMING__` cflag switch set, then the `tick_start` and `tick_finish` fields will contain SPU tick start and stop timing information.
- Field `dummy[1]` fills out the signal Vector to 16 bytes for a full 16-byte Vector DMA transfer.

C.3.4 PPU Shared Data Structure

The PPU reserves the following data space for communicating with the SPU. The SPU receives a pointer to this structure when it is initialized so that it can read to and write from it.

```
typedef struct ppu_md5_shared_data_t {
    md5_ppu_to_spu_t signal_to_spu;
    md5_spu_to_ppu_t signal_to_ppu;
} ppu_md5_shared_data_t;
```

- Field `signal_to_spu` contains the Vector for signaling to the SPU.
- Field `signal_to_ppu` contains the Vector for receiving signaling from the PPU.

C.3.5 SPU Shared Data Structure

The SPU Shared Data Structure is the SPU's data space for communicating with the PPU. When the SPU initializes, it sends a pointer to this structure to the PPU so that the PPU knows where to signal to the SPU.

```
typedef struct spu_md5_shared_data_t {
    md5_ppu_to_spu_t signal_to_spu;
    md5_spu_to_ppu_t signal_to_ppu;
    md5_hash_t md5_hash __attribute__ ((aligned (16)));
    unsigned char data[2][BUFFER_SIZE] __attribute__ ((aligned (16)));
} spu_md5_shared_data_t;
```

- Field signal_to_spu contains the Vector for receiving signaling from the PPU.
- Field signal_to_ppu contains the Vector for signaling to the PPU.
- Field md5_hash holds the hash value being hashed. It is loaded from the data context space via DMA Get before hashing starts and is used while computing intermediate hash values. It is loaded back to the data context space via DMA Put after hashing is updated.
- Field data[2][BUFFER_SIZE] contains data to be hashed, and is double-buffered.

C.3.6 MD5 SPU Context Structure

The MD5 SPU Context tracks the SPU and the process thread that is associated with it, and holds information for communicating between the PPU and SPU.

```
typedef struct md5_spu_ctx_t {
    spe_context_ptr_t spe_ctx; // pointer to SPU's SPE context
    pthread_t spu_thread; // pthread
    ppu_md5_shared_data_t ppu_shared_data; // shareable data on PPU
    spu_md5_shared_data_t *ls_spu_shared_data; // LS of shareable data on
} md5_spu_ctx_t;
```

- Field spe_ctx points to the SPE context which identifies the associated SPU.
- Field spu_thread identifies the PPU thread under which the associated SPU runs.
- Field ppu_shared_data contains the shared data structure on the PPU side that is associated with the SPU. Data within this structure is shared with the SPU.
- Field ls_spu_shared_data contains the local store address of the shared data structure on the SPU side that is associated with the SPU. The PPU uses this pointer to communicate to the SPU.

C.4 shared_context.h Shared Interfaces

These interfaces manage the opening, reading, and closing of files being hashed. Actual data read is maintained by data interfaces.

C.4.1 File Interfaces

- file_open(filename) creates a file context and opens a file for reading. Initial count values and starting file hash values are set.
- file_read_sectors(file_ctx_ptr, data_ctx_ptr) reads file data from the file identified by the file context into space in the data context. The number of bytes read and the running number of bytes read are set. The data context must be available for this operation to work. Specifically, the data context state must be DATA_EMPTY.
- file_close(file_ctx_ptr) closes a file that has been opened for reading and deallocates memory assigned to the file context. An error is returned if a data context is still associated with the file.

C.4.2 Data Interfaces

These interfaces provide data management support. Data is read into data resources using file interfaces and is hashed using SPU hashing interfaces.

- `data_alloc(sectors_per_read)` allocates a data context for reading files and hashing data, sets read values to zero, and sets the data context state to DATA_EMPTY.
- `data_free(data_ctx_ptr)` deallocates a data context and associated memory space. An error is returned if a data context is still active.
- `data_clear_hashed_state(data_ctx_ptr)` disassociates any file context from the data context, sets read values to zero, and sets the data context state to DATA_EMPTY. An error is returned if the data context is still active, which is the case when the data context state is DATA_READ or DATA_HASHING.

C.4.3 Hash sizes and Types

These declarations define the fundamental SHA-256, MD5, and SHA-1 hash types.

```
// hash constants
#define SHA256_HASH_SIZE 32 // 256 bits
#define MD5_HASH_SIZE 16 // 128 bits
#define SHA1_HASH_SIZE 32 // 160 bits

// types for hashes
typedef unsigned char sha256_hash_t[SHA256_HASH_SIZE];
typedef unsigned char md5_hash_t[MD5_HASH_SIZE];
typedef unsigned char sha1_hash_t[SHA1_HASH_SIZE];
typedef sha256_hash_t *sha256_hash_ptr_t;
typedef md5_hash_t *md5_hash_ptr_t;
typedef sha1_hash_t *sha1_hash_ptr_t;
```

C.4.4 Hash Flags

These declarations identify the six hash algorithms that may be selected.

```
// hash flags
#define FILE_SHA256 0x01
#define FILE_MD5 0x02
#define FILE_SHA1 0x04
#define SECTORS_SHA256 0x08
#define SECTORS_MD5 0x10
#define SECTORS_SHA1 0x20
```

C.4.5 Data Context States

These declarations define the four states that a data context may be in. File, data, and hash interfaces will fail if their associated data contexts are in incorrect states. Also, calculated hash values are only valid when data contexts are in their correct states.

```
// data context state
#define DATA_EMPTY 0
#define DATA_READ 1
#define DATA_HASHING 2
#define DATA_HASHED 3
```

- DATA_EMPTY indicates that the data context has no data, has no file attached to it, and has no calculated hash values. It is ready for a file read operation via `file_read_sectors`. A hashing operation may be performed on the empty data, but no action will take place.
- DATA_READ indicates that data has been read and a file has been attached to it, but the data not been hashed.
- DATA_HASHING indicates that the data is being hashed via a hashing operation such as `lanes_hash`. A poll operation returning completion status is required for transitioning the data context state from DATA_HASHING to DATA_HASHED.
- DATA_HASHED indicates that data has been hashed and hash values are available. Sector hashes are available. If the data context is at EOF, file hashes are also available. Issue `data_clear_hashed_state` to release the file attached to it and to transition the data context state from DATA_HASHED back to DATA_EMPTY. The file cannot be closed via `file_close` until it is released.

C.4.6 Data Context Type Declarations

A data context holds data read from a file, tracks the file the data came from, and provides space for holding Sector hashes.

When a data context is created, memory space is allocated for it that is the size of `data_ctx_ptr` plus the size of the data and Sector hash space required based on the number of sectors that it can hold.

```
// data types
typedef struct data_ctx_t
{
  unsigned int sectors_per_read; // private
  unsigned long long num_previous_bytes; // private
  unsigned int num_current_bytes; // private
  unsigned int eof; // public
  unsigned int data_ctx_state; // public
  file_ctx_ptr_t file_ctx_ptr; // private
  unsigned char *data; // private malloc
  sha256_hash_t *sha256_sector_hash_ptr; // public malloc
  md5_hash_t *md5_sector_hash_ptr; // public malloc
  sha1_hash_t *sha1_sector_hash_ptr; // public malloc
  unsigned char heap[0] __attribute__ ((aligned (16))); // start of heap s
} data_ctx_t;

typedef data_ctx_t *data_ctx_ptr_t;
```

- Field `sectors_per_read` defines the maximum number of sectors that may be read during a `file_read_sectors` file read operation. This value is established during the `data_alloc` call. The amount of data and Sector hash space reserved is based on this value.
- Field `num_previous_bytes` tracks how much data has been read before the current read, and is used for tracking Sector numbers and determining the EOF condition.
- Field `num_current_bytes` identifies how many bytes are read in the current read, and is used to determine the number of Sectors read, the number of blocks read, and the EOF condition.
- Field `eof` indicates that there are no more bytes to process. Note that this is different from field `eof` in `file_ctx_t` which may reach EOF while more than one data context may be available containing data from the same file.
- Field `data_context_state` contains the state of the data context, which is one of DATA_EMPTY, DATA_READ, DATA_HASHING, or DATA_HASHED.
- Field `file_ctx_ptr` points to the file context from which the last file read was made, or is NULL if the file context has been released using `data_clear_hashed_state`.
- Field `data` points to the start of the data space to be read using `file_read_sectors`. Field `data` starts at the address of field `heap` and is 16-byte aligned for DMA transfers. Its size is determined by the number of sectors per read, specifically `sectors_per_read * 4096`.
- Fields `sha256_sector_hash_ptr`, `md5_sector_hash_ptr`, and `sha1_sector_hash_ptr` point to their respective arrays of SHA-256, MD5, and SHA-1 Sector hashes. They immediately follow the allocation for field `data`. Their size is determined by the number of sectors per read, field `sectors_per_read`.

C.4.7 File Context Type Declarations

A file context contains information regarding an opened file including a pointer to its name, its file size, how many bytes have been read so far, and the file hash values being calculated for it.

When a file context is created, memory space is allocated for it that is the size of `file_ctx_t`.

```
// file types
typedef struct file_hash_t
{
  sha256_hash_t sha256_hash __attribute__ ((aligned (16))); // private
  md5_hash_t md5_hash; // private
  sha1_hash_t sha1_hash; // private
} file_hash_t;

typedef struct file_ctx_t
{
  char *filename; // public
  int file_descriptor; // private
  off_t file_size; // private
  off_t total_bytes_read; // private
  int eof; // public
  unsigned int data_ctx_attach_count; // private
  file_hash_t file_hash; // public
} file_ctx_t;

typedef file_ctx_t *file_ctx_ptr_t;
```

- Field `filename` points to the name of the opened file.
- Field `file_descriptor` points to the opened file.
- Field `file_size` indicates the size of the file.
- Field `total_bytes_read` indicates the number of bytes read so far.
- Field `eof` indicates that there are no more bytes to read.
- Field `data_ctx_attach_count` tracks how many file reads are active.
- Field `file_hash` contains the SHA-256, MD5, and SHA-1 file hashes. File hash values are transient or contain running hash values until 1) the associated data context state reaches DATA_HASHED and 2) the associated data has reached EOF.

D NPS Cell SPU Hash Utilities Overview

These utilities provide optimized hashing algorithms written in assembly and run on SPUs. SHA-256, MD5, and SHA-1 hashing algorithms are provided. Optimizations for maximum total throughput, maximum single-file throughput, and Sector hashing are provided.

The NPS Cell SPE Hash Utilities are described in MAN pages available in our source tree at `src/hash_algorithms/cell_hash_man/spu_hash`.

D.1 Maximum Total Throughput

These interfaces provide maximum total throughput by hashing four separate datasets in lanes. These interfaces should be used when multiple files may be hashed concurrently.

- `sha256_hash_4lanes`
- `md5_hash_4lanes`
- `sha1_hash_4lanes`

D.2 Maximum Single-file Throughput, Pipelined

These interfaces provide maximum single-file throughput by hashing using two pipelined SPUs. SPU pipeline 1 prepares a 16 KByte precomputation table from 4 KBytes of input, while SPU pipeline 2 hashes the 16 KBytes of precomputation data. Pipelined processing is available for SHA-256 and SHA-1 and is not available for MD5.

- `sha256_hash_pipe1`
- `sha256_hash_pipe2`
- `sha1_hash_pipe1`
- `sha1_hash_pipe2`

D.3 Maximum Single-file Throughput, Single-SPU

These interfaces provide maximum single-file throughput by hashing a single stream on a single SPU. For SHA-256 and SHA-1 processing, use Maximum Single-file Throughput, Pipelined interfaces, if possible, and use Maximum Single-file Throughput, Single-SPU interfaces when finalizing the last block. For MD5 processing, use the Maximum Single-file Throughput, Single-SPU interface to hash the entire single file.

- `sha256_hash`
- `md5_hash`
- `sha1_hash`

D.4 Sector Hashing

These interfaces provide maximum Sector hashing throughput by hashing four Sectors concurrently, one in each lane. These interfaces combine hash initialization, hashing, and finalization within the same assembly call for improved performance.

- `sha256_hash_4sectors`
- `md5_hash_4sectors`
- `sha1_hash_4sectors`

E Application `hash_files` for Maximum Throughput

We provide maximum hashing throughput via program `hash_files`.

E.1 `hash_files` User Interface

Usage:

```
hash_files -l <filename> -a <hash algorithm> [-s]
           [-r <num_4K_sectors>] [-n <number of SPUs>]
```

`-l <filename>` specifies the filename of a file containing a list of filenames to hash, one filename per line, no spaces, each line terminated by a carriage return. This selection is required.

`-a <algorithm>` identifies the hashing algorithm(s) to use. Options are `sha256`, `md5`, `sha1`, and `all` for all three.

`-s` specifies to also perform Sector hashing in addition to file hashing.

`-r <num_4K_sectors>` identifies the number of sectors to read per file read. A value of two may be optimal for many small files, and 16 may be optimal for mixed small and large files. 1024 may be optimal for many large files. Large values may cause page caching.

`-n <number of SPUs>` identifies the number of SPUs to use. We recommend using all SPUs or else enough SPUs to keep the hashing process I/O bound rather than processing bound. I/O performance is based on the hardware and file systems providing the files.

E.2 Output

Output is sent to `stdout` and consists of multiple file and Sector hash values separated by carriage returns. File hash output is as follows:

```
<filename> <algorithm> <hash value>
```

Sector hash output is as follows:

```
<filename> Sector <sector number> <algorithm> <hash value>
```

Text `<filename>` is the path of the filename obtained from the file list file, `<algorithm>` is one of `SHA-256`, `MD5`, or `SHA-1`, and `<hash_value>` consists of sets of eight-digit hexadecimal values separated by single space characters. For Sector hashes, the word `Sector`, a space, and the integer Sector number starting at zero are also provided.

An example MD5 file and Sector hash for 14 KByte file `myfile` follows:

```
myfile Sector 0 MD5 14c43752 b777ad9c 107fc589 b6faa235
myfile Sector 1 MD5 1390b353 cba445e0 f033d799 626c975a
myfile Sector 2 MD5 456bed94 e3b5cb37 21c20109 a98e599a
myfile MD5 bbc725c7 eaabd331 252dde6b 5d2cc600
```

E.3 Design of the `hash_files` Application

The `hash_files` application runs on top of NPS Cell PPU Hash Utilities. The NPS Cell PPU Hash Utilities drive one or more SPUs which in turn make calls to Cell SPE Hash Utilities for performing the actual hashing. The `hash_files` application is available in our source tree at `src/hash_algorithms/cell_hash`.

E.3.1 Local Data Structures

- `lanes_spu_ctx_ptr[MAX_SPUS]` contains the lane SPU context pointers for each SPU performing lane processing. Lane SPU context pointers identify the SPU to use when hashing on SPUs in lanes.
- `data_ctx_ptr[MAX_SPUS][2][4]` contains a matrix of data context pointers. A set of pointers is defined for each SPU. On each SPU, two sets are provided, allowing double-buffering of file reads. Four pointers per SPU set are provided to support processing in four lanes.
- `file_ctx_ptr_queue[MAX_SPUS][4]` contains a matrix of file pointers, one per lane per SPU.

  When new files are read, they are bound to a position in the `file_ctx_ptr_queue` and in the `data_context_ptr`. When a file is read and it reaches EOF, its file context is removed from `file_ctx_ptr_queue` so that a new file context may be placed there, but the file context is not deallocated. File contexts are deallocated by local helper function `clear_hash_state` after hashing for the file has fully completed. Managing file contexts this way allows lanes to have data from a new file on the next cycle after the previous file, allowing a new file to be loaded and read before the hash for the previous file is calculated and reported.

E.3.2 Local Helper Functions

These functions assist the `hash_files` program by wrapping functions into conceptual tasks in groups of four for lane processing.

- `start_hashing()` schedules the hashing of four data contexts on an SPU based on the SPU and dataset index values.
- `report_hashes()` prints calculated Sector hashes for the four datasets identified by the given SPU and dataset index. Where file contexts indicate EOF, calculated file hashes are also printed.
- `clear_hash_state()` calls `data_clear_hash_state()` to clear the hash state for each of the four datasets identified by the given SPU and dataset index. When datasets are at EOF and hashes have been consumed, their associated files are closed.

- `read_files()` calls function `file_read_sectors()` to read four files into data contexts identified by the given SPU and dataset index. If a file is at EOF, it is removed from `file_ctx_ptr_queue` and a new file is loaded and read in its place.

E.3.3 `hash_files` Main Entry

The main entry parses input to identify the following runtime parameters:

- `num_spus` identifies the number of SPUs to parallel-process on. One SPU context is created for each SPU.
- `hash_flags` identifies the hash algorithms to be performed on the files, up to six algorithms: file and Sector hashes for SHA-256, MD5, and SHA-1 algorithms.
- `list_filename` identifies the file containing the list of filenames to be hashed.
- `sectors_per_read` identifies the maximum number of sectors to read in a file read operation.

Once these runtime parameters are set, the main program calls function `process_files_lanes()` which allocates processing resources, iteratively reads and hashes files using lane processing, then deallocates resources and exits.

E.3.4 Wrapper Function `process_files_lanes()`

This function opens SPU and data resources for hashing in lanes, iteratively hashes files using double-buffering, closes SPU and data resources, then exits. Flow is as follows:

```
create n SPU contexts using lanes_open_spu(), create a matrix of data contexts using data_alloc(),
and open the list of filenames to be hashed using local helper function open_list()
loop
    for each SPU n do
        poll SPU availability using function lanes_poll()
        if poll status indicates that the SPU is ready then
            start hashing the next set of lanes on the SPU using local helper function start_hashing()
            report hashes from the current set of lanes using local helper function report_hashes()
            clear hash states from the current set of lanes using local helper function clear_hash_state()
            read files into the current set of available lanes using local helper function read_files()
            swap the current index and next index values for SPU n for the next loop through
        end if
    end for
    exit loop if the filename list is exhausted and all data context states are at DATA_EMPTY
end loop
close the list of consumed filenames using local helper function close_list(), close all data
contexts using data_free(), and close the n SPU contexts using lanes_close_spu()
```

Note that the first iteration of the loop hashes and reports empty data. This is acceptable because hashing and printing empty data does nothing. Also note that file reads are double-buffered and hashing is started immediately after polling indicates that the SPU is ready. This design strategy minimizes the delay between when an SPU is ready and when the SPU is tasked with work, improving performance.

E.3.5 PPU Call Tree

The PPU call tree is shown in Figure 9.

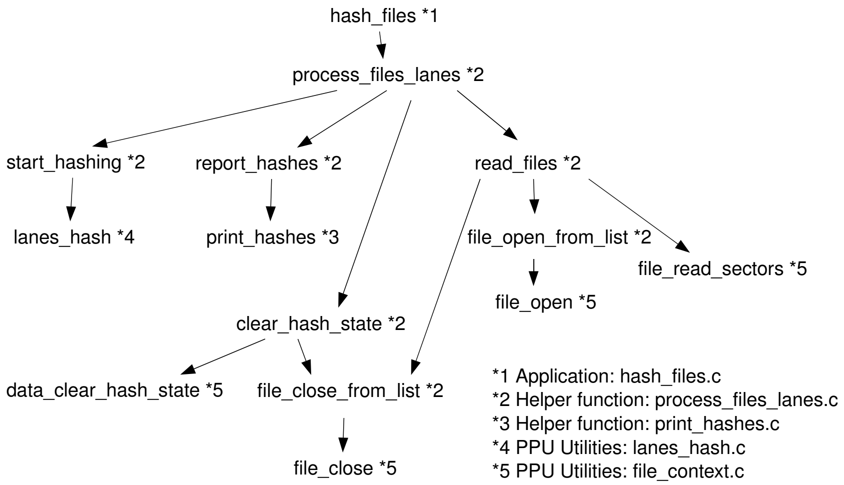
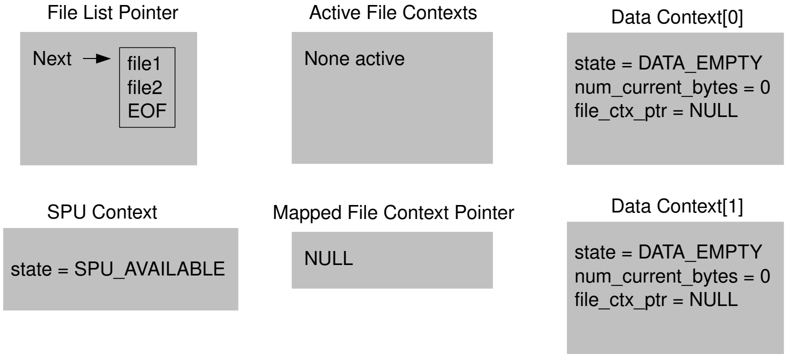


Figure 9: PPU call tree.

E.3.6 Example Flow

The following example steps illustrate state transition flow as two files are loaded and hashed on one SPU. For simplification, only one SPU and one lane are shown.

1. Start: No files have been read, data contexts are in their initial condition, and the SPU is ready. See Figure 10.



state = DATA_EMPTY
num_current_bytes = 0
file_ctx_ptr = NULL

state = SPU_AVAILABLE

state = DATA_EMPTY
num_current_bytes = 0
file_ctx_ptr = NULL

Figure 10: Step 1: Resources in their initial starting state.

2. Poll until SPU done: The SPU has not been tasked, so this returns immediately.
3. Start hashing `data_ctx[1]`: The data context is empty, so the SPU is not tasked.

4. Report hashes for `data_ctx[0]`: The data context is empty, so nothing is reported.
5. Clear hashes for `data_ctx[0]`: The data context is empty, so no action is taken.
6. Read file `file1` to `data_ctx[0]`:
   (a) The file context pointer mapped to this lane is NULL, so get the next filename `file1` from the file list pointer, open an active file context for it, and map it to the data context.
   (b) Read into `data_ctx[0]` from the newly mapped `file1`. Because the file is small, the file context reaches EOF on this first read and the data context is also marked EOF.

   See Figure 11.

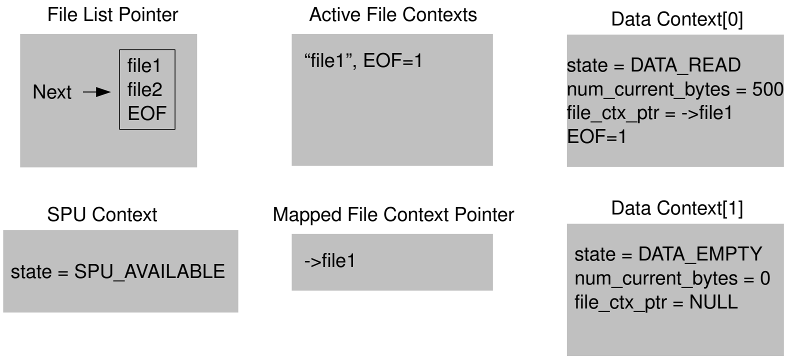


Figure 11: Step 6: Resources after reading file `file1`.

7. Continue the processing loop because the file list has not reached EOF yet.
8. Poll until SPU done. The SPU has still not been tasked, so this returns immediately.
9. Start hashing `data_ctx[0]`:
   (a) The SPU state transitions to SPU_NOT_AVAILABLE.
   (b) Data context state for `data_ctx[0]` transitions to DATA_HASHING.
   (c) The PPU signals the SPU to start hashing.

   See Figure 12.

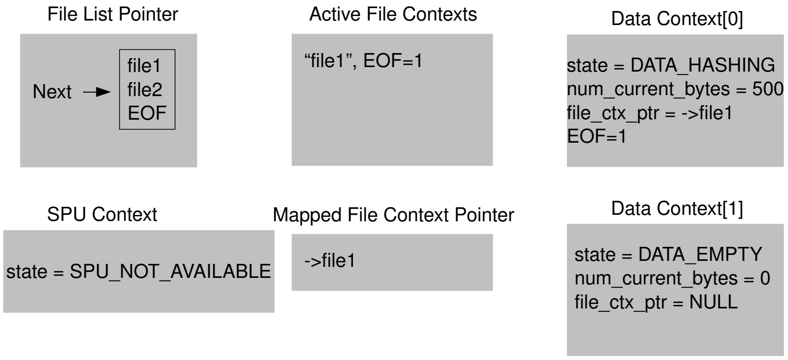


Figure 12: Step 9: Resources after starting to hash `data_ctx[0]`.

10. Report hashes for `data_ctx[1]`: Data contexts are still empty, so nothing is reported.

11. Clear hashes for `data_ctx[0]`: Data contexts are already empty, so no action is taken.
12. Read file `file2` to `data_ctx[1]`:
    (a) The file context pointer mapped to this lane is NULL, so get the next filename `file2` from the file list pointer, open an active file context for it, and map it to the data context.
    (b) Read into `data_ctx[1]` from mapped `file2`. Because the file is small, the file context reaches EOF on this first read and the data context is also marked EOF.

    See Figure 13.

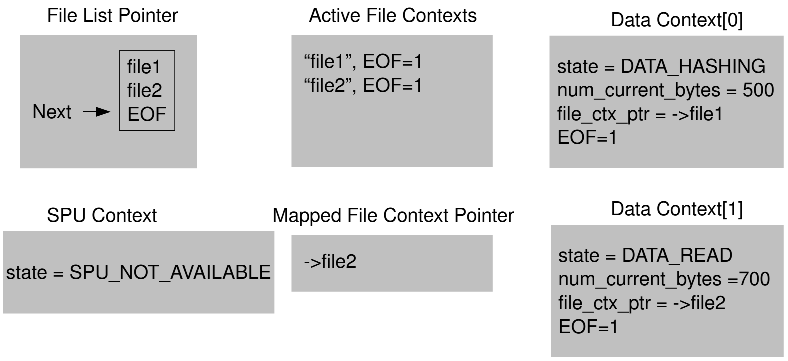


Figure 13: Step 12: Resources after reading file `file2`.

13. Continue the processing loop because the file list has not reached EOF yet.
14. Poll until SPU done to make sure the SPU has completed hashing `data_ctx[0]`. When done:
    (a) The SPU state has transitioned to SPU_AVAILABLE.
    (b) Data context `data_ctx[0]` is transitioned to DATA_HASHED.
15. Start hashing `data_ctx[1]`:
    (a) The SPU state transitions to SPU_NOT_AVAILABLE.
    (b) Data context state for `data_ctx[1]` transitions to DATA_HASHING.
    (c) The PPU signals the SPU to start hashing.

    See Figure 14.

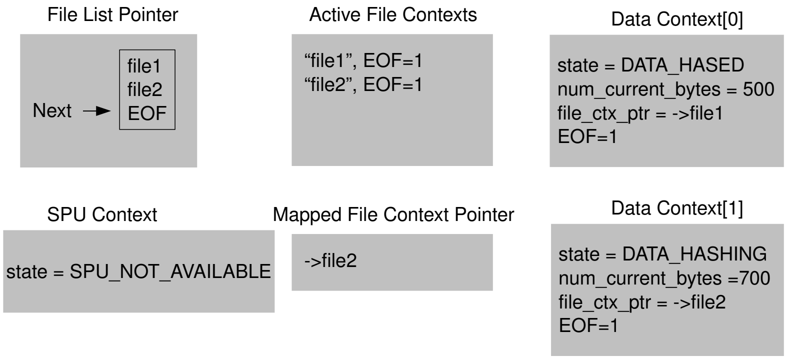


Figure 14: Step 15: Resources after starting to hash `data_ctx[1]`.

16. Report hashes for `data_ctx[0]`:
    (a) Report file hashes because `data_ctx[0]` indicates EOF.
    (b) Do not report Sector hashes. The file size is less than 4 KBytes so no Sector hashes are computed.
17. Clear hashes for `data_ctx[0]`:
    (a) `data_ctx[0]` indicates EOF so close the active file context for `file1` indicated by `data_ctx[0]`.
    (b) The `data_ctx[0]` state transitions to DATA_EMPTY.

    See Figure 15.

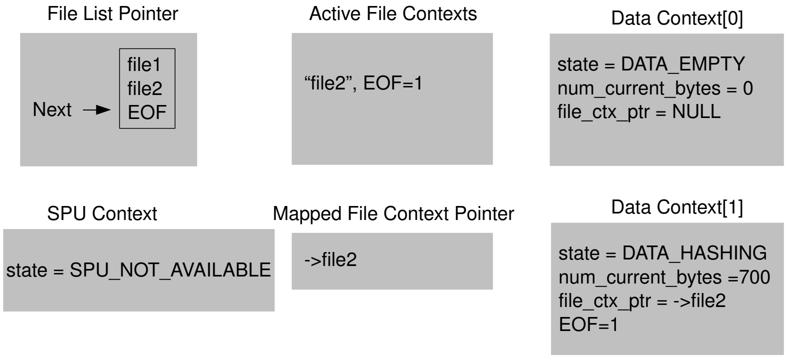


Figure 15: Step 17: Resources after clearing `data_ctx[0]`.

18. Read nothing because the read returned EOF.
19. Continue the processing loop because although the file list is at EOF, `data_ctx[1]` state has not transitioned back to DATA_EMPTY.
20. Poll until SPU done to make sure the SPU has completed hashing `data_ctx[1]`. When done:
    (a) The SPU state transitions to SPU_AVAILABLE.
    (b) Data context `data_ctx[1]` transitions to DATA_HASHED.

    See Figure 16.

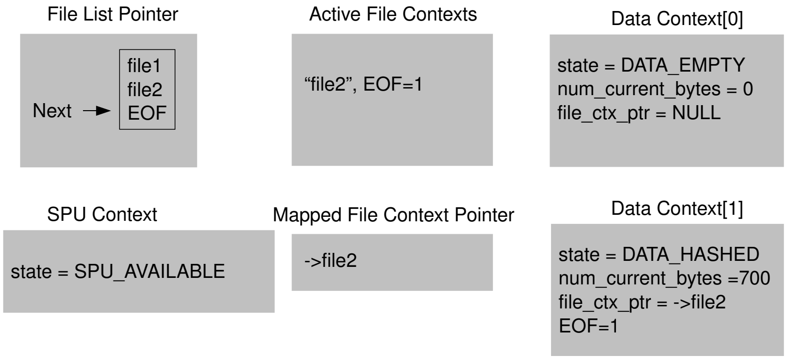


Figure 16: Step 20: Resources after waiting for hash `data_ctx[1]` to complete.

21. Start hashing `data_ctx[0]`:

(a) The SPU state transitions to SPU_NOT_AVAILABLE.
(b) Data context state for `data_ctx[0]` transitions to DATA_HASHING.
(c) The PPU signals the SPU to start hashing.

See Figure 17.

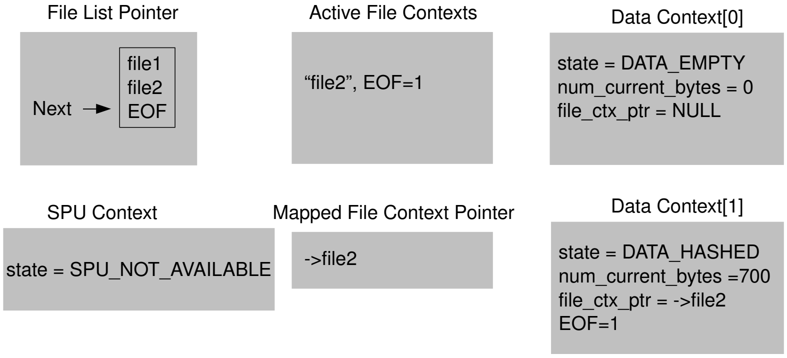


Figure 17: Step 21: Resources after starting to hash `data_ctx[0]`.

22. Report calculated hashes for `data_ctx[1]`:
    (a) Report file hashes because `data_ctx[1]` indicates EOF.
    (b) Do not report Sector hashes. The file size is less than 4 KBytes so no Sector hashes are computed.
23. Clear hashes for `data_ctx[1]`:
    (a) `data_ctx[1]` indicates EOF so close the active file context for `file2` indicated by `data_ctx[1]`.
    (b) Transition the `data_ctx[1]` state to DATA_EMPTY.

    See Figure 18.

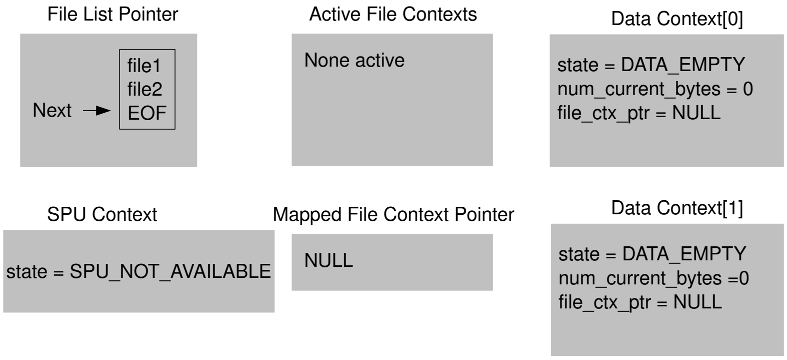


Figure 18: Step 23: Resources after clearing `data_ctx[1]`.

24. Read nothing (again) because the file list pointer is at EOF.
25. Terminate the processing loop because the file list is at EOF and all `data_ctx[*]` states have transitioned back to DATA_EMPTY.

E.4 Design of SPU Program `lanes_spu`

The `lanes_spu` Program initializes, then loops receiving and processing hashing jobs until the SPU_DONE signal is received.

The main SPU program flow is as follows:

```
obtain PPU shared data space ea_ppu_shared_data from parameter input
obtain the SPU index used for diagnostics from parameter input
initialize incoming and outgoing signal values to inactive
mbox mail the SPU shared data space ls_spu_shared_data to the PPU
loop
    continue loop until signal changes from SPU_NO_SIGNAL
    take the tick_start timestamp if timestamp is compiled in via the __TIMESTAMP__ compiler
    flag
    quit if signal is SPU_DONE
    run the hash job by calling function spu_process_lanes()
    take the tick_finish timestamp if timestamp is compiled in via the __TIMESTAMP__ compiler
    flag
    signal SPU_AVAILABLE to the PPU to indicate that the SPU is done and is ready for a new job
end loop
```

The `spu_process_lanes` program flow processes individual hashing assignments by reading four data contexts and performing hashing on them as follows:

```
start and finish DMA get set of four data context pointers
start and finish DMA get four data contexts
start and finish DMA get three starting file hash values
set sector_index to 0 for looping processing one Sector at a time
start DMA get current (first) data buffer
while more data do
    start DMA get next data buffer using DMA tag for next index
    wait for DMA get of current DMA tag to complete (this includes the data buffer and sector hashes)
    perform a file hash of the current data buffer, finalizing any file hashes that are at EOF
    perform a sector hash of the current data buffer
    start DMA put of current calculated sector hashes using DMA tag for current index
    swap current and next index values
end while
start and finish DMA put three calculated file hash values
wait for DMA put of final sector hash using DMA tag for the last index, which is now next
```

E.4.1 SPU Call Tree

The SPU call tree is shown in Figure 19.

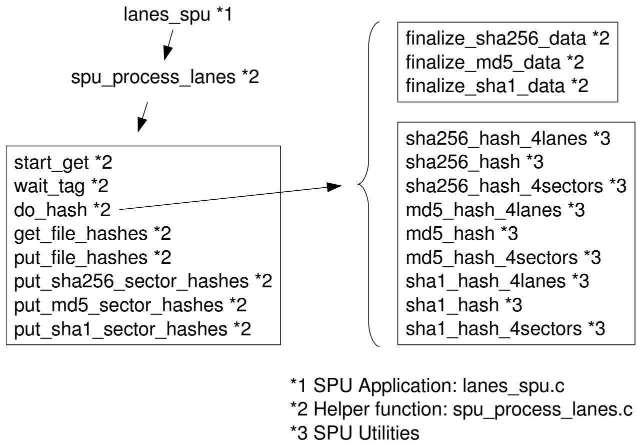


Figure 19: SPU call tree.

F Application hash_single_file for Maximum Single-file Throughput

Maximum single-file throughput is achieved using application hash_single_file.

F.1 hash_single_file User Interface

Usage:

```
hash_single_file -f <filename> -a <hash algorithm>
           [-r <num_4K_sectors>] [-c] [-v]
```

-f <filename> specifies the name of the file to be hashed. This selection is required.

-a <algorithm> identifies the hashing algorithm to use, and is one of sha256, md5 or sha1.

-r <num_4K_sectors> identifies the number of sectors to read per file read. The value should be large enough to reduce iterative reading, but small enough to fit read data in system memory without causing page caching. Devault is 1024.

-c displays SPU resource information.

-v displays input parameters.

F.2 Output

Output is sent to stdout and consists of one file hash value as follows:

```
<filename> <algorithm> <hash value>
```

Text <filename> is the path of the filename hashed, <algorithm> is one of SHA-256, MD5, or SHA-1, and <hash_value> consists of sets of eight-digit hexadecimal values separated by single space characters.

An example MD5 file hash for file myfile follows:

```
myfile MD5 bbc725c7 eaabd331 252dde6b 5d2cc600
```

F.3 Design of the hash_single_file Application

The hash_single_file Application runs on top of NPS Cell PPU Hash Utilities. The NPS Cell PPU Hash Utilities drive either a single SPU for hashing MD5 or a pair of SPUs for hashing SHA-256 or SHA-1. The SPUs make calls to Cell SPE Hash Utilities for performing the actual hashing. The hash_single_file application is available in our source tree at src/hash_algorithms/cell_hash.

The hash_single_file Application uses file process_pipe.c to hash SHA-256 or SHA-1 and uses file process_md5.c to hash MD5. Both files are identical except that process_pipe.c makes calls to Pipe Hash interfaces and process_md5.c makes calls to Single MD5 Hash interfaces.

F.3.1 Local Data Structures

- pipe_spu_ctx_ptr (or md5_spu_ctx_ptr) contains the SPU context pointer for the SPUs (or SPU) performing the hash processing.
- data_ctx_ptr[2] contains two data context pointers for supporting hashing using double-buffering.
- file_ctx_ptr points to the file context that will be active through the duration of hashing.

F.3.2 Local Helper Functions

These functions assist the hash_single_file program by wrapping functions into conceptual tasks for double-buffered hash processing.

- start_hashing() schedules the hashing of the data context.
- report_hash() resets the data context for the next read. Where file contexts indicate EOF, the calculated file hash is also printed.
- clear_hash_state() calls data_clear_hash_state() to clear the hash state for the dataset. When clear_hash_state() is called and the associated data context is at EOF, the associated file is closed.
- read_file() reads the file into space in the data context. If the file is at EOF, it is removed from file_ctx_ptr.
- data_done() indicates when data processing has completed by identifying when both data contexts have returned to the DATA_EMPTY state.

F.3.3 hash_single_file Main Entry

The main entry parses input to identify the following runtime parameters:

- hash_flags identifies the file hash algorithm to be performed on the file, and is one of SHA-256, MD5, or SHA-1.
- filename identifies the file to be hashed.
- sectors_per_read identifies the maximum number of sectors to read in a file read operation.

Once these runtime parameters are set, the main program calls function `process_pipe()` (or `process_md5()`) which allocates processing resources, iteratively reads and hashes the specified file, then deallocates resources and exits.

F.3.4 Wrapper Function `process_pipe()` (or `process_md5()`)

This function opens SPU and data resources for hashing, iteratively hashes the file using double-buffering, closes SPU and data resources, then exits. Flow is as follows:

```
create an SPU context using pipe_open_spu() (or md5_open_spu()), create the two data con-
texts using data_alloc(), and open the file to be hashed, assigning it to file_ctx_ptr
loop
    poll SPU availability using function pipe_poll() or md5_poll()
    if poll status indicates that the SPU is ready then
        start hashing the next file read on the SPU using local helper function start_hashing()
        report, if done, the hash from the current read using local helper function report_hash()
        clear the current hash state and, if done, close the file, using local helper function clear_hash_state()
        read from the file into space in the current data context using local helper function read_file()
        swap the current index and next index values for the next loop through
    end if
    exit loop if both data context states are at DATA_EMPTY
end loop
close both data contexts using data_free() and close the SPU context using pipe_close_spu()
(or md5_close_spu())
```

Note that the first iteration of the loop hashes and reports empty data. This is acceptable because hashing and printing empty data does nothing. Also note that file reads are double-buffered and hashing is started immediately after polling indicates that the SPU is ready. This design strategy minimizes the delay between when an SPU is ready and when the SPU is tasked with work, improving performance.

F.4 Design of SPU Program `pipe1_spu`

After the `pipe1_spu` Program initializes, it loops receiving and processing hashing jobs until the SPU_DONE signal is received.

The main outer SPU flow is identical to that of the `lanes_spu` program except that it additionally receives the EA of the pipe2 processing space during initialization so that the two pipes can interact.

The inner flow is managed by function `spu_process_pipe1` which processes individual hashing assignments by reading the data context provided by the PPU and creating preprocessed schedule data from data identified in the data context as follows:

```
start and finish DMA get of the data context
done if there are 0 bytes to process
set sector_index to 0 for looping processing one Sector at a time
start DMA get current (first) data buffer using DMA tag for current sector index
```

```
loop
    if more data is available on the next sector_index then
        start DMA get next data buffer using DMA tag for next index
    end if
    wait for DMA get of current DMA tag to complete
    wait for pipe2 to indicate that the current schedule data buffer may be changed
    calculate current schedule data from current data
    signal pipe2 that the current schedule data buffer is ready
    if last iteration then
        if data context indicates EOF then
            wait for pipe2 to indicate that the current + 1 schedule data buffer may be changed
            finalize current data into the current + 1 schedule data buffer
            signal pipe2 that the current + 1 schedule data buffer has finalization data
        end if
        break from loop
    end if
    increment current by 1
end loop
```

F.5 Design of SPU Program `pipe2_spu`

After the `pipe2_spu` Program initializes, it loops pulling and hashing schedule data until the SPU_DONE signal is received.

The main outer SPU flow is identical to that of the `lanes_spu` program except that it additionally receives the EA of the pipe1 processing space during initialization so that the two pipes can interact.

The inner flow is managed by function `spu_process_pipe2` which processes individual hashing assignments by reading the data context provided by the PPU and performing hashing based on information from the data context and from schedule data prepared from pipe1 as follows:

```
start and finish DMA get of the data context
done if there are 0 bytes to process
start and finish DMA get of the starting hash value
set sector_index to 0 for looping processing one Sector at a time
wait for pipe1 to signal that schedule data at the current sector index is available
start DMA get current (first) schedule data buffer from pipe1 using DMA tag for current index
loop
    if more data is available on the next sector_index then
        wait for pipe1 to indicate that next schedule data is available
        start DMA get next schedule data using DMA tag for next index
    end if
    wait for DMA get of current DMA tag to complete
    signal pipe1 that the current schedule data buffer may be changed
    calculate the running hash from the current schedule data
    if last iteration then
        if data context indicates EOF then
            wait for pipe1 to indicate that the schedule data buffer at index current + 1 has finalization data
            start and finish DMA get of the schedule data from pipe1 current + 1 index to pipe2 current + 1 index
            finalize the calculated hash by hashing the finalization schedule data at the current + 1 index
        end if
        break from loop
    end if
    increment current by 1
end loop
start and finish DMA put of the calculated hash value
```

F.6 Design of SPU Program md5_spu

After the md5_spu Program initializes, it loops receiving and processing hashing jobs until the SPU_DONE signal is received.

The main outer SPU flow is identical to that of the lanes_spu program.

The inner flow is managed by function spu_process_md5 which processes individual hashing assignments by reading the data context provided by the PPU and then hashing the data identified in the data context as follows:

```
start and finish DMA get of the data context
done if there are 0 bytes to process
start and finish DMA get of the starting hash value
set sector_index to 0 for looping processing one Sector at a time
start DMA get current (first) data buffer using DMA tag for current sector index
loop
    if more data is available on the next sector_index then
        start DMA get next data buffer using DMA tag for next index
    end if
    wait for DMA get of current DMA tag to complete
    hash the current data
    if last iteration then
        if data context indicates EOF then
            finalize current data
        end if
        break from loop
    end if
    increment current by 1
end loop
start and finish DMA put of the calculated hash value
```

References

[1] Raghav Bhaskar, Pradeep K. Dubey, Vijay Kumar, Atri Rudra, and Animesh Sharma. Efficient Galois field arithmetic on SIMD architectures. June 7–9 2003.

[2] Thomas Chen, Ram Raghaven, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. *developerWorks*, November 29 2005. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`.

[3] Neil Costigan and Michael Scott. Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061, 2007. `http://eprint.iacr.org/`.

[4] Owen Harrison and John Waldron. AES encryption implementation and analysis on commodity graphics processing units. pages 209–226, September 13 2007. LNCS Volume 4727/2007.

[5] NIST. Secure hash standard (SHS). Technical Report FIPS PUB 180-3, National Institute of Standards and Technology (NIST), October 2008.

[6] R Rivest. The MD5 message-digest algorithm. Technical Report RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, April 1992.

[7] Jon Stokes. End of the line for ibm's cell, November 23 2009. `http://arstechnica.com/hardware/news/2009/11/end-of-the-line-for-ibms-cell.ars`.

[8] Jason Yang and James Goodman. Symmetric key cryptography on modern graphics hardware. pages 249–264, 2007. LNCS 4833.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research and Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, California