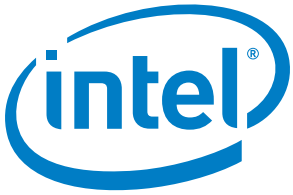


Intel® QuickAssist Technology

Performance Optimization Guide

September 2014

Document Number: 330687, Revision: 1.0

Contents

Figures

Tables

Revision History

Date	Revision	Description
September 2014	1.0	Initial release.

§

1 Introduction

This performance optimization guide for Intel® QuickAssist Technology can be used both during the architecture/design phases and the implementation/integration phases of a project that involves the integration of the Intel® QuickAssist Technology software with an application stack. Accordingly, the guide is divided into two main sections:

- Software Design Guidelines – Architecture and design guidelines on how best to integrate the Intel® QuickAssist Technology software into the application software stack. Tradeoffs between various design choices are described together with recommended approaches.
- Application Tuning – Guidelines to further increase the performance of Intel® QuickAssist Technology in the context of a full application.

The intended audience for this document includes software architects, developers and performance engineers.

In this document, for convenience:

Acceleration drivers is used as a generic term for the software that allows the QuickAssist Software Library APIs to access the Intel® QuickAssist Accelerator(s) integrated in the following processors:

- Intel® Communications Chipset 8900 to 8920 Series (codenamed Cave Creek)
- Intel® Communications Chipset 8925 to 8955 Series (codenamed Coleto Creek)
- Intel® Atom™ processor C2000 product family (codenamed Rangeley)

1.1 Terminology

Term	Description
IA	Intel® architecture CPU
Latency	The time between the submission of an operation via the QuickAssist API and the completion of that operation.
Offload Cost	This refers to the cost, in CPU cycles, of driving the hardware accelerator. This cost includes the cost of submitting an operation via the Intel QuickAssist API and the cost of processing responses from the hardware.
PCH	Platform Controller Hub
Throughput	The accelerator throughput, usually expressed in terms of either requests per second or bytes per second.

1.2 Where to Find Current Software and Documentation

Visit the Intel® Open Source Technology Center for a list of the Public versions of many Intel® QuickAssist documents go to: https://01.org/packet-processing/intel%C2%AE-quickassist-technology-drivers-and-patches

The collateral lists contain product documentation along with download instructions for obtaining the software package.

1.2.1 Product Documentation

Table 1 lists the documentation supporting this release. All documents can be accessed as described in Section 1.2.

Table 1. Product Documentation

Document	Number
Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide	330753
Intel® Communications Chipset 8925 to 8955 Series Software Programmer's Guide	330751
Intel® Atom™ Processor C2000 Product Family Software Programmer's Guide	503877
Intel® QuickAssist Technology API Programmer's Guide	330684
Intel® QuickAssist Technology Cryptographic API Reference Manual	330685
Intel® QuickAssist Technology Data Compression API Reference Manual	330686

2 Intel® QuickAssist Technology Software Overview

This section provides a very brief overview of the Intel® QuickAssist Technology software. It is included here to set the context for terminology used in later sections of this document. Refer to Table 1, *Programmer's Guide,* for your platform for more details.

The Intel® QuickAssist Technology software consists of an Intel® QuickAssist Technology API that is implemented by a driver which in turn drives the Intel® QuickAssist Accelerator hardware. The acceleration driver can run in either kernel space or in user space. When running in user space, the acceleration driver accesses the hardware directly from user space.

The Intel® QuickAssist Technology API supports two acceleration services:
1) Cryptographic and 2) Data Compression.

The acceleration driver interfaces to the hardware via hardware-assisted rings. These rings are used as request and response rings. Rings are grouped into banks (16 rings per bank). Request rings are used by the driver to submit requests to the accelerator and response rings are used to retrieve responses back from the accelerator. The availability of responses can be indicated to the driver using either interrupts or by having software poll the response rings.

At the Intel® QuickAssist Technology API, services are accessed via "instances". A set of rings are assigned to an instance and so any operations performed on a service instance will involve communication over the rings assigned to that instance.

3 Software Design Guidelines

This section focuses on key design decisions that should be considered, in order to achieve optimal performance, when integrating with the Intel® QuickAssist Technology software. In many cases the best Intel® QuickAssist Technology configuration is dependent on the design of the application stack that is being used and so it is not possible to have a “one configuration fits all” approach. The trade-offs between differing approaches will be discussed in this section to help the designer to make an informed decision.

The guidelines presented here focus on the following performance aspects:

- Maximizing throughput through the accelerator.
- Minimizing the offload cost incurred when using the accelerator.
- Minimizing latency.

Each guideline will highlight its impact on performance. Specific performance numbers are not given in this document since exact performance numbers depend on a variety of factors and tend to be specific to a given workload, software and platform configuration.

3.1 User Space vs. Kernel Space

The decision to use the user space version or kernel space version of the acceleration driver will be primarily dictated by the application stack that is being integrated with the Intel® QuickAssist Technology driver. If the application stack resides in user space, the user space version of the driver should be used and vice versa.

From a performance perspective, the same performance can be achieved by using a kernel space driver and application as can be achieved using a user space driver and application, assuming similar configurations.

It is not recommended to configure the user space Intel® QuickAssist Technology driver to use interrupts. Although this configuration is functional, it leads to sub-optimal performance. See Section 3.2 for more details on interrupts vs. polling.

3.2 Polling vs. Interrupts

Software can either periodically query the hardware accelerator for responses or it can enable the generation of an interrupt when responses are available. Interrupts or polling mode can be configured per instance via the platform-specific configuration file. Refer to Table 1, *Programmer’s Guide,* for your platform for configuration parameter details.

The properties and performance characteristics of each mode are explained below followed by recommendations on selecting a configuration.

3.2.1 Interrupt Mode

When operating in interrupt mode, the accelerator will generate an MSI-X interrupt when a response is placed on a response ring. Each ring bank has a separate MSI-X interrupt which may be steered to a particular CPU core via the CoreAffinity settings in the configuration file.

To reduce the number of interrupts generated, and hence the number of CPU cycles spent processing interrupts, multiple responses can be coalesced together. The presence of the multiple responses can be indicated via a single coalesced interrupt rather than having an interrupt per response. The number of responses that are associated with a coalesced interrupt is determined by an interrupt coalescing timer. When the accelerator places a response in a response ring, it starts an interrupt coalescing timer. While the timer is running, additional responses may be placed in the response ring. When the timer expires, an interrupt is generated to indicate that responses are available. Refer to Table 1, *Programmer's Guide,* for your platform for details on how to configure interrupt coalescing.

Since interrupt coalescing is based on a timer, there is some variability in the number of responses that are associated with an interrupt. The arrival rate of responses is a function of the arrival rate of the associated requests and of the request size. Hence, the timer configuration needed to coalesce X large requests is different from the timer configuration needed to coalesce X small requests. It is recommended that the timer is tuned based on the average expected request size.

The choice of timer configuration impacts throughput, latency and offload cost:

- Configuring a very short timer period maximizes the throughput through the accelerator, minimizing latency, but will increase the offload cost since there will be a higher number of interrupts and hence more CPU cycles spent processing the interrupts. If this interrupt processing becomes a performance bottleneck for the CPU, the overall system throughput will be impacted.
- Configuring a very long timer period leads to reduced offload cost (due to the reduction in the number of interrupts) but increased latency. If the timer period is very long and causes the response rings to fill, the accelerator will stall and throughput will be impacted.

The appropriate coalescing timer configuration will depend on the characteristics of the application. It is recommended that the value chosen is tuned to achieve optimal performance.

When using interrupts with the user space Intel® QuickAssist Technology driver, there is significant overhead in propagating the interrupt to the user space process that the driver is running in. This leads to an increased offload cost. Hence it is recommended that interrupts should not be used with the user space Intel® QuickAssist Technology driver.

3.2.2 Polling Mode

In polled mode, interrupts are fully disabled and the software application must periodically invoke the polling API, provided by the Intel® QuickAssist Technology driver, to check for and process responses from the hardware. Refer to Table 1, Programmer's Guide, for your platform for details on the polling API.

The frequency of polling is a key performance parameter that should be fine-tuned based on the application. This parameter has an impact on throughput, latency and on offload cost:

- If the polling frequency is too high, CPU cycles are wasted if there are no responses available when the polling routine is called. This leads to an increased offload cost.
- If the polling frequency is too low, latency is increased and throughput may be impacted if the response rings fill causing the accelerator to stall.

The choice of threading model has an impact on performance when using a polling approach. There are two main threading approaches when polling:

1. Creating a polling thread that periodically calls the polling API is often the simplest to implement, allows for maximum throughput, but can lead to increased offload cost due to the overhead associated with context switching to/from the polling thread.
2. Invoking the polling API and submitting new requests from within the same thread. This model is characterized by having a "dispatch loop" that alternates between submitting new requests and polling for responses. Additional steps are often included in the loop such as checking for received network packets or transmitting network packets. This approach often leads to the best performance since the polling rate can be easily tuned to match the submission rate so throughput is maximized and offload cost is minimized.

3.2.3 Recommendations

A polling model is recommended since a properly tuned polling model will have a lower offload cost compared to an interrupt model.

Summary of recommendations when using polling mode:

- Fine-tuning the polling interval is critical to achieving optimal performance.
- The preference is for invoking the polling API and submitting new requests from within the same thread rather than having a separate polling thread.

Summary of recommendations when using interrupt mode:

- Enable interrupt coalescing.
- Choose an interrupt coalescing timer configuration based on the average request size.
- Tune the timer configuration to achieve optimal performance.
- Do not use interrupts with the user space Intel® QuickAssist Technology driver.

3.3 Use of Data Plane (DP) vs. Traditional API

The cryptographic and compression services provide two flavours of API, known as the traditional API and the Data Plane API. The traditional API provides a full set of functionality including thread safety that allows many application threads to submit operations to the same service instance. The Data Plane API is aimed at reducing offload cost by providing a “bare bones” API, with a set of constraints, which may suit many applications. Refer to the *Intel® QuickAssist Technology Cryptographic API Reference Manual* for more details on the differences between the DP and traditional APIs for the crypto service.

From a throughput and latency perspective, there is no difference in performance between the Data Plane API and the traditional API.

From an offload cost perspective, the Data Plane API uses significantly fewer CPU cycles per request compared to the traditional API. For example, the cryptographic Data Plane API has an offload cost that is almost 10x lower than the cryptographic traditional API.

Note: One constraint with using the Data Plane API is that interrupt mode is not supported.

3.3.1 Batch Submission of Requests Using the DP API

The Data Plane API provides the capability to submit batches of requests to the accelerator. The use of the batch mode of operation leads to a reduction in the offload cost compared to submitting the requests one at a time to the accelerator. This is due to CPU cycle savings arising from fewer writes to the hardware ring registers in PCIe* memory space.

Using the DP API, batches of requests can be submitted to the accelerator using either the `cpaCySymDpEnqueueOp()` or `cpaCySymDpEnqueueOpBatch()` API calls for the symmetric cryptographic data plane API and using either the `cpaDcDpEnqueueOp()` or `cpaDcDpEnqueueOpBatch()` API calls for the compression data plane API. In all cases, requests are only submitted to the accelerator when the `performOpNow` parameter is set to `CPA_TRUE`.

It is recommended to use the batch submission mode of operation where possible to reduce offload cost.

3.4 Synchronous (sync) vs. Asynchronous (async)

The Intel® QuickAssist Technology traditional API supports both synchronous and asynchronous modes of operation. The Intel® QuickAssist Technology Data Plane API only supports the asynchronous mode of operation.

With synchronous mode, the Intel® QuickAssist Technology API will block and not return to the calling code until the acceleration operation has completed.

With asynchronous mode, the Intel® QuickAssist Technology API will return to the calling code once the request has been submitted to the accelerator. When the accelerator has completed the operation, the completion is signalled via the invocation of a callback function.

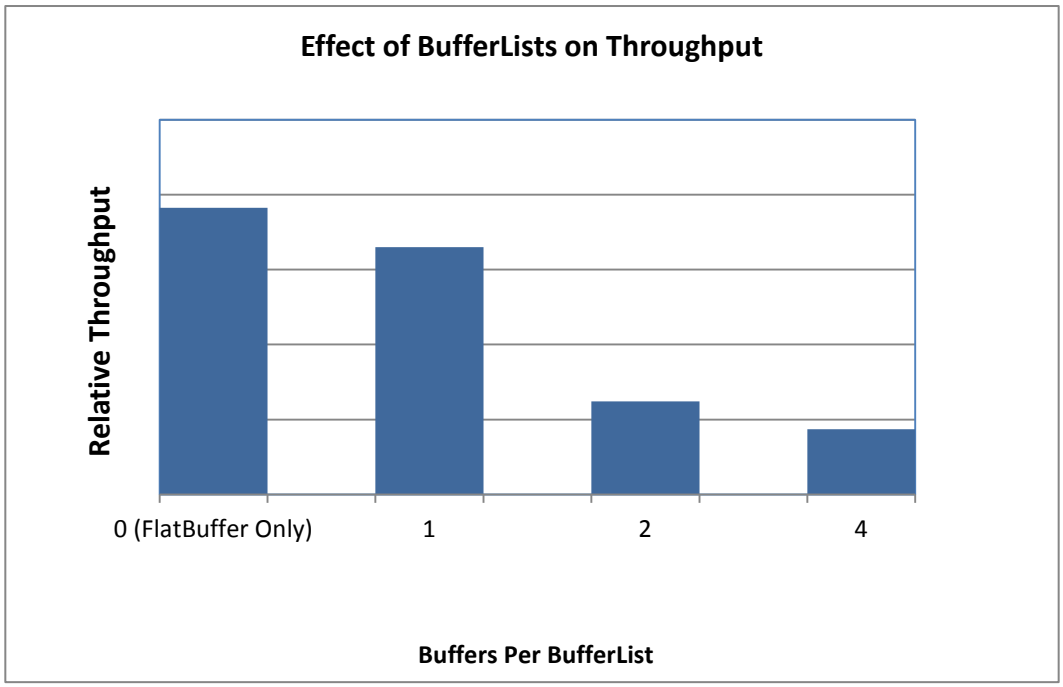
From a performance perspective, the accelerator requires multiple inflight requests per acceleration engine to achieve maximum throughput. With synchronous mode of operation, multiple threads must be used to ensure that multiple requests are inflight. The use of multiple threads introduces an overhead of context switching between the threads which leads to an increase in offload cost.

Hence, the use of asynchronous mode is the recommended approach for optimal performance.

3.5 Buffer Lists

The symmetric cryptographic and compression Intel® QuickAssist Technology APIs use buffer lists for passing data to/from the hardware accelerator. The number of elements in a buffer list has an impact on throughput with performance degrading as the number of elements in a buffer list increases. This is illustrated in Figure 1. In order to minimize this degradation in throughput performance, it is recommended to keep the number of buffers in a buffer list to a minimum. Using a single buffer in a buffer list leads to optimal performance.

Figure 1. Performance Impact of Using Multiple Buffers in a Buffer List



Note: Specific performance numbers are not given in this document since exact performance numbers depend on a variety of factors and tend to be specific to a given workload, software and platform configuration.

When using the Data Plane API, it is possible to pass a flat buffer to the API instead of a buffer list. This is the most efficient usage of system resources (mainly PCIe bandwidth) and can lead to lower latencies compared to using buffer lists.

In summary, the recommendations for using buffer lists are:

1. If using the Data Plane API, use a flat buffer instead of a buffer list.
2. If using a buffer list, a single buffer per buffer list leads to highest throughput performance.
3. If using a buffer list, keep the number of buffers in the list to a minimum.

3.6 Maximum Number of Concurrent Requests

The depth of the hardware rings used by the Intel® QuickAssist Technology driver for submitting requests to, and retrieving responses from, the accelerator hardware can be controlled via the configuration file using the `CyXNumConcurrentSymRequests`, `CyXNumConcurrentAsymRequests` and `DcXNumConcurrentRequests` parameters. These settings can have an impact on performance:

- As the maximum number of concurrent requests is increased in the configuration file, the memory requirements required to support this also increases.
- If the number of concurrent requests is set too low, there may not be enough outstanding requests to keep the accelerator busy and so throughput will degrade. The minimum number of concurrent requests required to keep the accelerator busy is a function of the size of the requests and of the rate at which responses are processed via either polling or interrupts (see Section 3.2 for more details).
- If the number of concurrent requests is set too high, the maximum latency will increase.

It is recommended that the maximum number of concurrent requests is tuned to achieve the correct balance between memory usage, throughput and latency for a given application. As a guide, the maximum number configured should reflect the peak request rate that the accelerator must handle.

3.7 Symmetric Crypto Partial Operations

The symmetric cryptographic Intel® QuickAssist Technology API supports partial operations. This allows a single payload to be processed in multiple fragments with each fragment corresponding to a partial operation. The Intel® QuickAssist Technology API implementation will maintain sufficient state between each partial operation to allow a subsequent partial operation for the same session to continue from where the previous operation finished.

From a performance perspective, the cost of maintaining the state and the serialization between the partial requests in a session has a negative impact on offload cost and throughput. To maximize performance when using partial operations, multiple symmetric cryptographic sessions must be used to ensure that sufficient requests are provided to the hardware to keep it busy.

For optimal performance, it is recommended to avoid the use of partial requests if possible.

Note that there are some situations where the use of partials cannot be avoided since the use of partials and the need to maintain state is inherent in the higher level protocol, e.g. the use of the RC4 cipher with an SSL/TLS protocol stack.

3.8 Stateful Compression

Stateful compression operations require the Intel® QuickAssist Technology API implementation to maintain state between subsequent API calls for a stateful compression session. Additionally, within a stateful compression session, one operation must complete before a subsequent operation for the same session can be submitted. This serialization can limit performance and so it is recommended that multiple stateful sessions are used at the same time to ensure that the hardware accelerator can be kept busy.

3.9 Load Balancing

When using the Intel® QuickAssist Technology API with Intel® Communications Chipset 8900 to 8920 Series or Intel® Atom™ processor C2000 product family (codenamed Rangeley), optimal throughput performance is achieved when operations are load balanced across the engines within the accelerator.

For example, for a top-SKU Intel® Communications Chipset 8900 to 8920 Series device, which has four cryptographic engines and two compression engines, a minimum of four cryptographic service instances and two compression service instances are required to maximize performance.

Note: Intel® Communications Chipset 8925 to 8955 Series (codenamed Coleto Creek) has multiple cryptographic and compression engines, but the hardware can load balance and provide full performance using only one service instance for cryptographic operations and one service instance for compression operations.

It is also recommended to assign each service instance to a separate CPU core to balance the load across the CPU and to ensure that there are sufficient CPU cycles to drive the accelerators at maximum performance.

When using interrupts, the core affinity settings within the configuration file should be used to steer the interrupts for a service instance to the appropriate core.

3.10 Best Known Method (BKM) for Avoiding PCI Performance Bottlenecks

For optimal performance, ensure the following:

- All data buffers should be aligned on a 64-byte boundary.
- Transfer sizes that are multiples of 64 bytes are optimal.
- Small data transfers (less than 64 bytes) should be avoided. If a small data transfer is needed, consider embedding this within a larger buffer so that the transfer size is a multiple of 64 bytes. Offsets can then be used to identify the region of interest within the larger buffer.
- Each buffer entry within a Scatter-Gather-List (SGL) should be a multiple of 64 bytes and should be aligned on a 64-byte boundary.

4 Application Tuning

This section describes techniques you may employ to optimize your application.

4.1 Platform Level Optimizations

This section describes platform-level optimizations required to achieve the best performance.

4.1.1 BIOS Configuration

Maximum performance is achieved with the following BIOS configuration settings:

CPU Power and Performance:

- Intel® SpeedStep® technology is disabled
- All C-states are disabled
- Max CPU Performance is selected

4.1.2 Memory Configuration

To achieve maximum performance, all memory channels on the platform should be populated with memory running at full speed.

4.1.3 Payload Alignment

For optimal performance, data pointers should be at least 8-byte aligned. In some cases, this is a requirement. See the API for details.

For optimal performance, all data passed to the Intel® QuickAssist Technology engines should be aligned to 64B. The *Intel® QuickAssist Technology Cryptographic API Reference* and the *Intel® QuickAssist Technology Data Compression API Reference* manuals document the memory alignment requirements of each data structure submitted for acceleration.

4.1.4 NUMA Awareness

For a dual processor system, memory allocated for data submitted to the acceleration device should be allocated on the same node as the attached acceleration device. This is to prevent having to fetch data for processing on memory of the remote node.

4.2 Intel® QuickAssist Technology Optimization

This section references parameters that can be modified in the configuration file or build system to help maximize throughput and minimize latency or reduce memory footprint. Refer to Table 1, Programmer's Guide, for your platform for detailed descriptions of the configuration file and its parameters.

There are two versions of the configuration file:

- Version1 is the original configuration file and allows full control of all the services available.
- Version2 is intended to make it easier for the user; some fine-grained control that is available in V1 is missing in V2. An example is the ability to set the bank configuration parameters such as `InterruptCoalesingTimer`.

This document refers to parameters available in Version 2 of the configuration file.

4.2.1 Disable Services Not Used

The compression service, when enabled, impacts the throughput performance of crypto services at larger packet sizes and vice versa. This is due to partitioning of internal resources between the two services when both are enabled. It is recommended to disable unused services as shown below.

For Intel® Communications Chipset 8900 to 8920 Series:

`ServicesEnabled = cy0;cy1;dc` (for all services)

Note: To use Wireless Firmware, you **must** enable the dc service, even though it is not used.

`ServicesEnabled = cy0;cy1` (crypto service only)

`ServicesEnabled = dc` (compression service only)

For Intel® Communications Chipset 8925 to 8955 Series (codenamed Coleto Creek):

`ServicesEnabled = cy;dc` (for all services)

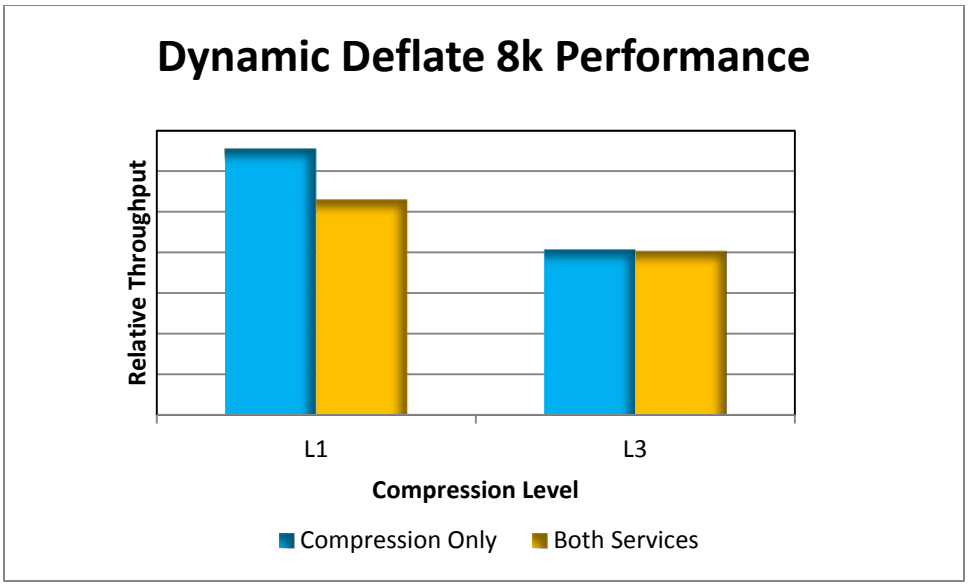
Note: To use Wireless Firmware, you **must** enable the dc service, even though it is not used.

`ServicesEnabled = cy` (crypto service only)

`ServicesEnabled = dc` (compression service only)

Figure 2 below shows the performance impact on compression when the crypto service is enabled but not used.

Figure 2. Comparison of Compression Performance with and without Crypto Service Enabled



4.2.2 Using Embedded SRAM

Note: This section is not relevant for Intel® Communications Chipset 8925 to 8955 Series (codenamed Coleto Creek) software.

Embedded SRAM can be used to reduce the PCIe* transactions that occur during dynamic compression sessions. This will have a small positive impact on performance, but also frees up PCIe bandwidth for other services.

Embedded SRAM can be set using the configuration file parameter:

```
dcTotalSRAMAvailable = 524288
```

The default value is 0, the maximum value is 1024 x 512 (524288).

4.2.3 Disable Parameter Checking

Parameter checking results in more Intel architecture cycles consumed by the driver. By default, parameter checking is enabled. The user can disable parameter checking by passing in ICP_PARAM_CHECK=n at build time.

The following steps can be followed:

```
setenv ICP_ROOT /QAT
setenv ICP_BUILDSYSTEM_PATH $ICP_ROOT/quickassist/build_system/
setenv KERNEL_VERSION `uname -r`
setenv KERNEL_SOURCE_ROOT /usr/src/kernels/$KERNEL_VERSION/
setenv ICP_BUILD_OUTPUT $ICP_ROOT/build
cd $ICP_ROOT/quickassist
make ICP_PARAM_CHECK=n
```

4.2.4 Adjusting the Polling Interval

This section describes how to get an indication of whether your application is polling at the right frequency. As described in Section 0, the rate of polling will impact latency, offload cost and throughput. Also Section 0 describes two ways of polling:

- Polling via a separate thread.
- Polling within the same context as the submit thread.

With option 1, there is limited control over the poll interval, unless a real time operating system is employed. With option 2, the user can control the interval to poll based on the amount of submission made.

Whichever method is employed, the user should start with a low frequency of polling, and this will ensure maximum throughput is achieved. Gradually increase the polling interval until the throughput starts to drop. The polling interval just before throughput drops should be the optimal for throughput and offload cost.

Note that this method is only applicable where the submit rate is relatively stable and the average packet size does not vary. To allow for variances, a larger ring size is recommended, but this in turns will add the maximum latency.

4.2.5 Using the Coalescing Timer to Tune Throughput and Latency

The coalescing timer is a configuration file parameter that allows interrupts to be coalesced by the driver and after a time expires, the interrupt fires, which then allows a number of responses to be processed at one time, rather the processor being continually interrupted for each response processed.

Users should be aware of the following:

- Increasing the coalescing time will increase the latency of packet processing and may reduce the IA cycle offload cost.
- Decreasing the coalescing timer will decrease latency, but may increase IA cycles and may decrease throughput.

4.2.5.1 Service Type

Due to the number of potential combinations of the API and the configuration parameters, we have categorized the services to generalize the effect of the coalescing time. For this document, services have been categorized from A to C as described in the following table.

Table 2. Service Categorization

Service Type	Type C	Type B	Type A
Symmetric	3DES, Kasumi > 512 Bytes	All other ciphers > 512-1024 bytes	All Ciphers < 512 bytes
Asymmetric	All Asymmetric Operations		
Compression	All Compression Operations		

The coalescing timer controls how often an interrupt is triggered when interrupt mode is used. The interrupt timer is controlled by the configuration file parameter:

```
InterruptCoalescingTimerNs
```

There is also:

```
InterruptCoalescingNumResponses
```

Note: This parameter is only valid when the default ring size is used. Refer to Table 1, Programmer's Guide, for your platform for permitted and default values.

(V1 configuration file provides a per-bank configuration. Refer to Table 1, Programmer's Guide, for more details.)

Optimal setting of this parameter is dependent on submit rate of the service being used. The rule of thumb for this setting is as follows:

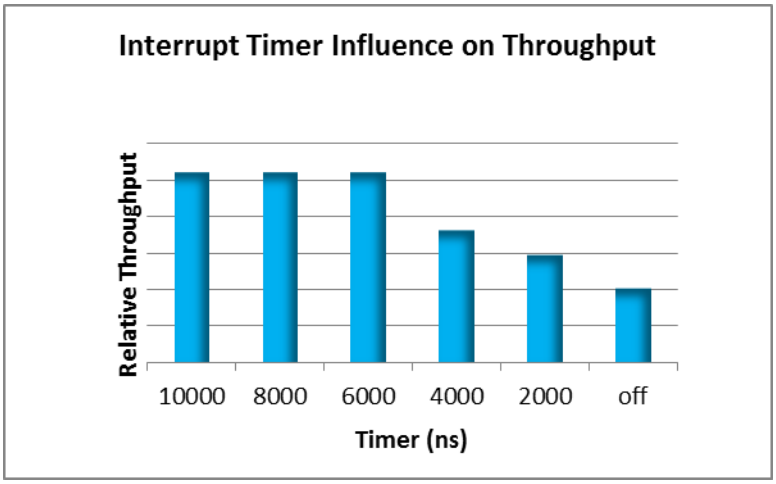
- The slower the service, the lower the Interrupt Timer
- Timer ~= 500 for type C service
- Timer ~= 2000-4000 for type B service
- Timer ~= 6000-7000 for type A service

See Section 4.2.5.1 for a description of the service speeds.

It is recommended that some additional tuning is done by the user around these ranges since there are too many combinations/specific scenarios to provide more concise guidance.

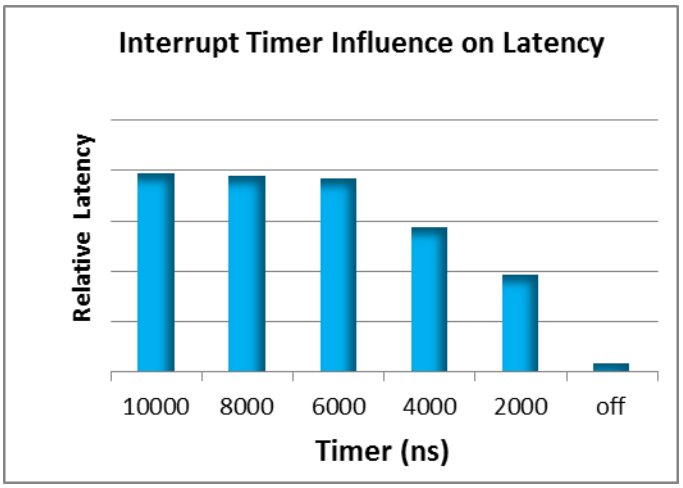
As an example, AES256 CBC is rated in this document as a type A service, the recommended timer = 6000. As can be seen below, the throughput starts to get affected when the timer is lower, the trade-off is improved latency. In the example following, if the user chooses 4000, the throughput suffers ~30% whilst the latency decreases ~25%.

Figure 3. Example of the Interrupt Timer Influence on Throughput



Note: Packet Size can affect the choice of Coalescing Timer. The larger the average size packet, the longer the processing time and subsequently a short timer maybe required to ensure max throughput is achieved.

Figure 4. Interrupt Timer Influence on Latency



4.2.6 Reducing Asymmetric Service Memory Usage

This section describes how to reduce the memory footprint required by using the asymmetric crypto API.

The asymmetric cryptographic service requires a far larger memory pool compared to symmetric cryptography and compression. The more logical instances that are defined in the configuration file, the more memory is consumed by the driver. This memory usage maybe unnecessary if only the symmetric part of the cryptographic service is used. Alternatively, the memory requirement may be reduced depending on the user's requirements on the asymmetric service.

Option 1: Asymmetric Not required

The minimum value for CyXNumConcurrentAsymRequests in the configuration file is 64, where X = the instance number.

Large values of the above configuration file parameter increase memory requirements and more instances will also cause increases. To minimize memory, the user should:

- Set MAX_MR_ROUNDS=1
- Set LAC_PKE_MAX_CHAIN_LENGTH=1
- Minimize the number of logical crypto instances

Details of these changes are described in option 2 and 3 following.

Option 2: Reduce Prime Miller Rabin Rounds

By default, the driver uses 50 rounds of Miller Rabin to test primality. If the user does not require this amount of prime testing, the following environment variable can be set at build time to reduce this:

```
export max_mr=NUM_ROUNDS
```

where NUM_ROUNDS is between 1 and 50.

Option 3: No Prime or ECDH services required

At build time:

- export max_mr=1
- in <ICP_ROOT> /QuickAssist/lookaside/access_layer/src/common/crypto/asym/include/lac_pke_utils.h

  set LAC_PKE_MAX_CHAIN_LENGTH=1

4.2.7 Improving Driver Loading Times with Multiple Devices

adf_ctl can be used via qat_service or independently to start or stop the acceleration driver. When using multiple acceleration devices in a platform and when not specifying a particular device, adf_ctl acts on the devices in series. For instance, in a platform with four acceleration devices, with the driver not loaded, and assuming that adf_ctl is in the current working directory, the driver can be loaded:

```
# ./adf_ctl up
```

This will start each of the four acceleration devices one after another. To start the devices faster, adf_ctl can be run in parallel on individual devices. For instance:

```
# ./adf_ctl icp_dev0 up &
# ./adf_ctl icp_dev1 up &
# ./adf_ctl icp_dev2 up &
# ./adf_ctl icp_dev3 up &
```

§