Accelerator Multi-Threading

Nic McDonald, Subhasis Das, Milad Mohammadi
{*nmcdonal, subhasis, milad*}*@stanford.edu*

Abstract

Special purpose accelerators are being used widely in most embedded and server processors to gain orders of magnitude gains in terms of efficiency as well as performance. Currently most accelerators are used such that a single application gets exclusive access to the accelerator at a time. This can cause the accelerator to become unfair to small jobs since they can be waiting for a longer job to complete. However granting exclusive access ensures that a single job operates at the maximum throughput possible since there are no overheads for context switching.

In this work, we propose a hardware framework to enable an accelerator to process multiple jobs simultaneously and compare it to a software based multithreading approach. We show that this achieves much throughput similar to no threading while maintaining fairness levels as fine grained switching. It is able to do so by decreasing the switching overhead by ∼150x.

1 Introduction

Special purpose accelerators are widely used for their high efficiency and low power consumption. Server processors frequently have cryptographic and CRC accelerators along with other accelerators such as regular expression accelerators and compression accelerators (e.g., [6]). Accelerators have also been proposed for doing high throughput jobs such as database processing (e.g., [5]).

A common problem with accelerators is that they are frequently designed to be single-threaded, i.e., they can process only a single job at any time. In future systems this will become a problem as accelerators are used more widely since exclusive access to an accelerator is fundamentally unfair to small-sized jobs. Under this scenario, the only method to share the accelerator between multiple jobs is to implement the multithreading in software and break up a job into multiple blocks which get scheduled according to some scheduling policy by a kernel space driver. Even in this case, the device has to have the ability to report the state of a partial computation to the OS and to resume a partially complete job.

The problem with a software based approach is the high latency of invoking a kernel interrupt handler. Our experiments indicate that it may take as long as 600 ns for an interrupt handler to get called and another 900 ns for it to finish handling the interrupt. On the other hand, depending on the internal state size of the hardware, it may be faster to store temporary state into a region adjacent to the accelerator with the accelerator itself managing the task scheduling. Many accelerators with low amount of state such as cryptographic or CRC can be made to switch jobs in single cycle. In our case, this resulted in a decrease in switch time by ∼ 150x. Since even in a software managed scenario, the accelerator needs to implement a context save/restore functionality, we can add the scheduler to the hardware with very low overheads.

Lower switching latencies in hardware enables one to do much finer grained task switching without decreasing the throughput of the accelerator by a significant amount. As we will show in Section 4.2, implementing multiple threads in the accelerator can achieve almost ideal fairness at very near to minimum latency.

Further, implementing the contexts in the accelerator enables one to keep the software interface simple since the software developer does not have to implement the context switching in the driver. As we will show,

this adds only one extra register to the accelerator which can be programmed at initialization time. The remaining interface remains similar to a single threaded accelerator.

Multi-threading has been partially implemented in network interface cards (NIC) in the form of multiple queues, which is managed by the hardware. However, a general interface for the hardware developers to implement queueing is still absent and open source accelerators such as cryptographic, CRC accelerators are all single threaded. Our interface is designed to be simple so that any accelerator can be ported to conform to the interface with little extra effort.

2 Motivation

Many system-on-chip designs today are integrating accelerators to improve the performance and efficiency of the system. As accelerators become more common, more software will be written to take use of the available accelerators present in a system. Most accelerators can run jobs of varying sizes. When multiple programs access an accelerator with variously sized data sets, a policy of first-come-first-serve gives high priority to larger jobs. If a small job requests access just after a large job requests access, the small job will incur the latency of the large job on top of the latency of its own job. In this situation the small job views the accelerator as having poor performance. If the jobs were submitted in reverse order, the small job would see full throughput and the large job would see a negligible decrease in performance. It is natural to assume that small jobs expect low latency, and large jobs expect high throughput.

Providing fair access to an accelerator requires a system to define the policy of fairness. We define *fair access* as having equal access to the accelerator in relation to the number of jobs requesting access. For example, if there are 4 jobs requesting access the the accelerator, each job is guaranteed 25% of the running time on the accelerator. We treat all jobs submitted to the accelerator as having equal priority regardless of the job size. There are many options when defining *fair access*, however, we believe that our fairness policy is the most general and provides sufficient system stability.

The standard method for sharing an accelerator is on a job-by-job basis where a user gains access to the hardware via a mutex or semaphore. According to the fairness policy previously defined, this method of sharing is severely unfair. A common alternative is to break each job into blocks and submit the job to the hardware on a block-by-block basis. If all programs have an equal opportunity to submit blocks to the accelerator, the fairness policy is upheld. To be completely fair, all job sizes must be greater than or equal to the block size. This insight infers that fine granularity of block sizes produces the most fair access given variable sized jobs.

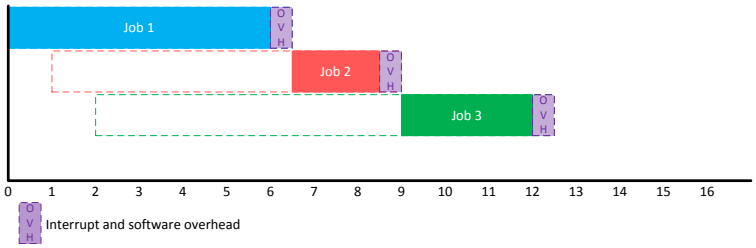


Figure 1: Software job-by-job accelerator sharing

Figure 1 shows a time diagram of a system where jobs are submitted to the accelerator on a job-by-job basis in order of first-come-first-served. Job #1 requests access to the accelerator at time 0 and immediately runs because there are no other job currently running. Job #2 requests access to the accelerator at time 1 and job #3 requests access at time 2, however, both are delayed because job #1 is already accessing the accelerator. Job #2 and #3 are queued in order of request time. Job #1 sees no degradation of performance because it did

not have to wait in line for the accelerator. Job #2 and #3 both observed a large degradation of performance because they had to wait for job #1 to complete. On top of that, job #3 also had to wait for job #2. The overhead shown at the end of each job is for accounting for the interrupt response and rescheduling time needed in software. The unfairness of this approach is portional to the difference between the minimum and maximum job sizes.

If we view each job as a set of blocks and divide the job into units of computation that can be performed in one time unit, job #1 is 6 block, job #2 is 2 blocks, and job #3 is 3 blocks. In an ideal system with zero hardware/software communication overhead, we can see that job #1 sees a performance of 1 block per time unit. It considers the accelerator performance as 1.00. Because job #2 requested when job #1 still had 5 blocks left to process, job #2 sees a performance of 2 blocks per 7 time units which is only 29% of job #1's performance. Job #3 sees a performance of 3 blocks per 10 time units which is 30% of job #1's performance. Having a greater than zero overhead makes the unfairness even larger.

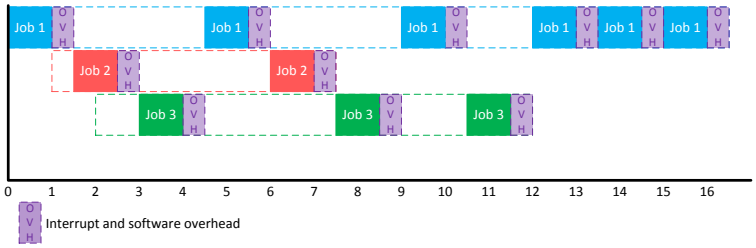


Figure 2: Software block-by-block accelerator sharing

Figure 2 shows a time diagram of a system where each job is divided into blocks and the blocks are scheduled on the accelerator in a round robin fashion until the jobs are finished. The percentage of time each job spends on the accelerator is inversely equal to the number of jobs being scheduled on the accelerator. For instance, from time 2 to time 7.5 there are 3 jobs that are sharing the accelerator. In this time frame, each job gets 33.3% of the time on the accelerator. From time 7.5 to tim 12 there is 2 jobs and each gets 50% of the time on the accelerator.

Given the fairness policy defined earlier, we consider this system to be 100% fair. However, the communication overhead that was previously only incurred at the end of each job is now incurred after each block. This example has an overhead of half of one time unit. Comparing the completion time of all jobs between figure 1 and figure 2 shows that the second system incurred an extra 4 time units of overhead. Depending on the granularity of block sizes and overheads, this could cause severe performance degradation of a system.

We declare the system shown in figure 2 as 100% fair because the block size is greater than or equal to the smallest job size supported on the accelerator. If the block size was set to 3 time units worth of computation, job #2 would see a performance decrease because it could wait 3 time units before running its job of only 2 blocks. This insight illustrates conjecture that a completely fair setup forces all job sizes must be greater than or equal to the block size. If jobs are submitted with sizes less than the block size, they should know that they are being treated unfairly.

Many accelerators support a wide variety of jobs sizes. The interrupt response and rescheduling overhead is the driving factor of the performance of a system that divides jobs into blocks. The overhead is generally constant and independent of the block size. For performance reasons, it is desirable to set the block size as large as possible since the overhead is incurred on each block switch. However, for fairness reasons, it is desirable to set the block size as small as possible.

Figure 3 shows the tradeoff between the block size and the performance. The top line shows the ideal system where there is zero overhead. In this case the accelerator can be used 100% of the time even when the block size is 1. The bottom line shows a system where the overhead is 1024. When the block size is 1,

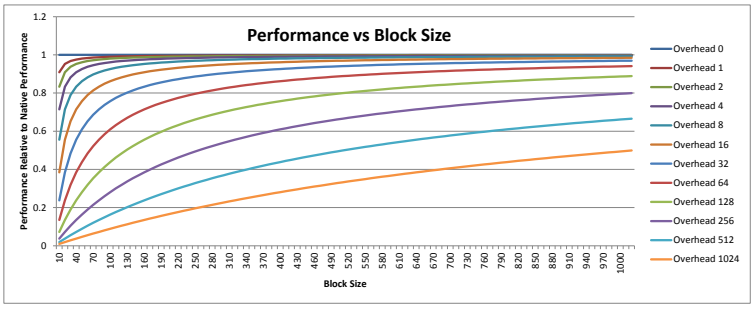


Figure 3: Performance vs Block Size with Variable Overheads

the accelerator is severely underutilized only getting one time unit of work done for every 1025 time units. Even when the block size is increased to 1024, the accelerator utilization is only 50%. As we showed earlier, the fairness of the system is inversely equal to the block size. As figure 3 shows, if a system allows job sizes to be small, the interrupt response and rescheduling overhead must be extremely small to uphold the fairness policy.

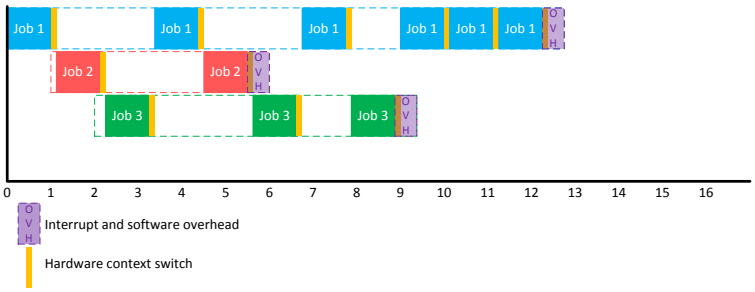


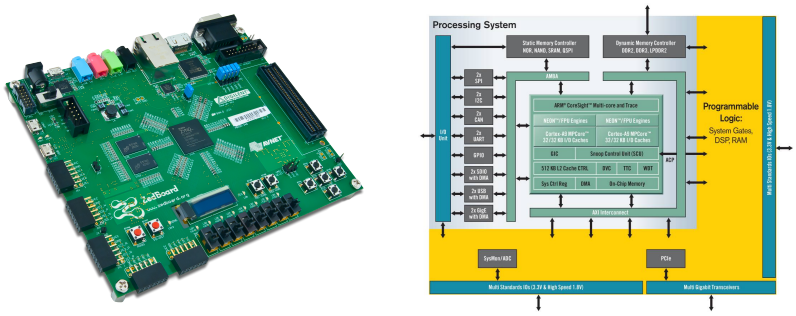
Figure 4: Hardware block-by-block accelerator sharing

Figure 4 shows a time diagram of a system where the accelerator has the ability to switch blocks without the intervention of the software. The accelerator in this system takes each job as a whole, but runs each job block-by-block while interleaving with other jobs' blocks. It essential performs the same function as the software did for the system shown in figure 2. However, because the accelerator takes each job as a whole, there only needs to be one interrupt per job. The context switching overhead of the accelerator is orders of magnitude faster now that it does not need software intervention. Figure 4 also shows that the hardware context switching overhead is overlapped by the interrupting overhead incurred at the end of each job. The accelerator continues working even though the interrupt is still in flight. It should also be noted that the accelerator's context switching overhead is very small, possibly even 0. When the logic resources are available, the accelerator can pipeline the context switch with the next job's block computation. This moves the top line of figure 3 from being a mythical ideal system to a realistic system. It should be noted that when comparing the systems of figures 2 and 4, the fairness, as defined by our policy, is exactly the same. This is because the smallest job size is greater than or equal to the block size. There would be a difference if the block size of the two systems was reduced. The system in figure 2 can't handle as small of job sizes as the system in figure 4. The major two differences are: the hardware context switching system finishes all 3 jobs in about 4 time units less; and the hardware context switching system allows for much smaller job and block sizes while maintaining perfect fairness.

3 Implementation

For an example accelerator we chose to implement a *Cyclic Redundancy Check* (CRC) engine. CRCs are most commonly used for data corruption detection, the most common use being 802.3 Ethernet frames. CRC is a good target for an accelerator because its function is extremely efficient in hardware and extremely inefficient in software. This can be seen in current networking systems. The Ethernet CRC functions are implemented in the network interface controller hardware but when data corruption checks are done in software it is common to use less secure algorithms, such as the TCP or UDP checksum as defined in RFC 793, which are faster for general purpose processors.

If TCP and UDP had access to fast CRC hardware, would they use less secure software-based algorithms? Questions like this stage the necessity for accelerator multi-threading. If TCP and UDP began to use CRC instead of their existing weak algorithms, there would potientally be a large contention for access to the CRC accelerator. Consider how scientific, data-center, and general purpose computing would all change if fast efficient accelerators were plentful, fair, and easy to access.



(a) Zedboard (b) Xilinx Zynq

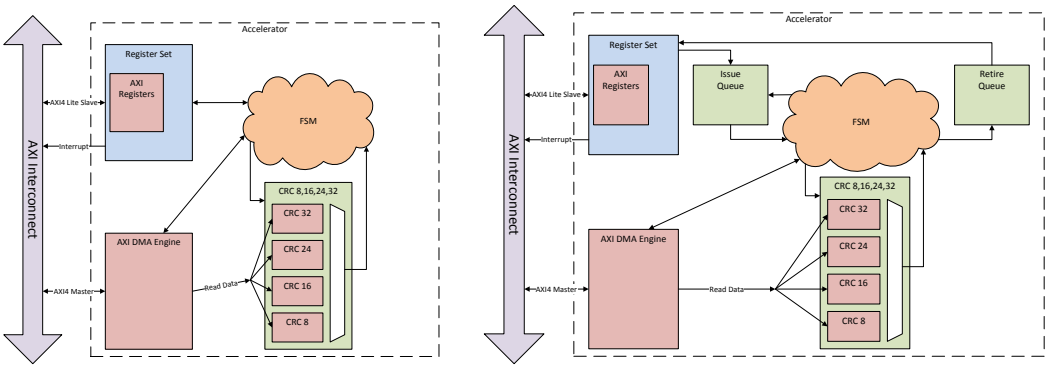
Figure 5: Accelerator Multi-Threading Development Platform

We chose to develop and test on the *ZedBoard* development platform. The ZedBoard is built around the *Xilinx Zynq All Programmable SoC* which is an integrated circuit that contains a dual-core ARM Cortex-A9 processing system embedded into *Field Programmable Gate Array* (FPGA) fabric. The FPGA fabric is connected to the ARM processing system through ARM's *AMBA eXtensible Interface* (AXI). This allows custom modules designed in the FPGA fabric to appear as memory mapped accelerators. The custom modules can also be masters on the AXI bus where they can issue direct memory reads and writes. The two CPUs run at 667 MHz while the FPGA devices run at 100 MHz. We placed a incrementing timer in the FPGA to measure the latencies of various operations. We developed the drivers for our accelerators to work with the Linux 3.3.0 kernel.

3.1 Single Threaded Accelerator

3.1.1 Hardware

The implementation of our baseline single threaded CRC accelerator is very simple and the methodology is similar to most accelerator designs. Figure 6a shows the block diagram of this design. A simple register set provides a controllable interface to the software. The accelerator only needs the physical memory address and length of the data. Triggering the *start* register starts the accelerator for one CRC operation. The state machine within the accelerator fetches the data, pushes it through the CRC function, saves the resulting CRC remainder, then interrupts the processor. Upon receiving the interrupt, the software then reads the result from register set.



(a) Single Threaded CRC Accelerator (b) Hardware Multi-Threaded CRC Accelerator

Figure 6: Accelerator Multi-Threading Development Platform

3.1.2 Software

Here we will discuss the generic interface of the linux driver developed for the single threaded accelerator with each job having exclusive access to the accelerator. We developed the driver to have no copying to or from user space so as to eliminate extra latency involved in copying. The basic system calls exposed to the user space are detailed next. This was achieved by returning an allocated memory region by a `mmap()` system call into which the process can write its data to be CRC-ed. After writing the CRC data, a process issues a `IOCTL_GO` `ioctl` to submit the job and a `read()` system call to get the CRC data.

3.2 Software Multi-Threaded Accelerator

3.2.1 Hardware

The implementation of our software block-by-block sharing accelerator took just a small modification to the accelerator shown in figure 6a. The CRC function module takes two control signals, one that indicates the first data beat and one that indicates the last data beat. The baseline accelerator set these signals according to the which data beat was being received from the DMA engine. The software block-by-block sharing accelerator has an extended register set that allows these two signals to masked off. This effectively allows the software to send partial jobs (blocks) to the accelerator for processing. Each block just needs to have the *first* and *last* signals properly set. With these modifications, the software can now interleave the jobs' processing block-by-block. The blocks from all jobs are processed with a round-robin approach to adhere to the fairness policy. This accelerator interrupts the processor after each block is processed.

3.2.2 Software

In the driver, the rescheduling is done at every interrupt by invoking the interrupt handler which goes through a circular queue of jobs to find the next schedulable job (i.e., one which has not finished).

The API presented to the user applications is exactly the same as before with the addition of an additional `ioctl`, `IOCTL_CHUNK` by which it can set the block size for software multithreading. Additional complexities are involved in this case since the interrupt handler needs to read the software queue while the `ioctl` and `read` functions needs to write to the software queue. Thus the queue structure is protected by a spin lock. Additional data sharing occurs since the `ioctl` needs to submit a job by itself to the device in case the device is idle. This is also ensured by a spin lock.

Thus we observe that doing multi-threading in software leads to several synchronization problems which

must be solved by mutexes or spin locks. Also as discussed earlier, the measured overhead of calling an interrupt handler is ∼150x compared to the cycle time of the accelerator and is thus unsuitable for doing fine grained switching.

3.3 Hardware Multi-Threaded Accelerator

3.3.1 Hardware

Figure 6b shows the design of our hardware multi-threaded CRC accelerator. The register set is nearly identical to the accelerator shown in figure 6a. The only major difference is that the hardware multi-threaded design has a register which allows the software to set the block size. The major difference between the two accelerators is that the multi-threaded design has two new design blocks: the issue queue and retire queue.

Each time the software submits a new job to the accelerator register set, it must give the job a unique tag. The tag is the job's identifying number as it is processed and eventually given back to the software. Each time a job is submitted to the accelerator it gets pushed into the issue queue. The issue queue is a RAM based structure similar to a *first-in-first-out* (FIFO) buffer where the priority is strictly set by age. It differs from a FIFO in that the oldest job also has the ability to be cycled to the back of the buffer while being modified. Each job in the issue queue has 5 values: current address, remaining length, unique tag, CRC, and first flag.

Each time the accelerator's state machine is free for processing another block, it looks at the issue queue's next job and determines whether it will be the last block of the job or not. It will either run a full block of computation or enough to finish out the job, whichever is smaller. At the end of the computation block, if there is remaining computation in the job, the address and length are updated and the job is cycled to the back of the issue queue. The *first* flag in the issue queue is always cleared when a job is cycled. If the job completed during computation block, it is removed from the issue queue and pushed into the retire queue. In either case, the state machine within the accelerator tracks the progress of the job based on the job's remaining *length* value. The *first* and *last* signals are applied to the CRC function based on whether it is the first and last data beat, respectively.

The retire queue is a simple FIFO structure that holds all the retire jobs until the software has removed them. The retire queue holds the tag and resulting CRC of the job. As with most FIFOs, our FIFO design has output signals for *full* and *empty*. To generate interrupts to the processor, we inverted the *empty* signal and tied it to the *interrupt* output signal. Since the processor responds to level sensitive interrupts, the accelerator will continue to interrupt the processor until the retire queue is empty. This provides a dynamic interrupt coalescing scheme without any additional logic. Interrupt coalescing is a method of batching multiple events into one interrupt to reduce the overhead incurred in interrupt overheads. Making this dynamic means that the batch size is varied according to conditions. Typically, making interrupt coalescing dynamic has high hardware complexity. In our case, we got it for free!

3.3.2 Software

In the case of a hardware multi-threaded accelerator, the driver is much simpler than the software multi-threaded case. In this case, we just grab a mutex to submit a job to the accelerator once a process issues the `IOCTL_GO` `ioctl`. Then we busy wait on the `START_OK` register of the accelerator until it goes high. Once it goes high (i.e., we have a free slot in the issue queue), we issue our job with an unique tag and release the mutex. In the interrupt handler, we read the whole retire queue until it is empty and wake up each job corresponding to the read `tag` register.

Thus, we see that we don't have multiple synchronization problems in the driver unlike the software multi-threaded case, thus reducing the development time as well as the possibility of bugs in the driver.

4 Results

4.1 Methodology

We performed the following experiments to measure the throughput and performance of our mechanism against no multithreading and software multithreading:

1. Single Threaded Throughput Analysis: this analysis discusses the throughput of the accelerator when a single job is partitioned into a range of block sizes. In other words, it illustrates the impact of task switching overhead on throughput. When the block size is equal to the job size, the throughput is 1 and as the number of blocks associated with the jobs increase, the throughput is expected to drop.

2. Throughput vs. Fairness Analysis: two of the main figures of merit in resource sharing evaluation are resource throughput and resource sharing fairness. Here, throughput is a measure of the time it spends in processes jobs vs. the total time it spends in processing jobs as well as in associated overheads; of course, the more task switching, the more the overheads and the lower the throughput. Fairness is a measure of how much resource allocation is given to any given job that is contending for the resource. Of course, the shorter the switching intervals, the more fair the system is to to small jobs. Since job switching implies incurring task switching overheads, here we discuss the tradeoff between fairness vs. high-throughput. We stress on the great potential of hardware multithreading in enabling greater fairness under high throughput conditions compared to software multithreading and no multithreading schemes.

3. Optimal Chunk Sizing: Our ultimate goal in this project is to gain best execution time for small tasks and reasonable execution times for large tasks. While fairness and throughput are critical measures for a robust design of such systems, we require to find the optimal input-data blocking size. While such a block size must maintain ácceptable fairness and throughputv́alues, it provides very high performance gains for small jobs and reasonable performance for large jobs.

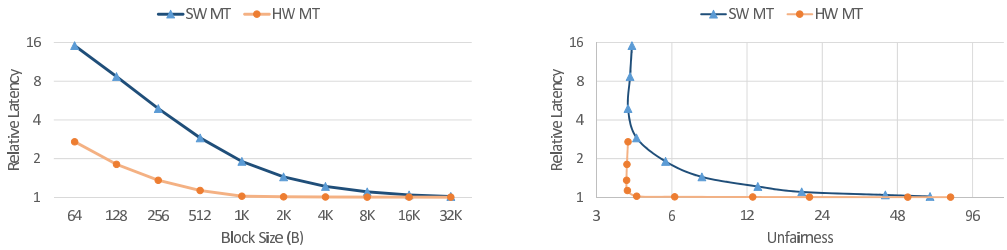
4. Performance: To compare our final design solution with the software multithreading and no-multithreading schemes, we collected the average execution time of a number of randomly sized jobs, normalized to the job sizes, running on a number of threads. As explained in a later section, this experiment is performed at different block sizes to enable observing multithreading performance benefit at different data-partitioning granularity points. Moreover, we evaluate the system performance under two sharing contention traffic scenarios: (a) a 100%-accelerator-traffic scenario in which threads continuously issue jobs, and (b) a real-system traffic scenario in which threads have some compute overhead between sending jobs.

4.2 Latency and Fairness

In order to measure the effect of block sizes on latency and fairness of the scheme, we created a testbench in which 4 threads submit several jobs with sizes drawn uniformly randomly from 4B to 32KB. We then ran the jobs

1. with no blocking and no multithreading. We call the latency of job i (i.e., the time from request to the final interrupt) as $T_{min,i}$

2. in isolation, i.e., without sharing the accelerator with any other thread. However the hardware or software as applicable still divides up the job into several blocks and submits them to the accelerator, thereby incurring the context switch overheads. We call the latency to run job i in this case as $T_{excl,i}$



(a) Latency vs Block Size

(b) Latency vs Unfairness for hardware and software based multi-threading

Figure 7: System Performance metrics vs Block Size

3. with the threads running in parallel and multithreading done either by the hardware or by the software driver. We call the latency of job i in this case as $T_{share,i}$.

We then define an average relative latency as the average of $T_{excl,i}/T_{min,i}$ taken over all the jobs.
We also define a measure of unfairness in the following way:

$$Unfairness = \max_i \frac{T_{share,i}}{T_{excl,i}}$$

The intuition behind this metric is to measure the maximum amount by which a job was slowed down due to the sharing.

Figure 7a shows the relative latency vs the block size. As we can observe from this figure, the latency of both the software multi-threaded as well as the hardware multi-threaded version increases by making the job switching more fine grained. However, the latency of the hardware multi-threaded version is smaller than the software one at any given granularity which enables us to do much finer grained switching in hardware and achieve better fairness results as shown in Figure 7b.

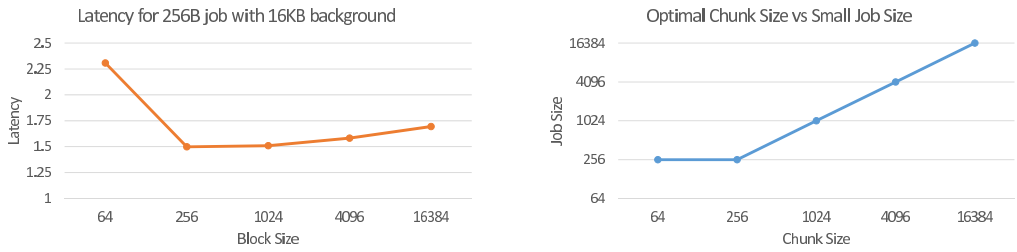
Thus we can see that the hardware implemented switching has $\sim$ 10x more throughput than the software switched scheme at finer switching granularities i.e., lower unfairness values.

4.3 Optimal Block Sizes

Figure 8a shows the performance results of running a of 256B (a small job) that is in continuous contention with a 16KB job (i.e. a large job). The bathtub curve graph indicates the optimal block size for this scenario to be at around 256B. The left side of the bathtub curve shows performance loss due context switching overhead of small jobs, and the right side shows performance loss due to small jobs being starved under large job block sizes that hog the resource. Figure 8b illustrates the summary of the same experiment while sweeping the size of one of the two contending jobs between 64B to 16KB, and maintaing the size of the other contending job at 16KB. The graph illustrates the optimal block size value tracks the size of the smaller job. This indicates the small job is better off running as a single block, as otherwise, the overhead associated with partitioning it will only hurt the performance. Intuitively, this makes sense as often partitioning the small job creates significant overheads for the small job and provides no performance benefit to the large job.

4.4 Execution Performance

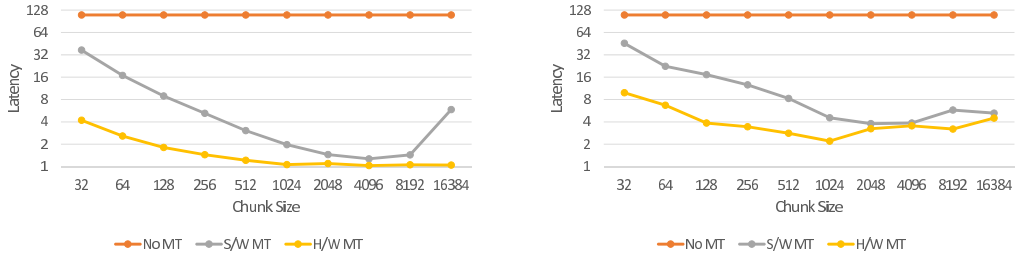
Here, the overall system performance of No-Multithreading, Software Multithreading, and Hardware Multi-threading implementation under the following conditions/configurations are compared. In both cases, hard-



(a) Latency vs. Block Size

(b) Optimal Block Size vs. Job Size

Figure 8: Latency for 256B job with 16KB background
(Normalized to Exclusive Execution)



(a) Performance vs. Block Size with Low Sharing Contention

(b) Performance vs. Block Size with High Sharing Contention

Figure 9: Latency for 256B job with 16KB background
(Normalized to Exclusive Execution)

ware multithreading performance proves to be superior to that of the other two schemes.

1. Traffic:
   (a) CPU compute overhead with $< 100\%$ load on the accelerator (Figure 9a)
   (b) no CPU compute overhead and 100% load on the accelerator (Figure 9b)
2. Job sizes: uniform-randomly chosen block size between [32B, 16KB] for all jobs
3. Job Count: 800 jobs in the experiment
4. Thread Count: 8

5 Related Work

This section describes the related-work in both the policies and mechanisms used in this work.

1. *Policy*: a similar mechanism for software multithreading has been studied by Beisel, et al. In this paper, the authors present an extension of the Completely Fair Scheduler (CFS) in Linux OS to support cooperative multi-tasking with time-sharing for heterogeneous elements in Linux. Processing tasks are broken into small units by the user/programmer. The OS then constructs one thread per unit of work and finds the most affine processor on which to schedule the task. Each unit of work is stored in

the run queue corresponding to the specified accelerator. The accelerator runs through the jobs in FIFO order. Tasks that are partitioned into small work units request the same resource repeatedly until the task is complete. The conclusions made in this paper are consistent with our software multithreading framework results [3].

2. *Mechanism*: to the authors knowledge, the Netlogic and Cavium multi-core chips have multithreaded interfaces for their multi-queue systems used for the NIC and for scheduling work between cores [1, 4]. Texas Instruments Keystone chips, also, have an elaborate queue management system for communicating to accelerators, including direct accelerator to accelerator communication [2].

6 Conclusion

We have shown the necessity for fast, fair, and efficient accelerator interfaces. We have also developed a simple framework that can be adapted to a large set of already existing accelerators. We used this framework to implement a hardware-managed multi-threaded CRC accelerator and showed its tremendous utilization and performance gains over no multi-threading and software-managed multi-threading.

7 Future Work

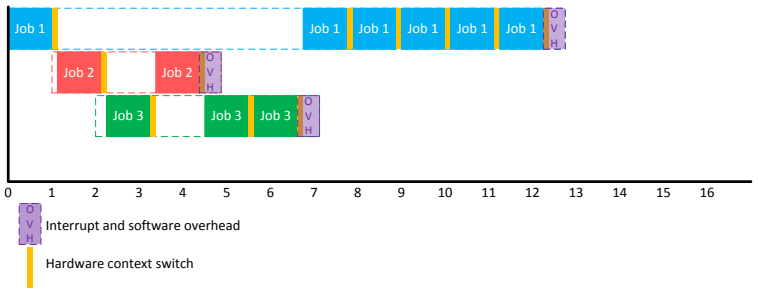


Figure 10: Multi-Threading with 1-bit Priority

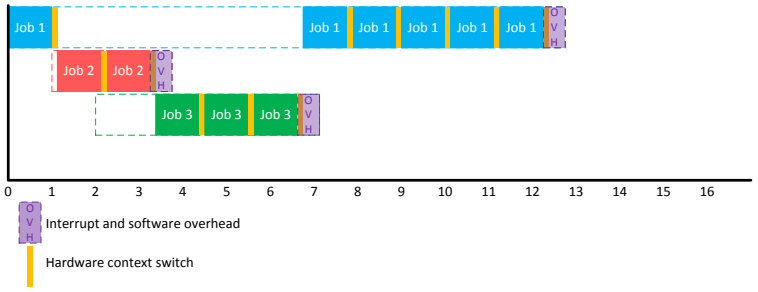


Figure 11: Multi-Threading with 2-bit Priority

We've shown a basic system for a multi-threaded accelerator. We believe our design can be easily adapted to most existing accelerators that are memory intensive with low state overhead. Additional work must be done to find an efficient method for hardware multi-threading where each job's intermediate state is large. Another option for future work is developing more efficient scheduling algorithms rather than the basic round-robin approach we used. There are many cases where different scheduling decisions can be made

that will increase the throughput without negatively effecting the fairness. Instead of having only one RAM inside the issue queue, multiple RAMs could create a scheme for supporting priorities. Figure 10 shows a time diagram where job #1 is low priority and jobs #2 and #3 are high priority. If we compare the completion times with that of figure 4 we can see that the low priority job, #1, didn't see any negative latency, however, jobs #2 and #3 get to finish earlier. We can extend this idea to multiple priorities as shown in figure 11. Job #1 is the lowest priority, job #2 is the highest priority, and job #3 is medium priority. Again, the lower priority jobs see no negative impact on latency while the higher priority job get completed earlier. A system that modifies its scheduler beyond the round-robin approach must be aware of starvation. The systems shown in figures 10 and 11 could result in starvation of the lower priority jobs if the higher priority jobs are scheduled at a high frequency.

References

[1] Broadcom to acquire netlogic microsystems, inc., a leader in network communications processors. http://www.broadcom.com/press/release.php?id=s604481, Sept. 9 2011.

[2] Keystone device architecture. http://processors.wiki.ti.com/index.php/Keystone_Device_Architecture, Oct. 10 2012.

[3] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler, 2011. pp. 223–226.

[4] A. Haider. Cavium networks introduces world's first multi service security processor family. http://www.cavium.com/newsevents_nitrox_plus.htm, July 7 2002.

[5] L. Huei. Database accelerator, Aug. 6 1996. US Patent 5,544,357.

[6] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. Power7: Ibm's next-generation server processor. *Micro, IEEE*, 30(2):7–15, 2010.