Towards OS kernel acceleration in heterogeneous systems

Alexander Kroh
University of NSW
akroh@cse.unsw.edu.au

Oliver Diessel
University of NSW
odiessel@cse.unsw.edu.au

ABSTRACT

While Moore's law continues to hold true, limits on the electrical distance between CPU core components have forced the maximum operating frequency of the CPU core to plateau. CPU vendors have introduced acceleration features to compensate for this frequency limit; however, these acceleration features are limited in their design because they cannot be tuned to target a specific high-level software system architecture. In this paper we evaluate the potential for an operating system kernel to be accelerated in the programmable logic part of a heterogeneous system. We consider accelerating the scheduler of the seL4 micro-kernel as a case study for feasibility. Lastly, we evaluate the proposed hardware-software co-design on commercial off-the-shelf hardware and identify architectural limitations of the approach.

Categories and Subject Descriptors

C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*;
D.4.1 [**OPERATING SYSTEMS**]: Process Management;
D.4.8 [**OPERATING SYSTEMS**]: Performance

Keywords

Operating System Performance, Reconfigurable Computing. Hardware Acceleration, Hardware Software Co-design

1. INTRODUCTION

In this paper, we explore the benefits and architectural limitations of provisioning a general purpose processor with programmable logic as a key accelerator for the operating system (OS). The focus of this acceleration is on the core of the system, the operating system kernel. While kernel operations are infrequent and short in nature, they are the most critical components of the system. Real-time (RT) systems, in particular, depend on the kernel for system reliability and schedulability. In an RT system, the Worst Case Execution Time (WCET) must be taken into consideration to ensure that critical tasks are completed in a timely manner. Failure to do so can lead to mission failure, personal injury or loss of life. The WCET is much larger than the average case execution time and computational resources will generally be over provisioned in such a system in order to ensure that it will operate safely in the unlikely event that the WCET is observed [4].

An example of such a system is an automotive airbag control unit. The controller must ensure that the airbag is inflated only after a collision, but well before injury can occur. In order to provide such a high guarantee on safety, a dedicated electronic control unit is deployed to ensure tight bounds on the WCET. The control unit remains idle until a collision is detected and, with no other software tasks executing on the CPU, it can act immediately on data received from the impact sensor. If a tight bound on the WCET can be assured in the presence of other software tasks, such as ABS breaking, the airbag deployment function could be integrated with other subsystems of the automobile. With less electronic control units (ECUs) present, the overall cost and complexity of the system is reduced. If hardware acceleration can bring the WCET closer to the average execution time, a tight bound on execution time can be assured for this purpose.

Hardware acceleration features are already provided by most CPU cores. These features include hardware page table walkers for virtual memory, branch prediction and speculative loading, out of order execution, and direct Java bytecode execution [2]. While these generic features are useful, they provide only fine-grained acceleration mechanisms that cannot leverage knowledge of the high-level architecture from the complete, and potentially large, computational problem. The focus of acceleration for this paper is on software subsystems such as the kernel scheduler. By accelerating these coarse-grained features, we aim to have a larger impact on system performance.

The operating policies of high-level software subsystems are chosen at design time and are specific to the intended application [14]. For example, Linux allows the user to select between three different scheduling policies for each process [5]. Two of these policies are specific to real-time processes while the third policy is dedicated to traditional processes. The Linux -rt patchset further improves the real-time capabilities of the Linux scheduler in order to tune the system for real-time applications [8]. It is not practical to introduce a fixed implementation for each application so we will rely

on programmable logic to provide the hardware platform for acceleration. The software developer can then choose from a range of hardware acceleration strategies that each implement a specific policy [13].

In order to compete with a high performance CPU with dedicated caches, a low-latency communication channel between CPU and FPGA is ideal. However, performance improvements may still be obtained with a high-latency communication channel if the hardware provides enough acceleration, or even if the acceleration feature reduces the cache footprint of the kernel. Such a feature would significantly reduce the WCET of the kernel by reducing the variance in execution time that arises from an unknown cache state. An operation that normally occupies a large number of cache lines can be offloaded to programmable logic, which can access RAM directly and in parallel with CPU execution.

This paper is organised as follows. Section 2 provides an overview of related work in the field. Section 3 provides a detailed architectural description of the proposed design for a hardware-accelerated operating system kernel. An evaluation of the proposed system is presented in Section 4. Section 5 outlines directions for future work before conclusions are drawn in Section 6.

2. RELATED WORK

Research in the area is primarily focused on accelerating an RTOS through programmable logic in order to reduce the WCET and system latency. Mooney et al. present a framework for real-time operating system construction in a hardware-software co-design [15]. The systems engineer is able to choose from a set of features that are provided in software and hardware implementation libraries. Hardware acceleration features include deadlock detection and dynamic memory management systems. Each of these have corresponding implementations in software such that the level and type of acceleration can be configured by the engineer.

Andrews et al describe how hardware acceleration can improve the performance of a number of operating system features [1]. The focus is again on real-time systems, where latency is a fundamental design constraint. The authors propose that a dedicated hardware task be deployed to manage hardware interrupt signals. In a traditional system, hardware interrupts are delivered immediately, regardless of the priority and criticality of the current thread of execution. The response to such an event is to simply mark the associated handler thread as runnable such that it may be scheduled at a later time. The proposed solution avoids unnecessary interruption of the current thread of execution by postponing the delivery of the interrupt until a time when the associated interrupt handler can and should be scheduled. This mechanism can also be extended to improve the efficiency of software synchronisation primitives. When a thread has completed the execution of a critical region in which mutual exclusion is required, the operating system is invoked to determine if any waiting thread should be scheduled. Hardware support can be used to avoid an interaction with the OS by allowing the hardware to interrupt the current thread of execution only if a rescheduling event is required.

Other contributions have focused directly on accelerating the RTOS scheduler. Gupta et al. evaluate the performance of three key scheduler acceleration policies [9]. First, they evaluate an unmodified scheduler which is to be used as a baseline for comparison. Next, they evaluate the use of a dedicated scheduler that executes on an independent and otherwise idle core. Finally, they evaluate a scheduler that is implemented in dedicated hardware. The CPU cycles required for a scheduling event were reduced from 185,937 cycles in both software schemes down to 142 cycles when the hardware scheduler was used.

Kuacharoen et al. identify a limitation that the provided RTOS scheduling policies are not flexible enough for real-world systems. The authors develop a system which supports the reconfiguration of the hardware scheduler to a variety of scheduling policies at run time [13]. The authors evaluate their system in a simulation environment and report the cost of modifying the list of runnable threads within the scheduler to be just one or two cycles, the cost of executing the instruction which triggers the change. Alternatively, Dodiu et al. propose an ASIC implementation of the scheduler where the user may select from a fixed set of policies [7].

More ambitious work involves the implementation of the RTOS completely within hardware [16, 17]. Ong et al implement an RTOS in programmable logic to support application software running on a NIOS soft-core processor [17]. The evaluation of this system shows that a hardware implementation of an RTOS can lead to 83.5% improvement in interrupt latency when compared to a system which runs on the NIOS soft-core processor alone. This improvement must be considered to be an upper bound because a performance improvement of the baseline system should be expected if the soft-core NIOS processor is replaced by a dedicated ASIC processor.

Thus far, related work has been limited to two broad system architectures. Most solutions propose that the FPGA fabric be provided with low-latency direct access to the CPU register file. These systems are evaluated in simulation because of the theoretical nature of the hardware architecture involved or are only applicable when a low performance soft-core processor is used. Alternatively, the proposed systems are provided with access to a high-latency off-chip FPGA. These systems amortize communication latency with parallel CPU execution. This architecture is becoming more popular as both Xilinx and Altera have begun to integrate high performance ARM application processors onto the same die as the FPGA. Dahlstrom et al. utilise this emerging technology to support the ARM CPU with a hardware implementation of the operating system scheduler [6]. However, the motivation for that work is not to improve system performance or to reduce latency, instead, the authors seek to improve operating system security. The authors propose that key operating system functions be migrated to hardware in order to obscure a malicious application's view of global system state. While the work is orthogonal to that which is presented in this paper, the system architecture is closely related.

Our work differs from previous work on hardware accelerated operating system subsystems because the evaluation of

our system is not conducted by simulation on theoretical hardware. Instead, our system is constructed and evaluated on commercial off-the-shelf hardware. The chosen hardware is the Xilinx Zynq®-7000 All Programmable System on Chip, which provides both a high-performance ARM processor and programmable logic on a single die. The operating system in our design executes on the high performance ARM processor rather than on a low performance soft processor. To our knowledge, this is the first "real" test of the potential to accelerate a high performance OS kernel executing in a commercial heterogeneous environment.

3. SYSTEM ARCHITECTURE

The seL4 micro-kernel [12] has been chosen as the OS kernel for acceleration. seL4 has a sound WCET analysis and there is movement to extend the kernel for RT applications [4]. seL4 is considered to be a micro-kernel because device drivers and other services are implemented as user-space applications rather than being provided directly within the kernel. A key advantage of this approach is that there exists only a minimum amount of software that must be trusted to ensure the correct operation of the system. Drivers, servers and applications all execute in a low-privilege operating mode of the CPU.

The chosen subsystem for acceleration is the kernel scheduler. Although the scheduler is not a long running operation, the performance of the scheduler is critical to system latency in terms of interrupt request (IRQ) delivery and efficient communication between client-server application software. Once the system has been initialised, the kernel provides 3 key functions, all of which result in an invocation to the kernel scheduler.

1. *IRQ delivery.* When an IRQ exception is received by the kernel, the kernel identifies a registered notification object, raises a flag within this object and unblocks any thread that was waiting on this notification. If the unblocked thread is of a higher priority than the active thread, the active thread must be inserted back into the scheduling queue and replaced by the waiting thread.
2. *Preemption IRQ.* When the preemption IRQ arrives, the current thread must be inserted back into the scheduling queue and a new thread is chosen and activated.
3. *Inter-Process Communication (IPC).* When a thread sends an IPC to another thread, the sender becomes blocked and the receiving thread is inserted back into the scheduling queue. The scheduler is then invoked to choose a new thread to be executed.

The communication latency between the CPU and the programmable logic is a key concern in the design of our hardware-software co-design. We must make sure that the cost of accessing the programmable logic does not exceed the performance improvement that a hardware implementation can provide. This is particularly true in our case because microkernel operations are designed to be short in nature.

seL4 provides a *fixed-priority preemptive scheduler* such that a thread will never execute while a runnable thread of higher priority exists in the system. It has been tuned for low latency IPC – we therefore expect it to be challenging to gain a benefit from hardware acceleration.

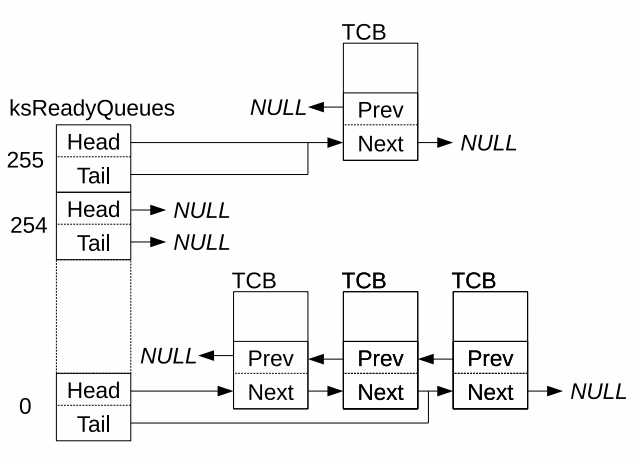

Figure 1: Software architecture of legacy scheduler.

3.1 Software architecture

The set of runnable threads in the seL4 micro-kernel is implemented as a doubly-linked list with one list for each of the 256 priority levels. The *next* and *previous* pointers of this list are maintained as part of the Thread Control Block (TCB) of each thread. A thread is appended to the end of its associated list when it has exhausted its allocated execution time interval or when it transitions from the blocked to the runnable state. If the thread has been preempted, perhaps because a higher priority thread has become unblocked and is now runnable, the remaining execution time of the thread is recorded and the thread is added back to the beginning of its associated list. The kernel maintains a set of head and tail pointers for each priority in global state. When the kernel scheduler is invoked, it walks this global list of runnable threads, known as the *ksReadyQueues*, from the highest priority (255) to the lowest priority (0) until it finds a non-empty list of runnable threads. If a non-empty list of runnable threads is found, the scheduler will remove a thread from the head of this list and mark it as the active thread. If there exists no runnable thread in the system, an implicit *idle* thread will be scheduled until an external IRQ causes a waiting thread to become runnable. The kernel scheduler data structure is illustrated in Figure 1.

The seL4 kernel implements a *fastpath*, a hand-optimised path for key operating system calls. A key feature of the *fastpath* is to allow IPC to higher or equal priority receivers to complete without invoking the scheduler. An IPC is a blocking call such that the thread that performs the IPC does not re-enter the *ksReadyQueues*. Instead, this thread is blocked waiting for a reply. Because the kernel implements a *fixed-priority preemptive scheduler*, we know that there does not exist a runnable thread with a higher priority than the sender. If the receiver is blocked waiting for an IPC and is of a higher or equal priority to the sender, we know that the receiver will be the new highest priority runnable thread in the system. For this reason, the receiver

can immediately become the new active thread without invoking the scheduler. In this way, IPC to a higher or equal priority thread need not invoke the scheduler function of the kernel.

Since the commencement of this work, the software architecture of the kernel scheduler has been further optimised by supplementing the design with a two-level bitfield lookup. Each bit in the second level lookup word corresponds to one of 32 priorities. If the bit is set, the associated priority contains at least one runnable thread. If the bit is clear, there are no runnable threads for the priority group. In the same way, the first-level bitfield reflects the presence of a runnable thread in each group of 32 priorities. The scheduler uses the Count Leading Zeros *CLZ* instruction on the first level bitfield and maps the result to the appropriate second level bitfield. The scheduler repeats this process on the second level bitfield to find the highest priority which has a runnable thread. The bitfields are adjusted when any thread is transitioned to the runnable state, or when the last thread at a particular priority leaves the runnable state.

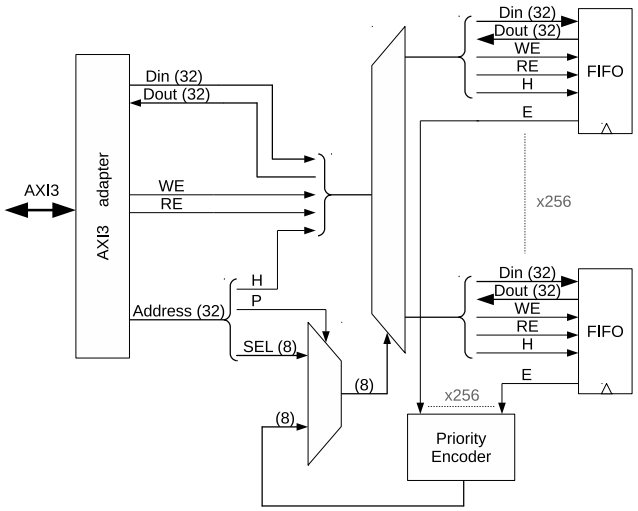

Figure 2: Hardware architecture of hardware scheduler.

3.2 Hardware architecture

The Avnet Zedboard is the chosen platform for system development. The Zedboard is a low-cost development platform that features the Xilinx Zynq®-7000 All Programmable System on Chip. The Zynq provides a dual ARM Cortex-A9 application processor and on-chip programmable logic. Communication between the ARM cores and the programmable logic is achieved through a range of ARM AXI communication buses [3].

The hardware design must provide the same features as the software design for compatibility. The hardware scheduler must allow a thread to be appended to both the head and tail of a *ksReadyQueue* and allow a thread to be removed from the head. Additionally, the hardware architecture must provide a level of acceleration which cannot be obtained in software.

The hardware architecture provides acceleration by allowing software to manipulate the head or tail of a *ksReadyQueue* in a single transaction to the AXI bus. This can be achieved because a transaction to an AXI slave peripheral conveys three core pieces of information. The transaction provides the data, a context for the transaction through the provided address, and additionally serves as a signal that a reaction to the provided stimulus is required. While the software implementation of the *ksReadyQueues* requires that software maintain a doubly-linked list of threads for each priority, a hardware implementation relieves this burden of list maintenance from software. It allows the software to add a thread to the head or tail of a specific priority by initiating a single write transaction. This transaction can contain a reference to the TCB of the thread as data and encode the priority and other manipulation parameters in the provided address. Similarly, software can remove a thread from a specific priority by initiating a single read transaction in which the desired priority is encoded in the provided address and a reference to the removed TCB can be returned as the data portion of the transaction. In either case, the underlying TCB data can still be read or written from the low-latency cache or from main memory. Finally, a single read transaction can be issued to both identify and remove a thread from the highest priority, non-empty *ksReadyQueue*. This can be done be encoding the parameters of this request in the address of the AXI bus transaction. The performance of locating the next thread to activate during a scheduling event can then be further improved by providing an O(1) lookup implementation in hardware.

The required hardware architecture is that of a priority queue. While there exists a lot of research on priority queue implementations [10, 11], the behaviour of insertions with equal priority are generally not well defined. Since priority queue hardware design and implementation is not the focus of this research, we used a trivial implementation to explore our ideas.

The hardware architecture, shown in Figure 2, follows closely that of the software architecture. The *ksReadyQueues* are replaced by FIFOs, where FIFO data represents a reference to the TCB of a thread in main memory. The head and tail of the FIFO can be swapped at any time by asserting the H signal. This signal allows a thread to be added to either the head or the tail of the FIFO. Write enable (WE) and read enable (RE) signals are provided to control the addition (push) or removal (pop) of a FIFO data item. If neither WE nor RE are asserted, a read operation will return the appropriate data from a FIFO without the side effect of item removal. Software is able to explicitly select the FIFO that is to be operated on by using the *SEL* signal. This allows software to select which FIFO a thread should be added to or removed from. Each FIFO additionally provides an E signal which reflects whether or not the corresponding FIFO is empty. Each E signal is routed to a 256-bit asynchronous priority encoder. When the state of any E signal changes, the priority encoder output will be updated to reflect this change before the next rising edge of the subsystem clock. It is this priority encoder that allows the highest priority runnable thread to be identified and removed from the list in a single operation. With a P signal asserted, the *SEL* signal is ignored and the highest priority non-empty FIFO

is selected by the output of the priority encoder.

Each FIFO signal is mapped to one or more bits of the address that is provided by software for the AXI bus transaction. Bits 0 and 1 of the address are reserved for 32 bit word alignment. The *H* signal is decoded from bit 2 of the requested address. By using this bit, we are consistent with the interleaved head and tail pointer layout of the software *ksReadyQueues*. Bits 3 to 10 represent the *SEL* signal and allows the software to specify a scheduling queue priority for the transaction. Bit 11 is used as the *P* signal. When this bit is asserted, bits 3 to 10 are ignored and the value at the output of the priority encoder is used in their place. Finally bit 12 can be used to mask the signals of *RE* and *WE*. This feature prevents the AXI transaction from having the side effect of modifying the content of the FIFOs and is useful for debugging. A read transaction in this case will return a reference to the TCB that is at the head of the list of runnable threads at the given priority without removing it from the list. A write operation will have no affect.

The priority queue described above was connected to a General Purpose (GP) AXI3 master port on the Zedboard platform through a protocol adapter. This protocol adapter serves only to translate between the complex AXI communication protocol and the trivial *RE* and *WE* protocol required by the FIFOs. Each transaction to the hardware scheduler is 32 bits in total size. For this reason, there is no benefit in supporting high bandwidth AXI burst mode transfers [3]. The hardware implementation of the *ksReadyQueues* is driven directly by the AXI clock. The maximum operating frequency of the implemented system is 100MHz.

4. EVALUATION

The kernel scheduler can be invoked in a controlled way by performing an IPC from one thread to a thread of lower priority. IPC of other priorities are handled by the *fastpath* and do not invoke the scheduler. An IPC benchmarking framework, known as *sel4bench* was used to benchmark the number of CPU execution cycles required to perform an IPC from a thread of the highest priority to threads of various lower priorities. When the *fastpath* is avoided and the scheduler is invoked, the priority of the sending thread has no impact on scheduler execution or performance.

We define the IPC execution cycles as the number of CPU cycles required for the sending thread to complete the IPC. This metric does not include the CPU cycles consumed by the receiving thread. The IPC that is used for the benchmark involves no data payload delivery and is made to a thread that resides in the same address space (a page table and Address Space ID switch are not performed). The benchmark was performed with a hot cache by using 16 cache warming iterations before performing 50 measurement iterations. Unless otherwise specified, the hardware scheduler AXI clock was configured to be 100MHz and the CPU operates at 666MHz.

The benchmark was run with various scheduling methods. The SW scheduler is the legacy scheduler described in Section 3.1. Described in the same section, the PB scheduler is the software extension to the legacy scheduler which provides O(1) lookup of the highest priority runnable thread using a bitmap. The HW scheduler uses the hardware acceleration features described in Section 3.2 to improve IPC performance. Receiver priorities were tested in the range of 250 to 0, however, no change in trends were observed beyond priority 220 so some measurements have been excluded for the purpose of clarity. The results of this benchmark are presented in Figure 3.

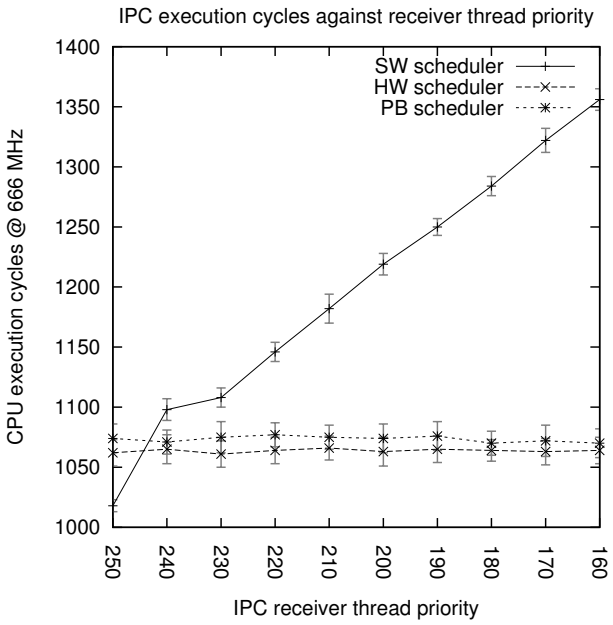

Figure 3: IPC execution cycles versus receiver thread priority.

We can see that the SW scheduler performance degrades linearly as the receiver thread priority decreases. This is because the scheduler traverses the *ksReadyQueues* from highest priority to lowest priority until it finds a runnable thread. The lower the priority of the receiver, the more entries the scheduler must examine before it finds the waiting receiver. This behaviour also increases the cache footprint of the scheduler. In the case of a receiver thread of priority 0, the scheduler reads from 256 queue heads. Because head and tail pointers are interleaved, this will result in 512 words (2 KB) being loaded into the cache. Another way to look at this is that the scheduling behaviour will result in 2 KB of data being evicted from the cache in the worst case. This evicted data may be data that is frequently used by an application; system performance may therefore be further degraded because this data must now be reloaded from high-latency main memory when the application is next scheduled.

The PB scheduler addresses this issue by extending the SW scheduler to include a hierarchical bitmap representation of the non-empty *ksReadyQueues*. This optimisation leads to an O(1) lookup complexity as can be seen by the results in Figure 3. From the results we also see that the legacy scheduler actually outperforms both the PB scheduler and the HW scheduler when a very high priority (250) receiver thread is targeted. This condition provides the best case

performance for the legacy scheduler because the iterative search need only examine a few priority levels before the highest priority runnable thread is located. The performance degradation in the PB scheduler reflects the additional operations required in the traversal and maintenance of the bitmap. The PB scheduler requires 3 additional reads to locate the highest priority thread. It must first read the domain bitmap which is not used in our configuration. Next it must read from each of the 2 remaining levels of the bitmap hierarchy. Once the thread has been identified, it will be marked as active and removed from the scheduling queue. Because this thread is the only runnable thread in the system, the bitmap scheduler must additionally mark the *ksReadyQueue* as empty in the second level, mark the priority group as empty in the first level and finally mark the domain as empty at the top level. For the sake of abstraction, the scheduler can not simply write 0 to these words, rather, it must perform the correct bit operation to clear only the relevant bit at each level. The result is that the bitmap scheduler must perform another 3 additional reads and 3 additional writes to update the bitmap, however, it is likely that these 6 memory accesses will operate on memory in the cache rather than suffering a penalty from loading data from main memory a second time.

Both the legacy scheduler and the PB scheduler require the manipulation of a doubly-linked list in order to remove a thread from the *ksReadyQueues*. The hardware scheduler, on the other hand, requires only one read operation to both identify and remove the highest priority thread from the *ksReadyQueues*, yet the performance curve shows similar results to that of the bitmap scheduler. The reason for this is the additional time required to access the off-core acceleration hardware.

Figure 3 shows an anomaly when the priority of the IPC receiver is 240. This feature was investigated by increasing the resolution of the benchmark as shown in Figure 4. The figure shows that the anomaly is reproducible and has a peak when the IPC receiver thread is at a priority of 240. The cause of this anomaly is currently still under investigation.

The *sel4bench* benchmark was also run with various AXI bus clock periods in order to find the relationship between the hardware access latency and HW scheduler performance. In this case, we examine only that benchmark which provides the best case performance of the legacy scheduler. This allows us to determine the required AXI bus clock frequency to achieve improved performance in all cases. Recall that IPC between two threads of the same priority is handled by the *fastpath* and hence that the software scheduler would not be invoked in this case. The best case performance of the legacy scheduler occurs when the scheduler need only examine the *ksReadyQueue* for priority 255 and 254 before it finds a runnable thread. For this reason, the chosen benchmark is an IPC from a thread of priority 255 to a thread of priority 254. The results obtained are presented in Figure 5.

The results present two distinct slopes as shown by the grey lines in Figure 5. The change in slope occurs between an AXI clock period of 45ns (22MHz) and 35ns (29MHz). By taking the slope of these curves and dividing by the CPU frequency, we can find the effective overhead of the AXI transaction,

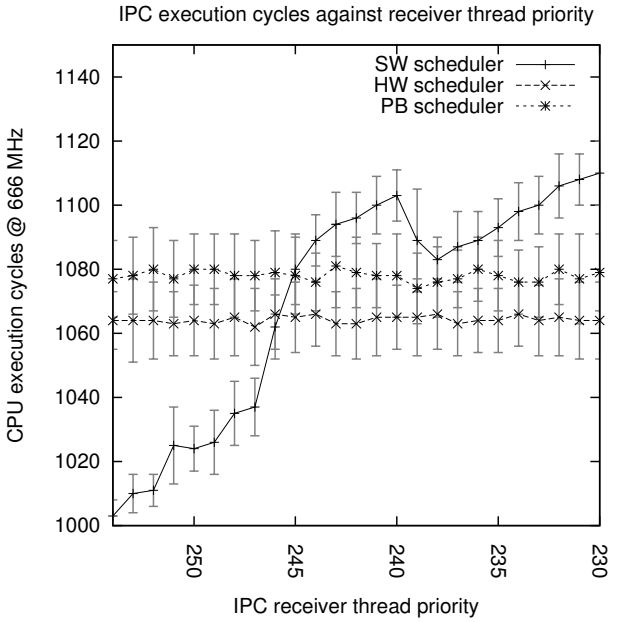

Figure 4: IPC execution cycles versus receiver thread priorities with focus on priority 240.

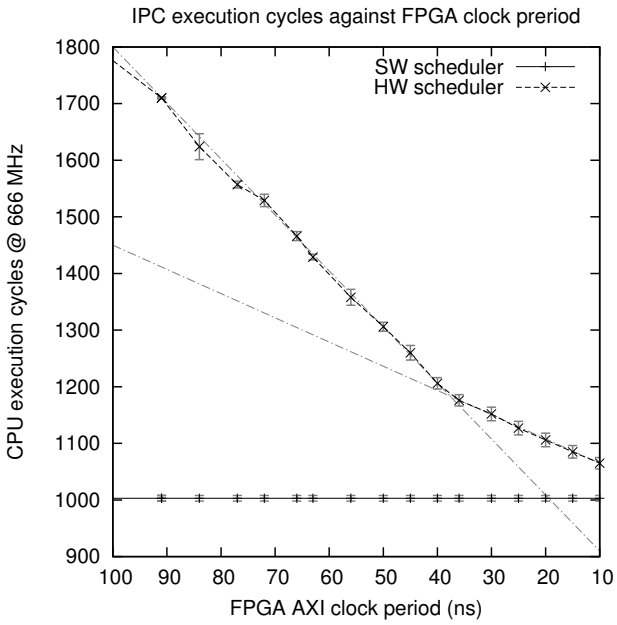

Figure 5: IPC execution cycles versus FPGA AXI clock period for an IPC from priority 255 to priority 254.

in terms of AXI clock cycles, for the HW scheduler. Below 22MHz, the AXI transaction is the bottleneck of system performance. The cost of accessing the hardware in this case is 15 AXI clock cycles. Above 29MHz, CPU execution begins to influence the performance curve and the AXI transaction decreases to 6.5 AXI clock cycles.

If we assume that the observed trend continues, we find that the number of execution cycles required for an IPC approaches 1020 as the clock period approaches 0. This suggests that, although we come close, the GP AXI-based HW scheduler solution cannot outperform the software implementation of the scheduler in all cases. If the application developer designs the system using only threads of a high priority, a GP AXI based hardware accelerated scheduler will reduce system performance.

5. FUTURE WORK

In our evaluation, we show that the acceleration of the seL4 kernel scheduler in hardware can improve the IPC performance of the system under most (but not all) operating conditions. However, problems were encountered when evaluating the performance of the bitmap scheduler. The implementation of the bitmap scheduler had to be back-ported to the version of the kernel on which the hardware scheduler was built. This was because the legacy scheduler were observed to suffer from a decrease in performance and also an increase in variance, and the hardware scheduler was observed to outperform the legacy scheduler under all operating conditions. The increase in variance suggests that the issue is related to nondeterministic acceleration features of the CPU. For example, the position of key data structures in memory may have trivially changed and led to a non-ideal cache layout. With this new layout, the cache footprint may have increased and forced the kernel to access high-latency main memory more frequently. One avenue for future work is the investigation into the cause of this degradation in performance to evaluate the sensitivity of performance to trivial changes in software source code. We may find that the robustness of custom hardware accelerator performance may outweigh the development and maintenance effort involved in software optimisations that exploit the acceleration features of the CPU.

A second outcome of our evaluation is a better understanding of the limitations of the chosen GP AXI port for interfacing between the CPU and programmable logic. Although this method allows the CPU to modify the scheduling queue with the execution of a single instruction, the cost of this access is sufficient for an optimised software solution to yield similar results. The Accelerator Coherency Port (ACP) provides cache-coherent access to memory such that the CPU could collect results from a hardware acceleration task directly from the cache. The Zedboard provides an ACP slave port but it does not provide an ACP master port. Unfortunatey, this means that we cannot signal to hardware that it is required to perform some function in the same way that was possible with a GP AXI slave port. A second avenue for future work is to investigate better low-latency signalling mechanisms between CPU and programmable logic.

Additionally, the ability to access the cache directly via the ACP port provides an opportunity for the use of cache warming to reduce variation in execution time. When the kernel is invoked, the accelerator could walk key areas of memory to ensure that they reside in the cache before they are required by the software system. This feature is already provided on the Zynq AP Soc by the PL310 cache controller [18], however, this feature operates on speculative cache line fills that are heuristically determined by observed memory access patterns. With insight into the operation and memory layout of a software system, a custom cache warmer could more accurately predict memory access patterns and more effectively ensure that key data is cache-resident before the memory is requested.

6. CONCLUSION

In this paper we presented and evaluated an approach to operating system software acceleration in programmable logic. This technique is only applicable to heterogeneous systems that provide programmable logic because the system must be flexible enough to support the implementation of custom policies of operation to suit a wide range of target applications.

We examined the seL4 kernel scheduler as a candidate for acceleration and found that the execution time of an IPC can be improved in the majority of cases. For cases where execution time is not improved, we examined the potential for improvement and found that a GP AXI acceleration peripheral cannot outperform the software solution in all cases. This is due to the latency of hardware access while a software solution has the benefit of operating on a low-latency cache.

We examined the performance results of a software scheduler extension that optimises performance with an improved O(1) algorithm and observed a similar performance curve to that of the hardware accelerated scheduler. These results came as a surprise because the hardware scheduler requires only one load instruction to identify and remove the highest runnable thread from the scheduler while the software implementation requires a total of 6 read operations and 3 write operations for the maintenance of scheduler data structures.

We found that performance penalties scale linearly with peripheral clock period, however, CPU acceleration features are able to reduce this latency from about 15 AXI clock cycles to 6.5 AXI clock cycles when the clock frequency of the bus is above 29MHz.

From the results presented in this paper, we conclude that the future of hardware acceleration of an operating system kernel depends heavily on the choice of communication between the CPU core and the programmable logic. The GP AXI slave interface is not suitable for applications such as a kernel scheduler, however, it may be more suited to longer running kernel operations such as resource management. We have identified alternative communication approaches that may improve results for the kernel scheduler for future work.

7. REFERENCES

[1] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid

FPGA-CPU computational components: A missing link. *Micro, IEEE*, 24(4):42–53, July 2004.

[2] ARM. *ARMv7-A Architecture Reference Manual DDI 0406C.b*, 2005.

[3] ARM. *AMBA®AXI™and ACE™Protocol Specification IHI 0022D (ID102711)*, 2011.

[4] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 339–348, Nov 2011.

[5] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[6] J. Dahlstrom and S. Taylor. Migrating an OS scheduler into tightly coupled FPGA logic to increase attacker workload. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 986–991, Nov 2013.

[7] E. Dodiu and V. Gaitan. Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – Concept and theory of operation. In *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, pages 1–5, May 2012.

[8] A. Garg. Real-time linux kernel scheduler. *Linux J.*, 2009(184), Aug. 2009.

[9] N. Gupta, S. Mandal, J. Malave, A. Mandal, and R. Mahapatra. A hardware scheduler for real time multiprocessor system on chip. In *VLSI Design, 2010. VLSID '10. 23rd International Conference on*, pages 264–269, Jan 2010.

[10] M. Huang, K. Lim, and J. Cong. A scalable, high-performance customized priority queue. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4, Sept 2014.

[11] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *Networking, IEEE/ACM Transactions on*, 15(2):450–461, April 2007.

[12] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[13] P. Kuacharoen, M. A. Shalan, and V. J. M. III. A configurable hardware scheduler for real-time systems. In *In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[14] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari. An experimental comparison of different real-time schedulers on multicore systems. *J. Syst. Softw.*, 85(10):2405–2416, Oct. 2012.

[15] V. Mooney and D. Blough. A hardware-software real-time operating system framework for SoCs. *Design Test of Computers, IEEE*, 19(6):44–51, Nov 2002.

[16] A. C. Nácul, F. Regazzoni, and M. Lajolo. Hardware scheduling support in SMP architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 642–647, San Jose, CA, USA, 2007. EDA Consortium.

[17] S. E. Ong, S. C. Lee, N. Ali, and F. Hussin. SEOS: Hardware implementation of real-time operating system for adaptability. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 612–616, Dec 2013.

[18] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.9.1)*, 2014.