Efficient fine-grained processor-logic interactions on the cache-coherent Zynq platform

ALEXANDER KROH and OLIVER DIESSEL, University of New South Wales, Australia

The introduction of cache-coherent processor-logic interconnects in CPU-FPGA platforms promises low-latency communication between CPU and FPGA fabrics. This reduced latency improves the performance of heterogeneous systems implemented on such devices and gives rise to new software architectures that can better use the available hardware.

Via an extended study accelerating the software task scheduler of a microkernel operating system, this paper reports on the potential for accelerating applications that exhibit fine-grained interactions. In doing so, we evaluate the performance of direct and cache-coherent communication methods for applications that involve frequent, low-bandwidth transactions between CPU and programmable logic.

In the specific case we studied, we found that replacing a highly optimised software implementation of the task scheduler with an FPGA-based scheduler reduces the cost of communication between two software threads by 5.5%. We also found that, while hardware acceleration reduces cache footprint, we still observe execution time variability because of other non-deterministic features of the CPU.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Hardware → Hardware accelerators**;

Additional Key Words and Phrases: Reconfigurable Computing, Cache-coherent interconnect

ACM Reference Format:
Alexander Kroh and Oliver Diessel. 2018. Efficient fine-grained processor-logic interactions on the cache-coherent Zynq platform. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2018), 22 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The emergence of tightly-coupled CPU-FPGA platforms is challenging the traditional model of offloading a large computational workload from the CPU. Such platforms provide cacheable data access to shared main memory that is coherent with that of the CPU. For the programmable logic part of the device, such systems offer low-latency data access to main memory that is similar in latency to memory local to an FPGA accelerator card or module. The need for bulk transfers of data from main memory to FPGA-local memory to improve data locality is thereby reduced. Additionally, CPU cache-coherent data access can further reduce latency and programming complexity.

While these CPU-offload accelerators may still benefit from the improved communication performance of tightly-coupled, cache-coherent platforms, the provided low-latency data access allows for a more co-operative hardware/software programming model. If the bulk transfer of data can be replaced by fine-grained, low-latency interactions, a task can be split into small subtasks that are better suited to either software or programmable logic.

Authors' address: Alexander Kroh, alex.kroh@unsw.edu.au; Oliver Diessel, o.diessel@unsw.edu.au, University of New South Wales, School of Computer Science and Engineering, Sydney, NSW, 2052, Australia.

1936-7406/2018/1-ART1 $$15.00
https://doi.org/0000001.0000001

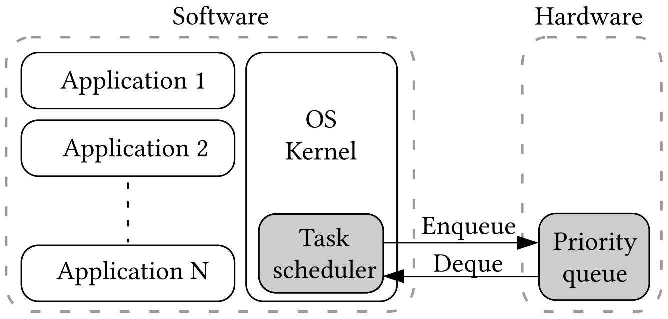


Fig. 1. Accelerated OS kernel scheduler architecture.

When a fine-grained co-operative system is CPU-bound, the completion time for a task is solely determined by the time required for the CPU subtasks to complete. In this case, although communication latency increases the response time of a hardware subtask, it is the communication overhead, as counted in CPU execution cycles, that impacts the overall execution time of the task. This overhead comes from data marshalling, the execution of data transfer and synchronisation instructions as well as data dependencies on high-latency sources. The communication overhead observed by the CPU must therefore be carefully balanced with the CPU-FPGA communication latency.

Anticipating the difficultly of accelerating short-running tasks, the study presented in this paper was primarily motivated by a desire to determine if migrating software functions to programmable logic can reduce the execution time jitter of a software task. Reduced jitter is particularly important for real-time systems, which require a known bound on execution time. Large jitter leads to pessimism in the worst case execution time (WCET) – a metric that determines if an external event can achieve a timely response.

Jitter is caused by the non-deterministic nature of execution on modern superscalar processors. Instruction throughput enhancing features, such as the cache hierarchy and branch predictor, reduce the amount of time that the CPU spends idle. This is typically done by predicting program and data flow with stochastic heuristics. For example, the instruction and data caches are typically configured for random cache-line replacement when new data requires space in the cache. This random replacement causes jitter in execution time as the choice of cache line determines future data access completion times. When software functions are implemented in hardware, the associated code and data no longer need to occupy the cache. This improves the likelihood that critical OS or application code stays resident in the cache.

In this paper, we evaluate the ability of the tightly-coupled Xilinx Zynq®-7000 series All Programmable System on Chip to support fine-grained interactions. The target for our case study is the software task scheduler of a microkernel Operating System (OS). Fine-grained interaction in this case involves the insertion and removal of a software thread handle from a priority queue that is implemented in hardware (Figure 1). Communication between the kernel software and the hardware-resident task queue involves the transfer of a single 32-bit word in both cases.

The task scheduler is the primary function of all OS. Not only is the scheduler invoked periodically to ensure CPU time-sharing between threads, it is also invoked on demand to facilitate communication or synchronisation between two threads, and when scheduling high-priority tasks in response to critical external events.

A large amount of effort is invested into the optimisation of the scheduler. The scheduler is a very frequently executed function of the OS and is a key element contributing to the WCET of event handling. When a critical event arrives, the appropriate event handler must be scheduled for execution. The handling of the event may also require communication with other software tasks in order to achieve the required goal.

Given that the OS scheduler is highly optimised and involves operations that are very short in nature, we expect it to be challenging to gain any improvement in execution time through hardware acceleration. The choice of this case study motivates us to optimise the communication methods used to the best of our ability. By careful selection of fine-grained communication methods between software and hardware, we have been able to achieve a 5.5% reduction in execution time for synchronous communication between two threads.

Hardware acceleration also provided a significant improvement in execution time variance. The migration of the software task scheduler into hardware removed most sources of non-deterministic execution time and reduced execution time variance by 58%. Unfortunately, further improvement was limited by the dominating influence of the branch predictor of the CPU.

This paper is organised as follows: Section 2 highlights related work on accelerating Operating System kernel schedulers. Section 3 describes both the software and hardware architectures of the OS kernel scheduler that we used in our study. Our experiment setup and evaluation of execution time and jitter is presented in Section 4. Future work is outlined in Section 5, while conclusions are drawn in Section 6.

2 RELATED WORK

Prior work has studied the migration of an OS kernel scheduler from software to hardware, primarily as a means to improve execution time and jitter in real-time systems. However, prior work has generally been limited to either soft-core systems [6, 15], simulated hardware [10, 11, 14] or loosely-coupled systems [5, 13]. To the best of our knowledge, the performance benefits of this migration has not been evaluated using low-latency, cache-coherent communication with fixed-core processors.

Ong et al. augmented a soft-core NIOS-II processor with a hardware-accelerated task scheduler. The accelerator was connected to the system interconnect bus and additionally provided a periodic timer and processor interrupt. The authors observed a 72% improvement in inter-task communication execution time and a reduction in interrupt request (IRQ) handler jitter from 25.4% to 1.59% [15]. Although these improvements are significant, the use of a soft- rather than fixed-core processor penalises the CPU in the evaluation. It is well known that fixed-circuits have higher maximum operating frequencies when compared to programmable logic.

Hardware assisted scheduling has shown promise in symmetric multi-core architectures. Using cycle-accurate simulation, Necul et al. showed that hardware-assisted scheduling can reduce the context switch time between two threads from 10,000 CPU cycles to 947 CPU cycles [14]. While the scheduler chooses the next thread for execution, it has no access to the CPU register file: saving and restoring CPU state must be done in software. The proposed system uses dedicated ports on an ARM926EJ-S processor for communication between the CPU and the scheduler, implemented in hardware. The authors measured throughput improvements in graphic filtering and network packet processing applications to be 46× and 10×, respectively.

Mooney et al. proposed a modular OS framework [13]. In their work, the system engineer can choose between hardware or software equivalent implementations for OS subsystems. Key subsystems include dynamic memory management, locking, and deadlock detection. Hardware-assisted locking aims to improve both execution time and the predictability of lock access times. By moving locking to hardware, deadlock detection improves system safety without significant run-time overheads. Simulated hardware experiments showed that hardware-assisted OS functions can provide speed-ups of 27% or more for database applications.

Hardware schedulers can be provided as programmable logic [10], or as dedicated circuits within an Application Specific Integrated Circuit (ASIC) [14]. While ASIC implementations provide fixed scheduling policies, programmable logic allows for flexible, application-specific scheduling policies that can be swapped on-demand.

The deployment of hardware schedulers has been explored as both independent coprocessors and integrated CPU features. In the latter case, the scheduler has direct access to the execution pipeline and register file of the CPU. This allows the scheduler to swap the entire thread context in a single cycle without degrading CPU pipeline performance [6]. Such an architecture requires modification of the CPU itself. Modern systems on chip (SoC), such as the Xilinx Zynq and Intel Cyclone V, offer programmable logic and a high-performance general purpose fixed-core processor on a single die, but these systems do not provide direct access to the CPU register file.

Prior work has implemented a hardware-accelerated software task scheduler on the Zynq SoC [5]. The focus of that work was on obscuring the view of thread context from other threads by storing it in programmable logic. The study showed up to 50% speedup (1500 CPU cycles), even though the entire thread context (68 bytes) had to be transferred between CPU and accelerator on every context switch. Although that work used a tightly-coupled CPU-FPGA system, it did not explore the use of the low-latency, cache-coherent communication that such coupling provides.

3 SYSTEM ARCHITECTURE

In our work, we decided to investigate accelerating the seL4 microkernel [9]. seL4 kernel operations are very short in nature and difficult to accelerate using the traditional CPU-offload model. Additionally, the kernel is the central gateway for all application resource management and communication: the performance of the kernel impacts all hosted applications.

seL4 has a complete WCET analysis [4, 16] and has recently been extended for RT applications [12]. seL4 is considered to be a microkernel because operating system services and device drivers are implemented as user-space applications rather than being provided directly by the kernel. A key advantage of this approach is that only a small amount of software 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 and are isolated by the memory management unit (MMU) hardware of the CPU. Because of this isolation, inter-process communication (IPC) is used frequently to communicate between tasks.

We decided to attempt to accelerate the kernel scheduler because it is a very frequently used function of the kernel. Although the scheduler is not a long-running operation, the performance of the scheduler is critical to IRQ handler latency and efficient IPC between client-server application software. Once the system has been initialised, the kernel provides 3 key functions, all of which can result in an invocation of the kernel scheduler.

(1) *IRQ delivery.* When the kernel receives an IRQ exception, the kernel unblocks any thread that is waiting for the IRQ. If the unblocked thread is of a higher priority than the currently active thread, the kernel must reinsert the active thread into the scheduling queue and replace it with the new highest priority runnable thread.
(2) *Preemption IRQ.* The preemption IRQ ensures the fair sharing of CPU time between threads of the same scheduling priority. When the preemption IRQ arrives, the kernel reinserts the current thread into the scheduling queue and replaces it with the next runnable thread in round-robin order.

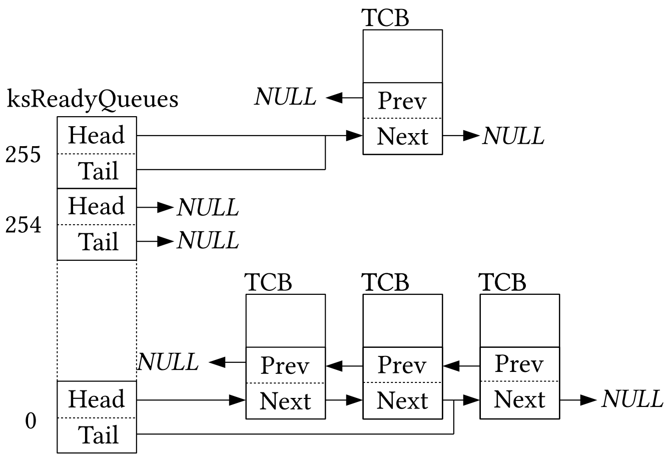


Fig. 2. Software architecture of legacy scheduler.

(3) *Inter-Process Communication.* IPC is a primitive for data transfer and synchronisation between threads. When a thread sends an IPC request to another thread, the kernel blocks the sender and inserts the receiving thread into the scheduling queue. The scheduler is then invoked to choose a new thread for execution.

seL4 provides a *fixed-priority preemptive scheduler* such that a thread never executes while a runnable thread of higher priority exists in the system. It has been carefully tuned for low-latency IPC through software optimisations that consider both the number of instructions executed, data locality and cache footprint. We therefore expect it to be challenging to gain a benefit from hardware acceleration.

3.1 Software scheduler

The set of runnable threads in the seL4 microkernel is implemented as a doubly-linked list with one list for each thread priority (Figure 2). The kernel maintains the *next* and *previous* pointers of this list along with the thread context as part of the Thread Control Block (TCB) of each thread. The kernel appends a thread to the tail 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 kernel records the remaining execution time of the thread and adds the thread to the head of its associated list, rather than the tail. The kernel maintains a set of head and tail pointers for each priority in a global structure known as the *ksReadyQueues*. When the kernel invokes the scheduler, it walks 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 removes the thread at the head of this list and marks it as the active thread. If no runnable thread is found in the system, the kernel schedules an implicit *idle* thread until an external IRQ causes a waiting thread to become runnable again.

The seL4 kernel implements a *fastpath*, a hand-optimised path for common operating system calls. The *fastpath* allows IPC from a low-priority sender thread to a higher- or equal-priority receiver thread to complete without costly scheduler invocations. Under these conditions, because the kernel implements a *fixed-priority preemptive scheduler*, the kernel knows that there is no runnable thread with a higher priority than the IPC sending thread. If the receiver is blocked waiting for this IPC and is of a higher- or equal-priority to the sender, the kernel knows that the receiver will be the new highest priority runnable thread in the system. For this reason, the sender can be blocked and the receiver can immediately become the new active thread without invoking the task scheduler of the kernel.

Since the commencement of this work, the software architecture of the kernel scheduler was further optimised by supplementing the design with a two-level bitmap lookup. Each bit in the second level bitmap 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 associated priority. In the same way, the first-level bitmap reflects the presence of a runnable thread in each group of 32 priorities. The scheduler uses the single-cycle Count Leading Zeroes *CLZ* instruction on the first level bitmap and maps the result to the appropriate second level bitmap. The scheduler repeats this process on the second level bitmap to find the highest priority at which a runnable thread can be found. Finding the highest-priority runnable thread thus becomes a constant time operation, irrespective of its location within the *ksReadyQueues*.

3.2 Accelerator design

The hardware design must provide the same features as the software design to ensure compatibility. It must allow a thread to be inserted at both the head and tail of a selected *ksReadyQueue* and allow a thread to be removed from the head of the highest priority non-empty *ksReadyQueue*.

A simple structure that satisfies these design goals is a priority queue. While much research has been undertaken on priority queue implementations [7, 8], the behaviour of insertions with equal priority are generally ill-defined. In our case, it is important that threads of equal priority are removed from the priority queue in the order in which they were inserted. Since our research is not focussed on priority queue hardware design and implementation, we used a trivial implementation to explore our ideas.

We used a hardware architecture that closely follows that of the software architecture for this study (Figure 3). We replaced the *ksReadyQueues* by FIFOs, where FIFO data represents a reference to the TCB of a thread in main memory. By transferring only the location of the TCB in main memory, we reduced the throughput requirement for priority queue transactions. This also preserved the ability of the CPU to optimise reads and writes to cacheable global memory when accessing thread context.

The *H* signal of the priority queue allows the scheduler to add a thread to either the head or the tail of a FIFO. Write enable (*WE*) and read enable (*RE*) signals control the addition (push) or removal (pop) of a FIFO entry. If the scheduler asserts neither *WE* nor *RE*, a read operation returns the appropriate entry from a FIFO without removal. The scheduler uses the *SEL* signal to select a specific priority (FIFO) for the priority queue operation. Each FIFO also provides an *E* signal, which indicates if 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 updates to reflect this change before the next rising edge of the subsystem clock. The priority encoder allows the scheduler to both identify and remove the highest priority runnable thread from the set of runnable threads in a single clock cycle. With the *P* signal asserted, the priority queue ignores the *SEL* signal and uses the priority encoder output in its place to select the highest priority non-empty FIFO as the target of the transaction.

The hardware implementation provides acceleration by offloading the task of manipulating the priority queue from software. The software implementation of the *ksReadyQueues* requires that software maintains a doubly-linked list of threads for each priority. Our hardware implementation relieves the burden of list maintenance from software by allowing software to manipulate the head

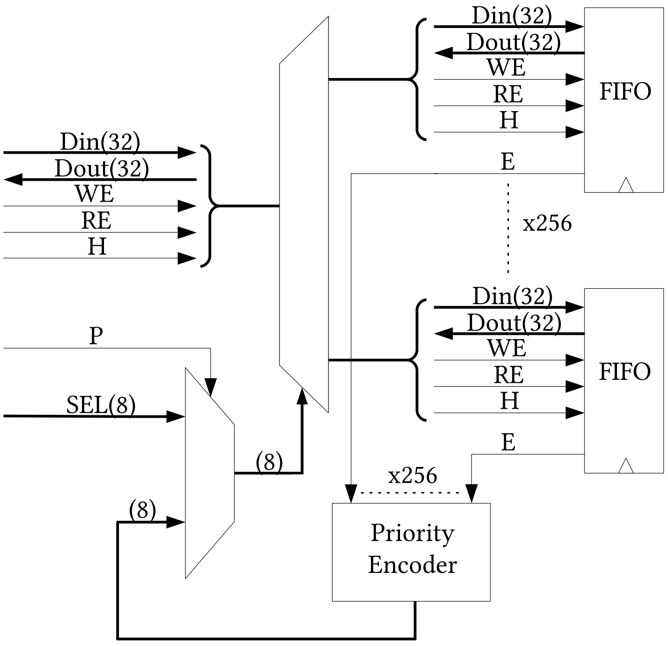


Fig. 3. Hardware architecture of the priority queue.

or tail of a *ksReadyQueue* with a single transaction to the accelerator. Additionally, the highest priority thread in the *ksReadyQueues* can be requested and removed from the schedule in a single transaction. This eliminates the need to search the scheduling queue or maintain a hierarchy of bitmaps.

We acknowledge that the use of fixed-size FIFO components presents scalability concerns in the design, however, more scalable priority queue implementations can also be supported. One must only ensure that the number of clock cycles required to update the highest priority runnable thread is less than the number of clock cycles between scheduler transactions. Alternatively, FIFO content can be stored in off-chip RAM with high-priority runnable threads cached in block RAM for low-latency access.

3.3 Target system hardware

We used Avnet's Zedboard to evaluate our design. The Zedboard is a low-cost development platform that features the XC7Z020 Zynq SoC. Zynq provides dual ARM Cortex-A9 application processors and on-chip programmable logic. Communication between the ARM cores and the programmable logic is achieved via a range of ARM AXI communication buses [2]:

- **GP AXI3**: The General Purpose (GP) AXI port offers 32-bit data transfers with bulk transfer sizes up to 64 bytes. Zynq provides the GP port in both master and slave interface configurations. The master is always responsible for initiating communication over the AXI bus and additionally provides the address of the transaction.
- **HP AXI3**: The High Performance (HP) AXI port offers 64-bit data transfers with bulk transfer sizes up to 128 bytes. Zynq provides the HP port as a slave interface only: the programmable logic is the master and is responsible for initiating all transfers. The HP port provides improved throughput over the GP slave equivalent due to the increased bus width.

11	10	9	8	7	6	5	4	3	2	1	0
P	SEL								H	0	0

Fig. 4. Address mapping of GP accelerator peripheral.

- **ACP AXI3**: Like the HP port, the Accelerator Coherency Port provides a slave interface port that offers 64-bit data transfers with bulk transfer sizes up to 128 bytes. Unlike the HP port, transfers can optionally be cache-coherent with the CPU.

In our study, we considered only GP AXI3 master and ACP AXI3 slave interfaces. The GP master port is the only port that provides direct communication between the CPU and the accelerator. The ACP provides the best performing shared memory interface due to its coherency with the CPU cache.

Communication between CPU and accelerator has three components:

(1) A **command** must be transferred to the accelerator to communicate the desired action. An example of such a command is to push a thread to the head of a particular scheduling queue.
(2) **Data** must be transferred to or from the accelerator. In our case, this data represents a reference to a thread in main memory.
(3) A **signalling** mechanism is needed to inform the accelerator that a new command is available and that a response is expected.

If we connect the accelerator as a slave peripheral on the GP AXI bus, signalling is a bi-product of the AXI handshaking protocol during a transfer. If we connect the accelerator as a master peripheral, communication is via shared memory and we must provide some other method for signalling. We can use a second GP master port for this purpose, or we can use the *EVENT_EVENTO* signalling method. Software can assert a single wire in the CPU for exactly one CPU clock cycle by executing the *Send Event* (*SEV*) instruction. This signal is generally used to signal an event to other embedded processor cores. However, the programmable logic can observe a toggled variant of this signal. The processor toggles the *EVENT_EVENTO* signal for each execution of the *SEV* instruction.

The following subsections describe the connection of the accelerator to the GP AXI master port and the ACP AXI slave port.

3.4 GP AXI accelerator

We designed the accelerator as a slave peripheral with the CPU communicating directly with the accelerator through memory-mapped IO. The accelerator decodes the address that is provided by the CPU to produce a command for the scheduler transaction as shown in Figure 4. Bits 0 and 1 of the address are reserved for word alignment, bit 2 is mapped to the *H* signal, which selects between the head or tail of the queue, and bits 3 through 10 are mapped to the *SEL* signal, which selects the priority for the transaction. Additionally, bit 11 is mapped to the *P* signal, which instructs the accelerator to ignore bits 3 through 10 and instead select the highest priority non-empty queue for the transaction. We developed an AXI adapter component to translate the complex AXI handshake protocol into a simple *read enable* (*RE*) and *write enable* (*WE*) signalling mechanism (Figure 5).

Software inserts a thread into the schedule at a specific priority by initiating a single write to memory. A handle to the thread is transferred as the data portion of the transfer, while the desired priority is encoded in the address portion of the transfer. Similarly, software removes a thread from the schedule by initiating a single read request from a specific memory address. A single read transaction can also be used to both identify and remove the highest priority runnable thread by executing a read instruction with bit 11 of the source address set.

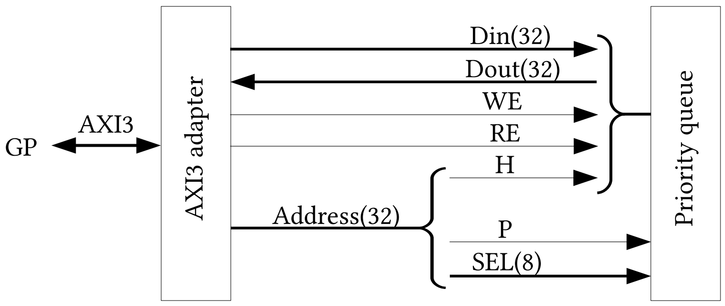


Fig. 5. Hardware architecture of GP AXI hardware scheduler communication.

Operation	31	30	29	28-10	9 - 8	7 - 0
Enqueue	0	0	H	&Thread	0	SEL
Dequeue	1	0	H	0	0	SEL
Dequeue (highest priority)	1	0	0	0	0	0

Fig. 6. Command and data encoding for ACP-based scheduler accelerator operations.

We clocked the hardware implementation of the *ksReadyQueues* directly from the GP AXI bus clock at 100MHz.

3.5 ACP AXI accelerator

Zynq provides the ACP only as a slave port. This means that the accelerator must be implemented as a master peripheral and initiate all transfers to the slave device. The slave device in this case is the Snoop Control Unit (SCU), which connects each ARM CPU to the system memory hierarchy and maintains cache-coherence between the connected masters and ACP-connected peripherals [3]. When the accelerator performs a read transaction, the SCU can retrieve the appropriate data from the shared L2 cache or the L1 cache of any connected CPU. When the accelerator performs a write transaction, the SCU writes the appropriate data into the L2 cache and invalidates any corresponding L1 cache lines in all CPUs.

Since both CPU and accelerator operate in master mode when communicating with the slave SCU, communication must be achieved indirectly through shared memory. The ACP bus is 64 bits wide: two words can be transferred per clock cycle. We assigned one word to represent the handle to the thread and encoded the desired command into the unused bits. The constraints of a seL4 thread object enforce an alignment of 10 bits and accessibility within the *kernel window* (0xE0000000 to 0xFFFFFFFF). These constraints provided 12 bits for command encoding (Figure 6).

The shared memory approach prevented us from using the AXI handshaking protocol to signal the accelerator when a new command was provided. Instead, we used the *SEV* processor instruction, which the accelerator detects as a state toggle of the *EVENT_EVENTO* wire in programmable logic. The transfer of the signal and data in this case was decoupled; we had to ensure that the data was observable by the accelerator before the signal of its presence arrived. We used the Data Synchronisation Barrier *DSB* processor instruction to ensure that any *store* operation had completed before the *SEV* instruction was executed.

In hardware, we extended the priority queue implementation to include a trivial finite state machine (Figure 7). After the processor signals the accelerator, the accelerator reads the provided command and data from shared memory by issuing a 64-bit read transaction on the ACP AXI bus.

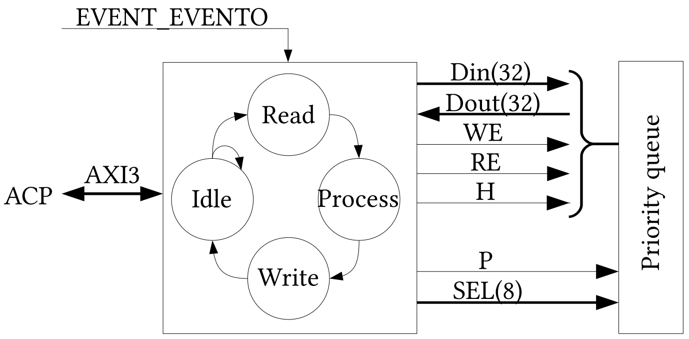


Fig. 7. Hardware architecture of ACP AXI hardware scheduler communication.

The data portion of the transfer is mapped directly to the data input of the priority queue while the command is decoded and routed to the relevant control wires. The priority queue operation is completed by the next clock cycle. The accelerator then sets the priority queue control signals such that the highest priority thread is reported. Finally, the accelerator issues a write transaction to the ACP. This write clears the command that was provided by software and reports the highest priority runnable thread through shared memory.

Once a scheduler transaction has completed any command, the first word in the 64-bit shared memory region is cleared to signal completion and the second word is speculatively filled with the highest priority thread in the schedule. In this way, the CPU can retrieve the highest priority thread from the low-latency cache at any time and then issue the appropriate command to remove this thread from the schedule. By always storing the highest priority runnable thread at a predetermined location, we make it possible for the CPU to retrieve this thread from the low-latency cache as soon as it is required. The CPU thus continues program execution while the accelerator processes the request and replaces the highest priority runnable thread in preparation for the next scheduling event.

We connected the ACP AXI accelerator described above directly to the ACP AXI3 slave port within the Zynq SoC. Each transaction requires both a read phase (to retrieve the provided command) and a write phase (to signal completion and update the highest priority runnable thread). Although this extends the time for the accelerator to complete a transaction, communication latency is reduced since data can be transferred asynchronously through the low-latency cache.

When deploying the described method of communication for multiple accelerators, one may find contention on the ACP as all ACP accelerators detect the *EVENT_EVENTO* signal at the same time. It may be beneficial to reserve another location in memory for the use of a bitmap to extend the resolution of the *EVENT_EVENTO* signal. Each bit of the bitmap would correspond to a signal for each accelerator. The CPU should then set the appropriate bit before executing the SEV instruction. A monitor, implemented in programmable logic, can then read the bitmap via the ACP and forward the signal to only those accelerators for which the corresponding bit is set. Although this adds an AXI transaction to the communication path, it reduces the total number of AXI transactions that will be issued by limiting the number of accelerators that respond to the signal.

ACP sharing introduces completion time variance as the transactions of one accelerator may delay the transactions of others. This variance can be controlled by assigning a high priority to transactions that originate from timing-critical accelerators. Transaction priorities are supported by the AXI bus specification and are enforced by the arbitration policy of the interconnects between the accelerators and shared memory.

3.6 Comparison of accelerator structures

The GP-connected accelerator is connected to the CPU via a single GP AXI3 port. In this arrangement, the accelerator is unable to initiate transfer to either CPU or shared memory because it is a slave peripheral on the connected bus. Each transaction from the CPU to the accelerator is a self-contained query, insertion or removal operation to the priority queue.

The cache-coherent ACP-connected accelerator contains the same functional logic as the GP-connected accelerator, but uses an alternative interfacing approach. This accelerator includes a state machine to manage accelerator-mastered transactions to cache-coherent shared memory via the ACP. Each transaction has two phases: The accelerator first uses a read transaction to pull the request from shared memory. Once the priority queue operation has been processed, the accelerator pushes the result back to shared memory.

Both the ACP and HP ports use the AXI3 communication protocol and provide a 64-bit wide data bus. The difference between these interfaces is the point at which the accelerator is connected to the memory hierarchy.

If cache-coherent communication is not required/desired, the ACP-connected accelerator can be connected to the HP port without modification. However, software must then manage data coherency with the accelerator. This is done by either configuring shared memory access to bypass the CPU cache, or by executing fine-grained cache maintenance instructions on the CPU for each transaction. In the latter case, the CPU must explicitly clean the appropriate cache line to ensure that commands are visible to the accelerator and invalidate this cache line when reading the result. While both methods increase the shared memory access latency of both CPU and accelerator, the latter approach also requires additional CPU cycles to maintain data coherency. Due to these additional overheads, neither method is further considered in this work.

4 EVALUATION

The seL4 kernel invokes the scheduler under four conditions:

(1) When a thread sends an IPC to a thread of lower priority;
(2) When an IRQ exception is received and the registered handler thread is of a higher priority than the current thread;
(3) When a thread becomes blocked waiting for an IPC or IRQ event; and
(4) When the periodic timer tick arrives to signal that the current thread has exhausted its execution time slice and another thread of the same priority should be scheduled. Threads of lower priority must wait for higher priority threads to become blocked before they are scheduled.

In the case of both sending an IPC and preemption timer tick, the scheduler issues both an insertion and a removal transaction to the priority queue. Evaluating the performance improvement under both of these cases is redundant and hence we evaluated only the IPC execution time and IRQ latency.

4.1 Inter-Process Communication

We can invoke the kernel scheduler in a controlled way by performing an IPC from one thread to a thread of lower priority. IPCs of other priorities are handled by the *fastpath* and do not invoke the scheduler. When the *fastpath* is avoided and the scheduler is invoked, the priority of the sending thread has no impact on scheduler execution time.

We used the ARM performance counters to count the number of CPU execution cycles required to perform an IPC from a thread of the highest priority to threads of lower priorities. In our benchmark, the IPC receiver thread was initially blocked, waiting for IPC. The sending thread first read the state of the CPU cycle counter and issued the IPC system call. Upon receiving the system call exception, the kernel unblocked the receiver thread and issued a scheduler transaction to insert it back into the schedule. The kernel then blocked the current thread as it waited on the IPC reply and issued a second scheduler transaction to find the new highest runnable thread in the system. By benchmark design, the highest priority runnable thread was always the IPC receiver and hence it was removed and scheduled for execution. Finally, the receiver thread read the CPU cycle counter. We took the resulting execution time of the IPC as the difference between the two readings of the CPU cycle counter.

In our study, the IPC merely served as a synchronisation signal between two threads. The IPC contained no data payload. Since transfer of data occurs in linear time across all scheduler architectures under investigation, varying the payload size was not expected to yield interesting information. Threads also shared the same virtual memory translations; a virtual address space switch was not performed as part of the study.

We ran the benchmark with a hot cache by using 16 cache warming iterations before collecting 250 samples. We set the hardware scheduler AXI bus clock and logic clock to 100MHz while the CPU operated at its maximum frequency of 667MHz. The software was compiled with arm-linux gcc 4.7.4.

We initially ran the benchmark with each of the four scheduler architectures described in Section 3. The *legacy* scheduler architecture was implemented entirely in software. The *bitmap* scheduler architecture addresses the scalability issues of the legacy scheduler, but at the cost of maintaining additional data structures. The *GP* scheduler architecture featured a hardware accelerated priority queue that the CPU communicated with directly via the GP AXI port. The *ACP* scheduler architecture used shared cache-coherent memory to communicate between CPU and accelerator and used the *SEV* signalling mechanism for synchronisation.

We tested receiver priorities in the range of 250 to 0, however, we observed no change in trend below priority 220; we have therefore excluded some of these measurements for clarity. The median results of this benchmark are presented in Figure 8. Error bars indicate 1st and 3rd quartiles.

We found that the execution time of the legacy software scheduler implementation generally increased linearly as the receiver thread priority decreased. This is because the scheduler traverses the *ksReadyQueues* from the highest priority to the lowest priority until it finds a runnable thread. The lower the priority of the receiver, the more entries the legacy scheduler must examine before it finds the waiting receiver. This behaviour also increases the cache footprint of the scheduler: when the receiver thread is priority 0, the scheduler reads from 256 queue heads. Because head and tail pointers are interleaved, this results in 512 words (2 KB) being loaded into the cache. Another way of looking at this is that the scheduling behaviour results in the eviction of 2 KB of data from the cache in the worst case. This evicted data can be data that is frequently used by an application; system performance will then be further degraded because that data must be reloaded from high-latency main memory when the application is next scheduled.

We investigated the discontinuities in the legacy scheduler curve and found that they correlate with an unusually high number of branch mispredictions (Figure 9). During execution, conditional branch instructions prevent the instruction prefetcher from always maintaining a full instruction pipeline. The prefetcher does not know which branch of execution to load until the branch condition is evaluated. In these cases, the branch predictor attempts to predict the path that execution will take. If the branch predictor is correct, the CPU continues uninterrupted. If the prediction is incorrect, the CPU must flush the instruction pipeline and wait for the prefetcher to fill the pipeline with the correct instruction stream. Although this micro-architectural feature improves CPU utilisation and application performance, it can add a significant amount of variance to execution time. Our results

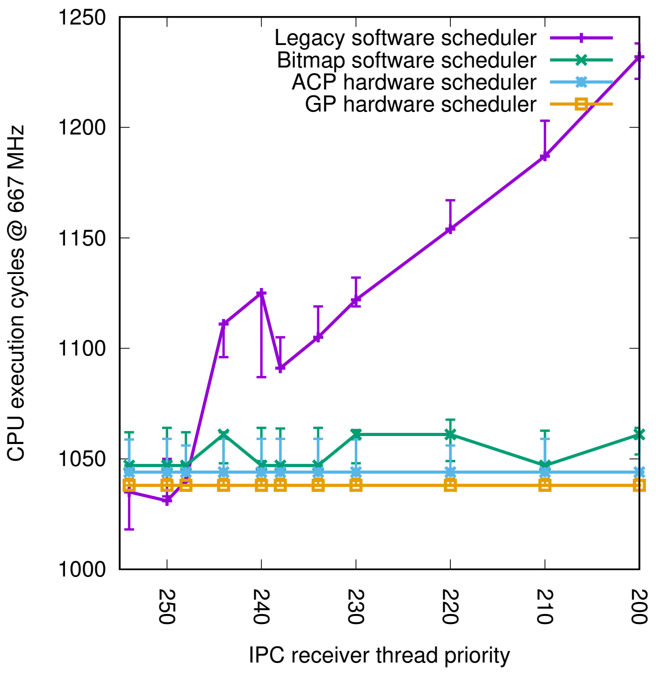


Fig. 8. Hot cache IPC execution cycles for given receiver thread priorities.

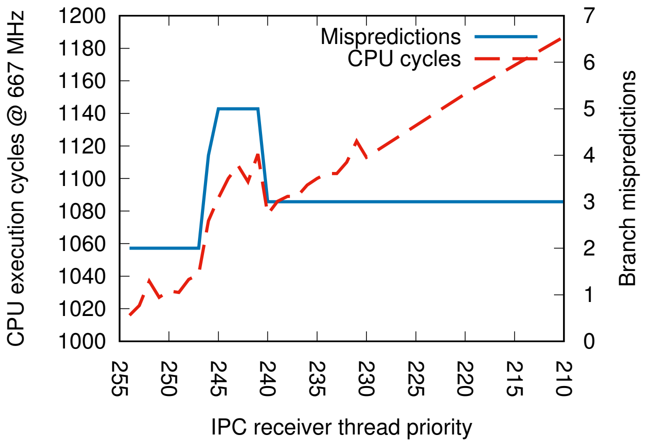


Fig. 9. Legacy scheduler anomaly investigation.

show that branch mispredictions on the ARM Cortex-A9 processor can, with high probability in some cases, result in more CPU cycles to perform fewer iterations of a software loop.

seL4 developers introduced a *bitmap scheduler* to address the scalability issue of the legacy scheduler implementation. This was done by extending the implementation to include a hierarchical bitmap representation of the non-empty *ksReadyQueues*. This optimisation leads to some overheads, but an O(1) lookup complexity was observed (Figure 8). The increased execution time of the bitmap scheduler for high-priority receiver threads reflects the additional operations required for the traversal and maintenance of the bitmap. The bitmap scheduler requires 2 additional reads to locate the highest priority thread: one for each level of the bitmap hierarchy. Once the scheduler identifies the thread to be scheduled, the scheduler marks the thread as active and removes it from the scheduling queue. Because this thread is the only runnable thread in the system, the bitmap scheduler additionally marks the *ksReadyQueue* as empty in the second level and marks the priority group as empty in the first level. For the sake of abstraction, the scheduler cannot simply write 0 to these words: it must perform the correct bit operation to clear only the relevant bit at each level. The result is that the bitmap scheduler performs another 2 reads and 2 writes to update the bitmap, however, it is likely that these 4 memory accesses operate on memory in the cache rather than suffering a penalty from loading data from main memory a second time.

We also see that the directly connected GP accelerator offered only marginal improvements over the optimised bitmap scheduler (Figure 8). While the entire task of priority queue management is offloaded to the accelerator, the performance gain is reduced due to the communication latency between the CPU and accelerator. We can see that the legacy software implementation still outperforms the GP-connected accelerator when the IPC receiver priority is very high. This condition provides the best case performance for the legacy scheduler because the iterative search only needs to examine two priority levels before the highest priority runnable thread is located.

Surprisingly, when communication is achieved using low-latency, cache-coherent shared memory rather than direct communication, we observed an increase in execution time relative to the GP accelerator. This is because the signalling mechanism is decoupled from the data transfer. The CPU must execute a data synchronisation barrier (DSB) in order to stall the CPU until data has reached the L1 cache. Only then can the CPU signal the accelerator with the *SEV* processor instruction. If the CPU executes the *SEV* instruction before this time, the accelerator may read a stale command from shared memory.

The decision to accelerate a short running OS kernel task scheduler on a high performance CPU was in part motivated by the challenges involved. We anticipated that accelerating the scheduler would be difficult and that it would drive us to find a truly optimal method for fine-grained interactions on an ARM-based CPU-FPGA heterogeneous platform. Our results show that the legacy software scheduler architecture still performs better than both the GP- and ACP-connected hardware scheduler for high-priority IPC receiver threads. We therefore took the best-case performance of the legacy software implementation as a target for the execution time of our hardware-accelerated task scheduler.

We constructed microbenchmarks to investigate why the direct communication of the GP approach out-performs the low-latency shared memory approach of the ACP-connected accelerator. We measured the median number of CPU cycles required to perform each scheduler operation (Table 1) The execution time was measured at the API level of the scheduler, which includes the cost of validating arguments and packing commands into the appropriate format for each accelerator architecture. The overhead of reading the ARM performance counters was measured and subtracted from the results. Note that such fine-grained benchmarking avoids some performance penalty events, such as cache misses.

From the microbenchmark results we found that the GP-connected accelerator requires the least number of CPU cycles to insert a thread into the schedule. These numbers do not reflect the latency of the transaction; they represent the CPU execution time to issue the command. The CPU can continue execution as soon as the transaction has reached the store buffers, well before it is received by the accelerator.

We also found that the GP-connected accelerator requires the greatest number of CPU cycles to retrieve the highest priority thread from the schedule. This is because of an immediate dependency

Table 1. Median scheduler operation cost (CPU cycles).

System	Enqueue	Dequeue highest priority	Total
Legacy (priority 255)	22	49	71
Legacy (priority 0)	22	954	976
Bitmap	30	59	89
GP	12	94	106
ACP	47	74	121
ACP (polling)	30	40	70
Hybrid	10	31	41

on the result of the transfer; we must test the validity of the returned data in case the schedule is empty. When the returned value is *NULL*, the kernel must activate the *idle* thread instead of the highest priority runnable thread. The CPU must stall until this branch in execution is determined.

The ACP results reported a lower execution time for retrieving the highest priority thread than the GP-connected accelerator. This is because the CPU can retrieve the result directly from the low-latency cache instead of waiting for a response from hardware. However, the synchronisation time adds significant overhead for insertions. The total time to perform both an insertion and removal of the highest priority runnable thread is thus higher for the ACP scheduler than for the GP scheduler.

To quantify the costs of data synchronisation and signalling, we modified the ACP-connected scheduler to operate in polling mode. In this mode, the accelerator continuously polls shared memory for a new command. We found that data synchronisation was still required when inserting a thread into the schedule. This is because there was not enough time between thread insertion and thread removal to allow the insertion transaction to complete. The barrier forces CPU store buffers to write through to the cache sooner, which reduces the observation time (and therefore completion time) of the accelerator. On the other hand, thread selection is performed as the last scheduler transaction before returning control to the chosen thread. The time required to restore the thread context and leave the kernel is large enough to avoid this memory barrier. The microbenchmark results of the ACP scheduler in polling mode are included in Table 1. The polled ACP scheduler requires the least number of CPU cycles to retrieve a thread across all systems that have been discussed so far.

We investigated a hybrid implementation to take advantage of the best performing communication channel for each operation. In this implementation, the scheduler uses the GP port for all priority queue commands. After processing any command, the accelerator provides the highest priority runnable thread via ACP in low-latency, cache-coherent shared memory.

We repeated the IPC benchmarks that were detailed at the beginning of this section for both the polling-mode ACP-connected accelerator and the hybrid accelerator. Our results focus on comparing the execution time of each architecture with the best case performance of the legacy software implementation (receiver thread priority 254). The distribution of collected samples is presented as box and violin plots in Figure 10 while numerical results are presented in Table 2. The violin plot shows the probability density of the collected samples: larger plot widths correspond to values at which we observed more samples.

The results show that the polling-mode ACP-connected accelerator and the hybrid accelerator perform better than both software approaches in all cases. The hybrid approach offers a 2.9%

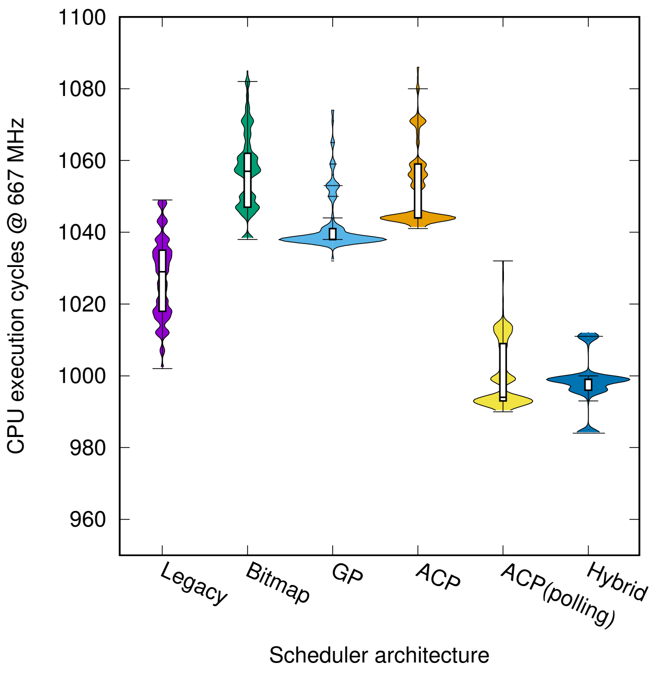


Fig. 10. Hot cache execution cycles with probability density for IPC from thread priority 255 to 254 for the scheduler architectures studied.

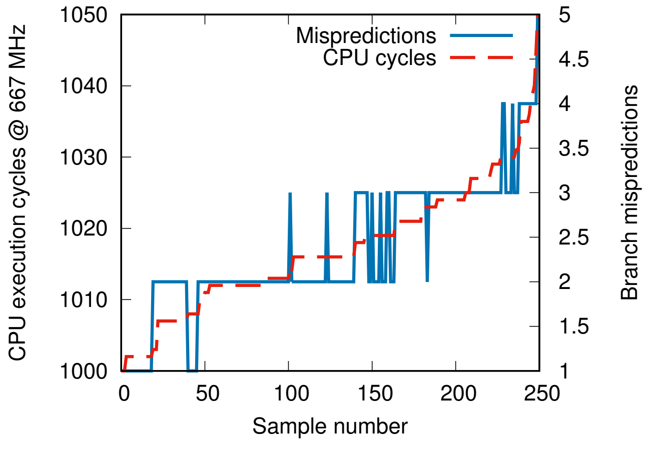
Table 2. Median IPC execution time for priority 254 receiver

System	Median IPC execution CPU cycles	Speedup
Legacy	1029	2.6%
Bitmap	1057	0.0%
GP	1038	1.8%
ACP	1044	1.2%
ACP (polling)	994	6.0%
Hybrid	999	5.5%

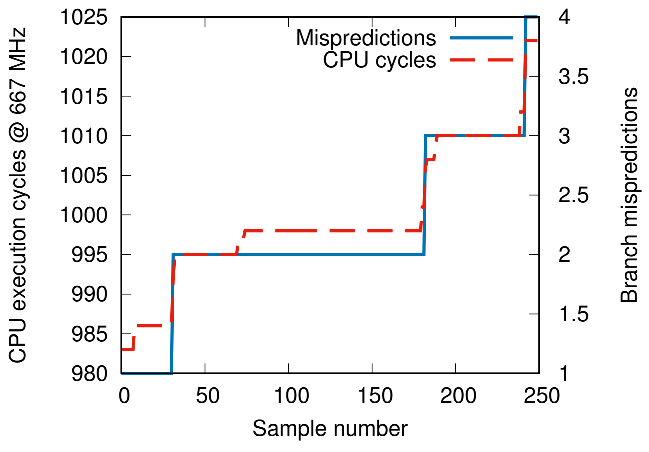
reduction in median execution time when compared with the best-case performance of the legacy software scheduler. We observed a 5.5% reduction compared with the bitmap scheduler.

A reduction in scheduler execution time allows more tasks to be scheduled in a given period of time. The time that would otherwise be consumed by kernel execution can now be used by an application to complete sooner, thereby allowing other tasks to begin sooner.

The results in Figure 10 also provide information about the execution time variance. The schedulers that use hardware acceleration showed well-defined modes. The ARM event counters [1] showed that these modes correlate with branch mispredictions (Figure 11). This pattern of mispredictions was also observed in the pure software implementations. However, noise from other sources of non-deterministic execution masked the effect.



(a) Legacy software scheduler



(b) Hybrid scheduelr

Fig. 11. Branch mispredictions correlated with CPU execution cycles. Samples sorted by execution time.

Hardware acceleration improved the execution time variance in all cases. The hybrid approach, in particular, showed improvements of 58% when compared to the legacy software implementation and 56% when compared with the bitmap optimisation (Table 3).

All other things being equal, the reduction in variance allows more tasks to be scheduled *safely* in a given period of time. Since the execution time is more deterministic, we can reduce the compute power that is reserved for ensuring that a critical real-time task will complete on time. One can then either schedule more tasks on the processor, or reduce the size of the compute resource to conserve energy and manufacturing costs.

Although we have reduced the variance of the system, the branch predictor remains a dominant source thereof. The branch predictor is an important micro-architectural feature for enhanced performance. Although it can be disabled to further reduce variance, this causes a significant reduction in CPU utilisation and an increase in program execution time.

Table 3. IPC execution time variance for priority 254 receiver

System	Variance	Improvement
Legacy	120	0%
Bitmap	120	0%
GP	65	46%
ACP	110	8%
ACP (polling)	80	33%
Hybrid	51	58%

4.2 IRQ latency

The secondary goal of our work was to reduce the latency and jitter of IRQ handling operations. In doing so, we aimed to improve the suitability of the high-performance ARM Cortex-A9 processor for real-time applications.

We measured the IRQ latency for the scheduler architectures studied by programming a timer to deliver an interrupt. Once the IRQ was received by the application thread to which it was bound, the application stopped the timer. We recorded the IRQ latency as the difference in time between when the IRQ was programmed to occur and the time at which the timer was stopped. As with the IPC benchmark, we recorded 250 samples after 16 cache warming cycles.

We designed the benchmark so that the IRQ event caused the current thread to be inserted back into the schedule while a higher priority IRQ handler thread was unblocked and activated. For each system we investigated, the scheduler transaction only involved the insertion of the current thread back into the schedule. The waiting thread thus bypassed the scheduler as it transitioned directly from being blocked to being the highest priority runnable thread in the system.

The scheduler can insert a thread into the priority queue in constant time; it is not sensitive to the priority at which it is to be inserted. However, the priority of the IRQ handler thread must be greater than the priority of the current thread; otherwise, the IRQ event does not result in the immediate scheduling of the IRQ handler thread. For our benchmark, we selected a priority of 255 for the IRQ handler thread and 254 for the thread that was to be inserted back into the schedule.

We expected that the GP and hybrid architectures would perform best in this case as the scheduler can send a command to the priority queue with little CPU overhead. Both software approaches require the manipulation of the linked list for the priority at which the thread is to be inserted. We expected the bitmap approach to require more time than the legacy software scheduler due to the overheads of updating the bitmap.

From the results shown in Figure 12, we see that the GP-connected accelerator consistently outperformed the other architectures. However, other scheduler architectures performed unexpectedly. In particular, the bitmap scheduler outperformed the legacy scheduler, although more operations must be performed for bitmap maintenance when inserting a thread into the schedule. An explanation for this may be that the bitmap scheduler benefits from a friendlier layout of memory in which the low-latency cache is used more effectively.

The code that is used to insert a thread into the schedule is shared between both GP and hybrid approaches. Although there is overlap in the samples acquired, the jitter that the hybrid approach experiences is larger than that of the GP approach.

4.3 Application case study

In this section we evaluate the impact of the improved hybrid scheduler using a hypothetical real-world application. Suppose an autonomous vehicle that makes use of a decryption server

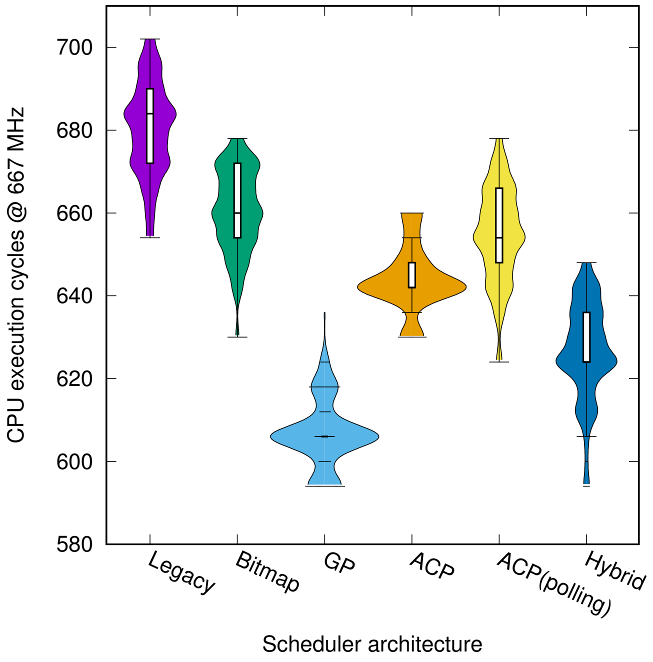


Fig. 12. Hot cache IRQ latency for the scheduler architectures studied.

for controlled access to sensitive data by N clients. One client may be responsible for processing packets that arrive on a secure radio link, while others may request the decryption of a subset of WiFi network packets. The dedicated decryption server is isolated from other tasks by using the MMU of the CPU. In this way, we protect the decryption key and the decryption service from would-be attackers and client programming errors.

With this experiment, we demonstrate that the throughput of the decryption server is increased when hardware acceleration is used to reduce the seL4 kernel task scheduler execution time. We evaluate only the best performing hardware (the hybrid scheduler) and software (the bitmap scheduler) approaches in our evaluation. The decryption server throughput was measured over a 5 second period. The experiment was repeated for various decryption workload sizes. We expect that, as the workload size increases, the scheduler overhead becomes small relative to decryption processing time.

Our experiment setup consists of a benchmarking control task, an AES decryption server and 4 client tasks. Each task is isolated by the MMU of the CPU. The benchmarking control task was set to the highest priority (255). After the control task had configured the system, it waited for the arrival of an IRQ from a periodic 5 second timer. With the highest priority thread blocked, the decryption service, at priority 254, was scheduled for execution. The decryption service then blocked its execution until a decryption request arrived from one of the 4 clients. Finally, when a client was scheduled for execution, it was programmed to continuously send decryption requests of the nominated size to the server using synchronous IPC. Each client shared a page of memory with the server such that the encrypted workload and decrypted result were transferred between client and server via a pre-determined location in shared memory. The client sent only the packet size to the server via synchronous IPC to begin packet decryption.

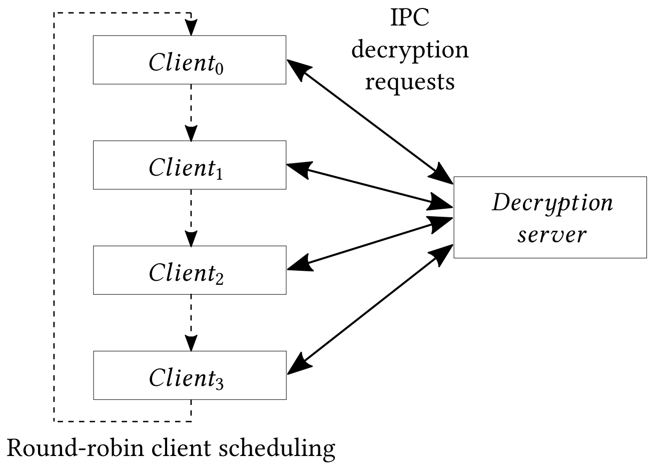


Fig. 13. System configuration for decryption application.

In a real system, each client would operate at a difference priority. The clients would become blocked as they perform IPC with other tasks or wait for external IRQ events. This would allow other clients to execute and deliver decryption requests to the server. In our system, we simulated this behaviour by programming each client to *yield* to another client of the same priority after the completion of each decryption task. The clients were thereby scheduled in round-robin order.

We programmed the clients to count the number of decryption requests that were completed by the server. When the timer IRQ was finally delivered to the control task, the control task accumulated the number of completed decryption requests from each client and reported the server throughput over the 5 second time interval.

For each client request, four scheduling operations were performed (Figure 13). The first pair of operations occurred during synchronous IPC between client and server as described in Section 3. The second pair of operations were for the round-robin scheduling of the next client thread. For round-robin scheduling, the kernel first appended the current thread to the tail of the queue associated with the current thread's priority. The kernel then removed the next thread from the head of this queue and scheduled it for execution.

For a packet size of 16 Bytes, the hybrid accelerator showed a 7% throughput improvement when compared to the bitmap scheduler (Figure 14). As expected, the benefits of accelerating the scheduler diminish as the workload size increases.

5 FUTURE WORK

In future work, we aim to evaluate how the "Intel® Xeon® with FPGA" platform compares with the Zynq for such fine-grained interactions. Intel provides similar direct and cache-coherent shared memory communication mechanisms between CPU and FPGA, however there is no low-latency signalling mechanism. On the other hand, the Intel platform does inform the programmable logic of cache line evictions from the FPGA-local cache (e.g. when the CPU invalidates the cache line by writing to its corresponding address).

We also seek to extend our study to data structures other than priority queues. We see an opportunity for HW/SW cooperation in memory management, sort and search algorithms that are

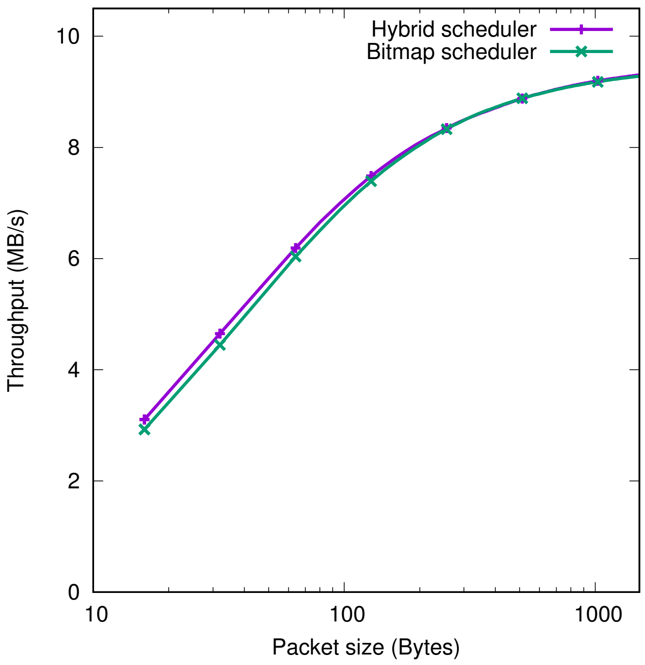


Fig. 14. Decryption server throughput when serving 4 client tasks.

commonly used in data centers. By targeting shared libraries, we could allow unmodified binaries to benefit from hardware acceleration.

6 CONCLUSION

We have evaluated the potential for tightly-coupled CPU-FPGA systems to improve the CPU execution time and jitter of frequent, short-running operations. The target application, the seL4 kernel task scheduler, proved difficult to accelerate and required careful selection of hardware-software communication strategies.

We evaluated communication via memory-mapped registers and also cache-coherent shared memory. No single strategy on its own was able to provide a performance improvement over the best case execution time of the legacy software. However, a carefully selected combination of these strategies was able to achieve a 5.5% improvement in CPU execution time. In this case, all communication from the CPU to the FPGA was made directly through memory-mapped registers to take advantage of buffering. All communication from the FPGA to the CPU was made through shared memory to prevent high-latency reads from stalling the CPU.

We improved the important real-time systems metric of execution time jitter by up to 58% with respect to a software-only implementation. We were able to eliminate most sources of variance, however, branch mispredictions remain as a dominant source of non-deterministic execution. Disabling branch prediction would dramatically reduce the remaining jitter, but at a significant cost in execution time.

Although the improvement in execution time is small, our results show that the use of FPGA-based accelerators in tightly-coupled systems need not be limited to coarse-grained acceleration.

They can also improve the execution time of very short-running tasks, such as appending to and removing from a linked list.

ACKNOWLEDGMENTS

This research was partially supported through an Australian Government Research Training Program Scholarship.

REFERENCES

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

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

[3] ARM limited. *ARM Cortex-A9 MPCore Technical Reference Manual DDI0407H*, 2012.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] P. Kuacharoen, M. A. Shalan, and V. J. Mooney 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.

[11] B.-C. C. Lai, P. Schaumont, and I. Verbauwhede. A light-weight cooperative multi-threading with hardware supported thread-management on an embedded multi-processor system. In *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, pages 1647–1651, October 2005.

[12] A. Lyons and G. Heiser. It's time: OS mechanisms for enforcing asymmetric temporal integrity. *CoRR*, abs/1606.00111, 2016.

[13] 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.

[14] 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.

[15] 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.

[16] T. Sewell, F. Kam, and G. Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.