INEXPENSIVE SCHEDULING IN FPGAS

W. W. Terpstra, D. Beck, M. Kreider, GSI, Darmstadt, Germany

Abstract

In the new scheme for machine control used within the FAIR project, actions are distributed to front-end controllers (FEC) with absolute execution timestamps. The execution time must be both precise to the nanosecond and scheduled faster than a microsecond, requiring a hardware solution. Although the actions are scheduled at the FEC out of order, they must be executed in sorted order. The typical hardware approaches to implementing a priority queue (CAMs, shift-registers, etc.) work well in ASIC designs, but must be implemented in expensive FPGA core logic. Conversely, the typical software approaches (heaps, calendar queues, etc.) are either too slow or too memory intensive. We present an approach exploiting the time-ordered nature of our problem to sort in constant time using only a few memory blocks.

INTRODUCTION

In a schedule-driven control system, pending actions include an execution timestamp. When the timestamp matches the current time, the responsible front-end controller (FEC) executes the action. Within the scope of the FAIR project, many physically distributed FECs will execute actions in concert. Actions are coordinated and distributed by a central unit, the data master, which controls beam production.

Unfortunately, the data master is quite complicated. The actions it requires a FEC to take may be delivered in an order different than the execution order. This paper describes how a FEC takes an incoming set of out-of-order actions and outputs them in sorted order. In principle, a single FEC may control many devices attached to many interfaces. However, for the purposes of this paper, we will concern ourselves only with the actions delivered by a FEC through a single interface. We also omit message processing.

Concretely, a FEC processes action tuples (a,x) , where a is the action to execute and x is the time at which it must be executed. At any given time t, the FEC has a set P_t of pending/yet-to-be-executed tuples. That is, $x \geq t$ for all $(a,x) \in P_t$. At time t, the FEC must output a if $(a,t) \in P_t$; this is illustrated in Figure 1. Obviously, it is physically impossible for a single interface to execute two actions concurrently. The data master would never ask a FEC to do something impossible. Therefore, we can assume $x = y \rightarrow a = b$ for $\{(a,x),(b,y)\} \subseteq P_t$.

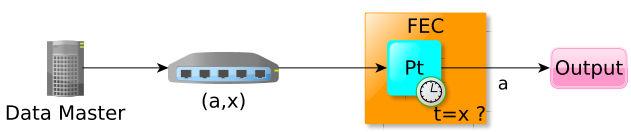

Figure 1: Actions (a,x) flow from the data master to the FECs. They are stored in P_t until $t = x$ and then output.

For FAIR, the control system is required to have nanosecond precision. This means that at least the execution/output of actions must be synchronized by hardware. Furthermore, the data master distributes the schedule via gigabit Ethernet. The FECs must be capable of accepting the schedule at the full rate, which suggests hardware may be required here as well. FECs include an FPGA and therefore we choose to solve the problem of receiving the schedule, sorting it, and outputting it, all in the FPGA.

AN FPGA CRASH COURSE

We can program an FPGA to contain customized hardware. Like an application-specific integrated circuit (ASIC), we can implement any digital circuit consisting of logic gates and registers. Unfortunately, FPGAs only have a limited number of comparatively slow logic elements to implement core logic (logic gates and registers). FPGAs also include many block memories, which are essentially small SRAM chips embedded inside the FPGA. These block memories have the same density and performance as they would in an ASIC. For these reasons, a good FPGA design tries to minimize core logic by leveraging block memory.

Digital logic is generally clocked. Registers take their values on the rising edge of a clock signal. In a modern FPGA with a reasonably complicated design, the clock speed is generally limited to <500MHz. That means that each clock cycle takes >2ns. The particular sorting circuit presented here can run at 325 MHz (≈3ns period) on an Altera Arria V chip. To achieve nanosecond precision, this means we need to be able to output an action in very few clock cycles. Furthermore, we need to accept new tuples at a similar rate.

Our goal is thus to accept a tuple (a,x) on clock cycle t to compute $P_{t+1} = P_t \cup \{(a,x)\}$. Furthermore, if there is $(a,t) \in P_t$, then we output a. In other words, on every clock cycle, we need to potentially accept a new action into the buffer and output the action whose timestamp is due.

APPROACH

Sorting a set of $n = |P_t|$ numbers requires $O(n \log(n))$ comparisons. If we must sort one timestamp every cycle, that means $\log(n)$ comparisons per cycle. For FAIR, timestamps are 64-bit numbers. In a modern FPGA, comparing even a single pair of 64-bit numbers in one clock cycle would reduce the maximum performance to ≈125 MHz. A hardware technique called pipelining can spread this work between multiple clock cycles. In our implementation, a 64-bit comparison is done in 3 steps to achieve the target performance. Unfortunately, this makes the comparator quite expensive. While there are techniques to spread out the work of all $\log(n)$ comparisons across multiple cycles using pipelining [1], these approaches cost significant hardware.

ISBN 978-3-95450-146-5

We solve the scheduling problem in a different way, requiring only 1 comparison per cycle. The key insight is that we do not actually need to solve the sorting problem. We only need to output the current action a for time t. Sorting requires finding in each step the next *smallest* timestamp. We only need to find actions with the current timestamp.

Imagine a gigantic table T that stores actions. At any given time t, you locate row t in the table and read out the action to execute; $a = T[t]$. Every time a new tuple (a,x) arrives from the data master, you just write a into entry x in the table; $T[x] := a$.

The only problem with this simple approach is that the table must have 2^{64} entries, one for each possible time x. We must find a way to store the table more compactly. The first thing to keep in mind is that an FPGA design has finite physical resources. We must define an upper limit to the number of pending actions which the FPGA design can store. This limit can be reconfigured whenever the FPGA is reprogrammed and currently we have set it to 256. Block memory in a modern FPGA is only available in ≥256-entry chunks, so anything less is no cheaper.

With only 256 entries, we cannot directly implement the gigantic look-up table. However, we can record anything that happens in the next 256 cycles. Just check if an incoming tuple (a,x) obeys $t <= x < t + 256$. If it does, then set $T[x \bmod 256] = a$. On clock cycle t, just look up $a = T[t \bmod 256]$ and clear $T[t \bmod 256]$ for re-use later.

Unfortunately, tuples arriving from the data master may be far in the future; $x \geq t + 256$. If we wrote them into the table anyway, we would execute them too early. However, we must still record these actions somewhere! Our scheme is to instead use two tables. There is one (unsorted) table used to store all pending actions, the *pending table*. There is another table used to record references to the pending table for those actions with timestamps due in the next 256 cycles, the *calendar*. This is illustrated in Figure 2.

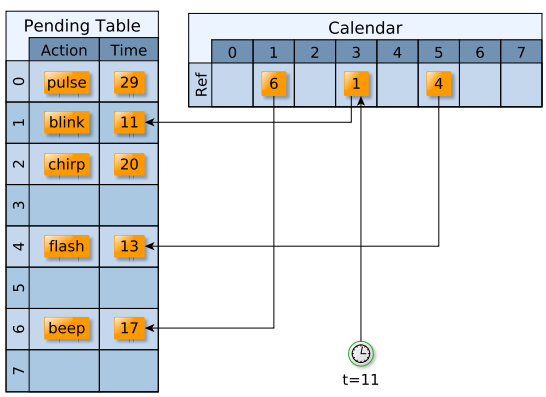

Figure 2: An example of 8-entry compressed tables.

Of course, these two tables alone do not solve the problem. Actions which execute more than 256 cycles in the future do not appear in the calendar. However, we can continuously scan the pending table's records, one record per clock cycle. If the pending record has a timestamp due within 256 cycles ($x < t + 256$), put a reference into the calendar. In fact, once we have this *scanner* process, we no longer need to write incoming tuples into the calendar at all. We need only write the tuple into the pending table and the scanner will put a reference into the calendar for us.

The reason this scheme works is that the scanner is guaranteed to fill the calendar before an action must be executed. Pick an arbitrary tuple (a,x) from the pending table. Define u as the time when the tuple was stored into the pending table. Because t grows without bound, eventually $x < t + 256$ and the scanner will put a reference in the calendar. Define time v to be the first time that the scanner does this. The scanner re-scans the records in the pending table every 256 cycles. Thus, at time $w = v - 256$, as long as (a,x) is already in the table ($u < w$), the scanner will have previously visited the record. However, since v was the first time t where $x < t + 256$, we know that $x \geq w + 256 = v$. In other words, a reference to (a,x) is placed into the calendar at time v, before it must be executed (x).

The above proof requires that the tuple is already in the pending table at time w. Recall that at time $t = v$, a reference will be placed into the calendar because $x < t + 256$, and thus $x - 256 < v$. Therefore, to ensure correct behaviour, we require tuples to be delivered early; $u < x - 256 - 256$. This implies that $u < v - 256 = w$, as required in the proof. What this means is that you must store a tuple into the pending table 512 cycles before its execution timestamp. Fortunately, in FAIR a time budget of 4μs is quite reasonable; this allows us to run as slowly as 125MHz (8ns period).

DESIGN

For a complete implementation, we must also manage the free entries in the pending table. A new incoming action must be written into an empty record in the pending table. When an action is executed, the dispatcher already removes the reference from the calendar. However, the entry in the pending table must be released as well. To facilitate allocation and release, we implement a *manager* that records free pending indexes in a *free stack*.

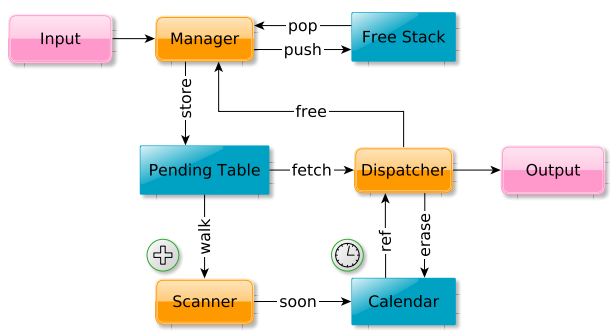

Figure 3: Block Diagram of block memories and processes.

As shown in Figure 3, we have in total three tables: pending, calendar, and free. They are controlled by three processes: dispatcher, scanner, and manager. On each cycle, the

ISBN 978-3-95450-146-5

dispatcher checks the current time in the calendar for a reference and erases whatever it finds. It then fetches the action from the pending table and outputs it. Finally, it tells the manager to free the pending entry. The scanner repeatedly walks the pending table. If the scanned record has a timestamp $< t + 256$ (the only comparator/adder in our design), the scanner writes a reference to that soon-to-be-executed record into the calendar. The manager accepts new tuples from the network and stores records into the pending table. If the manager must both allocate and free a record, it overwrites the freed record with the new tuple. If it must only free, it pushes the free pending-table index to the free stack. If it must only allocate, it pops the free stack and stores the tuple to the free index.

One very annoying constraint in FPGA design is that a block memory can only execute a single write and a single read per cycle. The free stack poses no problem here; the manager at most reads or writes one entry per cycle. The manager is also the sole writer to the pending table. The action column of the pending table is only read by the dispatcher, so this poses no problem. The timestamp column (in our actual implementation) is read by both the scanner and the dispatcher. However, two readers and one writer can be implemented by duplicating the table. What is not so straight-forward is the calendar.

The calendar is only read by the dispatcher. Unfortunately, it is both written by the scanner (to add references) and erased by the dispatcher (to remove references). On some platforms, one can implement same-address get+erase using one half of a true dual port memory. Sadly, this trick is not portable (same-port old-data) and does not work on the Arria V. Instead, we used two bank-interleaved block memories. Since t is incremented every cycle, it changes parity every cycle. Thus, we can read from the next even memory address $t + 1$ at the same time as we erase the odd memory address t.

To achieve high performance, each of the processes is heavily pipelined. Unfortunately, this creates a structural hazard. The scanner might write a reference into the calendar where the dispatcher is only halfway-done executing. To solve this, we just increase the width of the calendar slightly, and forbid the scanner from writing near the position of the dispatcher. Pending records near the scanner thus get scanned twice before being executed by the dispatcher, and only one of those times can be a structural hazard.

OUTLOOK

We presented a way to output actions scheduled according to their timestamps. While the actions arrive in unsorted order, we are still able to accept and output one action every clock cycle. Due to careful pipelining, the design can run at up to 325MHz on an Altera Arria V. The design uses only three memory blocks, one 64-bit comparator, and three very simple processes. The low cost of this solution was achieved by formulating the real-time scheduling problem as a lazily-updated look-up table.

Our approach has many similarities to calendar queues [2]. However, unlike calendar queues, time spent inspecting empty calendar entries is not time wasted. In a calendar queue, one must retrieve the *next* scheduled action. In real-time scheduling, we need only retrieve the action scheduled for the current time. Calendar queues thus suffer the same sorts of problems as bucket- and radix-sort [3]; they depend on the distribution of execution timestamps and performance can become significantly degraded.

Unfortunately, even our approach is not completely immune to a poor timestamp distribution. If, for some reason, more than 256 pending actions need to be stored at once, the block memory used for the pending table will be exhausted. For this reason, the FAIR data master only schedules actions that will be executed in the immediate future.

Compared to a heap-based approach [1], our approach deals poorly with two actions scheduled to occur simultaneously. While this situation must be avoided in any case, as the relative order between two simultaneous actions is undefined, a heap implementation would at least output these two actions back-to-back. Our approach will instead delay one of the actions by 256 cycles. We think that the significant area and performance benefit inherent to our approach justifies this small concession.

Finally, in an ASIC, it might make sense to use a shift-register-based approach [4]. To implement this in an FPGA would be prohibitively expensive as the storage of the records must be done in registers, not block memory. Even in an ASIC, the necessary CAM-like shift-register setup will cost far more than an equivalent SRAM. Indeed, it is hard to imagine any solution exists which is significantly smaller or faster than the one we propose here.

REFERENCES

[1] Wojciech M Zabołotny. Dual port memory based heapsort implementation for FPGA. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2011*, pages 80080E–80080E. International Society for Optics and Photonics, 2011.

[2] Randy Brown. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[3] D. Knuth. Sorting by distribution. In *The Art of Computer Programming, Volume 3: Sorting and Searching*, chapter 5.2.5, pages 168–179. Addison-Wesley, 3 edition, 1997.

[4] Chen-Yi Lee and Jer-Min Tsai. A shift register architecture for high-speed data sorting. *Journal of VLSI signal processing systems for signal, image and video technology*, 11(3):273–280, 1995.

ISBN 978-3-95450-146-5