Run-Time Scheduling Support for Hybrid CPU/FPGA SoCs

Jason Agron
jagron@ittc.ku.edu
Acknowledgements

• I would like to thank…
  • Dr. Andrews, Dr. Alexander, and Dr. Sass for assistance and advice in both research and class work.
  • Wes, Garrin, and Erik for help with tools and coding problems.
Publications


• David Andrews, Wesley Peck, Jason Agron, Erik Anderson, Jim Stevens, Fabrice Baijot, Presentation on the KU Hybrid Threads Project, MOCHA Design Conference, January 2006


Overview

- HybridThreads Project.
  - Primary Goals
  - Basic Architecture
- Purpose of my work.
- Comparison to related works.
- Designs, Implementations, Results.
- Conclusion.
- Questions.
HybridThreads Project

• Hybrid architecture = CPU + FPGA.
  • Portions of programs can now be in SW or HW.
• Create a unified programming model.
  • Threaded programming model based on POSIX standard.
  • All computations are viewed as threads.
    – HW, SW, or both.
• HW/SW co-design of OS services.
  • OS services can be implemented in HW to give a uniform interface to hybrid computations.
    – Anyone that can “talk” on the bus can use the services.
    – No need to interrupt the CPU to access services.
  • HW implementations allow for more parallelism to be exploited.
    – OS services themselves run in parallel with application execution (coarse-grained).
    – Internals of each OS service can be parallelized (fine-grained)
• Improves accessibility to resources of the FPGA.
Traditional Architecture (All SW)
HybridThread Architecture
Purpose of my work

• Provide scheduling support for “hybrid” threads.
  • Uniform APIs, regardless of thread “type”.

• Allow for high-level scheduling policies.
  • Add priority scheduling.
  • Separate OS policy concerns
    – More modular (scheduling != management).
    – Easier to extend and scale within the framework.

• Minimize overhead and jitter!!!
  • Reduce overhead/jitter of system by streamlining the scheduler.
Benefits of a HW-based Scheduler

<table>
<thead>
<tr>
<th>SW-Based Scheduler Events</th>
<th>HW-Based Scheduler Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer interrupt goes off</td>
<td>Timer interrupt goes off</td>
</tr>
<tr>
<td>ISR context switches to scheduler</td>
<td>ISR retrieves sched. decision from HW</td>
</tr>
<tr>
<td>Scheduler begins running</td>
<td>New thread begins to run</td>
</tr>
<tr>
<td>New thread begins to run</td>
<td></td>
</tr>
</tbody>
</table>

- SW-based scheduler is “invoke on demand”.
  - Starts to get “ready” when needed.
- HW-based scheduler is “ready on demand”.
  - Always “ready” ASAP.
Benefits of a HW-based Scheduler

• Scheduler can be invoked without interrupting the CPU.
  • Scheduler is the sole “bookkeeper” of the R2RQ.
    – Scheduler doesn’t require CPU time to execute.

• Traditional ISRs are translated into ISTs (Interrupt Service Threads).
  • Traditional ISRs are akin to threads with priority level of $\infty$.
  • CPU can be shielded from external interrupts by transforming them into scheduling requests.
    – Interrupts are then fielded by the scheduler, not the CPU.
    – Scheduler decides when to interrupt CPU based on status of R2RQ.
    – Jitter from interrupts can be controlled:
      – Critical interrupts = high priority \(\rightarrow\) interrupt all user threads.
      – Non-Critical interrupts = low priority \(\rightarrow\) interrupt some user threads.

• Scheduling algorithm can be parallelized.
  • Reduction of overhead and jitter.
Related Works

• RealFast/Malardalen – RTU: Real-Time Unit, A Real-Time Kernel in Hardware.
  • Systolic array implementation of R2RQ.
  • EDF, PRI, RM capable.
  • Can handle 16 tasks with 8 priority levels.

• Georgia Tech – Configurable Hardware Scheduler for Real-Time Systems.
  • Systolic array implementation of R2RQ.
  • EDF, PRI, RM capable.
  • 421 logic elements (slices), and 564 registers for queue of size 16.

  • Systolic array implementation of R2RQ.
  • EDF capable.
Problems with Systolic Arrays

- Systolic arrays are fast and easily allow for dynamic changes in priority.
  - Registers within cells allow for parallel accesses
- Systolic arrays are easily scaled through cell concatenation.
  - Each cell requires registers, multiplexers, comparators, and control logic.
    - Scaling systolic arrays requires lots of logic resources!
- But HW threads require logic resources of FPGA!
  - HybridThread OS modules need to be as small as possible to save space for HW threads.
  - BRAMs are to be used instead of registers to hold ready-to-run queue structure.
- Pros:
  - Scalable: more space $\rightarrow$ use more BRAM and slightly more logic.
    - BRAMs don’t take up CLBs
    - BUT address decode logic and pointers will grow slightly.
  - Fast: 2 clock cycle reads, 1 clock cycle writes.
    - Almost as fast as registers.
- Cons:
  - Serial: Only 1 or 2 accesses at a time.
    - Dynamic priority changes can’t happen in parallel.
Comparison to Related Works

• HybridThreads is compatible with the POSIX (Pthread) thread standard.
  • RealFast/GaTech/Valencia use their own custom APIs.
    – Doesn’t allow for easy portability between systems.
• HybridThreads R2RQ is of size 256.
  • RealFast/GaTech only have R2RQs of size 16.
    – Systolic queue of size 16 requires ~421 slices
    – BRAM queue of size 256 requires ~484 slices.
• HybridThreads must support scheduling of both SW and HW resident tasks.
  • Other systems only handle SW threads.
• HybridThreads system is real → simulatable, synthesizable, and usable.
  • Valencia’s scheduler is only theoretical.
  • RealFast/GaTech have real systems
    – BUT you must learn their custom APIs to use their systems.
    – Pthreads applications can be ported to HybridThreads for “free”
      – Using our pthread to hthread wrapper.
Initial Design

- Break scheduling services out of TM.
  - Define a standard interface.
  - Create a R2RQ that is separate from management data structures.
- Add priority scheduling services (while still remaining backwards compatible).
Internals of First Redesign

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>1-bit</td>
<td>1 = Queued, 0 = Not Queued</td>
</tr>
<tr>
<td>N6:N7</td>
<td>8-bit</td>
<td>Ready-to-run queue next pointer</td>
</tr>
<tr>
<td>L0:16</td>
<td>7-bit</td>
<td>Scheduling priority-level</td>
</tr>
</tbody>
</table>

Thread Data BRAM

- 0:12 = [0x0 to 0x12]
- 13:31 = [0x13 to 0x31]
- 32:51 = [0x32 to 0x51]
- 52:60 = [0x52 to 0x60]
- 61:67 = [0x61 to 0x67]
- 68:71 = [0x68 to 0x71]
- 72:75 = [0x72 to 0x75]
- 76:79 = [0x76 to 0x79]
- 80:83 = [0x80 to 0x83]
- 84:87 = [0x84 to 0x87]
- 88:91 = [0x88 to 0x8B]
- 92:95 = [0x92 to 0x95]
- 96:99 = [0x96 to 0x99]
- 100:103 = [0xA0 to 0xA3]
- 104:107 = [0xA4 to 0xA7]
- 108:111 = [0xA8 to 0xAB]
- 112:115 = [0xAC to 0xAD]
- 116:119 = [0xAE to 0xAF]
- 120:123 = [0xB0 to 0xB3]
- 124:127 = [0xB4 to 0xB7]
- 128:131 = [0xB8 to 0xBA]
- 132:135 = [0xBC to 0xBF]
- 136:139 = [0xC0 to 0xC3]
- 140:143 = [0xC4 to 0xC7]
- 144:147 = [0xC8 to 0xCB]
- 148:151 = [0xCC to 0xCF]
- 152:155 = [0xD0 to 0xD3]
- 156:159 = [0xD4 to 0xD7]
- 160:163 = [0xD8 to 0xDB]
- 164:167 = [0xDC to 0xDF]
- 168:171 = [0xE0 to 0xE3]
- 172:175 = [0xE4 to 0xE7]
- 176:179 = [0xE8 to 0xEB]
- 180:183 = [0xEC to 0xEF]
- 184:187 = [0xF0 to 0xF3]
- 188:191 = [0xF4 to 0xF7]
- 192:195 = [0xF8 to 0xFB]
- 196:199 = [0xFC to 0xFE]
Initial Results

- R2RQ of size 256 with 128 priority levels.
  - Linear traversals $\Rightarrow O(n)$.

- Synthesized on Virtex-II Pro 30:
  - 484 out of 13,696 slices, 573 out of 27,392 flip-flops, 873 out of 27,392 4-input LUTs, 1 out of 136 BRAMs.
  - Max. operating frequency of 166.7 MHz.

- Scheduling decision requires $\sim$40 ns per thread in R2RQ.
  - Variable execution-time based on R2RQ length.
O(n) Timing Results

<table>
<thead>
<tr>
<th>No. of Threads in R2RQ</th>
<th>Time (ns)</th>
<th>Est. Time/Thread (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>10060</td>
<td>40.24</td>
</tr>
<tr>
<td>128</td>
<td>5140</td>
<td>40.16</td>
</tr>
<tr>
<td>64</td>
<td>2610</td>
<td>40.78</td>
</tr>
<tr>
<td>32</td>
<td>1330</td>
<td>41.56</td>
</tr>
<tr>
<td>16</td>
<td>690</td>
<td>43.13</td>
</tr>
<tr>
<td>2</td>
<td>130</td>
<td>65</td>
</tr>
</tbody>
</table>

• O(n) → Variable scheduling decision delay based on R2RQ length.
• Context switch requires ~ 2µs
  • (R2RQ ≤ 32 threads) → decision completes before C.S. completes
  • (R2RQ > 32 threads) → decision completes after C.S. completes
• As R2RQ length increases so does the possibility of a scheduling event occurring while the next scheduling decision is still being calculated, thus introducing jitter into the system.
O(n) Timing Results

- **Raw interrupt delay** = time from when an interrupt fires to when the CPU enters the ISR.
  - System dependent, cannot be changed – due to variable length execution times of atomic instructions that delay interrupt acknowledgement.
    - Not affected by number of threads in R2RQ
  - Mean = 0.79 μs and Jitter = (Max – Mean) = 0.73 μs.

- **End-to-end scheduling delay** = time from when an interrupt fires to when the C.S. is about to complete (old context saved, new context about to be loaded).
  - Mean = 2.0 μs and Jitter = (Max – Mean) = 2.4 μs with 250 threads in R2RQ.
    - Jitter is caused by scheduler module and cache.
  - Raw interrupt delay makes up a significant portion (~ 1/3) of end-to-end scheduling delay.
Accomplishments of 1st Design

• Developed a standard scheduling interface.
  • Enforces policy, while leaving the mechanism abstract.
• Provides HPF, FIFO, Round-Robin scheduling services.
• \( O(n) \) – scheduling decisions.
  • FIFO R2RQ – requires traversal.
  • Decision is slower than context switch sometimes.
    – Long R2RQ = Long traversal.
• Conclusion:
  • Performance could be better.
    – 2.0 \( \mu s \) end-to-end scheduling delay with 250 threads with 2.4 \( \mu s \) of jitter is pretty good.
      – Linux delay is in the millisecond range!
      – RealFast/Malardalen’s RTU is in the 40 \( \mu s \) range!
  • Still need control of both SW and HW threads.
Second Redesign

- Change R2RQ structure to reduce overhead and jitter.
- Solution = Partitioned R2RQ + Priority Encoder!
Priority Encoder

- Priority Encoder calculates the highest priority level active in the system.
  - Input register: 1-bit per priority level. (1 = active, 0 = non-active).
  - Output register: highest active priority level in the system.
- Requires 4 clock cycles to execute.
# Internals of Second Redesign

## Thread Data BRAM

<table>
<thead>
<tr>
<th>Field Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q? (1-bit)</td>
<td>1 = Queued, 0 = Not Queued.</td>
</tr>
<tr>
<td>10:16</td>
<td>(7-bit) Scheduling priority-level</td>
</tr>
<tr>
<td>P0:P7</td>
<td>(8-bit) Ready-to-run queue previous pointer</td>
</tr>
</tbody>
</table>

## Priority Data BRAM

<table>
<thead>
<tr>
<th>Field Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0:M7</td>
<td>(8-bit) Priority-queue head pointer</td>
</tr>
<tr>
<td>T0:T7</td>
<td>(8-bit) Priority-queue tail pointer</td>
</tr>
</tbody>
</table>

## Diagram

- **Dequeue Begin**
  - TID = Thread_ID to DeQ
  - l_entry = Thread_Data[TID]
  - de-assert next_thread_valid

- **Lookup Priority Entry**
  - PRI = l_entry.priority
  - l_entry.Q = 0
  - p_entry = Priority_Data[PRI]

- **Check Encoder Input**
  - old_head = p_entry.head
  - l_entry = Thread_Data[old_head]
  - equal_flag = (p_entry.head == p_entry.tail)
  - equal_flag = 1

- **Set Q To Empty**
  - encoder_input[PRI] = 0

- **Write Back Entries**
  - Thread_Dict[TID] = l_entry
  - Thread_Data[old_head] = l_entry
  - Priority_Data[PRI] = p_entry

- **Lookup Highest Priority Entry**
  - h_entry = Priority_Data[encoder_output]
  - exist_flag = (encoder_input <= 0)

- **Return Next Thread Valid**
  - next_thread_id = h_entry.head
  - assert next_thread_valid

- **Return Next Thread Invalid**
  - No active threads in the system
  - (No threads are Crit)
2nd Redesign Results

- R2RQ of size 256 with 128 priority levels.
  - No traversals needed \(\rightarrow\) fixed execution times.
  - \(O(1)\).
- Synthesized on Virtex-II Pro 30:
  - 1,034 out of 13,696 slices, 522 out of 27,392 flip-flops, 1,900 out of 27,392 4-input LUTs, 2 out of 136 BRAMs.
  - Max. operating frequency of 143.8 MHz.
- Scheduling decision executes in fixed amount of time (\(\sim 24\) clock cycles).
O(1) Timing Results

<table>
<thead>
<tr>
<th>No. of Threads in R2RQ</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>240</td>
</tr>
<tr>
<td>128</td>
<td>240</td>
</tr>
<tr>
<td>64</td>
<td>240</td>
</tr>
<tr>
<td>32</td>
<td>240</td>
</tr>
<tr>
<td>16</td>
<td>240</td>
</tr>
<tr>
<td>2</td>
<td>240</td>
</tr>
</tbody>
</table>

• O(1) → Constant scheduling decision delay regardless of R2RQ length.

• All scheduling operations execute in fixed amount of time that is less than C.S. time.
  • Scheduling operations do not inject any jitter into the system!
O(1) Timing Results

- Raw interrupt delay was re-measured and found to be the same as in the system with O(n) R2RQ.
  - Mean = 0.79 µs. and Jitter = (Max – Mean) = 0.73 µs.
- End-to-end scheduling delay changed based on redesign of scheduler.
  - Mean = 1.9 µs. and Jitter = (Max – Mean) = 1.4 µs with 250 threads in R2RQ.
    - Jitter is caused by the cache.
  - O(1) R2RQ helped to reduce the jitter by approximately 1 µs!
2\textsuperscript{nd} Redesign Accomplishments

- **Partitioned R2RQ + Priority Encoder:**
  - Executes quickly and in constant time.
    - $O(1)$.
    - $\sim 24$ clock cycles.

- **Priority Encoder:**
  - Responsible for scheduling decision (priority level selection).
    - Functionality can be changed (hierarchical).

- **Conclusion:**
  - Scheduling overhead and jitter have been reduced.
  - Still need support for both SW and HW threads.
Third Redesign

- Provide services for “hybrid” threads.
  - SW threads – covered.
  - HW threads - ???.
- What else changes?
  - All policies deal with threads; which ones need to know “where” they are?
    - Management – allocation, creation, status of TIDs.
    - Scheduling – which TID should run when and where.
- Only the scheduler needs to be changed in order to “hybridize” the system!
  - Could these changes be encoded in the scheduling parameter...
Internals of Third Redesign

Thread_Data BRAM

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q?</td>
<td>1-bit</td>
<td>1 = Queued, 0 = Not Queued</td>
</tr>
<tr>
<td>NH:N7</td>
<td>8-bit</td>
<td>Ready-to-run queue next pointer</td>
</tr>
<tr>
<td>L0:L6</td>
<td>7-bit</td>
<td>Scheduling priority-level</td>
</tr>
<tr>
<td>P0:P7</td>
<td>8-bit</td>
<td>Ready-to-run queue previous pointer</td>
</tr>
</tbody>
</table>

Priority_Data BRAM

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NH:N7</td>
<td>8-bit</td>
<td>Priority-queue head pointer</td>
</tr>
<tr>
<td>T0:T7</td>
<td>8-bit</td>
<td>Priority-queue tail pointer</td>
</tr>
</tbody>
</table>

Param_Data BRAM

<table>
<thead>
<tr>
<th>Field</th>
<th>Width</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0:s31</td>
<td>32-bit</td>
<td>Scheduling parameter</td>
</tr>
</tbody>
</table>
3rd Redesign Results

• R2RQ of size 256 with 128 priority levels.
  • O(1).

• Synthesized on Virtex-II Pro 30:
  • 1,455 out of 13,696 slices, 973 out of 27,392 flip-flops, 2,425 out of 27,392 4-input LUTs, 3 out of 136 BRAMs.
  • Max. operating frequency of 119.6 MHz.

• Scheduling decision executes in fixed amount of time (~ 24 clock cycles).

• Uniform support for both SW and HW threads.
  • Thread type encoded in scheduling parameter
    – Low-overhead and jitter scheduling support for SW and HW threads!
3rd Redesign Accomplishments

• Encode HW/SW distinction in sched. parameter.
  • SP < 128 → (SW thread, priority)
  • SP >= 128 → (HW thread, address of cmd. reg.)

• Only the scheduler had to change!
  • Add Master-IPIF.
  • Add storage for larger SP.
  • ENQ – check SP.
    – SW threads: Add SW thread to R2RQ.
    – HW threads: Send “START” command to HW thread.

• Entire system is now truly “hybridized”.
Hybrid O(1) Timing Results

- Timing results for tests involving only SW threads remain the same as for non-hybrid O(1) scheduler.
- System is now “hybrid” compliant.
  - Operations can be used by both SW and HW threads.
Overall Accomplishments

- HW/SW co-design of scheduling services allows for:
  - Complexity of OS to be pushed into hardware.
    - Relieves CPU of duties
      - OS coprocessors do the work.
      - Less kernel code.
  - More parallelism within the system can be exploited.
    - Internals of scheduler are parallelized, and application and OS run in parallel.

- Breaking up management and scheduling through a standard interface allows for:
  - Greater maintainability
    - Mechanisms of each can be modified independently.
  - Separation of concerns
    - All OS components became “hybridized” by “hybridizing” the scheduler.
Results & Conclusion

- Three design iterations:
  - 1\textsuperscript{st} - Enable priority scheduling $\rightarrow$ O(n).
    - Still needed to improve performance.
  - 2\textsuperscript{nd} - Improve performance $\rightarrow$ O(1).
    - Needed to provide hybrid support.
  - 3\textsuperscript{rd} - Combine O(1) with “hybrid” features.
    - Provides FULL system support for ALL threads in a system!!!

- The Result:
  - OS services with generalized support for SW and HW threads.
    - Super low overhead and jitter.
    - Highly scalable due to on-chip BRAMs.
  - Standard interface defined for scheduling operations and data storage.
Questions???

- More information at…
- I can be contacted at…
  - jagron@ittc.ku.edu.