MTE325: Lab 1

Thursday June 12th, 2014

Kelvin Tse - kktse - 20421996
Mike Werezak - mwerezak - 20303777

Introduction

The purpose of this lab is to investigate different synchronization and interfacing techniques using a computer system built on the Altera Cyclone II evaluation board. The lab is split into two sections: phase 1 and phase 2. Phase 1 uses synchronization techniques to interface with user input and outputs. Phase 2 uses different synchronization techniques to service inputs while a background process is running.

Phase 1

Desired Functionality

The code takes inputs from the DIP switches, and blinks the state of the eight DIP switches sequentially on an LED or a seven segment display depending on a button input. The LED and seven segment display must blink at a rate of one blink per second. The code should respond to a button press at any time.

Implementation

The code is separated into two major sections: a main loop, and the push button interrupt service request. The main loop is a while loop which updates the state of the outputs, then waits until the timer interrupt indicates that the one second cycle has completed. The interrupt service request triggers when the push button is pressed, stores the state of the DIP switches and passes this information to the main loop. Figure 1 shows a simplified visual representation of how this is handled in the code.

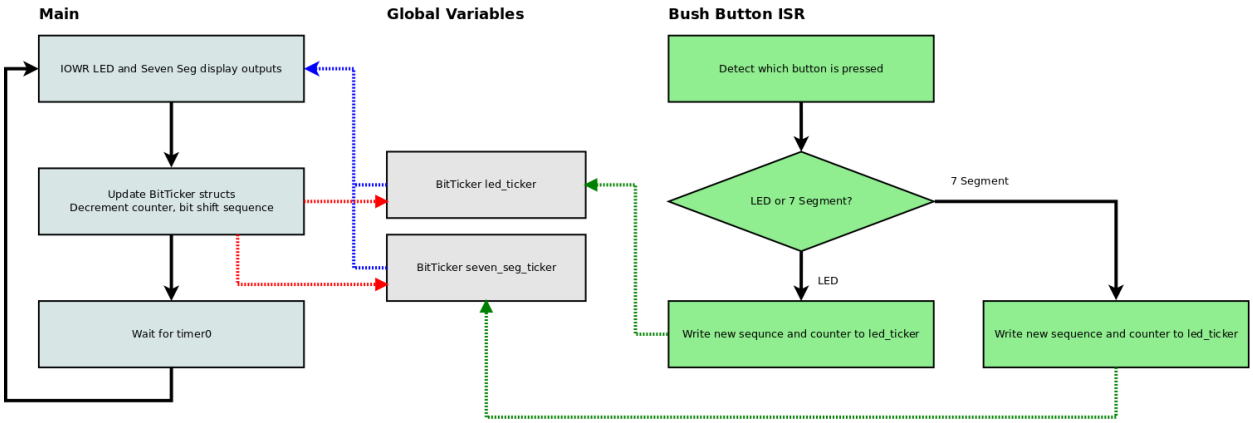

Figure 1: Simplified block diagram of phase 1 code

The `led_ticker` and `seven_seg_ticker` variables store information about what to output onto the LED and seven segment display respectively. Below is the code describing `struct BitTicker`.

```
typedef struct BitTicker {
    alt_u8 bit_sequence;
    alt_u8 counter; //when zero there are no bits left
} BitTicker;
```

The member `alt_u8 bit_sequence` stores the sequence of the eight DIP switches. The member `alt_u8 counter` keeps track of how many bits remain to be displayed on the LED or seven segment display. Pressing the push button triggers the interrupt service routine to update the two `BitTicker` structs with the appropriate data at the time of the request, which then is passed into the main loop to update the outputs.

The main loop is simply a while loop which reads the data stored in `led_ticker` and `seven_seg_ticker`, updates the LED and seven segment display outputs, updates the `led_ticker` and `seven_seg_ticker`, and waits the timer interrupt to trigger. Updating the `led_ticker` and `seven_seg_ticker` involves decrementing `alt_u8 counter` and shifting `alt_u8 bit_sequence`. Timer interrupts are handled in a separate interrupt service routine, which simply updates a flag to indicate that the elapsed time has passed.

Synchronization Methods

Two synchronization methods were attempted for phase one: tight polling and periodic polling. These two methods were used to service the push button. Using tight polling, a while loop would continuously check the state of the buttons.

Tight polling synchronization is blocking, meaning that it does not allow computer to process tasks while waiting. Responding to the button request is low latency, but this is at the cost of throughput and ignoring other requests. However, this solution is easy to implement, requiring only one line of code to handle the push button.

This synchronization method is incapable of fulfilling the requirements for phase 1. This is because the program is in either one of two states. When the program is waiting for pushbutton input, it cannot perform its task of flashing the LED or updating the 7-segment display. When it is updating the outputs, additional pushbutton input is ignored. Since we want to handle both push buttons without blocking the other, tight polling is insufficient. It is difficult to process requests in the background when the polling loop prevents any other code from executing.

Periodic polling for phase 1 used timer flag generated from a timer interrupt. The push button state is handled by an interrupt service request on the rising edge of the button signal. Servicing required outputs occurs only when the next timer interrupt is raised. In other words, the code processes requests once between the previous timer interrupt and next timer interrupt. Once the

previous request has been processed, the timer interrupt flag is polled before repeating the cycle. This synchronization worked the best for phase 1 since it reliably serviced the push buttons and the timer interrupt, versus using polling loops which could block tasks wishing to execute in parallel. Periodic polling allowed for a block of time for servicing the push buttons and outputs without blocking other requests.

Periodic Polling - Pros & Cons

Periodic polling is good since it guarantees that some time will be dedicated to processing, meaning that it will always do some useful processing. It is simple to implement, only polling when convenient to do so. Its ease of implementation also makes it simple to debug. Since code still runs in a sequential manner, it is easy to step through the code and identify where problematic behaviour occurs. Periodic polling is not the most efficient synchronization method, nor does it guarantee a maximum latency on requests. If the processing time between polling is less than the timer interrupt interval, then there is time wasted to polling and not responding to requests. If the processing time between polling is greater than the timer interrupt interval, then there system will lag the timer interrupts. For phase one, the processing time between polling is less than the timer interrupt interval. The max latency for responding to push button requests is one second.

Testing Strategy

The implementation of phase 1 was completed in sections. The required functionality was split into sub-features which needed to be implemented. For phase 1, those features are controlling the LED, controlling seven segment display, using the timers and detecting push buttons. Each of these features were initially implemented independently from each other and tested independently. These features were combined to perform specific tasks, and were tested as subtasks. For example, using the push button to blink the LED. This strategy ensured that each individual component worked, and allowed narrowed down problems to logic or behavioural problems.

This strategy catched a problem waiting for an interrupt call within an interrupt service request. In an early version of phase one, the LED blink sequence was implemented within the push button interrupt service routine. The blink LED sequence was implemented using the timer interrupt. During a midpoint implementation check, the system appeared to enter an infinite loop when it was supposed to wait for the timer. Using this testing strategy, we identified that implementation of the timer and LED blink worked individually outside the interrupt service routine, eliminating an implementation error in these two areas. At this point, we were able to identify it is either an interaction between the LED and timer or some interaction within the interrupt service routine. We removed the LED from the ISR routine and kept the timer alone in the interrupt service routine along with some debug messages. At this point, we identified that the timer did not work within the interrupt service routine. Knowing that interrupts are disabled during the interrupt service routine, we identified that the timer flag would never be changed if the timer interrupt could never be triggered. The solution to this problem was to come up with different logic since this strategy was limited by this behaviour.

Phase 2

Synchronization Methods

The first implementation used occasional polling to check for the EGM trigger. This synchronization technique is passive. The background process is executed for a number of iterations. Once the background task is done, the status of the EGM is checked. Based on the EGM status, the appropriate response is made to handle the EGM trigger. The process repeats until all the iterations of the background task are complete. The pseudocode for this implementation is below.

```
while(true) {
      background(iterations); //perform background task
      egm_state = check_egm(); //check the status of the EGM
      respond_to_egm(egm_state); //respond with last known EGM state
}
```

This method guarantees that the background task will not be starved of resources. It is also relatively simple to implement since it does not require interrupts to implement.

The second implementation used interrupt synchronization, or active synchronization. The background task runs continuously in main, while the EGM raises an interrupt which triggers an interrupt service routine to execute. The EGM is handled within the interrupt service routine. Once the interrupt service routine is complete, then the background task continues executing. This process is described by Figure 2.

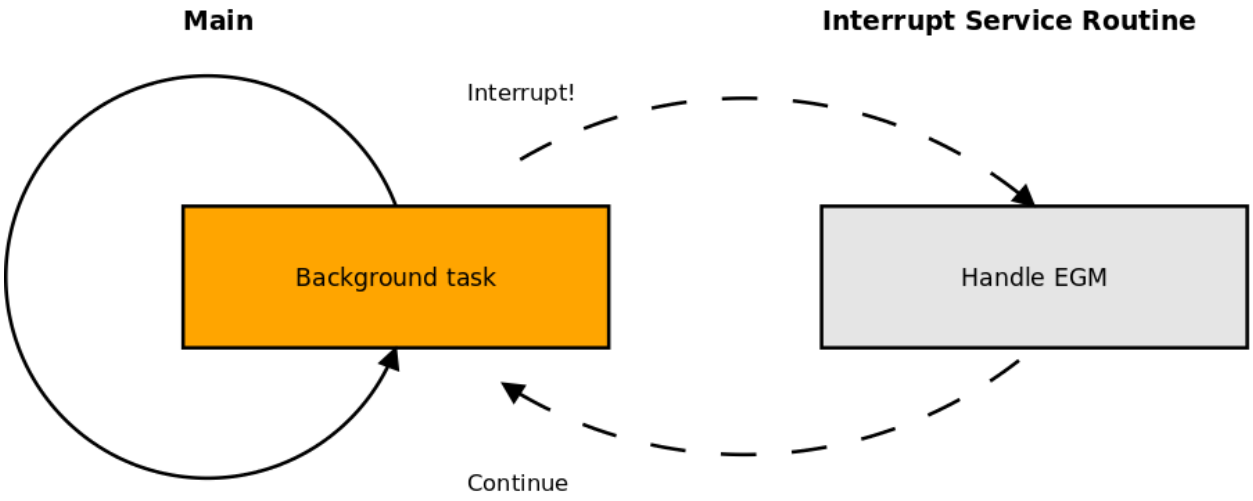

Figure 2. Interrupt Synchronization

This method guarantees that the EGM is handled, so long as the period between EGM triggers does not exceed the time taken to handle the EGM.

Evaluation Parameters

The synchronization methods are evaluated on a runtime of 16000 iterations of the background task. Each method is ran across a sweep of different granularity, period and duty cycle. Evaluation of each synchronization method is based on three parameters: total elapsed time, EGM response time, and missed EGM events.

The primary metric of interest is missed events. This is the number of events generated by the EGM for which the synchronization method could not respond to correctly before the next event. Both methods of synchronization are written with the goal of minimizing the number of missed EGM triggers. In other words, targeting zero missed events.

Total elapsed time is a metric of throughput. The number of tasks completed is only a meaningful statistic if the time allotted to complete the tasks is fixed. We found it was easier to measure the amount of time to complete a fixed number of tasks instead.

EGM response time is a metric of latency. Lower time to respond indicates lower latency. This criteria was not included in our discussion of the results, due to space constraints. Analysis that includes latency can be found in the appendices.

Results

Missed events versus period and grain size demonstrates the varying characteristics of occasional and interrupt synchronization. Figure 3 shows a plot of missed events versus period and grain size at a fixed duty cycle of 50% using occasional polling. For occasional polling, number of missed events increases with as grain size increases and period decreases. The worst case scenario for occasional polling was 668 missed events with a 1600 grain size and a period of 164 us.

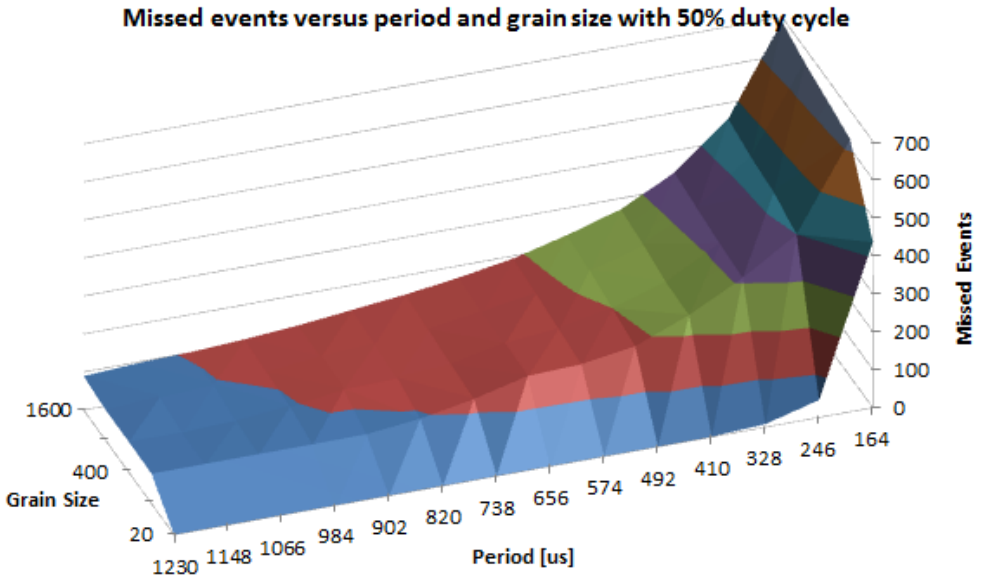

Figure 3. Occasional polling - missed events vs. period and grain size

In contrast to interrupt synchronization, the number of missed events is two orders of magnitude lower than occasional polling. Figure 4 shows a plot of missed events versus period and grain size for interrupt synchronization. In this case, the worst case for missed events was 3 with grain size of 20, a period of 164 us and 50% duty cycle. Missed events increases as grain size decreases and period decreases.

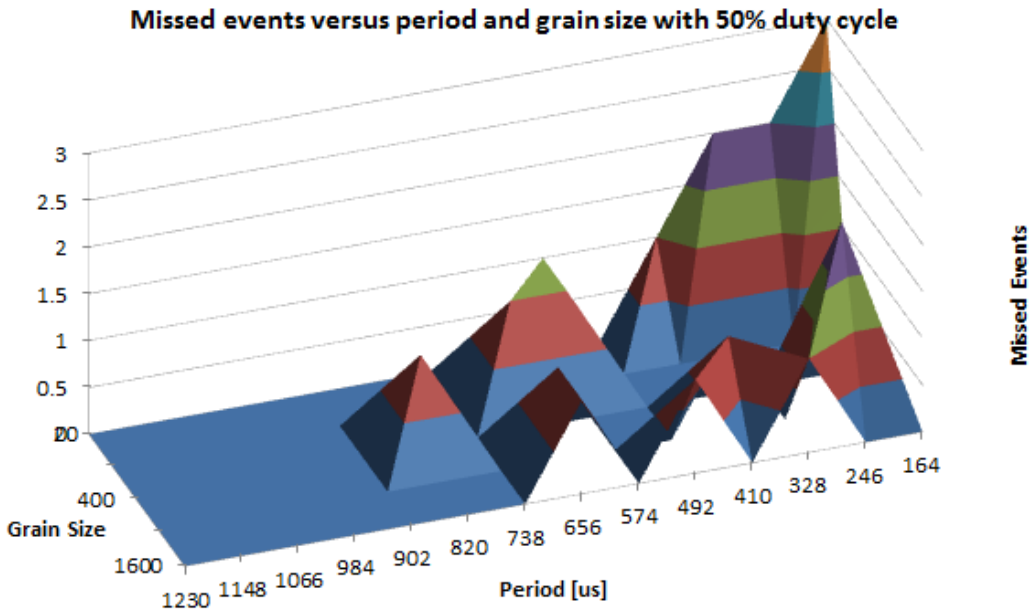

Figure 4. Interrupt synchronization - missed events vs. period and grain size

For occasional polling, duty cycle has minimal effect on the number of missed events. Figure 5 shows a plot of missed events versus duty cycle and period. Missed events remains the same as duty cycle increases and increases as period decreases. The worst case performance was 648 missed events with a period of 164 us.

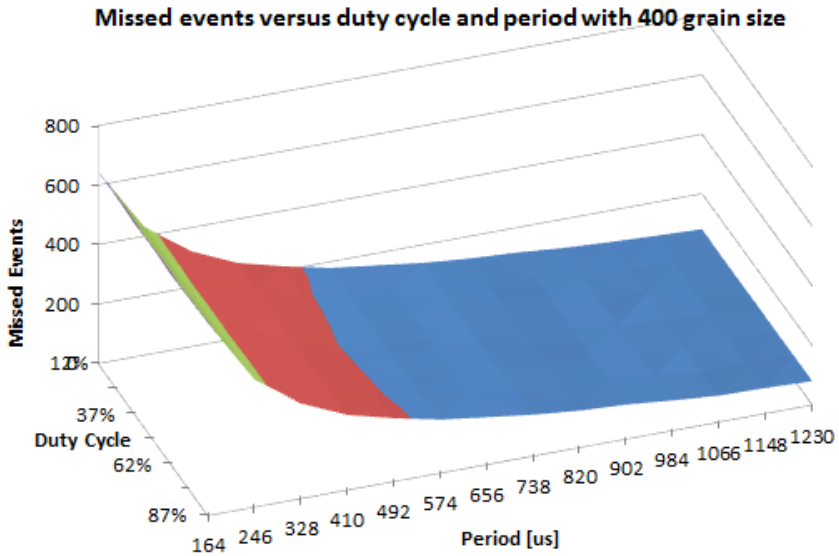

Figure 5. Occasional polling - missed events vs. duty cycle and period

Interrupt based synchronization consistently misses zero events, except for at the two extremes of duty cycle at a period of 164 us. Figure 6 shows a plot of missed events versus duty cycle and period for interrupt based synchronization. The worst case scenarios were 162 and 158 missed events with a period of 164 us at 12% and 87% duty cycle respectively. These results make sense for interrupt synchronization. At the two extremes of duty cycle, the time between a rising-edge the following falling-edge is extremely small, and vice-versa. This may be occurring because the time taken to handle the first interrupt exceeds the time the time when the next interrupt is triggered. This will only occur at the extremes of the duty cycle. However, even it the worst case scenario, occasional polling misses four times as many events than interrupt based synchronization.

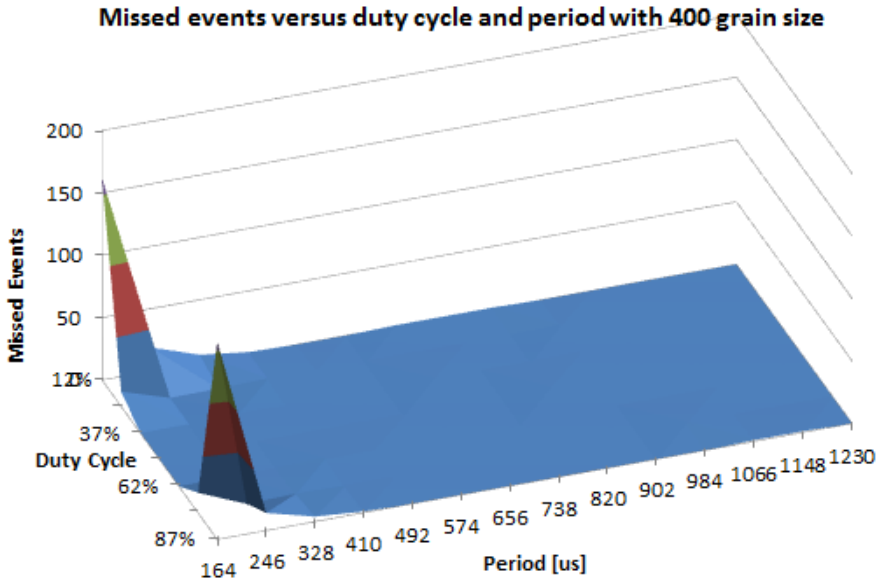

Figure 6. Interrupt synchronization - missed events vs. duty cycle and period

As a side note, occasional polling was found to have a higher throughput of background tasks than interrupt based synchronization. This can be seen by comparing Figures 8 and 15 in the appendices. This difference was more pronounced at lower periods (~160 ms to complete 16k iterations for interrupts, 107 ms for occasional polling), and less pronounced at higher periods (both methods were nearly the same for throughput). This was as expected, as the overhead for interrupts is incurred when the interrupt is triggered.

Findings

To minimize the number of missed events, interrupt based synchronization performed significantly better than occasional polling. In some cases, interrupt based synchronization missed approximately 100 times less events than occasional polling. Interrupt based synchronization often met the ideal target of zero missed events when sweeping across duty cycle, period and grain size. The only identified weakness of interrupt based synchronization is at high and low duty cycles, where the number of missed events significantly increase. However, in all cases, interrupt based synchronization performed better.

Contribution and Reflection

Mike

In pair programming terminology, I took on the role of the “driver” for this lab. This meant that I was responsible for writing the code that would meet the goals of this lab. This involved identifying the logic that each software component in our design would need to follow, and producing the C code that implemented that logic, as well as collaborating with Kelvin for debugging.

In addition to that, I configured the Cyclone II to include the EGM in our “myESystem” hardware for phase II, which involved importing the EGM hardware description into Quartus II, wiring the new components, verifying that the EGM was working as intended and dealing with any issues.

For me, the most important learning outcome from this lab was gaining further familiarity with embedded systems and challenges involved in developing embedded software, as well as hardware development tools such as Quartus II. In addition, I felt I gained a greater appreciation for the challenges involved in synchronizing hardware.

Kelvin

In the pair programming practice, I took on the ‘observer’ role during the course of this lab. I contributed to how to proceed through the lab and made design proposals in relation to code implementation. Specifically, in phase 1, I proposed which features to implement and which features to implement next to culminate to the lab deliverables. This involved checking the lab manual for requirements, searching for Nios II documentation and collaborating with Mike to measure progress and debug code. Lastly, I created the diagrams used for explaining the implementation in this report.

The most important learning outcome was learning how to communicate engineering requirements in plain language, especially programming requirements. In this lab, extracting requirements was a two fold process: first understanding the requirement in the lab manual, then turning that into an implementation requirement. In this lab, I acted as a mediator between the first and second steps in this process. As a result of this lab, I am able to more effectively interpret requirements and translate them into an actionable item.

Appendix A - Occasional Polling Results

This appendix presents results and analysis for occasional polling that were deemed non-essential for the report, but are given here for completeness.

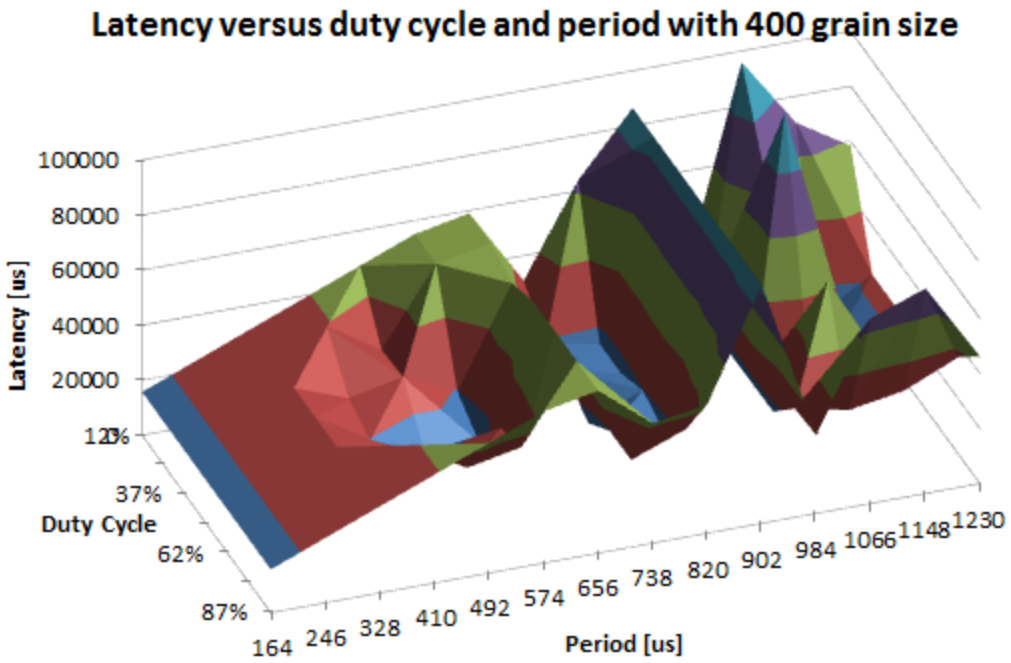

Figure 7. Occasional polling - latency vs. duty cycle and period

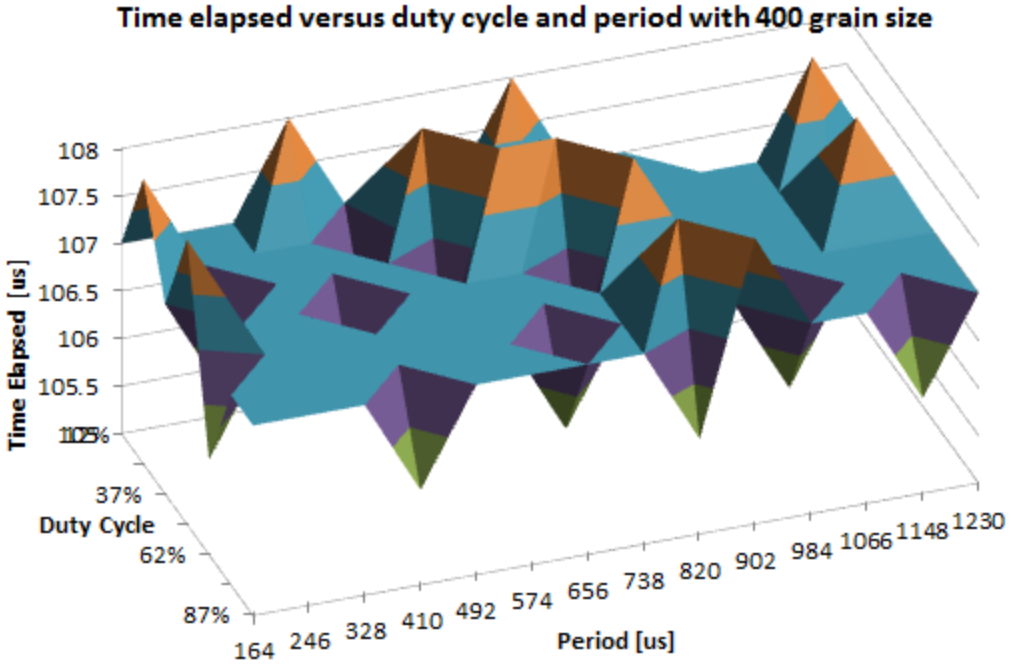

Figure 8. Occasional polling - time elapsed vs. duty cycle and period

Note: Time elapsed is relatively constant (note scale) for periodic polling. This makes sense.

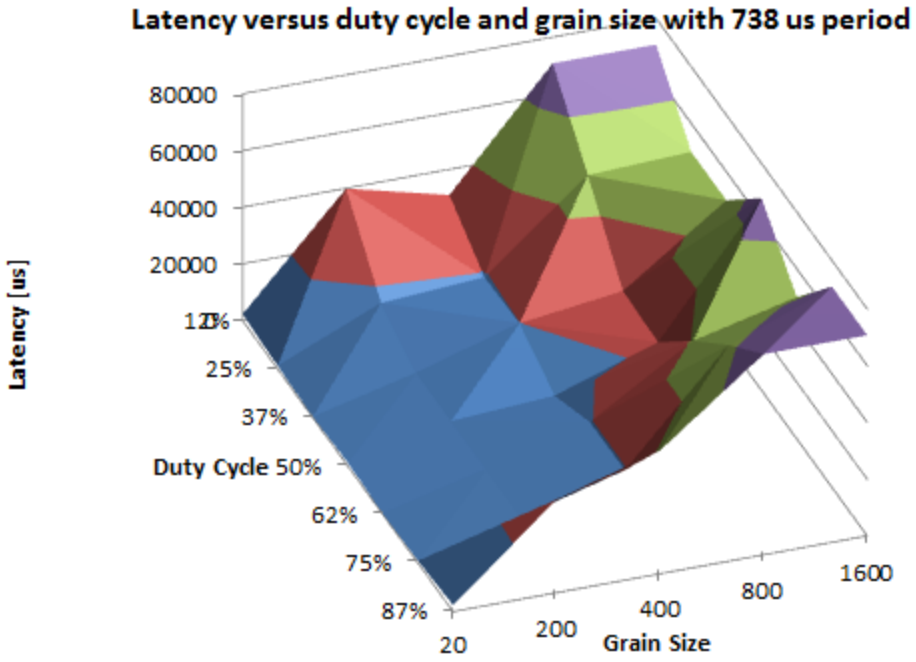

Figure 9. Occasional polling - latency vs. duty cycle and grain size

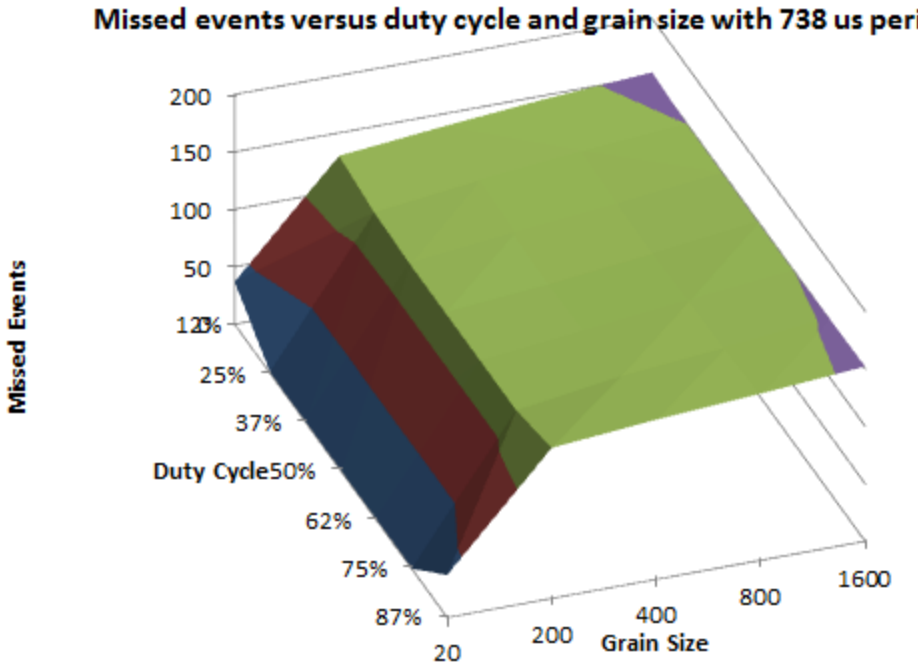

Figure 10. Occasional polling - missed events vs. duty cycle and grain size

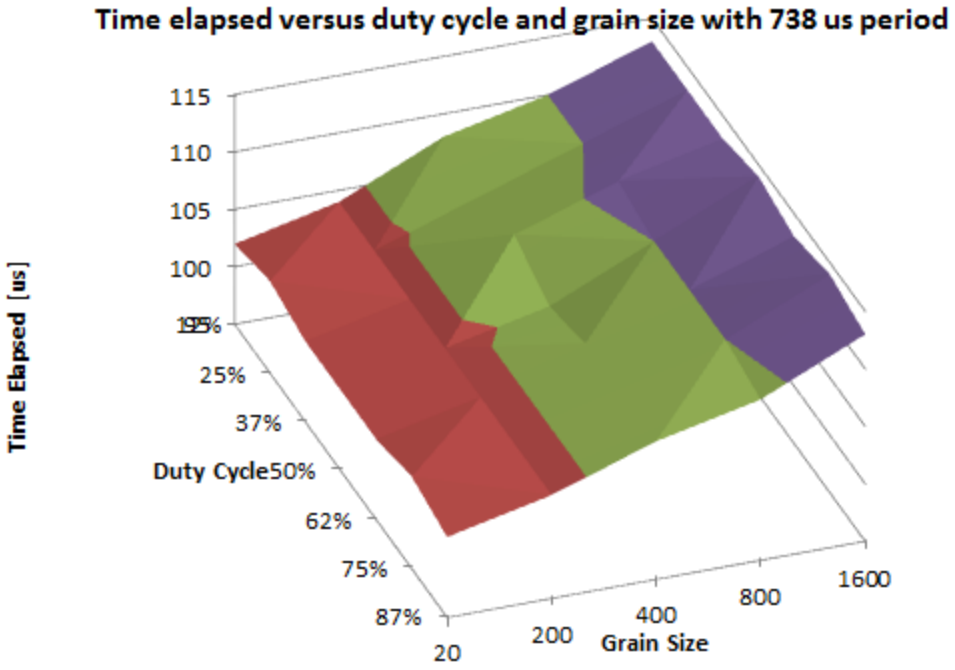

Figure 11. Occasional polling - time elapsed vs. duty cycle and grain size

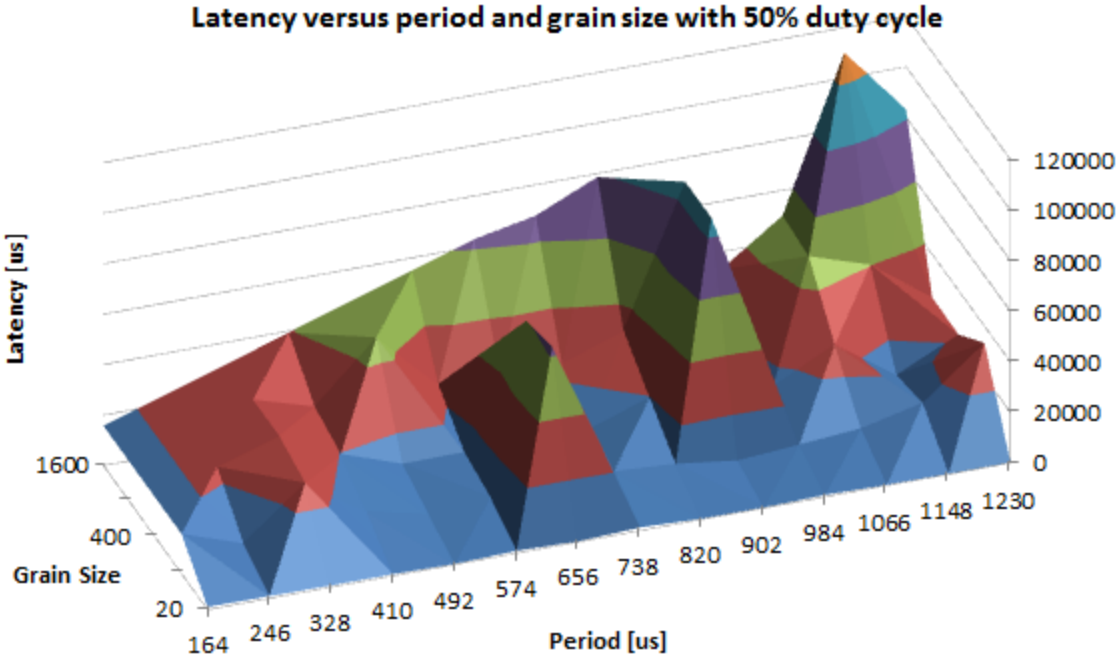

Figure 12. Occasional polling - latency vs. period and grain size

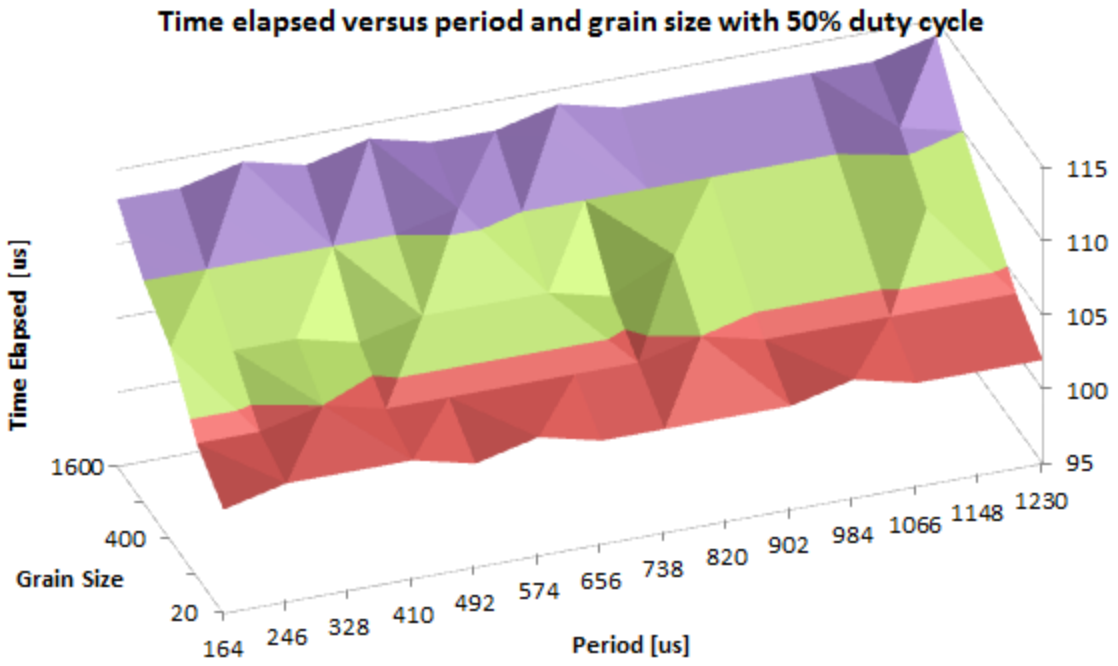

Figure 13. Occasional polling - time elapsed vs. period and grain size

Appendix B - Interrupt Synchronization

This appendix presents results and analysis for interrupt based synchronization that were deemed non-essential for the report, but are given here for completeness.

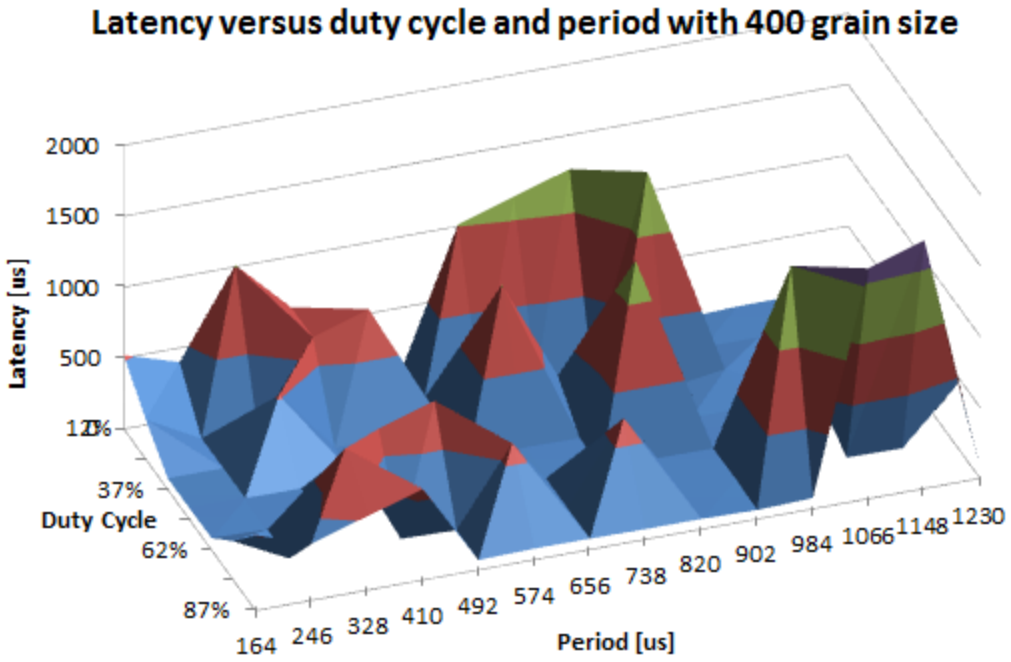

Figure 14. Interrupt synchronization - latency vs. duty cycle and period

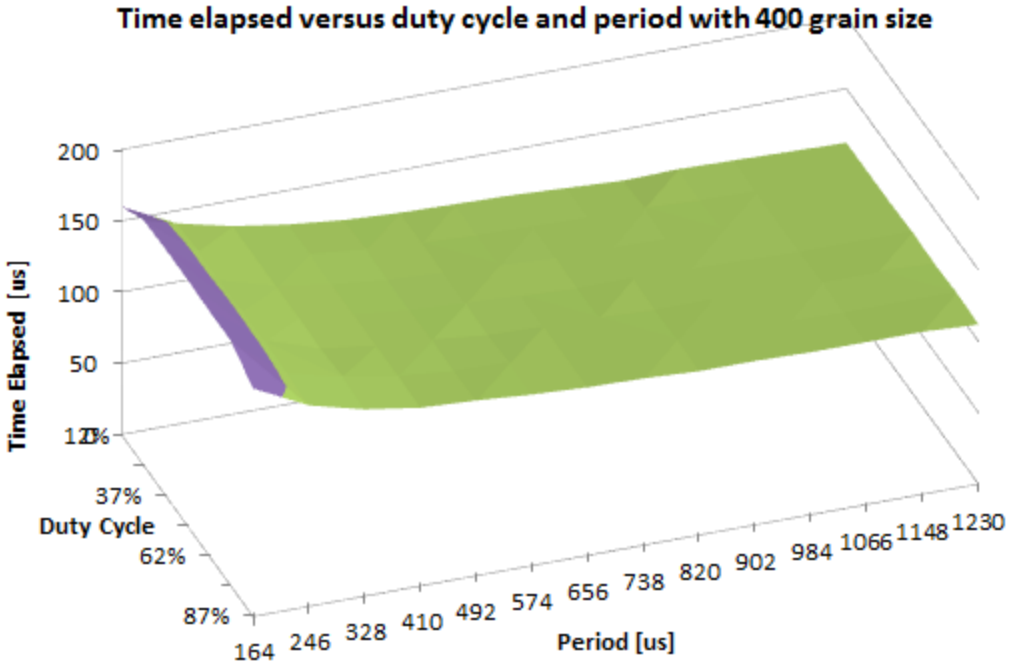

Figure 15. Interrupt synchronization - time elapsed vs. duty cycle and period

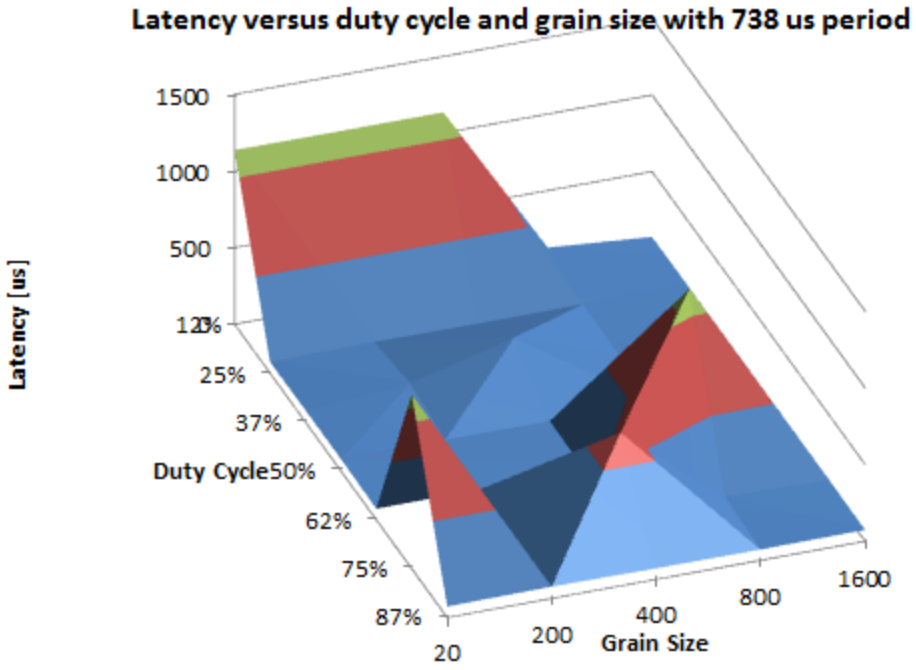

Figure 16. Interrupt synchronization - latency vs. duty cycle and grain size

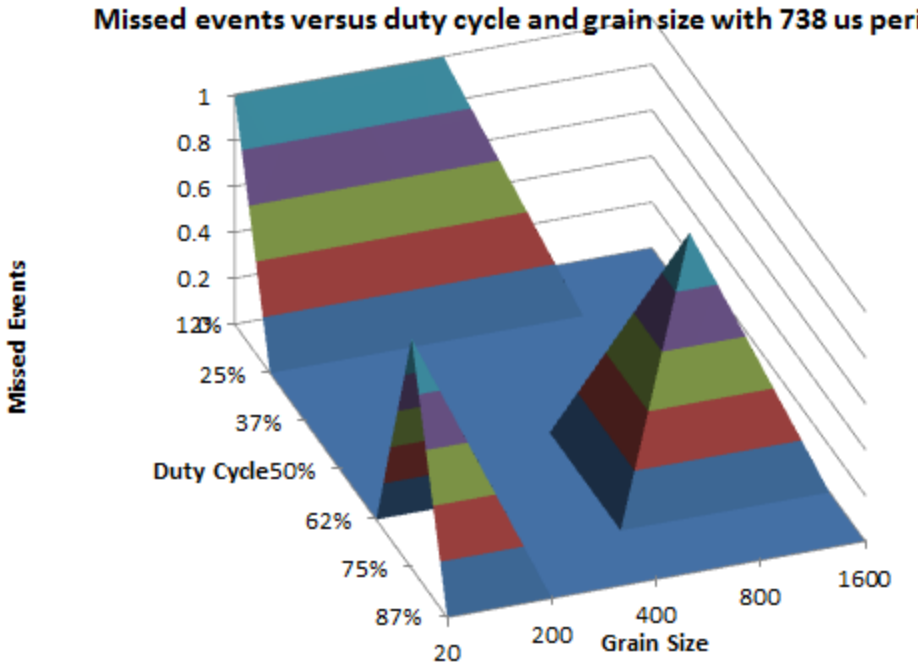

Figure 17. Interrupt synchronization - missed events vs. duty cycle and grain size

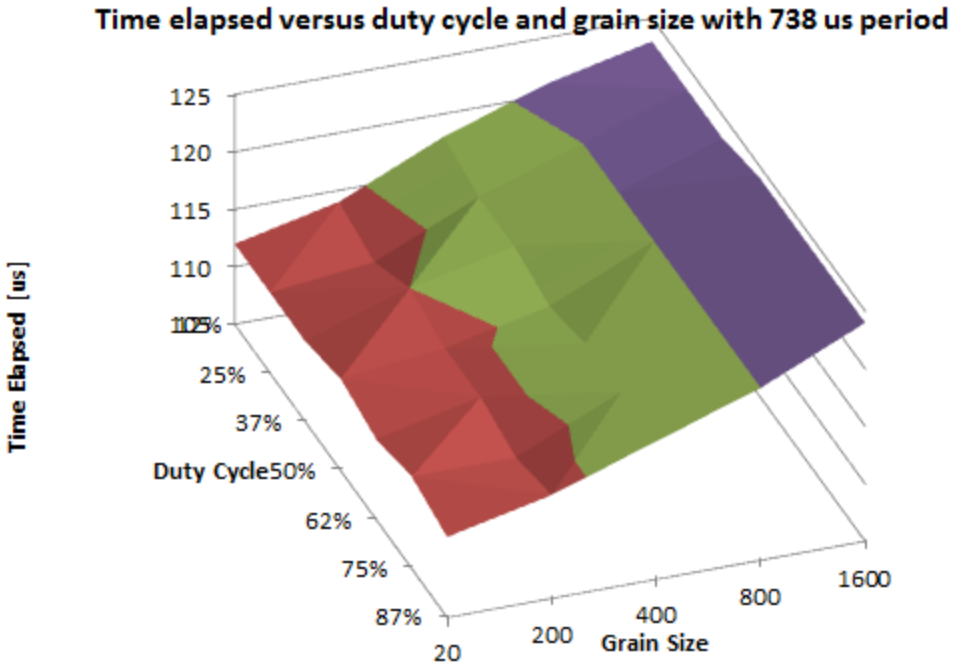

Figure 18. Interrupt synchronization - time elapsed vs duty cycle and grain size

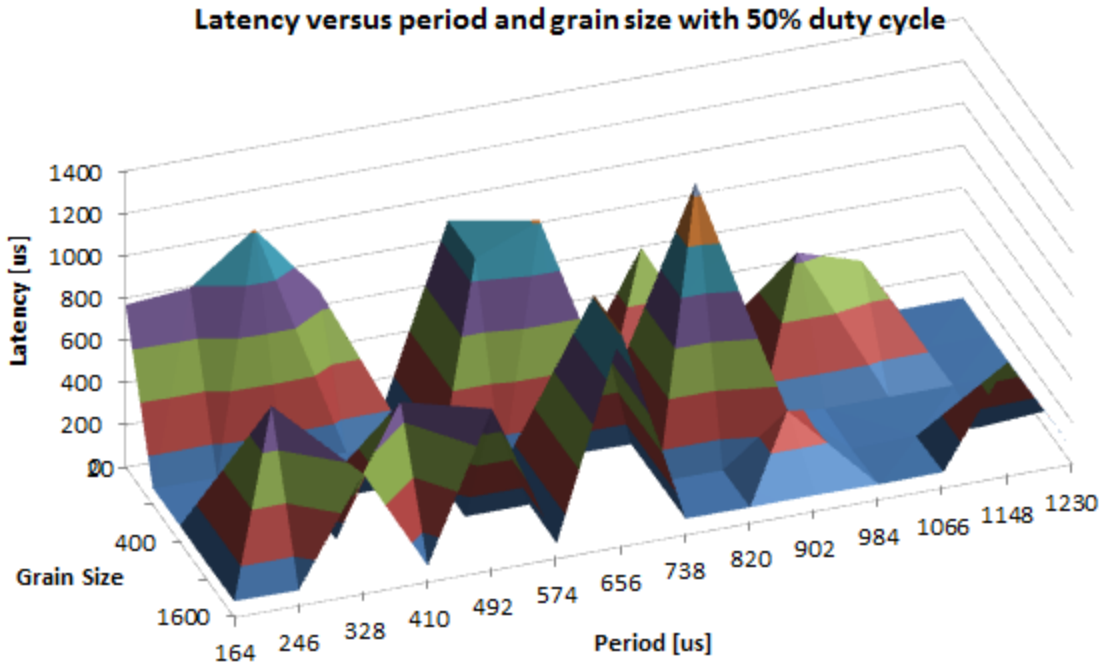

Figure 19. Interrupt synchronization - latency vs. period and grain size

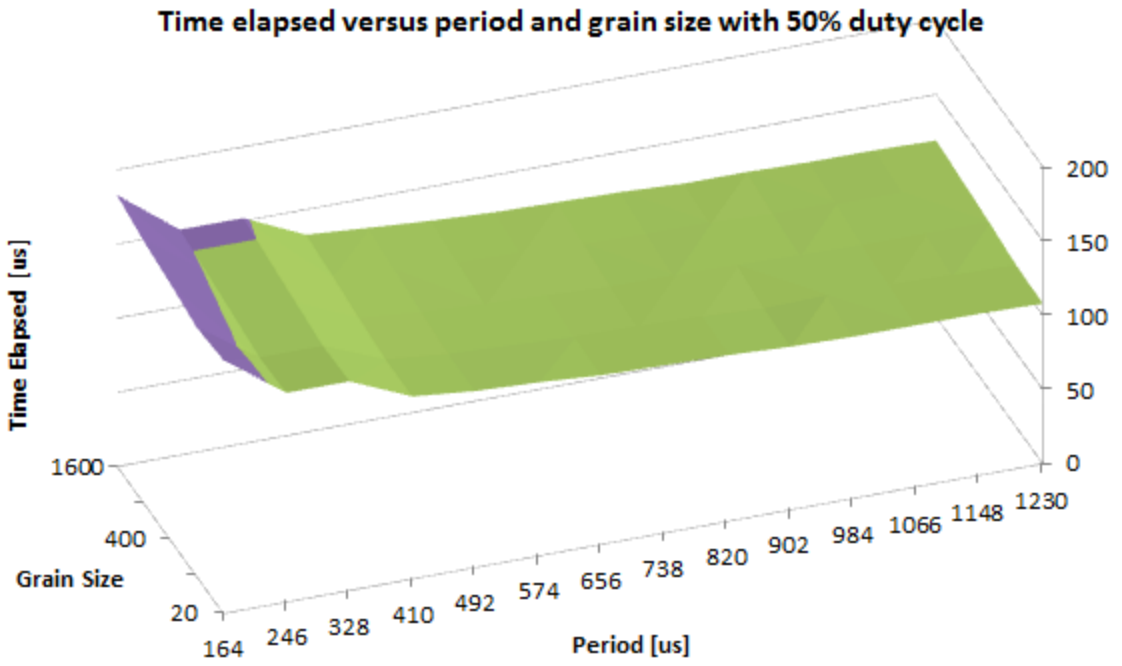

Figure 20. Interrupt synchronization - time elapsed vs. period and grain size