Part II – Design and Implementation of an Interrupts Simulator

We built a C++ interrupt simulator and ran 20 test cases to see how context save/restore time and ISR execution time affected it. Every testcase used the same program trace (so the useful CPU work is constant at 2,972 ms). The only variables we changed were the context save/restore to 10, 20, 30 ms and ISR to values from 40 to 200 ms. For each test case we calculated the Makespan, CPU time, Kernel Overhead, and Kernel %.

The makespan is computed as Makespan = CPU (2,972 ms) + Kernel Overhead, and the kernel fraction is Kernel% = Overhead / Makespan. This lets us compare the measured logs against predicted totals for simple consistency checks. For this specific trace, the overhead is linear: Overhead (ms) ≈ 60 × ISR + 90 × Context + 240. This represents the switch, save, vector lookups, ISR, IRET, restore, switch, and the counts reflect how many times each step appears in the trace.

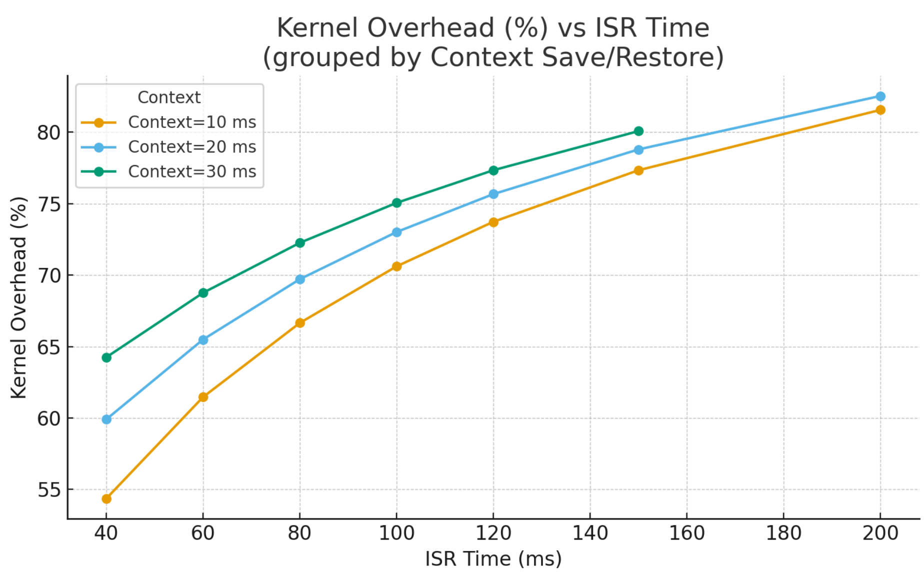
Why these coefficients

- The coefficient 60 multiplies ISR because the ISR body executes 60 times in the log. The coefficient 90 multiplies Context because the total number of context saves and restores across all interrupt episodes adds up to 90. The constant +240 adds the fixed 1-ms steps (mode switches, vector lookups, and IRET) over the whole run. .
- At a fixed context time, increasing the ISR by +20 ms adds about 60 × 20 = 1,200 ms of overhead, so Kernel% rises by a few percentage points. At a fixed ISR time, increasing the context by +10 ms adds about 90 × 10 = 900 ms of overhead, which also produces a noticeable increase in Kernel%.
- For every run, we grepped the execution.txt logs to confirm ISR durations and the context save/restore lines, saved each run's output as a separate snapshot, and compared those measured values to the closed-form prediction. This ensured that the reported Makespan, Overhead, and Kernel% were consistent.

Analysis of Simulation Results

The simulation logs show the relationship between interrupt handling parameters and overall system performance. Analysing the simulation results shows a linear growth in kernel overhead as the ISR duration increases. For any fixed context time, the Kernel% rises with ISR duration which means that a longer ISR execution time directly increases the proportion of time the CPU spends in kernel mode. The data shows that the most gains come from using shorter ISRs and lighter context paths.

What the chart shows (Kernel% vs. ISR)



Context = 10 ms

ISR 40: 54.36% kernel
ISR 80: 66.65% kernel
ISR 200: 81.55% kernel
Context = 20 ms (same ISR)
ISR 40: 59.90%
ISR 80: 69.71%
ISR 200: 82.53%
Context = 30 ms
ISR 40: 64.24%
ISR 80: 72.26%
ISR 200: 83.61%
(Exact values are in the CSV right at the end)

Analysis of ISR Activity Time

The ISR execution is an important part of interrupt handling, and the data below shows that a more complex or time-consuming ISR increases the total system overhead.

When we had a constant context time of 10ms, increasing the ISR time from 40ms to 100ms caused the Makespan to jump by 54% (from 6512ms to 10049ms). The kernel overhead percentage increased from 54.36% to 70.60%. This is because a longer ISR uses more CPU time in kernel mode which creates a bottleneck.

This means that when an ISR's execution is too long, it becomes a bottleneck. This is because the CPU is only busy handling the interrupt and cannot process any other tasks during this time. The data demonstrates a nearly linear relationship between the ISR time and the Makespan, which shows that when the ISR is not used efficiently it can affect the system in bad way.

Impact of Context Save/Restore Time

The data shows a proportional relationship between the context save/restore time and overall system performance. As the time required to switch between user and kernel modes increases, the total Makespan of the program also increases. When the context time was increased from 10ms to 20ms (with a constant ISR time of 40ms), the Makespan increased by 12.8% (from 6512ms to 7349ms). This adds a 9.4% to the kernel overhead, showing that for every millisecond added to a context switch, the system's overhead is affected. When the context time was increased to 30ms, the Makespan for the same ISR time rose to 8249ms, representing a 26.6% increase from the original 10ms context time. This is because a longer context switch means the CPU is performing non-productive work. This analysis shows that even a small increase in context time has an impact on total execution time. Therefore, a kernel must prioritize reducing the time spent on context switches to maintain efficiency.

Combined Effects and Conclusion

The interaction between the two parameters shows an increasing effect. The worst-performing simulation was the one with the highest values for both parameters (execution_ctx30_isr200.txt), which had a Makespan of 14040ms and an overhead of 83.61%. This shows an effect where a slow context switch with a complex ISR leads to a massive increase in completely non-productive CPU time.

In conclusion, this shows the importance of reducing interrupt overhead in an operating system design. Both context save/restore time and ISR activity time are very important factors and a system that cannot handle interrupts efficiently will have poor performance, even if the hardware is fast. The data from this analysis shows that time spent on context switching and complex ISRs can quickly add up a program's total execution time.

Excel File With All the Test Case Values and Calculations:

File name	Context Save (ms)	Context Restore (ms)	ISR Time (	Final Timestamp	Last Duration	Makespan (ms)	CPU time	Overhead	Kernel%
execution_ctx10_isr40.txt	10	10	40	6449	63	6512	2972	3540	0.543611794
execution_ctx10_isr60.txt	10	10	60	7649	63	7712	2972	4740	0.614626556
execution_ctx10_isr80.txt	10	10	80	8849	63	8912	2972	5940	0.666517
execution_ctx10_isr100.txt	10	10	100	10049	63	10112	2972	7140	0.706091772
execution_ctx10_isr120.txt	10	10	120	11249	63	11312	2972	8340	0.737270156
execution_ctx10_isr150.txt	10	10	150	13049	63	13112	2972	10140	0.773337401
execution_ctx10_isr200.txt	10	10	200	16049	63	16112	2972	13140	0.815541212
execution_ctx20_isr40.txt	20	20	40	7349	63	7412	2972	4440	0.599028602
execution_ctx20_isr60.txt	20	20	60	8549	63	8612	2972	5640	0.654900139
execution_ctx20_isr80.txt	20	20	80	9749	63	9812	2972	6840	0.697105585
execution_ctx20_isr100.txt	20	20	100	10949	63	11012	2972	8040	0.730112604
execution_ctx20_isr120.txt	20	20	120	12149	63	12212	2972	9240	0.75663282
execution_ctx20_isr150.txt	20	20	150	13949	63	14012	2972	11040	0.787896089
execution_ctx20_isr200.txt	20	20	200	16949	63	17012	2972	14040	0.825299788
execution_ctx30_isr40.txt	30	30	40	8249	63	8312	2972	5340	0.642444658
execution_ctx30_isr60.txt	30	30	60	9449	63	9512	2972	6540	0.687552565
execution_ctx30_isr80.txt	30	30	80	10649	63	10712	2972	7740	0.722554145
execution_ctx30_isr100.txt	30	30	100	11849	63	11912	2972	8940	0.750503694
execution_ctx30_isr120.txt	30	30	120	13049	63	13112	2972	10140	0.773337401
execution_ctx30_isr150.txt	30	30	150	14849	63	14912	2972	11940	0.800697425