Flappy Bird

IS1200 Advance Project Analysis Report

Panayiotis Charalambous - Marios Stavrou (group 90)

September 2025

Project Description

We have implemented a version of the game *Flappy Bird* that runs on our DTEK-V board. The game renders graphics through a VGA interface, generates pipes at vertical positions, and allows the bird to be controlled via the board's button using interrupts. The program updates game states and graphics in real time using interrupts from the internal clock. Furthermore, we display live score on the 7-segment displays. We have achieved smooth frame updates on the VGA display, responsive input and correct collision detection with the pipes and screen boundaries.

Methodology

We gathered performance data using the approach from **Introduction to DTEK-V Hardware Performance Counters**, issuing inline assembly to access the counters. We implemented two helpers—`clear_counters`, which resets all counters to zero, and `read_counters`, which captures their current values. For each run, we invoked `clear_counters` immediately before the game begins and called `read_counters` right after a loss (end of the game). Counter values were printed to the terminal using the lab's 2 printing routines. We recorded all ten counters—`mcycle`, `mcycleh`, `minstret`, and `mhpmcounter3-9`. After sampling, we imported the results into Microsoft Excel to compute derived metrics.

Experiments were performed using our Flappy Bird implementation, where workload is defined by the **number of pipes passed** ($N = 1, 2, 4, 8$). For each (N we collected two measurements: (a) an unoptimized build compiled with `-O0`, and (b) an aggressively optimized build compiled with `-O3`.

Results

	Unoptimized (O0)				**Optimized (O3)**			
Metric	**N=1**	**N=2**	**N=4**	**N=8**	**N=1**	**N=2**	**N=4**	**N=8**
Game Time (s)	33.16	41.58	61.85	102.19	10.61	13.98	20.78	34.31
IPC	0.6876	0.6875	0.6873	0.6871	0.7173	0.7152	0.7129	0.7114
I-cache hit rate (%)	99.998	99.998	99.998	99.998	99.998	99.998	99.998	99.998
D-cache hit rate (%)	99.36	99.36	99.37	99.37	98.50	98.51	98.51	98.52
Data-hazard stalls (%)	24.77	24.81	24.87	24.92	3.91	3.90	3.88	3.87
ALU-stall ratio (%)	0.00	0.00	0.00	0.00	0.0003	0.0002	0.0001	0.0001
#Memory instructions	213 712 453	267 876 294	398 287 904	657 827 784	39 256 483	51 810 401	77 168 262	127 631 926
#Instructions	683 989 814	857 480 137	1 275 205 453	2 106 526 744	228 365 850	299 929 747	444 487 919	732 171 455
#D-cache stalls	112 511 284	140 548 861	207 964 097	342 104 161	21 076 928	27 739 214	41 258 898	67 733 950
#Cycles	994 708 097	1 247 267 839	1 855 427 476	3 065 723 852	318 363 541	419 370 246	623 455 321	1 029 190 702

IPC (Instructions Per Cycle) achieved

We achieve a peak IPC of 0.7173, running with compiler optimizations and problem size N=1. As the problem size grows, the IPC achieved declines marginally. The lowest IPC we observe is 0.6871, for the unoptimized version and N=8.

What is preventing you from reaching a peak (IPC=1) of your application?

The main reason is data cache performance and data hazard stalls. We observe that for IPC=0.7173 (best case scenario), the D-cache hit rate is 98.50% while the data hazard stalls 3.91%. This translates to 1.5% of cycles used for cache stalls, and 3.91% of cycles used for data hazard stalls. This is total of 17223467 cycles missed. Instruction cache performance (>99.9%) and ALU stall ratio (<0.001%) have negligible impact to performance.

What is the impact of compiling whith no optimizations (-O0) and heavy optimizations (-O3) on the application and on the above metric?

Comparing optimized and unoptimized execution for the same problem size, we notice that the execution time for the unoptimized version is significantly larger. Practically, for our game, this means that the frames render much slower and the game seems chunkier. On average, the time the unoptimized version takes is 3 times larger than the optimized version.

Observing our metrics, this mainly happens due the significant increase on data hazard stalls when running with no optimizations (average 24.84%). This is due to the fact that our game logic uses a lot of branching to determine the sate of the game. For example whether a collision has occured, or when a pipe is passed through. Without any optimizations the miss-prediction penalties are dramatically higher.

Furthermore, we also notice that in the unoptimized version, the number of instructions and memory instructions executed are consideerably higher, with non-memory instruction being 5 times more and memory instructions 3 times more.

This happens mainly because when the compiler runs with no optimizations translates C code literally line by line. This means redundant loads and stores are left in (for example reading the same array element multiple times), temporary variables often spill into memory instead of being kept in registers. Moreover loop unrolling is not applied, for instance in the nested for loops used to display pixels on the screen. This results to many more instructions being executed, and more memory accesses because of poor register reuse.

In contrast, when the compiler uses heavy optimization, repeated calculations are done once, values that don't change inside the loops are lifted out, values are kept in registers instead of memory and fewer, more efficient instructions are executed. Overall, far fewer total instructions, and fewer memory instructions are executed because loads/stores are minimized

What architectural improvements you would do to the processor and why you would do them in order for the program to run faster?

There are a lot of improvements on the processor that could help the program run faster. Firstly, we could increase D-cache size or associativity which will reduce D-cache miss penalty.

Secondly, our game loop collision checks involve repeated conditionals. A simple static predictor could be replaced by a dynamic branch predictor to reduce pipeline flushes. This will keep the pipeline filled, and avoid wasted cycles.

Lastly, issuing two instructions per cycle, or increase processor's cores to allow parallel processing. This would significantly improve tasks like graphics updates (lots of independent pixel stores).