PLATFORM-LEVEL PERFORMANCE ANALYSIS

Bus based systems add another layer of complication to performance analysis platform level performance involve much more than the CPU we often focus on the CPU because it processes instructions but any part of the system can affect total system performance. More precisely the CPU provides an upper bound on performance but any other part of the system can slow down the CPU merely counting instruction execution times is not enough.

Consider the simple system we want to move data from memory to the CPU to process it to get the data from memory to the CPU we must:

- Read from the memory
- Transfer over the bus to the cache
- Transfer from the cache to the CPU

The time required to transfer from the cache to the CPU is included in the instruction execution time, but the other two times are not.

Bandwidth as performance

The most basic measure of performance we are interested in is bandwidth the rate at which we can move data ultimately if we are interested in real time performance we are interested in real time performance measured in seconds but often the simplest way to measure performance is in units of clock cycles however different parts of the system will run at different clock rates. We have to make sure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds

Bus bandwidth

Bandwidth questions often come up when we are transferring large blocks of data for simplicity let's start by considering the bandwidth provided by only one system component the bus consider and image of 320 pixels with each pixel composed of 3 bytes of data this gives a grand total of 230400 bytes of data if these images are video frames, we want to check if we can push one frame through the system within the 1/30 sec that we have to process a frame before the next one arrives.

Let us assume that we can transfer one byte of data every microsecond which implies a bus speed of 1 MHz. In this case we would require 230400 µs = 0.23 sec to transfer one frame that is more than the 0.033 sec allotted to the data transfer we would have to increase the transfer rate by 7xto satisfy our performance requirement.

We can increase bandwidth in two ways we can increase the clock rate of the bus or we can increase the amount of data transferred per clock cycle for example if we increased the bus to carry four bytes or 32 bits per transfer we would reduce the transfer time to 0.058 sec if we could also increase the bus clock rate to 2 Mhz. then we would reduce the transfer time to 0.029sec, which is within our time budget for the transfer.

Analyze the Program-Level Performance

Because embedded systems must perform functions in real time we often need to know how fast a program runs. The techniques we use to analyze program execution time are also helpful in analyzing properties such as power consumption. In this section, we study how to analyze programs to estimate their run times. We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis.

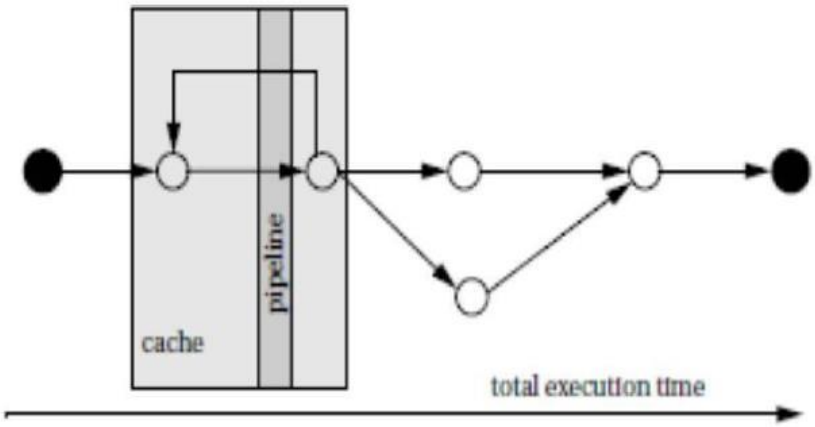

Figure 1.9.1 The CPU Pipeline and Cache act as Windows into Program

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

It is important to keep in mind that CPU performance is not judged in the same way as program performance. Certainly, CPU clock rate is a very unreliable metric for program performance. But more importantly, the fact that the CPU executes part of our program quickly does not mean that it will execute the entire program at the rate we desire. As illustrated in Figure 1.9.1, the CPU pipeline and cache act as windows into our program. In order to

understand the total execution time of our program, we must look at execution paths, which in general are far longer than the pipeline and cache windows.

The pipeline and cache influence execution time, but execution time is a global property of the program. While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:

- The execution time of a program often varies with the input data values because those values select different execution paths in the program. For example, loops may be executed a varying number of times, and different branches may execute blocks of varying complexity.
- The cache has a major effect on program performance, and once again, the cache's behavior depends in part on the data values input to the program.
- Execution times may vary even at the instruction level. Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction but on the instructions around it in the pipeline.

We can measure program performance in several ways:

- Some microprocessor manufacturers supply simulators for their CPUs: The simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.
- A timer connected to the microprocessor bus can be used to measure performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.

- A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution. The length of code that can be measured is limited by the size of the logic analyzers buffer. We are interested in the following three different types of performance measures on programs:
  - ✓ **Average-case execution time** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
  - ✓ **Worst-case execution time** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
  - ✓ **Best-case execution time** This measure can be important in Multirate real-time systems.

First, we look at the fundamentals of program performance in more detail. We then consider trace driven performance based on executing the program and observing its behavior.

Elements of Program Performance

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behaviour, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program. The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed iteration

bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming.

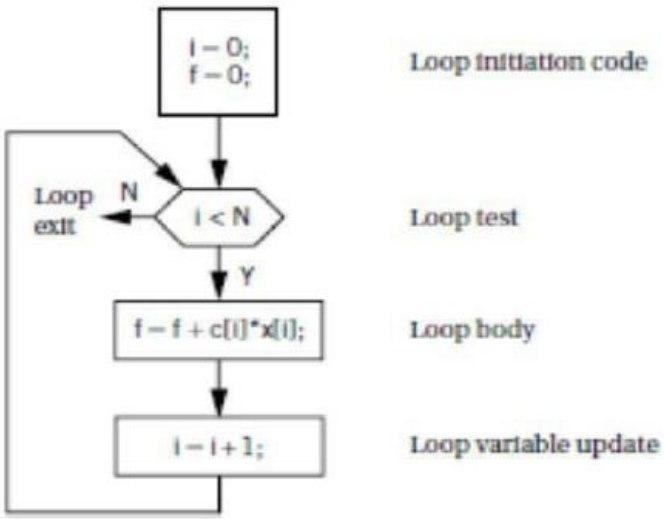

Figure 1.9.2 Segments of the program

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

A precise estimate of performance also relies on the instructions to be executed, since different instructions take different amounts of time. (In addition, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it. To measure the longest path length, we must find the longest path through the optimized CDFG since the compiler may change the structure of the control and data flow to optimize the program's implementation. It is important to keep in mind that choosing the longest path through a CDFG as measured by the number of nodes or edges touched may not correspond to the longest execution time. Since the execution time of a node in the CDFG will vary greatly depending on the instructions represented by that node, we must keep in mind that the longest path through the CDFG depends on the execution times of the nodes.

In general, it is good policy to choose several of what we estimate are the longest paths through the program and measure the lengths of all of them in sufficient detail to be sure that we have in fact captured the longest path. Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, which means we need only count the instructions and multiply by the per-instruction execution

time to obtain n the program's total execution time. However, even ignoring cache effects, this technique is simplistic for the reasons summarized below.

- Not all instructions take the same amount of time. Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU's pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load-store instructions are examples of longer-executing instructions in the ARM architecture. Floating point instructions show especially wide variations in execution time—while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.
- Execution times of instructions are not independent. The execution time of one instruction depends on the instructions around it. For example, many CPUs use register bypassing to speed up instruction sequences when the result of one instruction is used in the next instruction. As a result, the execution time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).
- The execution time of an instruction may depend on operand values. This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

Measurement-Driven Performance Analysis

Most methods of measuring program performance combine the determination of the execution path and the timing of that path: as the program executes, it chooses a path and we observe the execution time along that path. We refer to the record of the execution path of a program as a **program trace** (or more succinctly, a **trace**). Traces can be valuable for other purposes, such as analyzing the cache behavior of the program. Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program. This problem has two aspects. First, we have to determine the actual input values. We may be able to use benchmark data sets or data captured from a running system to help us generate typical values. For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time.

The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out. When we are designing a large system, it may be difficult to extract out part of the software and test it independently of the other parts of the system. We may need to add new testing modules to the system software to help us introduce testing values and to observe testing outputs.

We can measure program performance either directly on the hardware or by using a simulator. Each method has its advantages and disadvantages. Physical measurement requires some sort of hardware instrumentation. The most direct method of measuring the performance of a program would be to watch the program counter's value: start a timer when the PC reaches the program's start, stop the timer when it reaches the program's end. Unfortunately, it generally isn't possible to directly observe the program counter.

However, it is possible in many cases to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end. While this doesn't give us direct information about the program trace, it does give us execution time. If we have several timers available, we can use them to measure the execution time of different parts of the program.
A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program. However, because logic analyzers have a limited amount of memory, this approach doesn't work well for programs with extremely long execution times.

Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, a branch trace message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace. The alternative to physical measurement of execution time is simulation. A CPU simulator is a program that takes as input a memory image for a CPU and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image.

For purposes of performance analysis, the most important type of CPU simulator is the **cycle-accurate simulator**, which performs a sufficiently detailed simulation of the processor's internals so that it can determine the exact number of clock cycles required for execution. A cycle-accurate simulator is built with detailed knowledge of how the processor

works, so that it can take into account all the possible behaviours of the micro architecture that may affect execution time. Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself.

A cycle-accurate simulator has a complete model of the processor, including the cache. It can therefore provide valuable information about why the program runs too slowly. The next example discusses a simulator that can be used to model many different processors.

Performance Optimization

Loop Optimizations:

Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion, induction variable elimination,** and **strength reduction**. Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on operations performed in the loop body, then we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations.

An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it implement the loop. Properly transformed, we may be able to eliminate some variables and apply strength reduction to others. A nested loop is a good example of the use of induction variables.

Cache Optimizations:

A **loop nest** is a set of loops, one inside the other. Loop nests occur when we process arrays. A large body of techniques has been developed for optimizing loop nests. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance.