Estimating CPU-Time From Microarchitecture Events for Real-time Applications

Safayet Ahmed

625.740 Data Mining Fall 2020, Final Project

1. Introduction

The goal of this project is to demonstrate an approach to estimate execution time from micro-architecture-level events. The motivation for this is rooted in real-time computing. This paper begins by describing real-time applications and the need to predict execution times.

Then, sources of variance in execution times are described including the processor. The basic operation of processors is described. A model is presented that relates execution time to cycles-per-instruction (CPI) and microarchitecture events. Then, a description is presented of experiments performed on hardware to collect data on microarchitecture events. Linear regression analysis is applied on the collected data.

It is shown that a linear model that takes microarchitecture events from all cores into account is an excellent predictor of CPI across a diverse set of workloads. This paper concludes with possible directions for future work.

2. Real-time Computing

For most applications, correctness is defined as producing the right output in response to inputs. Real-time applications need to produce not only the correct output, but produce it on time. Specifically, the computation must complete by some **deadline** relative to the start time. Examples of real-time applications include control software, and multimedia.

The duration between the start and completion of real-time code is called the **response time**, and it's not constant. The source of variation in response time can be separated into three components.

First, the code may contain decisions (conditional statements), and the parts of the code executed may depend on those decisions. As a result, the amount of computation performed (the numbers and types of CPU instructions executed) may be dependent on the data provided as input.

Second, the processor executing the real-time code may be shared with other applications, and all of them are managed by the OS. The OS controls when any of these applications run. This problem of scheduling different real-time applications on a shared CPU while allowing them to meet deadlines is called real-time scheduling. Real-time scheduling has been a subject of several decades of research [1,2].

Finally, the processor itself adds variance to the run time of the code. The reason why the same set of instructions can take different amounts of

time on the same processor is discussed in more detail in the next section.

Providing even a probabilistic guarantee that real-time code will complete by it's deadline requires guarantees on the response time. Towards that end, the focus of this paper is estimating run time from events in the CPU's operation.

3. Basic Operations of a Processor

From fairly early on, computers have been designed according to the Vonn Nauman architecture. The key feature of this design is that there is a CPU and separate memory as illustrated in Figure 1 [3]. The CPU **fetches**, **decodes**, and **executes** a sequence of instructions from memory and operates on data located in the same memory.

Internally, the CPU is designed like an assembly line called a **pipeline**. Just like an assembly line, the pipeline has stages performing different operations on instructions. Specifically, the front-end stages fetch instructions from memory. Intermediate stages decode the instruction, acquire the necessary source data required by the instruction, and perform the computation. Back-end stages commit the results of the computation. The stages are synchronized and operate to a beat called the clock signal. At each clock beat (**clock cycle**), each stage of the pipeline is loaded with the results from the previous stage.

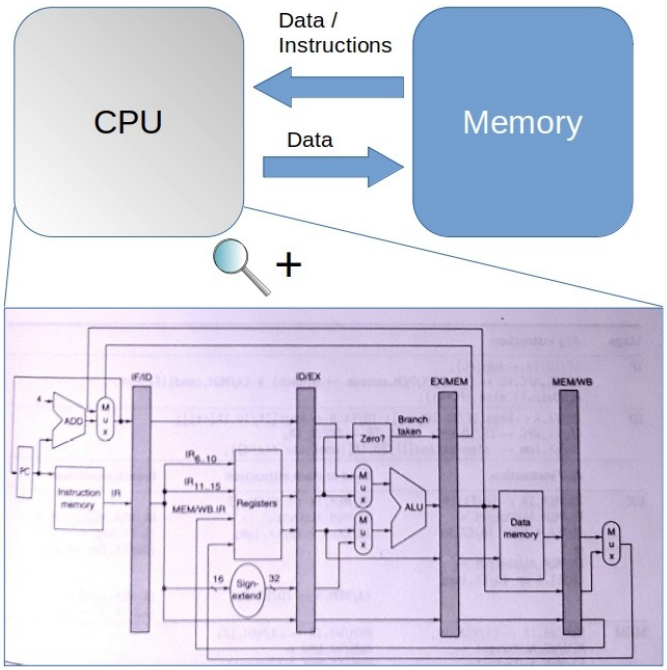

Figure 1.von Neumann architecture and CPU pipeline

3.1. Branch Prediction

Instructions are of different types, such as load and store instructions that cause the processor to load or store data to or from memory, and arithmetic operations and comparisons.

Also, there are control-flow instructions. A processor will fetch most instructions sequentially, these instructions are placed sequentially in memory. However, jump instructions cause the processor to fetch the next instruction from a different address. A branch instruction is a conditional jump that only behaves like a jump (**the branch is taken**) if the result of a previous instruction meets some condition or continues sequentially to the next instruction (**the branch is not taken**). Control-flow instructions are used to implement conditional statements, loops, and function calls.

Because of such instructions, the task of fetching the next instruction is actually more complex. For performance reasons, the processor must fetch the next instruction even before it gets a chance to “look at” (decode) the last fetched instruction. As a result, the front-end stage has to predict (1) which instructions are branch instructions, (2) whether the branch is taken or not, and (3) the branch destination. More importantly, any instruction fetched and processed as a result of the prediction is speculative. If the prediction was incorrect, the processor needs to **flush the pipeline** of mispredicted instructions (pretend as if the wrong instructions were never fetched). Such mispredictions reduce performance and are one of the sources of randomness.

3.2. Caches

Another aspect of processors is that they operate much faster than memory. For each instruction the processor executes, it must load both the instruction and any associated data. Further, higher-performance processors tend to be **superscalar**, fetching, executing, and retiring multiple instructions each clock cycle; this adds more pressure on access to memory. To make matters worse, most current processors are actually multi(core) processors, with each core operating independently, and sharing the same memory.

As a result, access to memory behaves like a performance bottleneck, called the Vonn-Neumann bottleneck. To mitigate this bottleneck, processors include multiple stages of caches: temporary stores of recently accessed data that can be accessed quickly.

Lower cache stages are several times smaller and faster than higher cache stages, and all cache stages are significantly smaller than memory. As a result, lower cache stages only store a small subset of the contents of higher cache stages and memory.

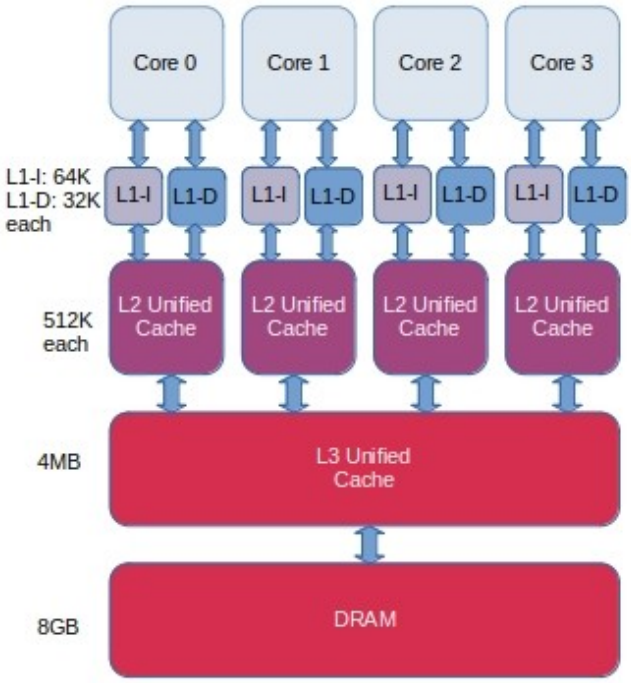

Figure 2.Cache configuration of a Ryzen 3 AMD processor with separate L1 instruction cache and data cache, and unified L2 cache and L3 cache.

When a processor must access a memory location, each cache stage is checked in succession. If a cache stage contains the desired data (**cache hit**), the memory operation completes quickly. However, if the desired data is not present in a cache stage (**cache miss**), the processor must check the next cache stage, and eventually, memory. Each additional cache stage

that is accessed adds to the latency; as a result, the instruction takes longer to complete. Missing all cache levels and accessing memory incurs the longest latency.

Further, since all cores share the same last-level cache (LLC) and memory, the different cores can contend for both space in the LLC and access to memory.

As a consequence of this complexity, each memory operation a processor must perform is a significant source of randomness. Figure 2 illustrates the cache configuration of a four-core AMD processor.

3.3. Instruction-Level Parallelism and Out-of-order Execution

As mentioned earlier, higher-performance processors are superscalar: processing and executing multiple instructions at the same time. However, this is only possible when instructions have no inter-dependencies. Where the result of one instruction feeds into another, the dependent instruction can't execute until the instruction generating it's data completes. So, the processor executes instructions out of order to increase throughput. Namely, if an instruction is stuck waiting for another instruction to complete, the processor will temporarily skip the waiting instruction and execute a following instruction. Such out-of-order execution can also add to the randomness.

3.4. A Model for CPU Time

Given some code and the same input, the instructions executed (and so number of instructions) should stay the same, even if the execution time does not. Taking into account the clock speed, f_{clock} in cycles per second (Hz), the CPU time and instructions are related as follows:

$$CPU\,time = (CPI \times Instructions)/f_{clock}$$

where CPI is the number of cycles required per instruction. Given that the number of instructions is fixed, and the clock speed is fixed, the main source of randomness is CPI.

We consider the AMD platform shown in Figure 2 with the three cache levels. The L3 cache is shared. From the discussion in earlier sections, the CPI is defined as follows. Similar models are presented in prior work [4].

$$\begin{aligned} CPI_{total} &= CPI_{base} \\ &+ Pr_{branchmiss} \times CPI_{branchmiss} \\ &+ Pr_{L1access} \times CPI_{L1access} + Pr_{L1miss} \times CPI_{L1miss} \\ &+ Pr_{L2access} \times CPI_{L2access} + Pr_{L2miss} \times CPI_{L2miss} \\ &+ Pr_{L3access} \times CPI_{L3access} + Pr_{L3miss} \times CPI_{L3miss} \end{aligned}$$

The total CPI is defined in terms of a base number of cycles required by every instruction, but augmented by the expected number of additional cycles incurred per instruction due to branch misses, cache accesses, and cache misses. Then, the problem of predicting CPI becomes a problem of predicting the probability of such events, and estimating the latency of such events.

In this paper, it is assumed that the probabilities and latencies are constant. Since the hardware remains fixed, constant latency is a reasonable assumption. Assuming constant probability may not be as realistic depending on the application. However, prediction of the probability of such event is left to future work.

In this paper, the viability of estimating the probabilities is considered by applying regression on instruction counts and event counts. The CPI penalties are estimated by applying regression on CPI and the rate of occurrence of the miss events.

For the most part, this is a simple linear model. However, additional complexity arises due to the shared L3 cache and memory bandwidth. The probability that load operations on one core misses L3 cache depends not only on the code running on the core, but also on code running on other cores contending for the same cache.

Further, the instrumentation used to count accesses to L3 cache and L3 misses can't attribute events to one core versus another; the counts of accesses and misses are aggregated. Despite this complexity, as seen in the results presented in Section 5.2, this linear model performs really well.

4. Collecting Data from Hardware

4.1. Counting Microarchitecture Events

Most current processors contain instrumentation to count the microarchitecture events described in previous sections. These are called hardware-performance counters, or performance-monitoring counters (PMCs). These are provided to profile (tune) software to improve performance.

Data was collected using such PMCs on a laptop with the Ryzen AMD processor described in Figure 2, running Ubuntu Linux 20.04. Linux provides a package called perf-tools to access and sample the PMCs [5].

The PMCs are configured to measure duration, clock cycles, total number of instructions, number of branch instructions, branch misses, cache accesses, and cache misses for L1 caches and L3 cache across all cores. In total, these are 35 events across four cores.

Note, this is more events than there are PMCs to count them, so the PMCs are multiplexed. Namely, the same PMC is used to count different events at different times, and the total number of occurrences of the events is estimated from this sample and the duration. Therefore, there is some measurement noise.

Also note, that L2 cache accesses and misses are not being counted, because those counters aren't available. That's acceptable since L2 is accessed when L1 is missed, and L3 is accessed when L2 is missed. Therefore the L2 cache events are counted indirectly and the regression models should handle that implicitly.

4.2. Workload

For the workload, stress-ng [6] is run with different stressors with different size and duration

configuration. Example stressors include various types of sorting and searching, solving combinatorial problems like the hanoi towers, compression workloads, and matrix operations (scalar multiplication and addition, matrix multiplication, and transpose). In addition, there are synthetic workloads specifically designed to cause branch misses, cache misses, and memory accesses.

For some workloads with configurable sizes (such as matrix sizes), the sizes are set to span 90% of L1-D cache size, 90% of L2 cache size, 90% of L3 cache size, and about 800% of L3 cache size to simulate workloads that are L1 bound, L2 bound, L3 bound, and memory bound. In total, 37 workloads are considered (including the four size configurations).

Each workload is run for durations of 5 seconds to 14 seconds in increments of 1 second. Further, for each workload and duration, the runs are repeated 6 times. In total, that's 2220 runs.

Separately, the perf tool was run without a workload to measure "background noise". As before the measurements were collected for durations of 5 to 14 seconds, each repeated 50 times. In total, that's 500 runs.

In a final set of experiments, stress-ng was run simultaneously on two cores. These runs spanned six different "background" loads on one core, and 36 "foreground" loads. The background loads were chosen to vary the degree of memory boundedness as mentioned above. All combination of background loads and foreground loads were run. The duration was varied between 5 and 14 seconds, but the runs weren't repeated. In total, that's 2160 runs.

Across these three sets of experiments, that's a total of 4880 runs.

The reason for choosing these workload, configurations, and durations was to span the measurement space of event counts as much as possible under the project time constraints. The goal is to approximate the function that relates the event-rates across cores to the CPI. By sampling more of the domain of this function, the hope is to learn a higher-fidelity approximation.

4.3. Reducing Background Noise

As mentioned earlier, the focus of this paper is variation in CPI due to the processor architecture. To eliminate scheduling-related noise from the OS, the stress workload was run on an isolated core using isolation features of the Linux kernel and docker containers. Further, the workload was run with real-time scheduling policy, and a high priority. These configurations prevent the OS from preempting or interrupting the workload, and help reduce measurement noise.

Second, the CPU model described in Section 3.4 assumes the clock frequency is fixed. By default, the OS scales the CPU frequency for power and performance management. So, the Linux cpufreq subsystem was used to configure the laptop to use a fixed frequency across all cores.

The Dockerfile and set of scripts used to collect data are available on github [7].

5. Analysis of Data

5.1. Estimating Event Rates

Recalling the model from Section 3.4, the probabilities of branch misses and cache misses are of interest. These probabilities are also the expected value of the number of such events with each additional instruction. So, simple-linear regression is applied to the event counts as dependent variable and instruction count as the independent variable. Since each workload is different, the regression is done separately for each workload.

In this analysis, the regression coefficient represents the rate of occurrence of the event with each instruction. For example, a coefficient of 0.25 would indicate an event that occurs once every four instructions. The regression score represents how accurately the number of instructions can predict the number of occurrence of events. High scores close to one are an indication that the rate of occurrence of a particular event is fairly consistent for the workload. Lower scores indicate that the rate of occurrence of events with each instruction is not constant.

Interestingly, the offset of these simple linear regression models were non-zero but low. The non-zero (and at times negative) offsets were unexpected, since if the number of instructions executed is zero, no microarchitecture events should occur. The number of events should never be negative. It is speculated that the non-zero and negative offsets exist due to transients when the workload starts. The parts of a code that run at startup are different from the parts of the code executed at steady state. An unloaded cache (cold cache) generally incurs more cache misses. Also,the CPU branch predictors, which are not trained at startup, will incur more branch misses.

First, the regression analysis is performed on runs with a single workload on a single core. A heat map of the coefficients for a representative subset of the workloads is shown in Figure 3. A coefficient of zero means the event occurred less than once every one hundred instructions. Based on this analysis, branch misses, instruction-cache misses and last-level cache accesses and misses are generally rare events.

The regression scores were generally high: in the 90%s in most cases and in many cases, more than 99%. The one exception was last-level cache accesses and misses: the scores were higher for memory-bound workloads and lower for workloads with smaller memory footprints. This was expected since the last-level cache is shared across cores. Also as expected, the regression scores for predicting cycles from instruction count were low for memory-bound workloads, because such workloads spend less time executing instructions and more time waiting to access last-level cache and memory.

Similar analysis was done for all runs including those with background workloads running on a second core. The regression scores were lower as expected but still consistent.

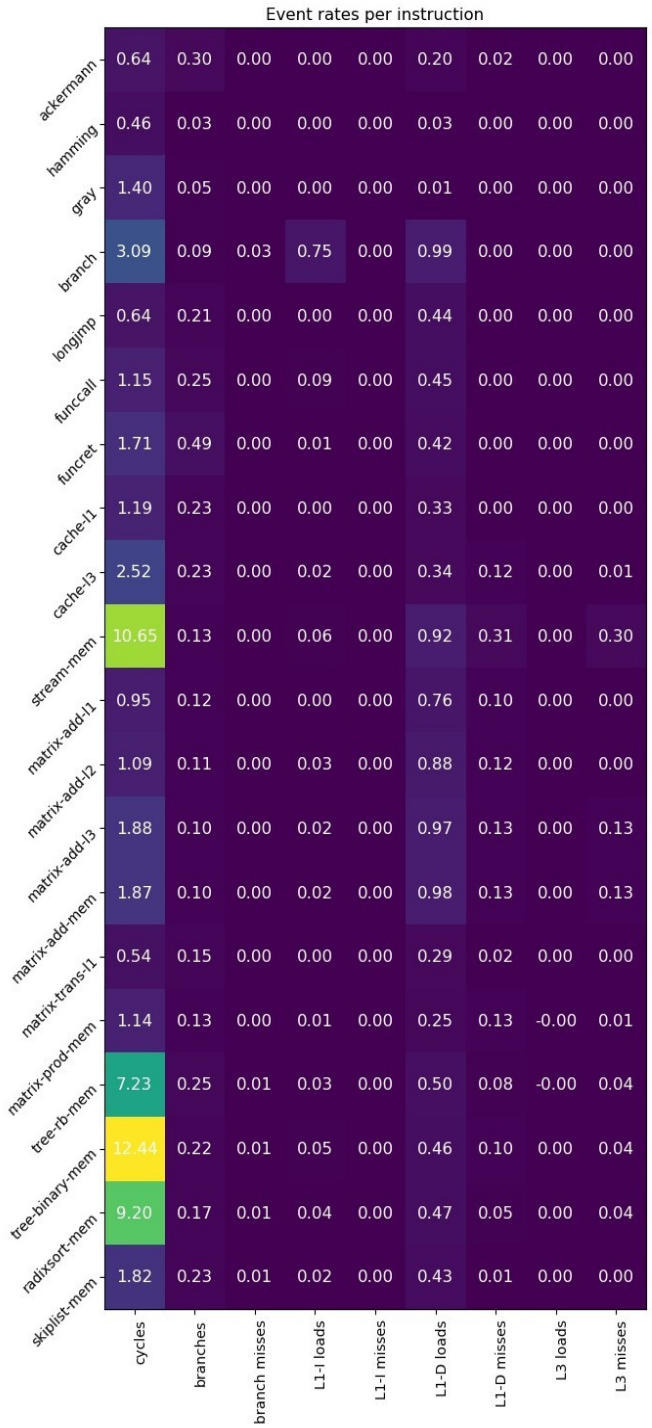

Figure 3. Coefficients of regression of event count (columns) against instruction count for a subset of workloads (rows).

Finally, when the same regression was applied to the combined data from one-core and two-core runs, the regression scores were significantly lower for cache-related events. Therefore, the rate of occurrence per instruction of cache-related events (cache accesses and misses) is severely affected by background loads.

5.2. Estimating CPI from Event Rates

Based on the model described in Section 3.4, linear regression was applied to the collected data. Namely, the number of cycles and all other events were divided by the instruction count. Then, regression analysis was done with Cycles-per instruction (CPI) as the independent variable and all other even rates as independent variables.

The data from the runs were randomly sampled to form a test set and training set. 80% of the runs were used as training set, and 20% as test set.

Two linear models were considered. In one model, only events rates from the single core on which the load ran are considered. In a second model, the event rates across all cores are taken into account.

Both linear models were trained and tested for just the single-core runs, and then separately for just the two-core runs. Finally, the models were trained and tested on combined data from all runs.

The regression scores for these two models and three data sets are shown in Table 1.

Table 1. Regression scores for different models relating CPI to microarchitecture event rates per instruction.

	Single-core runs only	Dual-core runs only	All runs
Single-core events only	93%	94%	69.9%
All-core events	99.4%	99.1%	98.9%

6. Conclusions and Future Work

It is clear, that across all data sets, a linear model that takes microarchitecture events into account from all cores is an excellent predictor of CPI.

It is surprising, that the one linear model trained across all the workloads in the training set performs so well on a separate test set of similarly diverse workloads.

Further, as apparent from the performance of the model that uses only single-core event rates, there is significant gains in using event rates from all cores.

Similar works have been published recently, applying machine-learning algorithms to counts of microarchitecture events to predict software performance [8] [9].

For future work, an easy next step would be to apply gradient-descent algorithms so that this linear model can be learned dynamically and online.

Further, additional models may be needed to predict the event rates, especially across different hardware with varying speeds and configurations. Namely, some method is needed to characterize a workload independent of hardware, and characterize hardware independent of workload, and a model that enables prediction of the performance of the workloads on hardware based on these characterizations.

7. References

[1] Liu, Chung Laung, and James W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." *Journal of the ACM (JACM)* 20, no. 1 (1973): 46-61.

[2] Davis, Robert I., and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems." *ACM computing surveys (CSUR)* 43.4 (2011): 1-44.

[3] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] Karkhanis, Tejas S., and James E. Smith. "A first-order superscalar processor model." *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*. IEEE, 2004.

[5] De Melo, Arnaldo Carvalho. "The new linux'perf'tools." *Slides from Linux Kongress*. Vol. 18. 2010.

[6] King, Colin Ian. "Stress-ng." *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)* (2017).

[7] Ahmed, Safayet N. "cputime-estimation." URL: https://github.com/sahmed62/cputime-estimation, *git/(visited on 22/11/2020)* (2020).

[8] Nemirovsky, Daniel, et al. "A general guide to applying machine learning to computer architecture." *Supercomputing Frontiers and Innovations* 5.1 (2018): 95-115.

[9] Gomatheeshwari, B., and J. Selvakumar. "Appropriate allocation of workloads on performance asymmetric multicore architectures via deep learning algorithms." *Microprocessors and Microsystems* 73 (2020): 102996.