Measuring Software Performance on Linux

Technical Report

November 4, 2018

Martin Becker
Chair of Real-Time Computer Systems
Technical University of Munich
Munich, Germany
martin.becker@tum.de

Samarjit Chakraborty
Chair of Real-Time Computer Systems
Technical University of Munich
Munich, Germany
martin.becker@tum.de

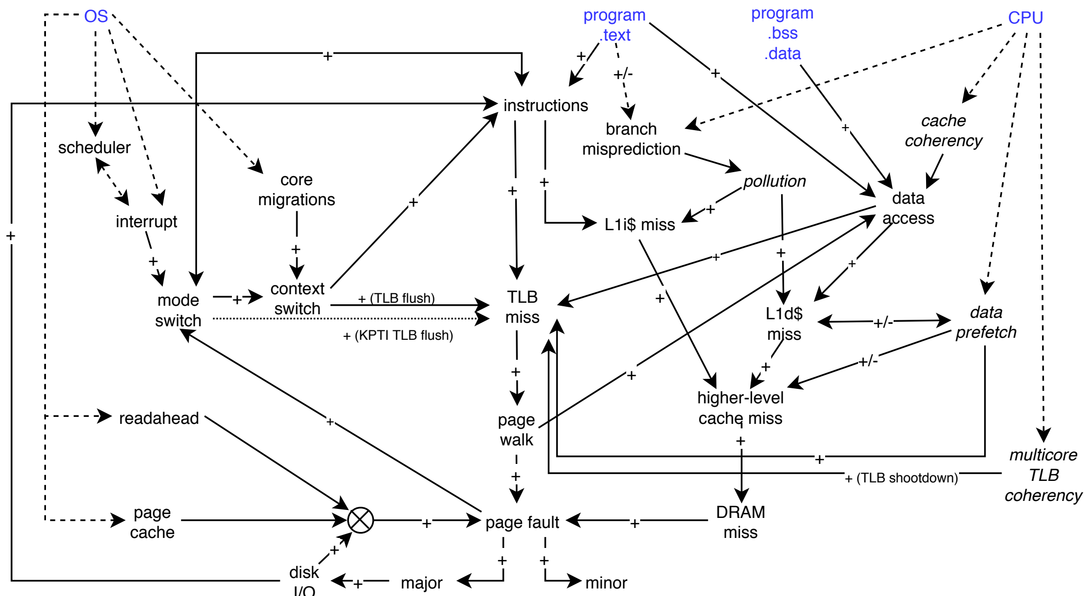


Figure 1. Event interaction map for a program running on a event Intel Core processor on Linux. Each event itself may cause processor cycles, and inhibit (−), enable (+), or modulate (⊗) others.

Abstract

Measuring and analyzing the performance of software has reached a high complexity, caused by more advanced processor designs and the intricate interaction between user programs, the operating system, and the processor's microarchitecture. In this report, we summarize our experience about how performance characteristics of software should be measured when running on a Linux operating system and a modern processor. In particular, (1) We provide a general overview about hardware and operating system features that may have a significant impact on timing and how they interact, (2) we identify sources of errors that need to be controlled in order to obtain unbiased measurement results, and (3) we propose a measurement setup for Linux to minimize errors. Although not the focus of this report, we describe the measurement process using hardware performance counters, which can faithfully reflect the real bottlenecks on a given processor. Our experiments confirm that our measurement setup has a large impact on the results. More surprisingly, however, they also suggest that the setup can be negligible for certain analysis methods. Furthermore, we found that our setup maintains significantly better performance under background load conditions, which means it can be used to improve software in high-performance applications.

CCS Concepts • **Software and its engineering → Software performance**;

Keywords Software performance, Linux, Hardware Counters, Microarchitecture, Jitter

1 Introduction

Countless performance tests of software are available online in blogs, articles, etc., but often their significance can be refuted by their measurement setup. Speedups are usually reported in units of time, yet without any introspection how exactly the differences came to live. Not only are the results

questionable for different machines, but even on identical processors with identical memory configurations and peripherals, there are many external factors that can influence the result. One such factor is the operating system (OS) itself. Different configurations and usages of OSes might nullify or magnify some effects, and thus the performance measurements do not necessarily reflect the characteristics of the software that we actually want to analyze and improve.

In this report, we describe how performance measurements of user software should be conducted, when running on a mainline Linux OS and a modern multi-core processor. Specifically, we are concerned with real-time measurements taken on the real hardware, providing quantitative information like execution time, memory access times and power consumption. We expect that we are able to greatly reduce measurement errors, and that the setup of the operating system and hardware makes a significant difference on the results.

This document is structured as follows: We start with a general survey of microarchitectural and OS elements affecting performance in modern processors in Section 2, in the next section we list possible sources of measurement errors, followed by a proposed measurement setup in Section 4. Finally, we show several examples of how the setup influences the results, before we conclude this report.

2 Microarchitectural and OS Performance Modulators

We start by giving an overview on hardware and OS features and how they influence the performance of software. In essence, this section is an elaboration of the interactions depicted in Figure 1. The well-informed reader might directly proceed with the subsequent section, which identifies sources of measurement errors based on the details presented here.

Although we try to keep the explanations generic, it would be impractical to make only statements that cover any conceivable processor. The details are therefore given for an Intel 2nd generation Core^{TM} x86-64 microarchitecture ("Sandy Bridge"). This is a pipelined, four-width superscalar multi-core processor, with out-of-order processing, speculative execution, a multi-level cache hierarchy, prefetching and memory management unit (MMU). This kind of processor is currently used in high-end consumer computers, and has well-proven architectural features that we expect to see in embedded processors in the coming years; many of them are already available in ARM SoCs [21]. The details of our machine are summarized in Tab. 1. As operating system (OS) we consider Linux, specifically in an *SMP* configuration.

Along this report, we will give some numbers to illustrate the magnitude of some effects. These numbers are given to the best of our knowledge, the best available vendor documentation, and supported by measurements that we have taken. The numbers are all based on the following system *penalties*, caused by the typical design of CPUs and OSes.

Unit	Properties
Processor	Intel Core i7-2640M @2.8GHz, dual-core
Microarchitecture	"Sandy Bridge", Microcode version 0x25
L1i/d cache	32kB, 8-way, 64B/line, per core
L2 unified cache	256kB, 8-way, 64B/line, writeback, per core
L3 uncore cache	4MB, 16-way, 64B/line, shared between all cores
TLB	2-level, second level unified, per core
OS	Debian 8.11 GNU/Linux SMP 3.16.51-3

Table 1. System specifications

Penalties: The various latencies for our system are summarized in Tab. 2, and discussed in the following. They are taken from the processor documentation [11], and extended by measurements using `lmbench` [24] and `pmbench` [32]. The values denoted "best case" are declared as such in the documentation. Our measurements show that these values are also the *most frequently* observed ones. Note, however, that penalties can (and should) be hidden by out-of-order processing. That is, not every page walk will delay computation for 30 cycles.

Event	Condition	Latency [cycles]
L1 cache hit	best case	4
L2 cache hit	best case	12
L3 cache hit	best case	26..31
DRAM access	best case	$\approx$ 200
branch misprediction	most frequent	20
TLB miss	2nd-level TLB hit	7
page walk	most frequent	$\approx$30
minor page fault	most frequent	$\approx$1,000 .. 4,000
major page fault	most frequent	$\approx$260k .. 560k
context switch	best case	$\approx$3,400

Table 2. Penalties for system in Table 1

Software Performance: In this report, we look at software performance, mainly from a timing point of view. Specifically, for the execution time of the process, we only consider the time where the processor executes instructions on behalf of the process (including kernel code and stalls), but not sleeping or waiting states, since the latter are either voluntary or depend on the execution context and not the program itself.

2.1 Microarchitecture

As an overview about the features discussed next, consider the microarchitectural block diagram shown in Fig.2.

2.1.1 Number of Cycles and Instructions

As a single number indicating program speedup, we first look at the number of processor cycles spent on execution. Lower is better. The number of clock cycles is primarily driven by the instructions being executed, whereas the exact

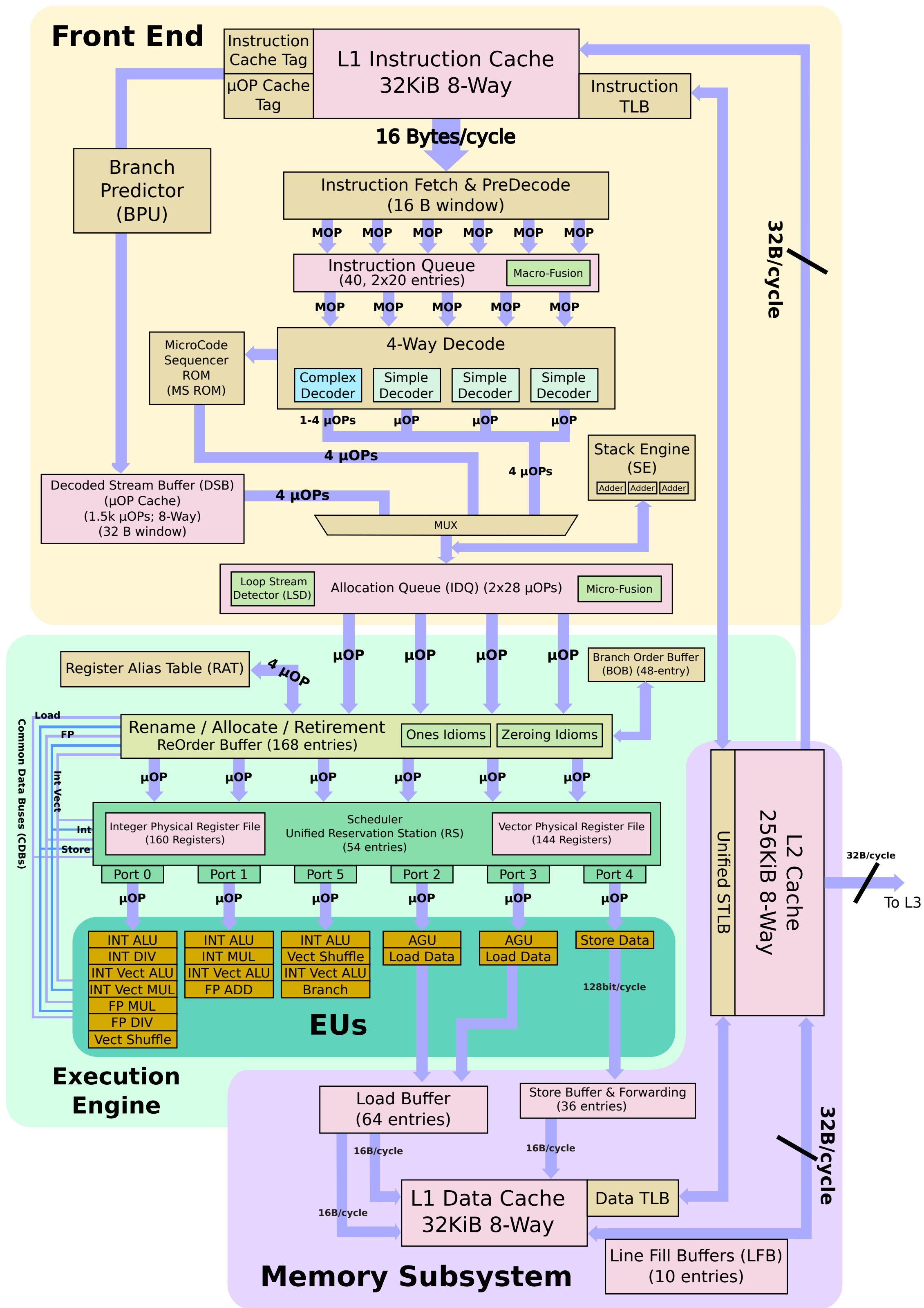


Figure 2. Microarchitecture of Intel Sandy Bridge as example of a superscalar out-of-order processor with caches, branch predictor and prefetcher. Taken from [3], and checked for consistency against manufacturer datasheets [11].

relationship is defined by the implementation of the microarchitecture. The relationship is usually nontrivial and not fully documented, therefore we have forgone to indicate in Fig.1 the relative influence of events of clock cycles. We will provide numbers in the following descriptions, as far as they are known.

If we count both the number of retired instructions and processor cycles, we can compute the ratio *instructions per cycle* (IPC). While often used as a first performance indicator, this figure is highly program-specific, and not helpful for judging our software. For example, a program may execute faster than before although the IPC has dropped, simply by virtue of a reduced number of instructions.

Micro-operations (uops): Some processors split instructions into multiple smaller operations [11, 21], which provides more possibilities for out-of-order processing [7]. On such targets, it is more meaningful to measure uops instead of instructions, since a single instruction may become a variable-length series of uops. Some, usually more complex instructions, may even invoke *Microcode*.

Microcode: On some processors, instructions are not hardwired but interpreted, to be broken down to hardwired machine code during execution. This abstraction layer can incur a slowdown, but it allows upgrading processors during their lifetime, e.g., to fix processor errata. On x86, microcode is only used for few complex instructions; specifically, only for instructions generating more than four uops in our processor [7]. The performance can be impacted in two ways: First, switching between the regular instruction stream and Microcode may have a penalty, and second, microcode generating a lot of uops may be limited by the frontend bandwidth.

Uops Cache: Instructions are decoded into uops by a hardware circuit that can be limited in throughput, causing a bottleneck if the average instruction length exceeds its capacity. Some processors add a cache to store decoded uops, in an attempt to alleviate this problem [7].

2.1.2 Pipelining

Instead of processing one instruction or uop after another, processing is often is temporally overlapping to reduce execution time, called *pipelining*. For example, while one instruction is being executed, the subsequent one can already be decoded to uops in parallel, and meanwhile the uops from the previous instruction can be retired. Today it is very rare to find processors or even microcontrollers that do not implement some form of pipelining.

2.1.3 Out-of-Order and Superscalar Processing

Modern processors implement dynamic scheduling of instructions [7, 21]. That is, the order in which instructions are executed, may deviate from the original order of the instruction stream given by the program counter. This enables the processor to hide various processing latencies, by performing other work while waiting for an instruction to finish. For example, an arithmetic division can take several cycles to complete, meanwhile successor instructions can be executed, up to the point where the result of the division is required. Similarly, memory access latencies can be hidden. This is one of the primary reasons, while it is nontrivial to predict the number of clock cycles that a certain event slows down a program.

As a side effect, such out-of-order (OoO) processing enables *superscalarity*. By having more than one instance of each functional unit (e.g., several ALUs or FPUs), it becomes possible to execute several instructions in parallel. Thus, OoO processors often issue multiple instructions at the same clock cycle; for our machine, four uops are issued simultaneously, thus the maximum IPC is four.

An algorithm for OoO was proposed by Robert Tomasulo in 1967 [28], and today's out-of-order processor implementations still follow the same concept [7, 11]. The OoO implementation is often called *execution engine*, and for the purpose of our performance measurements, the following constituents must be considered: 1. Register Renaming and 2. Scheduler.

Register Renaming: Register names used by the compiler are those defined by the instruction set architecture (ISA), called *architectural* registers. Due to the limited number of registers, the compiler might me forced to re-use registers for computations that are otherwise not related, creating false data dependencies. However, actual implementations of the ISA may have more *physical registers* than architectural ones. Therefore, the first step is to map architectural to physical registers, while resolving some false dependencies. This renaming process therefore improves superscalar and OoO processing [7].

Scheduler: The renamed operations now enter the main core of OoO processing, the *scheduler*. Here operations are queued up for execution units in *reservation stations*, and will be held there until all operands become available. When this becomes the case, the operation is started on the execution unit. As soon as it completes, results are propagated to all subscribers (e.g., registers or reservation station entries waiting for a result), and the operation is forwarded from the reservation station to the Reorder Buffer. From here, most schedulers also takes care of retiring the instructions in their original order. A bottleneck may occur if the scheduler stalls execution because no free reservation station is available.

2.1.4 Branch Mispredictions

Many microarchitectures perform some kind of branch prediction, to hide the latency for loading the instructions and data after a branch [7, 21]. Predictions for both the outcome of two-way branches, and the (possibly multi-way) target address of indirect branches, are being made by the branch prediction unit.

If the branch is incorrectly predicted, then the pipeline and other resources must be flushed, which means that there is a

time penalty to fetch and decode the correct instructions. The magnitude of the penalty primarily depends on the pipeline depth. In case of Sandy Bridge, the penalty for flushing and resuming execution is about 20 cycles [7, 11].

2.1.5 Speculative Execution

The time window between branch prediction and learning the actual branch outcome is spent with *speculative execution*. That is, the processor continues the control flow at the assumed branch target, and buffers the results until the actual branch target becomes known. Whenever the prediction was incorrect, it flushes the pipeline as described above. If the prediction was correct, the speculatively executed instructions are allowed to be released from the buffer ("retire"), and no time was lost waiting for the branch outcome.

There is one lesser known side effect, however, dubbed *pollution* in Fig.1. Although instructions executed during mis-speculation are not retired, they can still cause changes in cache and buffer states. These effects are *indirect* cost of branch mispredictions which manifest themselves during later execution, and they have recently been exploited in the *Spectre* and *Meltdown* vulnerabilities [18].

Last but not least, even perfect speculation might become a performance bottleneck in some cases. All speculatively taken branches are stored in the *branch order buffer* (BOB) until they are confirmed. However, too many speculations in a short time window might cause the BOB to fill up, which in turn stops the issue of new uops [11].

2.1.6 Machine Clears

When multiple threads can run truly in parallel (as on SMP systems and especially with OoO processing), the ordering of memory accesses must be monitored and ensured. If the CPU detects that accesses complete differently to program order, a *machine clear* is performed. This entails undoing some operations, flushing the pipeline, and re-starting with the correct operands [11]. The cost is comparable to a branch misprediction, which is discussed next. Further causes for machine clears are self-modifying code and illegal AVX addresses.

2.1.7 Cache Hierarchy and Misses

Although memory access is fast these days, it can still be orders of magnitudes slower than the processor, which has become known as the *Memory Wall* [31]. Consider the penalty for a major page fault (disk access) shown in Tab. 2; at least five orders of magnitude lie between the processor and the disk access, although we use a comparatively fast Solid State Disk (SSD). *Caching* is the omnipresent approach to counteract this issue, by preloading or latching all data in faster, smaller memories, and exploiting the *principle of locality*. That is, data that has been recently accessed, is likely to be accessed soon in the future again. It thus makes sense to buffer recently used data in caches. Every time a data item is requested ("data access" in Fig.1), we check for the desired data in the faster cache, before accessing the slow memory. If we find the data, called a *cache hit*, we have circumvented waiting for the slow memory. If the cache does not contain the data, called a *cache miss*, we have to pay the penalty for accessing the slow memory (usually DRAM). After this, the data is placed in the cache for future reference.

Since caches have to be small to be fast, there are inevitably situations when data is not in the cache, and needs to be loaded from slower memory. Even if we were able to perfectly predict what data (including instructions) is needed, then there are still compulsory misses on first access. As a result, execution can be slowed down by one or two orders of magnitude even with fast caches and very high hit ratio. For example, let us assume each instruction takes one cycle, and that each cache miss costs five extra cycles. Then, even with $p = 95\%$ hit ratio, a program with X instructions would take $X + X(1 - p)5 = 1.25X$ cycles, i.e., experience a 25% slowdown. Measuring cache behavior is therefore important.
Victim Caches: These are small and fully associative caches, holding items evicted from a larger, not fully-associative one. Each miss in the large cache is first looked up in the victim cache, before the slower memory is consulted. This masks miss times for temporally close conflict misses. From a practical point of view, the victim cache need not be considered separately; instead, a victim hit can be seen as a hit in the faster cache. Vice versa, a victim miss can be seen as a hit in the next-slower memory or cache.
Hierarchy: As most modern processors, our machine uses a *hierarchy* of caches. That is, there are three caches in a cascade (three "levels") before the slow secondary storage is accessed. The first level (L1) is the fastest/smallest and separate for data (L1d) and instruction (L1i), see Tables 1 and 2. Leaving aside special architectural tricks, this is the only level that can be accessed directly by the CPU. Higher levels (L2, L3) are larger and slower, and usually unified. Depending on the processor, some caches can be shared with other devices.

2.1.8 DRAM Access

The next-larger memory after the cache hierarchy, is Direct Random Access Memory (DRAM). Conceptually, the cache hierarchy always acts as buffer to DRAM accesses. Only if the lookup of data or instructions missed at all levels in the cache hierarchy (Fig.1 "higher-level cache miss"; not necessarily sequentially, though), then the DRAM is consulted. Accesses to DRAM are typically 100 times slower than the CPU, thus the penalty missing the entire cache hierarchy becomes steep. Even worse, DRAM is often accessed via the *Northbridge*, possibly suffering contention with other CPUs and DMA transfers [6].

2.1.9 Hardware Data Prefetching

To further reduce access times to slow memory, many processors have a *data prefetcher* circuit, which predicts future data accesses and actively pre-loads the data into caches. Most prefetchers are triggered by certain access patterns in cache misses [6]. Some newer processors may even cross page boundaries [11, §2.4.7]. In summary, prefetch events both depend on and influence L1d cache events, as shown in Fig.1.

2.1.10 ISA Extensions, Streaming/Vector/SIMD

Processors may extend the ISA with instructions applying the same operation on multiple data items (vectors, single-instruction-multiple data). Examples are the extensions AVX, SSE, MMX on Intel and AMD, and NEON on ARM processors. Using these can greatly speed up certain calculations, but switching to these modes may also cause extra penalties [7, §9.1.2]. In general, any switch between ISA modes or extensions may cause extra penalties.

2.1.11 Direct Memory Access (DMA)

Accessing secondary storage, such as hard drives, but also traffic from network cards, usually takes place via DMA, which enables peripherals to exchange data directly with the DRAM.

2.1.12 Cache Coherency Protocols and ring bus

On multi-core systems, further cache accesses are caused by cache coherency protocols between the caches of the different cores [6]. It becomes active during core migrations, but also in the presence of data shared between cores.

2.1.13 Neglected Features

There are many peculiarities to each microarchitecture. We have omitted many of such details, in an attempt to focus on those features prevalent in most processors. Of course, these details can be important for the performance, and a careful study of the microarchitecture is required to see which need to be measured. Omissions include instruction and uop fusion, loopback buffers, register stalls, exhaustion of execution ports, cache bank conflicts, limited number of register ports, misaligned memory access, and store forwarding stalls. These are not considered to have a large or systematic performance impact on Sandy Bridge [7].

2.2 Microarchitecture and OS Interaction

2.2.1 Virtual Memory and Paging

Virtual Addressing is common in larger general-purpose and application processors, and is done for a variety of reasons, chief among which are process isolation and provision of a contiguous address space from the process' point of view. However, it also enables *paging*, which helps to significantly mitigate the latency of accessing slow secondary storage, such as hard drives. Unfortunately, Virtual Addressing comes with a performance penalty that can vary highly. Translation needs to be performed for every instruction and data reference (see Fig.1), and thus there is an obvious incentive to minimize delays. Therefore, in practice the address translation process is very intricate and specific to the microarchitecture. An in-depth description can be found in [6].

Usually a hardware unit called *Memory Management Unit* (MMU) is responisble for the address translation. To ensure translations do not stall execution, most MMUs have their own caches, called *Translation-look-aside buffers* (TLBs), which hold the most recently translated addresses. In case the buffers do not provide the needed translation (TLB miss), then a slow search in memory has to be conducted, called a *page walk*, see Fig.1. On x86 machines, this means that a dedicated hardware circuit starts looking for page table entries in memory, which incurs a penalty depending on memory access latency. For other architectures, like some PowerPC, this might be done in software, and thus is even slower. In case the information was found in the page table, the TLB is updated and execution resumes. Otherwise, the operating system is signaled a *page fault*, and has to decide whether the access is allowed, and if so, update the page table for the requested translation, and possibly bring missing data to the main memory. The cost for virtual address translation therefore is consisting of cycles spent in TLB lookup (hardware), plus page walk cycles (often hardware), plus cycles to handle page faults (OS).

Translation-Lookaside-Buffer: TLBs are regularly flushed by the OS, since it is responsible for the coherency between TLBs and page tables in memory, and that the TLBs do not hold stale translations from a formerly running process. They can either be flushed or selectively updated, depending on the OS and hardware capabilities. Accesses after a flush are TLB misses, therefore flushes degrade performance. In the x86 architecture, there furthermore exists the case of *TLB shootdown*. This stems from the need to have consistent TLB entries between multiple cores in the presence of sharing. Since x86 does not have a coherency protocol in place, TLB contradictions between cores are avoided by triggering flushing in hardware. Last but not least, TLB and cache access can be executed in parallel, to reduce latency.

Page Walks: Examining the page tables in memory incurs a penalty that depends on whether the tables are cached (L1d, L2, L3), or whether the slow DRAM needs to be consulted. On our machine, a typical page walk costs between 20 and 60 cycles. In an extreme case ("TLB trashing"), this could happen for every instruction, and thus become very expensive. On some microarchitectures, as in Intel Sandy Bridge, page walks go through the caching hierarchy. That is, page tables are buffered in L1d and below, and thus caches can be modified by page walks. Consequently, caching behavior and TLB misses cannot always be separated, even in the absence

of page faults. Additionally, it has recently been disclosed that Intel processors speculatively work on possibly invalid cache entries in parallel to page walks (see “L1TF” vulnerability [5]). It can therefore be assumed that even the latency of page walks is partially hidden.

Page Faults: If a translation cannot be found in the page table, a *page fault* is signalled from the MMU to the OS (Fig.1 bottom). There are three fundamental types of page faults: 1. Invalid page fault, 2. major page fault, or 3. minor page fault.

Invalid page faults are those caused by an attempt to access addresses that are beyond the process' address space, or where privileges are insufficient. An example are segmentation faults. We do not discuss them further, since they are pathological events pointing to faulty software.

Major page faults require disk access, which is orders of magnitude slower than the effects we are trying to observe here. To give a number here: For our SSD-equipped host, we used `pmbench` [32] and measured the most frequent latency for major faults as between 262 thousand and 524 thousand cycles (coinciding with the median), same for both read and write accesses. Additionally, the distribution is tail-heavy, similar as in [32]. That is, the estimated average is about one million cycles, due to some accesses exceeding several dozens of milliseconds. Closely related to page faults, Linux implements a *page cache:* the virtual memory buffers data blocks of recently used files. When files are read (e.g., via `fread` or `mmap`), then access is by default buffered via the page cache. Therefore, if a program is executed a second time and there was sufficient memory, there are ideally no major page faults due to file access. To further reduce first-time access latency, the Linux kernel proactively reads file data from disk before it is demanded (“read ahead”), which does no longer is counted as major page fault. Finally, if DRAM is exhausted and *swapping* is enabled, unused pages are temporarily written to secondary storage (“swapped out”). When they are needed again, they have to brought back to DRAM, which similarly causes major page faults [8].

Minor page faults are caused by memory allocation without disk access, but still are still problematic for our measurements. The memory can either be immediately allocated, or only reserved (“lazy”). In the latter case, which is the default case, the page is only created when the first write occurs (“copy-on-write”). This means that the penalty of minor page faults consists of two parts: the cost for the fault handler itself, and the conditional copy-on-write cost. Measuring the cost of a minor page fault is unequally more complicated than major faults. First, the latency is relatively short, resulting in inaccuracy when tried to measure using software. Tracing kernel functions is one option that we have exercised, but it has some non-negligible overhead in our kernel version, only giving us an upper bound. Hardware measurements are not possible in this case. The result is a somewhat wide range for the minor page fault cost. Using `pmbench`, we found that the mode lies between 1,024 and 4,096 cycles, again coinciding with the median. Note that this includes lazy allocations. The numbers match recent measurements of Torvalds on the successor microarchitecture *Haswell* [29].

2.2.2 Context Switch

Switching between processes happens frequently (discussed later in Section 2.2.6), an is an expensive operation that influences the TLB and caches, see Fig.1.

Besides executing a number if instructions to save and load hardware registers, stack pointer and PC of suspended and waking process, the pipeline is flushed, and the TLB must also be updated [20]. On most Linux versions the TLB update takes the form of a flush, accompanied by switching the page table pointer. This can only be avoided with hardware that supports process context identifiers (PCIDs) and with newer kernels supporting this feature – such as x86 on Kernel 4.14 onwards [22]. The penalty for such TLB invalidation is called the *direct cost* of a context switch [20].

However, the CPU caches are also affected, which incurs *indirect costs*: Because processes share caches among themselves also with the kernel, the waking process may not find its data in the caches as had been left when it got suspended, and experience cache misses. This effect magnifies with growing working set size [20]. Context switches can happen involuntarily due to interrupts, or voluntarily due to system calls (both discussed later). Using `lmbench`, we found that context switches on our system take at least 3,400 cycles, with a frequent value around 30,000 cycles. Note that this number would increase if a core migration happens at the same time, which is explained next.

2.2.3 Core Migrations and Load Balancing

The OS (and the hardware, if Hyperthreading is enabled) may migrate a running process between cores (“core migrations” in Fig.1), to balance load or thermal stress. This causes a context switch with additional overhead. The process must be stopped, copied to another core's run queue, cache lines are moved to the new core [6], and only then both scheduling domains are released. Naturally, this requires to run a lot of kernel code, which in turn increases chances for events like branch mispredictions, cache pollution and page walks. We have observed latencies of beyond 100,000 cycles for migrations, depending on the working set size.

2.2.4 Mode Switch

Switching between user and kernel mode is not a context switch, and thus has lower overhead. These switches are caused by system calls done by the running program (thus the bidirectional interaction between instructions and mode switches in Fig.1) for which the kernel is supposed to perform some work on behalf of the calling process. The kernel is then said to be in *process context.*

In Linux on x86, these calls involve copying the arguments to registers, triggering a trap (CPU changes to kernel mode), executing the trap handler (which copies the arguments from the registers to the kernel stack and then performs its work), and eventually returning to user mode. Depending on the processor and OS, the TLB might be invalidated. This is the case for x86 since the *Spectre* and *Meltdown* vulnerabilities in 2017, where now *kernel page table isolation* (KPTI) invalidates entries during each mode switch to protect the kernel from attacks, unless the hardware supports PCIDs (see above). Finally, when exiting the kernel mode, a context switch to a different process might take place.

Our kernel as KPTI enabled, but not PCID support. We created a test program to measure the direct penalty from KPTI. We found that the fastest syscall drops from at most 660 cycles down to less than 130 cycles, when KPTI is disabled [1]. Kernel developers have measured large slowdowns as well, up to 30% in networking code [4]. That is significant, and thus needs to be considered during measurements.

In summary, mode switches also have a direct penalty caused by the call overhead, and indirect penalties caused by TLB and cache effects, depending on kernel version and processor.

2.2.5 Interrupts

Typically, a few dozens of interrupts[2] can arrive at any point in time. For example, there are *thermal interrupts* in case the hardware overheats, and *machine check exceptions* indicating hardware errors in the CPU. Some interrupts cannot be avoided, while others can be disabled or redirected to different cores. Interrupts do not directly cause context switches as described above. Instead, the CPU itself saves and restores very few registers (notably, the PC), and then a mode switch happens [2], see also Fig.1. After this, the interrupt handler must take care of the rest. Specifically, the kernel then is running in *interrupt context*, as opposed to process context, and the work being done here is not attributed to any user process.

The interrupt handler itself is supposed to be fast and must not suspend execution. This is also called the *top half*. Interrupts which require longer-lasting or I/O work to be done, must defer such work for later processing in a so-called *bottom half* [15].

Once the handler finishes, the OS switches back to user mode. As explained before, a context switch might happen during this transition. Specifically, if the interrupt had deferred some work in a bottom half, then a context switch to a kernel thread might happen at this point in time.

[1]This upper bound was measured as the minimum latency over many calls of `getuid`, with the program given in Appendix A. This program had experienced a 3x speedup!

[2]see `/proc/interrupts`

One special periodic interrupt is the *local timer interrupt* driving the scheduler, thus also called "scheduling clock ticks", and inspected next.

2.2.6 Scheduler Ticks

By default, the Linux scheduler runs periodically, to select which process is executed on the hardware for the next time slice. This is achieved by a timer interrupt that fires typically every four milliseconds. Considering that interrupts cause mode switches and possibly context switches (see Fig.1), running the scheduler itself may unnecessarily suspend our user task. Luckily, there are two options to configure the kernel differently, commonly referred to as "tickless" [13].

1. *Dyntick-Idle*, enabled by the kernel configuration option `CONFIG_NO_HZ_IDLE=y`, describes a kernel configuration that omits scheduling ticks when there is no task to be executed. While this is often the default for desktop and embedded systems to save energy, it is not useful for our purpose.
2. *Adaptive ticks*, enabled by kernel configuration option `CONFIG_NO_HZ_FULL=y`, describes a kernel configuration that additionally turns off ticks when there is only one runnable task, thus preventing unnecessary interruptions. On top of the configuration option, it is necessary to add a kernel boot parameter specifying for which CPUs it shall be applied, RCU calls need to be offloaded to other CPUs.

Both of these configurations have disadvantages, as well, including increased number of instructions to switch to/from idle mode, and more expensive mode switches [13]. Using again the program shown in Appendix A, we have found that mode switch times in our system increase to about 700 cycles with adaptive ticks and KPTI disabled. Consequently, tickless operation may not be beneficial for workloads with many syscalls, or those that often go idle.

2.2.7 Neglected Features

The most important mechanisms of OS-hardware interaction have been explained. Nevertheless, there are many other features and effects that depend on the specific version of OS. Some omissions include speculative paging here [10], as that is a recent development not commonly used, yet, and page migration between NUMA nodes, since our target is a single NUMA node.

3 Measurement Errors

This section summarizes features that can create measurement errors, building on the explanations given before and visualized in Fig.1. By *error* we subsume all effects that stem from sources other than the software under analysis and, when not properly controlled, would therefore mislead us in a systematic or random direction. For example, we consider effects caused by other processes running in the background

of an operating system as measurement error. Inter alia, they share caches with the software under analysis, and thus the caching behavior of our software can look significantly worse if there exists a data-heavy background process.

Table 3 depicts all major sources of measurement errors, together with a classification whether they are caused by hardware (HW) or software (SW), their measurable impact, and the time when the effects manifest in the measurements. Additional explanations are given in the notes below.

Notes for Table 3:

(a) Only traffic not caused by the software under analysis is considered an error. Such traffic might be caused by all other processes running on the same core, including the kernel. See process isolation.

(b) If the number of demand data accesses shall be measured, then the prefetcher skews the results, otherwise it is not considered a source of error. Also, prefetching has the same effect as demand access on caches and TLB, thus it can skew TLB access metrics.

(c) Only interrupts not caused and required by the software under analysis are considered source of errors. Interrupts may cause other processes to be scheduled to execute bottom halves.

(d) The mode switches themselves are often required to execute system calls. But Mode switch overhead can be highly different depending on OS and HW. Therefore, it may or may not contribute a significant error.

(e) Allocators for virtual memory have different characteristics. Some may consume significantly more/less memory than others, and some may force context switches.

(f) Non-tickless kernels may generate unnecessary context switches and impair TLB, caches and thus performance. None of those switches are caused by the software under analysis, and thus considered errors.

(g) Major page faults cause disk access. If the effects to be measured are rather small compared to disk access times, then major faults would mask those effects due to large jitter, and must be avoided. Furthermore, it is worth noting that a process accessing a file might be sent into waiting state, and then the number of unhalted processor cycles is not incremented. Consequently, I/O is not directly visible in counter-based measurements, and even I/O-limited applications can still exhibit a high IPC. Indirectly, however, it can be seen that the process spends less CPU time than wall-clock time, pointing to an I/O-bottleneck.

(h) Depending on the microarchitecture, DMA might slow down DRAM access because of increased traffic on the Northbridge [6]. Naturally, DMA also generates interrupts. Furthermore, cache coherency might be a concern [25], and generate extra cache traffic.

(i) Some caches might be shared with peripherals and thus be influenced by their operation. For example, the GPU and the processor share the L3 cache on our machine. This means that even different monitor setups can change cache miss statistics and bandwidth benchmarks[3].

3.1 Short-Living Programs

When an OS is used, there is necessarily an overhead for starting and terminating a process. This can become significant when we measure the performance of programs that have a low execution time. After all, most developers are not trying to optimize the OS or its libraries, but their own software. To give a rough number, programs that execute less than a million instructions usually fall into this category, with the specific number being subject to the amount of work the OS has to do.

As an example, consider the callgraph of a program (the `nsichneu` from the Mälardalen WCET benchmarks, compiled with gcc and `-O3`) shown in Figure 3a. The user code, all in a single function (black node in the upper right half), executes about 40,000 instructions. The entire program, from startup to after termination, requires about 150,000 instructions. The OS performs tasks such as *dynamic loading* (starts in upper left corner and spreads about two thirds into the picture), which locates and loads into memory all dynamic libraries used by the program. The actual user code is executed thereafter, and followed by another cascade of calls performing cleanup actions. Figure 3b shows the same program with static linking. The remaining overhead is mainly from the C library, although this programs does not make any explicit library calls, as evident from the call graph. In this example, the user code is only responsible for about 26% of the program with dynamic linking, and for 75% with static linking.

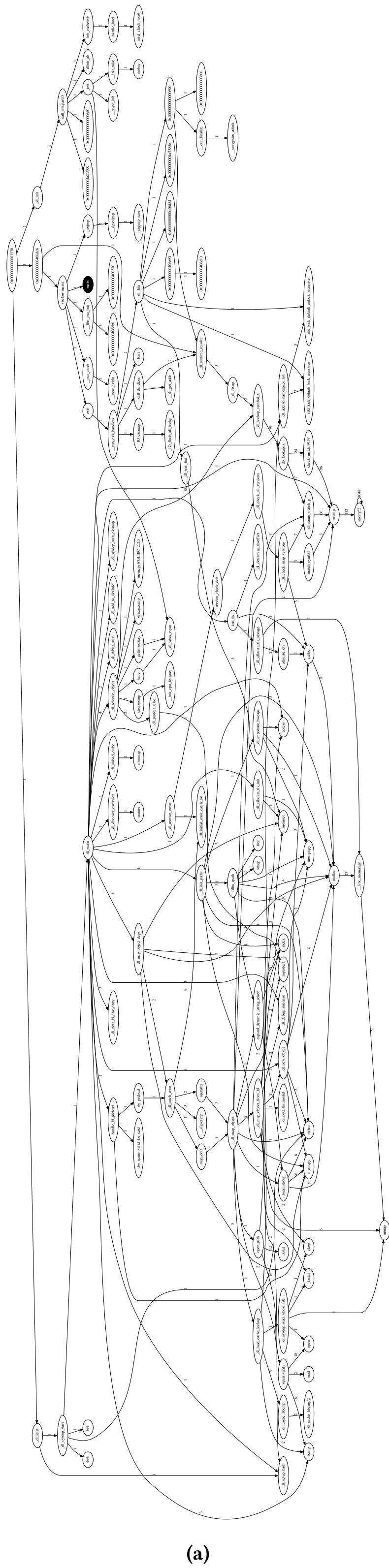
In summary, if we are not aware of such OS and library overhead when looking at the measurement results, we may draw conclusions that merely reflect the operating system and its libraries, rather than the software that we are trying to analyze and improve.

4 Measurement Setup

In this section, we propose a measurement setup on SMP Linux, that minimizes measurement errors and thus provides a faithful quantification of the software under analysis, while suppressing other influences as far as possible.

All event interactions described in previous sections are highly depending on the microarchitecture, the operating system and its configuration, next to the program under analysis itself. We describe our measurement setup for the system detailed in Table 1, as a representative of an advanced superscalar out-of-order processor with operating system, MMU, caches, prefetchers and so on. For different targets or OSs, the setup has to be adapted accordingly.

[3]We have observed a 10% decline in throughput in the STREAM benchmark, when execued on a dual-monitor setup vs. single-monitor



(a)

(b)

Figure 3. Callgraphs of a short-living program, suggesting that most of the work being done is OS “overhead”. The user function is highlighted in black. Both figures show the same program, where (a) is dynamically linked and (b) statically linked.

Source	Class	Impact	When	Note
Speed Stepping/Frequency Scaling	HW	varying execution time	immediate	
TLB Shootdown	HW	slower address translation	lagging	
DMA Transfers	HW/SW	slower memory access	immediate	(h)
Cache Coherency*	HW	additional accesses to cache	immediate & lagging	(a)
Hardware Prefetcher*	HW	more cache & TLB accesses, less or more misses	immediate & lagging	(b)
Load Balancing/Migrations	SW	more & TLB accesses, less or more misses	immediate & lagging	
Interrupts	HW/SW		immediate & lagging	(c)
Context Switches	SW		immediate & lagging	
Mode Switches	SW		immediate & lagging	(d)
Allocator	SW		immediate & lagging	(e)
Scheduler	SW		immediate & lagging	(f)
Major Page Fault*	SW/HW		immediate & lagging	(g)
Peripherals	HW		immediate & lagging	(i)

Table 3. Summary of sources for measurement errors. Sources marked with (*) may or may not be considered creating measurement errors, depending on the intended observation.

The main focus of our setup is how to control variables that would otherwise lead to the measurement errors listed in the previous section. Therefore, many of the suggestions given here are about parameters and configuration of both the OS and the CPU. In the end, we briefly describe the measurement process itself.

4.1 Control Variables

4.1.1 Isolating and Pinning the Process

First, one or more (yet not all) CPU cores were isolated from the scheduler and SMP balancing, which prevents other userspace tasks from interfering (kernel boot parameter `isolcpus`). We recommend to leave out CPU0, since one core is required to process the offloaded tasks, and this core usually has interrupts that cannot be moved to other cores (e.g., some DMA controllers). Additionally, Hyperthreading was turned off in BIOS, to avoid hardware context switches, same-core migrations and some known errata with the hardware counters. The process under analysis was subsequently pinned to the isolated cores with command line tool `taskset`. Note that each subprocess/thread needs pinning if a range of cores is isolated, since otherwise migrations might happen.

Interrupts: The affinity of interrupts was set to the non-isolated cores, preventing them from skewing the measurements[4]. *Thermal interrupts* were prevented by throttling the CPU speed such that full load does not lead to critical temperatures. This can be done by setting the governor's limits in *sysfs* or tools like `cpufreq`. *Machine check exceptions* were disabled by kernel boot parameter `mce=off`.

Ensuring tickless operation: To minimize the number of context switches, a kernel with adaptive ticks should be used, and enabled with kernel boot parameter `nohz_full`. The remaining challenge is to keep kernel threads off the run queue, to prevent ticks from getting generated. There are several factors that can cause runnable kernel threads, which are discussed in [14]. One such factor are RCU calls, which should be disabled for the isolated CPUs with kernel boot parameter `rcu_nocbs`. Another reason are bottom halves of interrupts as discussed earlier, which can be prevented by setting the interrupt affinity.

Allocator Selection: Yet another reason why tickless operation fails, may be the kernel's memory allocator. Depending on the distribution, different algorithms (called "SLAB", "SLOB" or "SLUB") can be in use, with different impact on cache miss/hit metrics and execution time. Notably, the SLAB allocator requires periodic cleanups, which enables a kernel process and prevents tickless operation. The best option is the newest, and in Desktop distributions most commonly used allocator, the SLUB allocator (one exception is Debian, using SLAB and thus rendering adaptive ticks ineffective). Changing the allocator requires building a custom kernel.

Watchdogs: More context switches could be caused by the watchdog. This can be prevented with kernel boot parameters `nowatchdog` and `nosoftlockup`.

Real-Time Priority: Depending on the Linux version, some kernel threads [14] are still active on isolated cores and can cause unwanted context switches (e.g., *kworker*). To avoid this, we perform our measurements at real-time priority using the command line tool `chrt`. Additionally, real-time throttling must be disabled (by setting `sched_rt_runtime_us` in *procfs*), since otherwise some Linux versions force one involuntary context switch per second for CPU-bound tasks.

Summary: The show configuration results in no other user processes running on the isolated cores, and in no CPU migrations taking place of our process(es). Some interrupts may

[4] `/proc/irq/default_smp_affinity`

still be left, but these should only cause mode switches. This can be verified using `perf`[5].

4.1.2 Fixing Processor Speed

If the processor supports speed stepping/frequency scaling, then the clock speed of the processor should be fixed for two reasons. First, if the absolute execution time is one desired measure, a non-fixed clock speed could yield different results in subsequent runs, depending on how the governor selects the frequency. Second, we conservatively prevent a potential impact of frequency scaling on processor cycles; usually there is not enough information in the data sheets to conclude there is no influence.

Further, fixing the clock speed also ensures more stable DRAM access times when measured in processor clock cycles – DRAM runs at its own bus frequency, which means that faster CPU frequency will make DRAM bottlenecks stand out more. This is important for backend-bound benchmarks, where the bottleneck becomes more severe with increasing processor frequency.

Fixing the processor speed is best done in BIOS by disabling technologies like speed stepping. However, it has to be ensured that the processor has sufficient cooling under full load, which often is not the case for mobile or Laptop devices. Alternatively, for Intel CPUs, the older ACPI driver can be activated with boot parameter `intel_pstate=disable`, which supports to fix the frequency of the processor.

4.1.3 Controlling TLB Flushes and Shootdowns

Even though we have isolated cores from the scheduler, shootdowns may still happen because the Linux kernel runs on the same core as the process it is serving. Thus, every core that is used executes a part of the kernel at some point, where shared kernel data can lead to shootdowns.

In our kernel version KPTI is enabled, yet PCID support is not available. Therefore, each mode switch flushes the TLB, significantly skewing our measurements (see Section 2.2.4). We thus disabled KPTI with kernel boot option `pti=off`[6]. This step is not recommended when PCIDs are supported by both OS and CPU.

4.1.4 Controlling Page Faults

As earlier, we assume the user wants to avoid major page faults as far as possible, since they have a penalty large enough to hide other effects that may be of interest. Also, there is not much the user can do if access to secondary storage is required, unless the logic of the program is reworked.

In Linux, reading or writing data files is by default buffered through the virtual memory. A read-ahead heuristic is used to bring data from the slow disk to RAM, as soon as a demand is foreseen. However, while this mechanism prevents major page faults quite effectively, it causes DMA transfers and might not be desirable. Once a file is buffered in virtual memory ("page cache"), data can be accessed without disk access taking place. Obviously, the page cache can only prevent all disk access if it is large enough to hold all files that will be in use, and not flushed in between.

The easiest way to make use of the page cache is to run the measurement twice, and discarding the results from the first run. Alternatively, there exist tools which allow to check and manipulate the page cache, e.g., `vmtouch`.

4.1.5 Controlling Hardware Data Prefetching

Although it is possible to distinguish some events by their cause (i.e., demand vs. speculation), it cannot be known how much of the prefetcher-induced penalties are hidden in out-of-order processing, or how many of certain events are caused by the software directly. One shortcut to answer this question, is to turn off hardware prefetching and run the benchmarks twice, then compare results. This may not possible on all CPUs, and is specific to every processor family. For Intel CPUs, machine-specific registers (MSRs) allow configuring the processor in many ways, inter alia turning off prefetchers.

4.1.6 Controlling Influences of Peripherals

If peripherals are sharing memory with the software under analysis, it might be helpful to turn them off or minimize their impact, in case an influence on the results cannot be ruled out. For example, the graphics engine in Sandy Bridge can be made to relinquish parts of the L3 cache by booting into a low-resolution mode.

4.2 Taking Measurements

The act of taking measurements itself is straightforward, and should provide meaningful results if the previous recommendations have been followed. We therefore summarize this only briefly.

4.2.1 Performance Monitoring Units

While there exist different ways and tools to measure performance, such as hardware tracing and countless tools, we briefly describe the use of the CPU's *performance monitoring unit* (PMU) [11], which provides event counters. These counters are processor-specific registers that are incremented on the occurrence of certain events, for example, the number of L1 cache misses, or the number of processor cycles. Under Linux, the `perf` tool allows setting up and reading these registers, and to separate them by process. A growing number of CPUs and SoCs offer an equivalent to the PMU, e.g., many ARM Cortex processors already do.

Additional to hardware events, the `perf` also reads kernel (software) counters, such as minor and major page faults, context switches and CPU migrations. We give the name of the used counters at the beginning of each following section.

[5]`perf record -e "sched:sched_switch"` ...

[6]Note that this may open a security vulnerability!

Note: We highly recommend to use the native names of the registers, to avoid misunderstandings (e.g., the `perf` tool counts *STLB hits* under the name *ITLB loads* on Sandy Bridge), and to watch out for errata (such as counter problems under HyperThreading).

Multiplex and Grouping: PMUs have a limited number of hardware counters (eight, in our case). If we request more events than counters, `perf` starts time-multiplexing, and extrapolating the results from the sampling window to the entire life time of the process. It is thus possible to miss certain events, if the specific counter is not currently active. Therefore, if the process behavior is not stationary for long enough, the results may appear inconsistent. One way around this issue is grouping of counters, which tries to multiplex all counters in a group at the same time. However, some processors have scheduling restrictions for which counters must not be used together. One way around this is to avoid multiplexing, possible if the workload is repeatable. With Boolean options for multiplexing (M) and grouping (G), there are four different possibilities to measure, each with its own consequences:

1. *Grouping and Multiplexing*: Possibly unsafe and incomplete results, because the process might be undersampled from M., and G. may create scheduling conflicts, which disables some counters.
2. *No Grouping and Multiplexing*: Possibly unsafe but complete, because the process might be undersampled and counters inconsistent towards each other, but groups can be built in arbitrary ways to resolve scheduling conflicts.
3. *No Multiplexing and no Grouping*: Possibly unsafe but complete, only works for repeatable workloads. Multiplexing, although not enabled, might still take place implicitly in an attempt to resolve conflicting counters.
4. *No Multiplexing and Grouping*: Safe but possibly incomplete, only works for repeatable workloads. The grouping prevents implicit multiplexing from taking place.

By *unsafe*, we mean that the counters may be both imprecise and contradict each other (e.g., it might be possible that the L2 miss count is greater than the L3 access count). By *complete*, we mean that every event that has been requested was indeed counted at some point in time.

Recently, *weak* groups have been introduced in `perf` [16], which can be broken to resolve scheduling conflicts, but otherwise ensure that certain events are counted synchronously. It is thus a mixture of cases 1 and 2, resulting in complete results with minimal multiplexing artifacts, applicable to non-repeatable workloads.

We suggest to use grouping and avoid multiplexing as long as the workload is repeatable, to obtain self-consistent and precise results. Otherwise, grouping and multiplexing should be used, preferably using weak groups.

For the system described in Tab. 1, the CPU supports around 475 different events, offering deep insights into the processor. However, looking at single counter values can be misleading. Counters can have several orders of magnitude in difference (e.g., page faults vs. cache misses), yet have a similar impact on the resulting performance. Especially when comparing two versions of the same program, large differences may become meaningless when compared to the absolute values.

4.2.2 Hierarchical Bottleneck Analysis

To identify the true bottleneck of an application, Yasin has proposed a hierarchical analysis [33]. The analysis follows the hierarchical anatomy of general OoO processors, depicted in Fig.4. As a first level, the user should only be concerned whether the majority of cycles is spent in the Frontend (entails decoding instructions, L1d access and Microcode switches), or in Bad Speculation (machine clears and branch mispredictions) or in the Backend (L1/2/3 and DRAM data access) or whether most of the time is spent in retiring instructions (ideally, 100%). Once the most time-consuming category is identified, the user should focus on its children at the next level, determine the most prominent one, and so on. For a full explanation, please refer to [33].

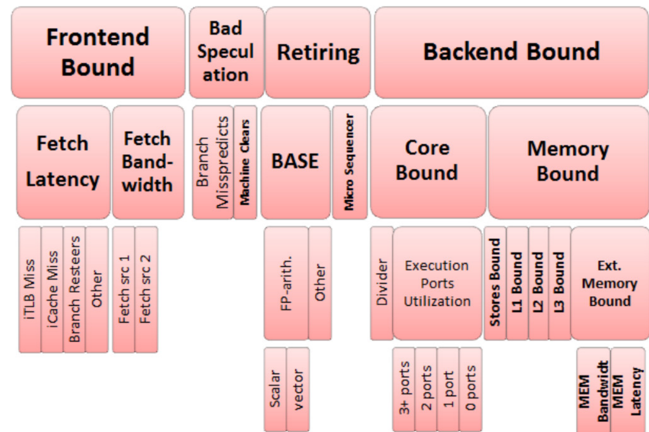


Figure 4. Hierarchical View on Processor Performance [33].

A free implementation of this analysis for Intel CPUs is available as a tool called `toplev`, which is part of Intel's `pmu-tools` [17]. It takes measurements using Linux' `perf` tool, and presents the results in the explained hierarchy, but in ratios rather than absolute numbers. Multiplexing and grouping is supported as described earlier. Taking the measurements thus boils down to invoking the tool.

Only after the bottleneck has been identified, individual counters should be inspected using `perf`, since now we know they are meaningful. Last but not least, it is worth noting that `pmu-tools` also includes a convenience wrapper for `perf`, called `ocperf`. This tool allows using native names for Intel CPUs, and better supports uncore events.

It should be noted that `toplev` does not allow localizing causes of undesired behavior in the source code, since there is no localization available. An approach for this is provided by `perf` in *sampling* mode, where a history of events can be logged, and attributed to source code locations. The tool also offers a lot of options given in the documentation [12].

4.2.3 Uncertainty Propagation

All measured events can be associated with a measurement uncertainty. This uncertainty indicates the precision of the measured values, and prevents us from drawing false conclusions if we are comparing two versions of a software.

Crucially, not all events can be measured directly on the hardware, and thus must be calculated from other, measured or itself calculated, ones. The calculated events therefore have an uncertainty based on their constituents, which needs to be properly tracked. As an example, on Sandy Bridge the cache hit ratio r must be estimated from access a and hit h counters, and its standard deviation σ_r depends on both measurements as follows

$$\sigma_r = \left|\frac{h}{a}\right| \sqrt{\left(\frac{\sigma_h}{h}\right)^2 + \left(\frac{\sigma_a}{a}\right)^2 - 2\frac{\sigma_{a,h}}{ah}}, \quad (1)$$

where $\sigma_{a,h}$ is the covariance between accesses and hits. Analogously, the uncertainty is propagated for multiplication, division, addition, and all other operations [19].

We have contributed patches to `toplev`, which track the standard deviation of counters across multiple runs (`--repeat`) through all calculations, while assuming statistically independent variables, i.e., $\sigma_{a,h} = 0$. The resulting uncertainty for each event is indicated with error bars in our plots.

Warning: When measurements are taken in system-wide mode across more than one logical processor core, then `toplev` currently forces `perf` to not aggregate the results (flag `-A`). This in turn suppresses `perf`'s output of the standard deviation, and leads to an optimistic uncertainty. System-wide mode is also forced when HyperThreading (HT) is activated, thus HT suppresses the standard deviation, as well. There are thus three options to capture measurement uncertainty correctly: (1) **Recommended:** Disable HT and avoid system-wide mode. The `perf` tool will "follow" the software under analysis to whatever core it is going (if not pinned). (2) **Alternative 1:** Disable HT and use system-wide mode while filtering only for only one single core. The software under analysis must be pinned to that single core. (3) **Alternative 2:** Keep HT and use system-wide mode, and specify `--single-thread`, which forces `toplev` to aggregate the results across all CPUs. The system should otherwise be idle. In all cases, the core where the software is running on should be isolated, as explained in Section 4. Future releases of `perf`/`toplev` might no longer have this caveat.

5 Experiments

We provide a few short examples illustrating the difference between our proposed measurement setup, and a default Linux environment. All measurements have been taken with similar parameters, using the performance monitoring counters (see Section 4.2.1. Specifically, multiplexing and HyperThreading were disabled, to prevent undersampling and to ensure proper uncertainty propagation, as explained in Section 4.2.2.

5.1 Hierarchical Bottleneck Analysis

We continue by showing two specific examples of the hierarchical analysis introduced earlier. Towards this, we have chosen two programs with different characteristics:

- *gnugo*: This is a game engine for the Chinese Go game. The program consists of many branches and jumps, many of which are hard to predict. Additionally, it has a low spatial locality of instructions, and will therefore like suffer from many instruction cache misses. Memory usage is low compared to the next program.
- *stream*: This is McCalpin's memory bandwidth benchmark [23], which stresses the memory subsystem heavily, but in a predictable pattern. Consequently, this program occupies a lot of memory (1.1GB in our case), but has few branches or jumps.
- *syscalls*: This is the program shown in Section A, which exercises mode switches. The memory usage is low compared to the others, but it causes many mode switches thus allows more context switches to take place. This program should keep the functional units busy.

All programs have been executed five times in a row to determine a standard deviation, which is used for our uncertainty propagation. We have repeated this for two different measurement setups; first with the default settings of the system described in Tab. 1, and another time with our proposed measurement setup described in Section 4.

Figure 5 shows the first level of the hierarchical analysis from `toplev`. It can be seen that the measurements all programs reflect the expected behavior: *gnugo* indeed spends a lot of time in its frontend (FE), due to many cache misses and branch mispredictions (see Fig.6). The *stream* benchmark spends most of its time in the backend (BE), because it is mostly memory-bound (see Fig.7). The *syscalls* program shows itself to be mainly backend-heavy, but this time due to core usage (see Fig.8).

The difference between the default setup and ours is however barely visible for this type of analysis. Only *syscalls* shows a larger difference. The results with our proposed setup make the program appear less back-end bound, in exchange for more time spent retiring instructions. This hierarchical analysis does not offer any more explanation, and thus we will revisit this in the next section.

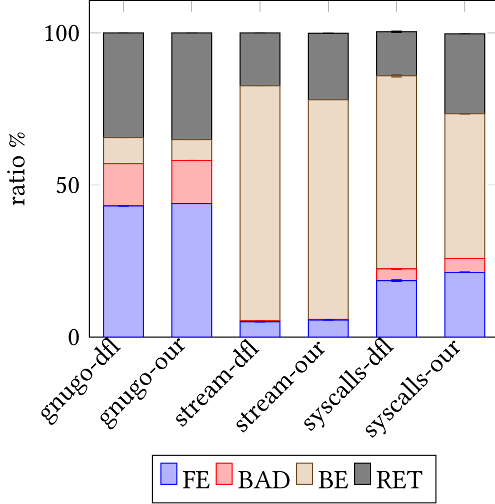


Figure 5. Results of hierarchical performance analysis for default setup (dfl) and our proposed setup (our).

All figures have error bars, but only very few of them are visible due to their low magnitude. This suggests that a default setup may be sufficient in terms of measurement uncertainty, and that the results of a hierarchical analysis may be close enough between the two setups, at least for the programs tested here. This is surprising, because the absolute values of the counters, as well as program performance, differ significantly, as we show next.

5.2 Absolute Event Counts and Absolute Performance

Hierarchical analysis has not been showing any larger differences between the measurement setups. It builds on ratios rather than absolute values, therefore large differences could become invisible. While this is acceptable and desirable for bottleneck analysis, it might be undesirable in other scenarios, such as when absolute performance or the precise event counts are required. These absolute counts differ significantly, as we show in the following.

Figure 9 shows the absolute event counts of *gnugo* for our proposed setup ("tune") and for the default one. First, it can be seen that the program runs faster under the default setup. The bar "task-clock (msec)" shows 11.4s vs 12.9s; note that task-clock only increments for those cycles where the program has been active on the processor, which is always less or equal than wall-clock time. As a better metric reflecting wall-clock time, all plots show a bar called "IPC0", which is calculated as $I/(f \cdot t_w)$, where I is the number of instructions, f the processor frequency, and t_w the wall-clock time of the program. In Fig.9, IPC0 is almost identical for both measurement setups, thus the programs would take about the same time to execute in both setups.

The next apparent difference is the absence of context switches (one switch is always necessary) and CPU migrations in our setup, followed by no major faults in our setup, and a different L1d caching behavior. All in all, the measurement uncertainty again is comparable when the counts are close.

The results for *stream* and *syscalls* are shown in Appendix B. Again, there are some differences in the absolute event counts. This time, IPC0 shows that *stream* runs faster in our setup ($0.6/0.8 = 25\%$), whereas *syscalls* is significantly slower ($0.68/0.15 = 450\%$), as expected due to the higher overhead for mode switches. Furthermore, *syscalls* also shows large differences in most counters. The entire characteristics seem to be driven by the differences in the mode switch cost, and even outweighs that the default setup performs many context switches and CPU migrations.

In conclusion, the measurement setup can make a large difference in absolute event counts and software performance. Our experiments suggest that neither setup is always dominating in performance, and that the choice of setup should depend on the software under analysis.

5.3 Stability towards Background Processes

So far, all measurements have been taken on an idle system, with no interactive user processes being executed. We now examine how measurements change when we run some background processes, as often the case in production systems. Specifically, we run four background processes which stress caches and DRAM (`stress-ng -C2 --vm 2 --vm-bytes=512m --vm-populate`).

5.3.1 Hierarchical Analysis

Figure 10 shows the results of a hierarchical analysis, which certify that all measurements under all setups are stable. There are small differences, which however do not change results fundamentally. Again, the measurement uncertainty surprisingly is always very close. The results suggest that the hierarchical analysis is robust against background processes, at least at the uppermost level.

5.3.2 Absolute Event Counts and Absolute Performance

The absolute event counts give a similar picture as before. However, the wall-clock time of the programs is considerably more stable in our setup.

Figure 11 shows again absolute event counts for both measurement setups of *gnugo*, when background processes are running. A very large change in IPC0 can be seen. In our setup, the program almost maintains the same performance as on an idle system (4% degradation, since the IPC0 drops from 1.25 to 1.21), whereas in the default setup performance significantly degrades (64% degradation, since IPC0 drops from 1.22 to 0.45). The result is similar for *stream*, which degrades by 16% in our setup, but by 67% in the default setup,

```
FE        Frontend_Bound:                                   43.94 +- 0.02 % Slots            <==
BAD       Bad_Speculation:                                  14.16 +- 0.01 % Slots
RET       Retiring:                                         35.07 +- 0.01 % Slots below
FE        Frontend_Bound.Frontend_Latency:                  22.77 +- 0.03 % Slots
FE        Frontend_Bound.Frontend_Bandwidth:                21.13 +- 0.03 % Slots
BAD       Bad_Speculation.Branch_Mispredicts:               14.07 +- 0.01 % Slots
FE        Frontend_Bound.Frontend_Latency.ICache_Misses:    17.66 +- 0.00 % Clocks_Estimated
FE        Frontend_Bound.Frontend_Latency.Branch_Resteers:  14.07 +- 0.01 % Clocks_Estimated
```

Figure 6. `toplev` output for *gnugo* with our proposed measurement setup.

```
FE        Frontend_Bound:                                    5.62 +- 0.00 % Slots below
BAD       Bad_Speculation:                                   0.24 +- 0.00 % Slots below
BE        Backend_Bound:                                    72.20 +- 0.00 % Slots
RET       Retiring:                                         21.94 +- 0.00 % Slots below
BE/Mem    Backend_Bound.Memory_Bound:                       52.55 +- 0.02 % Slots            <==
BE/Core   Backend_Bound.Core_Bound:                         19.65 +- 0.02 % Slots
BE/Mem    Backend_Bound.Memory_Bound.L1_Bound:               0.00 +- 0.00 % Stalls below
BE/Mem    Backend_Bound.Memory_Bound.L2_Bound:               1.73 +- 0.02 % Stalls below
BE/Mem    Backend_Bound.Memory_Bound.L3_Bound:              17.80 +- 0.03 % Stalls below
BE/Mem    Backend_Bound.Memory_Bound.DRAM_Bound:            36.13 +- 0.04 % Stalls below
BE/Mem    Backend_Bound.Memory_Bound.Store_Bound:           10.27 +- 0.01 % Stalls below
BE/Core   Backend_Bound.Core_Bound.Ports_Utilization:       21.92 +- 0.03 % Clocks
```

Figure 7. `toplev` output for *stream* with our proposed measurement setup.

```
FE        Frontend_Bound:                                   21.40 +- 0.05 % Slots
BE        Backend_Bound:                                    47.52 +- 0.08 % Slots
RET       Retiring:                                         26.31 +- 0.01 % Slots below
FE        Frontend_Bound.Frontend_Latency:                  12.94 +- 0.03 % Slots below
BE/Mem    Backend_Bound.Memory_Bound:                        0.00 +- 0.00 % Slots below
BE/Core   Backend_Bound.Core_Bound:                         47.52 +- 0.08 % Slots
BE/Core   Backend_Bound.Core_Bound.Ports_Utilization:       39.88 +- 0.20 % Clocks           <==
```

Figure 8. `toplev` output for *syscalls* with our proposed measurement setup.

and for *syscalls*, which degrades by 60% in the default setup, but only by 2% in our setup.

In summary, the absolute event counts are considerably different in our setup, and additionally the performance of the program under background load is significantly better maintained in our setup.

5.4 Other Influences

When performance counters are used, there is a low but visible overhead, as with many other measurement methods. We have neglected this overhead in our experiments.

6 Related Work

Hardware Performance Counters: Since all performance counters are vendor- and processor-specific, different profiling software exists to access them. A generic tool is Linux' `perf`, which also supports some many AMD, ARM, ARC, Blackfin, MIPS, PowerPC and SPARC processors [30]. Specific to Intel CPUs, we have `toplev`, which we have mentioned earlier [17], and the commercial tool *VTune*. Interestingly, some processors also provide counters for power events [27], for which our setup might be equally important.

Software-based Measurements: There exist many well-known tools for debugging software performance. Perhaps best known are software profilers, such as `gprof` (execution time) and Visual Studio Profiler. More generally, there exists the larger class of dynamic program analyzers, which observe programs during run-time, sometimes by instrumentation, other times through simulation. A survey about those was recently given in [9]. One successful framework worth mentioning is Valgrind [26], which provides tools to track execution time and memory usage. However, such tools change the program's characteristics through instrumentation or simulation, and furthermore do not provide any microarchitectural explanations. Finally, we should also mention `ftrace`, which can collect software events during the execution of a program, and is integrated with `perf`.

Bottleneck Analysis: An alternative hierarchical model to Yasin's [33] is given in the Intel's Optimization Reference Manual [11, §B.7] as "Drill-Down Technique".

Reducing OS Noise: Another attempt at reducing the OS' impact on workloads has been described by Akkan et. al [1]. However, their report is mainly concerned with the OS, and furthermore neglects some important effects (e.g., Hyper-Threading, ftrace overhead and tick length). Nevertheless,

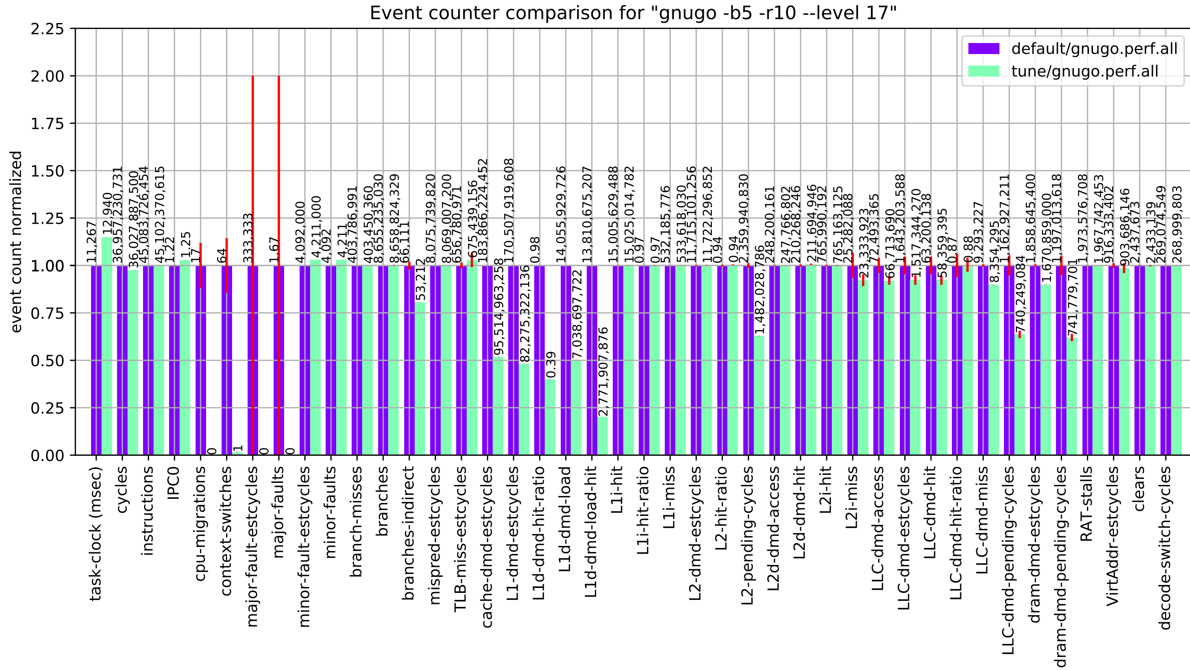


Figure 9. Absolute event counts for *gnugo* for default setup (dfl) and our proposed setup (our). Red error bars indicate measurement uncertainty.

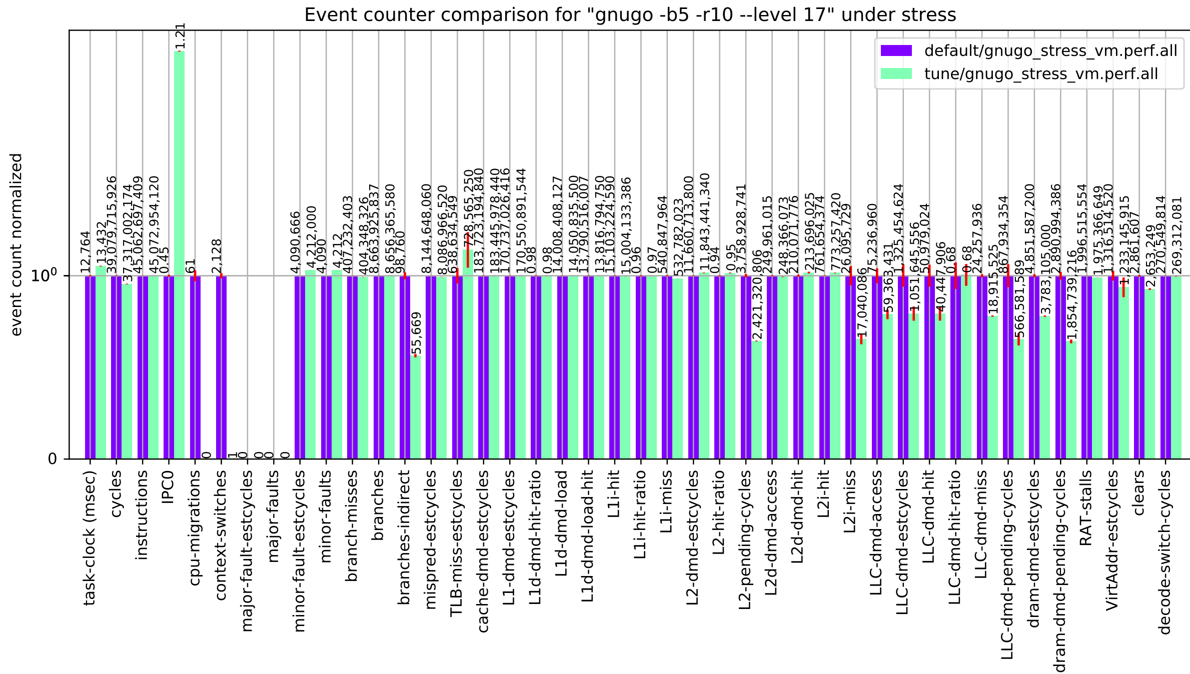


Figure 11. Absolute event counts for *gnugo* for default setup (dfl) and our proposed setup (our) with background processes. Red error bars indicate measurement uncertainty. Large performance differences are visible.

some pointers can be found there. For technical details in various aspects on Linux, the *Linux Kernel Mailing List* (LKML), the kernel docs, and LWN have been invaluable primary sources for understanding the inner workings of the kernel.

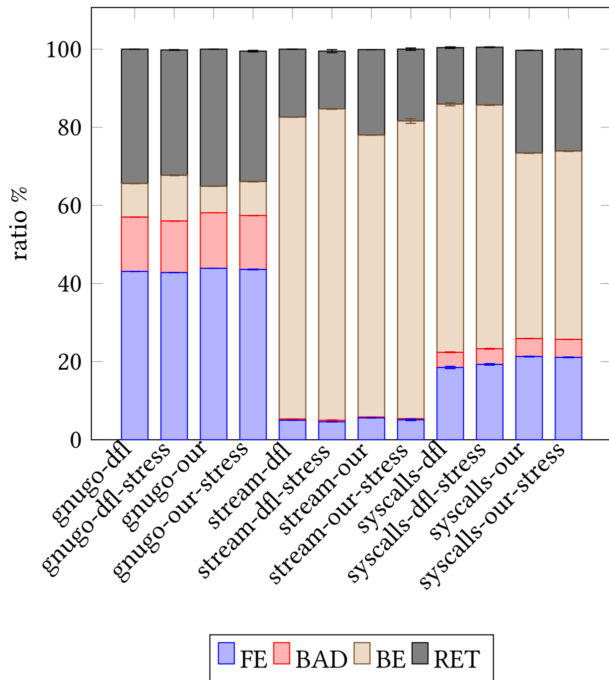


Figure 10. Results of hierarchical performance analysis with and without highly active background processes, for default setup (dfl) and our proposed setup (our).

We have given the references where applicable, and many more can be found by the interested reader. Last but not least, to erase even the faintest doubt of imprecise or deprecated documentation, the Linux source code itself serves as an instructor.

7 Concluding Remarks

We have described the most important features of both the Linux operating system and a modern out-of-order superscalar processor, and have how they can influence software performance. Based on the identified influences, we have proposed a measurement setup that aims to reduce measurement errors, such that not the OS or hardware is characterized, but the program under analysis.

Towards that, we have extended a hierarchical performance analysis method with uncertainty propagation, to indicate the measurement errors in percentages of the obtained values. Surprisingly, our experiments showed that this hierarchical and ratio-based performance analysis method, the measurement setup makes little difference. For absolute event counts, e.g., when cache access counts shall be correlated to source code, our setup has shown significant improvements.

As a side effect, we found that the proposed setup is more immune to active background processes than a default setup. Programs show considerably better stability in observable performance under our setup, whereas a performance degradation of up to 64% has been observed in a default setup. Our proposed setup can therefore be used as a guideline to tune the system for high-performance applications.

Naturally, the results presented here may change with different processors and OS versions. We have given many references to evaluate which parts of the setup may become relevant, but an extrapolation of the presented results to all targets would be unsafe.

In conclusion, the ultimate measurement setup does not exist. It depends on both the requirements to the measurements and the characteristics of the software under analysis. For more robust measurements, we recommend our proposed setup. This may however degrade performance for some programs, and thus of the presented configurations might be chosen to achieve optimal results.

References

[1] Hakan Akkan, Michael Lang, and Lorie Liebrock. 2013. Understanding and isolating the noise in the Linux kernel. *The International Journal of High Performance Computing Applications* 27, 2 (2013), 136–146.

[2] Robertp Arcomano. 2018. TLDP Kernel Analysis HowTo. Online. https://www.tldp.org/HOWTO/KernelAnalysis-HOWTO-3.html, retrieved 2018 Oct. 30.

[3] At32Hz. 2017. Sandy Bridge Block Diagram. Online. https://en.wikichip.org/wiki/File:sandy_bridge_block_diagram.svg, retrieved 2018 Oct 30.

[4] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. Online. https://lwn.net/Articles/738975/, retrieved 2018 Oct 30.

[5] Jonathan Corbet. 2018. Meltdown strikes back: the L1 terminal fault vulnerability. Online. https://lwn.net/Articles/762570/, retrieved 2018 Oct 30.

[6] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11 (2007), 2007.

[7] Agner Fog. 2018. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. online. Retrieved 2018 Sept 3rd.

[8] Mel Gorman. 2007. *Understanding the Linux Virtual Memory Manager*. Open Publication License. Available at https://www.kernel.org/doc/gorman/html/understand/.

[9] Anjana Gosain and Ganga Sharma. 2015. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Springer, 113–122.

[10] Nur Hussein. 2017. Another attempt at speculative page-fault handling. Online. https://lwn.net/Articles/730531/.

[11] Intel Corp. 2018. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corp.

[12] kernel.org. 2018. Linux kernel profiling with perf. Online. https://perf.wiki.kernel.org/index.php/Tutorial.

[13] kernel.org. 2018. NO_HZ: Reducing Scheduling-Clock Ticks. Online. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt, retrieved 2018 Oct 30.

[14] kernel.org. 2018. Reducing OS jitter due to per-cpu kthreads. Online. https://www.kernel.org/doc/Documentation/kernel-per-CPU-kthreads.txt, retrieved 2018 Oct 30.

[15] kernel.org. 2018. Software Interrupt Context: Softirqs and Tasklets. Online. https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html, retrieved 2018 Oct. 30.

[16] Andi Kleen. 2017. Support standalone metrics and metric groups for perf. Online. https://lwn.net/Articles/732567/, retrieved 2018 Oct 30.

[17] Andi Kleen. 2018. pmu-tools. github repository.

[18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[19] Harry H Ku et al. 1966. Notes on the use of propagation of error formulas. *J. Res. Nat. Bur. Standards* 70, 4 (1966).

[20] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2.

[21] ARM Ltd. 2016. *Cortex-A57 Software Optimization Guide* (arm uan 0015b ed.).

[22] Andy Lutomirski. 2017. PCID and improved laziness. Online. https://lkml.org/lkml/2017/6/14/16, retrieved 2018 Oct 30.

[23] John D McCalpin. 1995. Sustainable memory bandwidth in current high performance computers. *Silicon Graphics Inc* (1995).

[24] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[25] David Miller, Richard Henderson, and Jakub Jelinek. 2018. Dynamic DMA mapping Guide. Online. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt, retrieved 2018 Oct 30.

[26] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[27] Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real Time Power Estimation and Thread Scheduling via Performance Counters. *SIGARCH Comput. Archit. News* 37, 2 (July 2009), 46–55. https://doi.org/10.1145/1577129.1577137

[28] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.

[29] Linus Torvalds. 2014. Google Plus. https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6.

[30] Vince Weaver. 2018. Linux support for various PMUs. Online. http://web.eece.maine.edu/~vweaver/projects/perf_events/support.html, retrieved 2018 Oct 30.

[31] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News* 23, 1 (1995), 20–24. https://doi.org/10.1145/216585.216588

[32] Jisoo Yang and Julian Seymour. 2018. Pmbench: A Micro-Benchmark for Profiling Paging Performance on a System with Low-Latency SSDs. In *Information Technology-New Generations*. Springer, 627–633.

[33] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

A Mode Switch Benchmark for x86

```c
#include <x86intrin.h>
#include <stdio.h>
#include <unistd.h>

#define TWENTY_MILLION 20000000
#define NTRIALS TWENTY_MILLION

int main(void) {
    unsigned long ini, end, now, best, tsc;
    int i;

#define measure_time(code) \
    for (i = 0; i < NTRIALS; i++) { \
        ini = __rdtsc(); \
        code; \
        end = __rdtsc(); \
        now = end - ini; \
        if (now < best) best = now; \
    }

    /* time rdtsc itself (i.e. no code) */
    best = ~0;
    measure_time (0);
    tsc = best;
    printf ("rdtsc: %li cycles\n", tsc);

    /* time one of the fastest syscalls */
    best = ~0;
    measure_time (getuid());
    printf ("getuid(): %li cycles\n", best-tsc);
    return 0;
}
```

B Absolute Event Counts

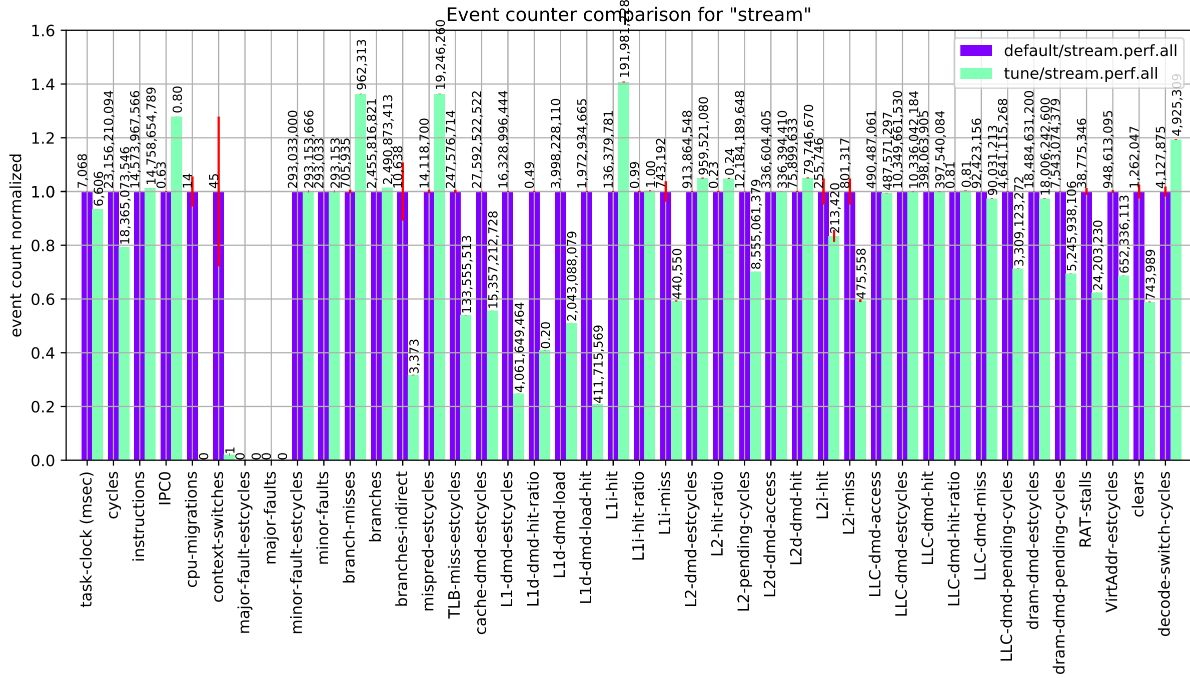


Figure 12. Absolute event counts for *stream* for default setup (dfl) and our proposed setup (our).

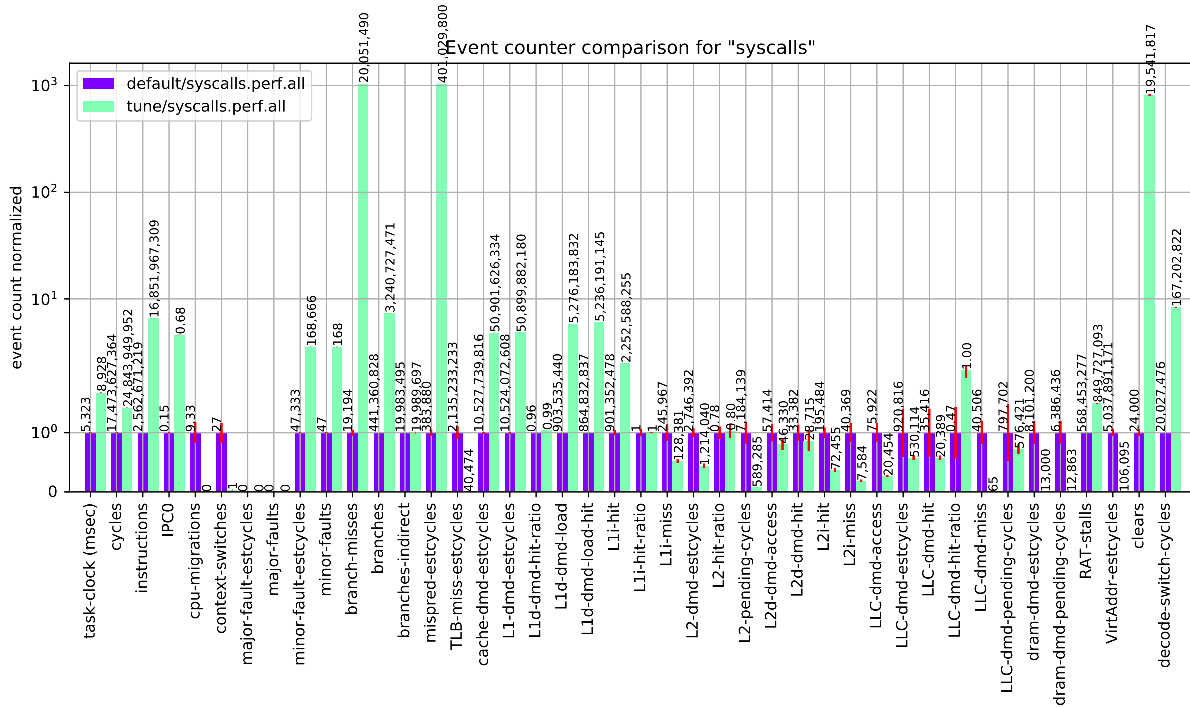


Figure 13. Absolute event counts for *syscalls* for default setup (dfl) and our proposed setup (our).

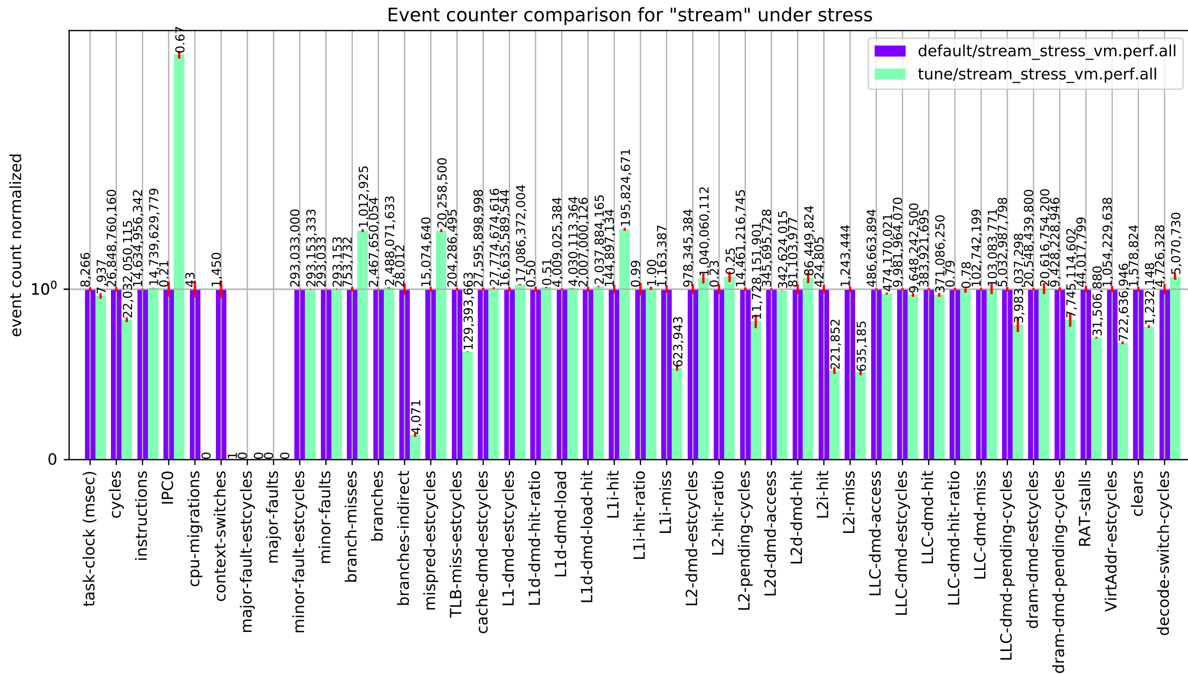


Figure 14. Absolute event counts for *stream* for default setup (dfl) and our proposed setup (our) with background processes.

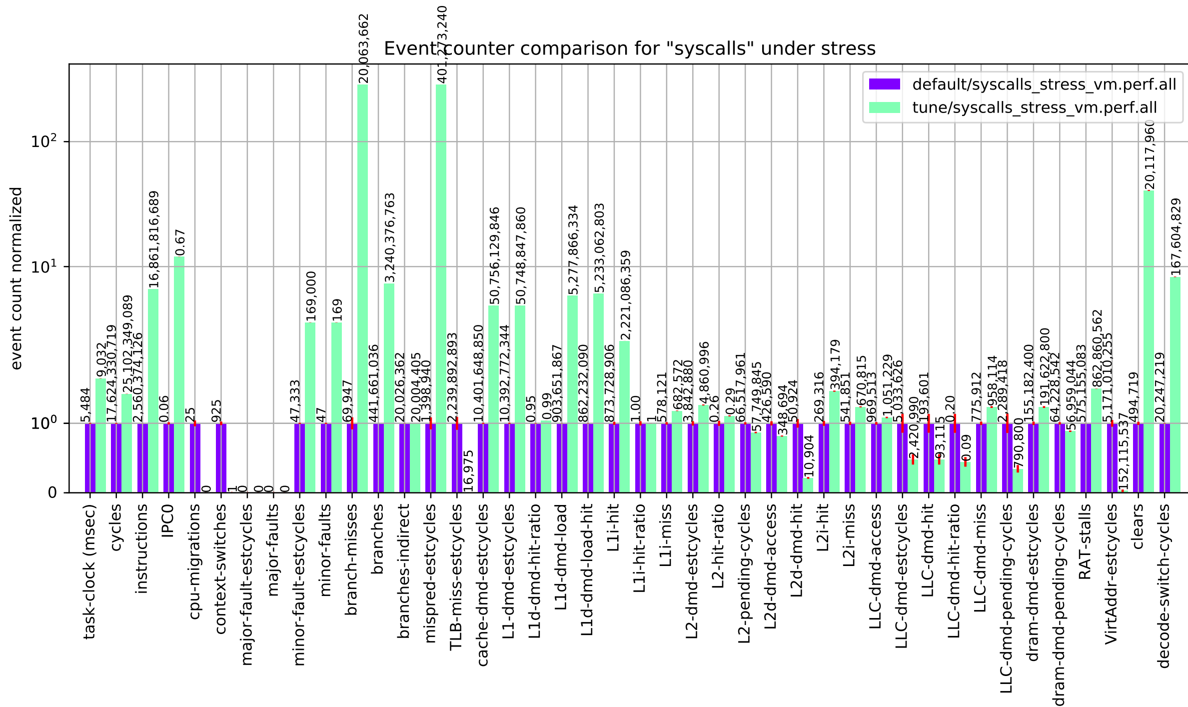


Figure 15. Absolute event counts for *syscalls* for default setup (dfl) and our proposed setup (our) with background processes.