Yamba: An Esoteric Language

Michael N. Gagnon

September 26, 2020

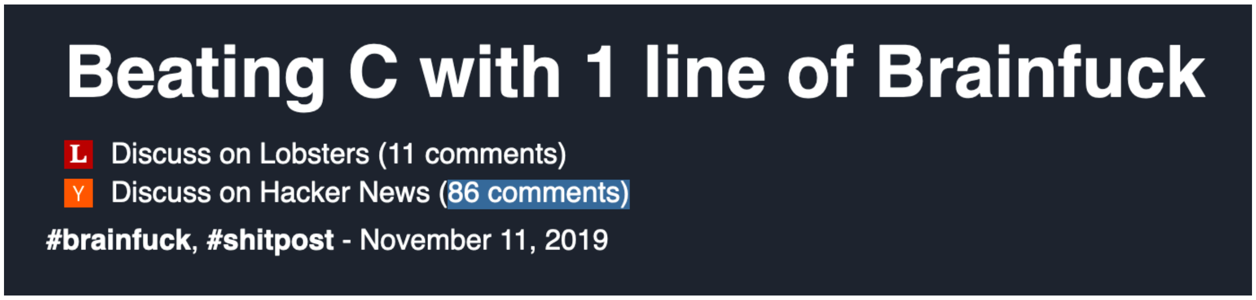
1. Brainfuck

For the fuck of it, hackers enjoy crafting and programming *esoteric languages*—languages that are very difficult to code upon, and/or languages that are very difficult to comprehend. Hackers can earn respect by performing interesting feats with esoteric languages, such as proving Turing completeness, or, better yet, inventing new esoteric languages.

The Brainfuck programming language is a favorite example. Here is a program in Brainfuck:

```
>>,+><[——————[---------------------[>+<<-[<+>+]>[-]]][-]]<[-]>>[<<+>>-]<,
+]<[-]<[>>+>+<<<-]>>>[<<<+>>>-]<<+>[<->[>++++++++++<[->-[>+>>]>[+[-
<+>]>+>>]<<<<<]>[-]++++++++[<++++++>-]>[<<+>>-]>[<<+>>-]<<]>]<[->>++++++++
[<++++++>-]]<[.[-]<]>++++++++++.
```

I haven't bothered to figure out what this program does, but it did score 86 points on Hacker News.[1]



Beating C with 1 line of Brainfuck

L Discuss on Lobsters (11 comments)

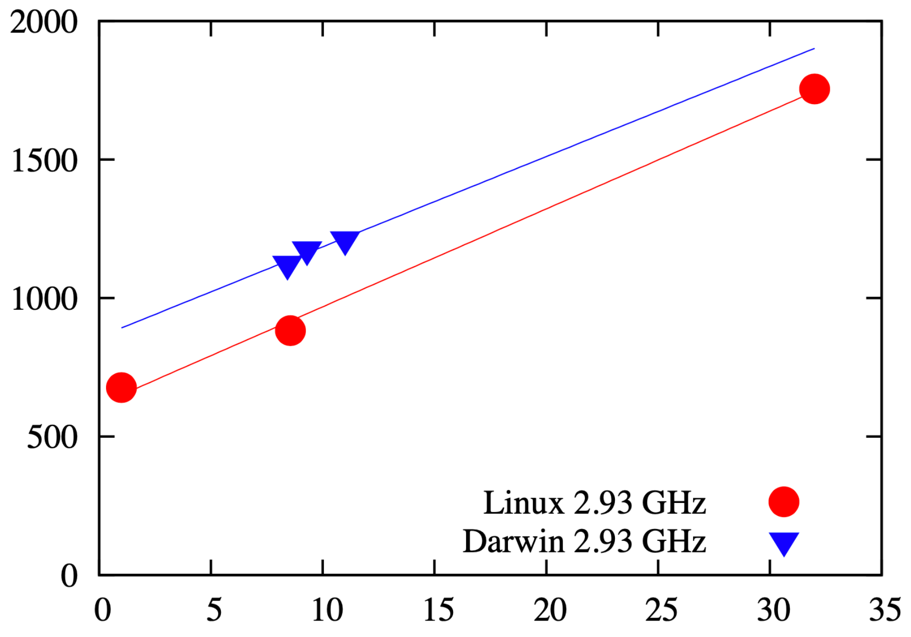
Y Discuss on Hacker News (86 comments)

#brainfuck, #shitpost - November 11, 2019

2. Introduction to Yamba

Yamba is the name of one of my esoteric programming languages. I first began developing Yamba as a student at Harvard, for a class project. I received a B instead an A, I think because (1) in my final report, I insulted hbench, which was a competing language developed by the professor who generated my grade, and (2) but really, most probably because I never got around to writing a compiler. I still haven't written a compiler for Yamba, but, nevertheless, I did mange to use Yamba to precisely compare and contrast pagefault latencies between Linux and Darwin.

[1] https://kiwec.net/blog/posts/beating-c-with-brainfuck/



3. Transclusion

Today, I am announcing a new Yamba macro. Here is a simple example:

```
#transclude “yamba.pdf”
```

Invoking this macro means transcend-and-include `yamba.pdf`. Transclude is thus a reference to both Ken Wilber[2] and my previous attack against Wikipedia,[3] for which the Pentagon invited me into their courtyard.[4]

[2] “What if we were to use all of the time and energy we invest to make other perspectives ‘wrong’ to instead find out what’s **right, true,** and **valuable** within them and *integrate* that with what we already know to be true? What if we aim to *acknowledge* the valuable partial truths in the model, *integrate* them with the valid insights from all other perspectives, and *transcend* the limitations of all of the models?” — Adam J. Pearson, six years ago: https://philosophadam.wordpress.com/2015/01/29/transcend-and-include-the-integral-attitude-to-competing-perspectives/amp/

[3] While developing and evaluating a security technology for DARPA, I studied the potential to attack Wikipedia by infecting transcluded pages with high-density workloads. See Section 6.2.10 of the Beer Garden Technical Report http://michaelgagnon.me/file/beergarden.pdf

[4] See Chapter “DARPA” on page 69 of my memoir: http://michaelgagnon.me/file/yana-zendo-and-the-powers.pdf

DARPA

Shortly after I joined Twitter, DARPA invited "boutique" hackers from across America to submit proposals for revolutionary cybersecurity research and development. DARPA is the military agency that invented the Internet, GPS, and stealth airplanes.

I proposed to fund myself and a colleague to develop and test a new defense against high-density cyberbombs, essentially picking up where my Harvard thesis left off. DARPA awarded me a six-month, six-figure grant, so I ended up working two full-time jobs for a period of time in Silicon Valley. I named my cyber defense: *Beer Garden.*

We built Beer Garden and tested it against a slew of attacks, targeting a variety of web applications, and it worked! Well, it only worked for half of the web applications we tested. The reason it didn't work for the other half, is because the "bouncer" component was prohibitively inefficient for web applications implemented in certain programming languages. But, in principle, with bespoke optimizations, Beer Garden could be adapted to suit any web application.

The Pentagon invited my colleague and me to present Beer Garden from their courtyard.

Experiments. We varied the attack rate and ran several experiments against both a conventional setup and against Beer Garden. The attack rate is important in this attack because each attack request must upload about 1.4 megabytes. Thus the attacker's connection speed to the server limits the attack rate. For example, if the attacker can upload at a rate of 6 mbps (megabits per second), then the attacker can sustain a rate of one request every two seconds (0.5 RPS).

At the lowest attack rate we tested, both the conventional and Beer Garden setups had relatively high success rates, though Beer Garden's was significantly better: 80% success for the conventional setup and 91% success for the Beer Garden setup. In addition to providing a better success rate, Beer Garden also provided better latencies. While the conventional setup completed 95% of requests within 26 seconds, Beer Garden completed 95% of authorized requests within 3 seconds and authorized 95% of requests within of 12 seconds.

Beer Garden's good performance here is due to the Bouncer evicting the attack requests, which alleviates CPU contention.

As the attack rate increased, the success rates dropped lower for both Beer Garden and the conventional setup. The conventional setup tended to have a better success rate, though we're not sure why. Success latencies remained mostly stable independent of the attack rate.

These experiments represent a moderate success of Beer Garden. Although the Doorman is ineffective against this attack, inherent bandwidth limitations assume its role. A single, bandwidth- constrained attacker will only be able to submit attack requests at a low rate. In this setting, Beer Garden provides good service levels during attack.

6.2.10 Infected-transclusion attack (sophisticated)

Summary. This attack is one of three attacks that demonstrates that if an attack can cause legitimate requests to become expensive, then the attack can evade Beer Garden. This attack causes legitimate requests to become expensive by modifying a Wiki page that is included in many other Wiki pages. The modification causes slow server-side rendering of infected pages.

Beer Garden limitation. The success of the Beer Garden defense rests on several assumptions, among them the assumption that the attacker cannot increase the resource consumption of legitimate requests. If that were possible then an attack could cause legitimate requests to become high-density requests, which would then be evicted at a high rate – just like malicious requests. Even worse though, each legitimate visitor has its own computer, which means that all legitimate visitors solve puzzles in parallel. Thus the Doorman's puzzles will not rate limit high-density requests since they are coming from many legitimate visitors.

This attack uses MediaWiki features to "infect" a large number of wiki pages, causing legitimate requests for those pages to become high-density requests.

MediaWiki vulnerability. MediaWiki stores wiki pages on the server using a markup language. When serving a page for a request, MediaWiki must translate the page's markup into HTML. It is well known that this translation step can be very expensive for certain pages [**5**].

Adaptable Platform-Dependent Micro-Benchmarks

Michael N. Gagnon
mikegagnon@gmail.com

Abstract

Micro-benchmark suites often strive for *portability*—the ability to apply the same micro-benchmark to multiple platforms. To achieve portability, existing micro-benchmark suites, such as lmbench, employ the *platform-independent* (PI) approach. The PI approach dictates that the same, unaltered micro-benchmarks should apply to multiple platforms. Though portability is desirable, the PI approach is troubled by many difficulties that undermine benchmark accuracy and precision. These problems compound, making micro-benchmark development an unnecessarily difficult task.

Through experimentation we demonstrate that the platform-independent approach suffers inherent limitations. We propose an alternative approach, the *adaptable platform-dependent* (APD) approach, which promises to overcome the deficiencies of PI micro-benchmarks. APD micro-benchmarks exploit platform-specific information and features to achieve better accuracy and precision. We also claim that APD micro-benchmarks are more straightforward to develop than equivalent PI micro-benchmarks. To be portable, APD benchmarks must be easily adaptable to new platforms—a key challenge.

To demonstrate the feasibility of achieving adaptability, we instantiate the APD approach with Yamba, a new micro-benchmark framework that allows programmers to develop adaptable micro-benchmarks. Experimental results with Yamba show that APD micro-benchmarks can achieve better accuracy and precision than equivalent PI benchmarks. We demonstrate the ease of adaptation by illustrating the labor involved in developing and adapting micro-benchmarks for new platforms.

1. INTRODUCTION

Micro-benchmarks measure the performance of primitive operations, such as function calls, network transmissions, language statements, and hardware operations. For decades, researchers and engineers having been using micro-benchmarks for a variety of purposes. Traditionally, researchers and engineers most often use micro-benchmarks to assess and compare the performance of system components.

For example, in 1962 Kilburn et al. used micro-benchmarks to assess the performance of the first virtual-memory system [5]. Using micro-benchmarks, they determined the effect of memory storage on the performance of several instructions. As a recent example, Baumann et al. used micro-benchmarks to assess the performance of a novel multi-kernel operating system, Barrelfish [1]. The Barrelfish operating system relies heavily on inter-core message passing. Thus, a key aspect of their evaluation was demonstrating, via micro-benchmarks, that inter-core communication is sufficiently efficient.

Beyond performance assessment, micro-benchmarks are also useful for a variety of other miscellaneous purposes. For example, researchers have used micro-benchmarks to rigorously predict application performance [8, 12]. Researchers have also used micro-benchmarks to facilitate dynamic self optimization. The X-Ray system uses *platform-independent* (PI) micro-benchmarks to determine uniprocessor hardware parameters, which educate dynamic self optimization mechanisms [13]. Similarly, P-Ray uses micro-benchmarks to determine hardware parameters for multi-core processors, and employs *platform-dependent* (PD) micro-benchmarks that take advantage of high-resolution timers and specialized operating-system features [2]. As a final example of micro-benchmarks applied to self-optimization, the Barrelfish operating system uses PD micro-benchmarks to make informed system-resouce-policy decisions.

Micro-benchmarks are also useful for exploiting timing side channels. Cryptanalysts have famously used micro-benchmarks to break cryptographic implementations [6]. Recently, several researchers have used micro-benchmarks to exploit information leaks in virtualized and cloud environments. For example, Garfinkel et al. propose using micro-benchmarks to expose sensitive information about virtual-machine monitors [3]. In a similar vein, Ristenpar et al. used micro-benchmarks to expose sensitive information in cloud-computing environments [7].

Often, as some of the above examples show, researchers and engineers develop custom micro-benchmarks because it gives developers the power to finely tailor their benchmarks—

enabling exact control over what the benchmarks measure and how they do it. However—as other examples show—it is often desirable to dispose of customization for the sake of *portability*—the ability to apply the same micro-benchmark to multiple systems.

1.1 Portability

Portability is desirable for many reasons. Namely, portability facilitates benchmark re-use, easy deployment, and performance comparisons over multiple systems. Existing portable micro-benchmark suites employ the *platform-independent* (PI) approach—essentially espousing the "write once, run anywhere" philosophy. The PI approach to micro-benchmark portability dictates that the same, unaltered micro-benchmarks should apply to multiple platforms.

For example, the *lmbench* portable micro-benchmark suite measures the performance of operating-system and hardware operations [9]. Lmbench espouses the PI approach by encoding all of its benchmarks in ANSI C and choosing only to interface with the OS through the POSIX interface. Thus any system that supports ANSI C and POSIX may run the lmbench benchmark suite. Due to its ease of use and wide applicability, many researchers and engineers use lmbench to assess and compare the performance of computer systems. Lmbench was first released in 1996 and over the years has become the de-facto standard operating-system micro-benchmark tool.

Unfortunately, as this paper demonstrates, PI micro-benchmarks deliver poor accuracy and precision when they rely on unsafe assumptions. Furthermore, these weaknesses represent a fundamental limitation to the PI approach to micro-benchmarking.

As existing micro-benchmark suites have aged and computer systems have continued to evolve, these problems have become more apparent. It is time to reconsider this standard design choice.

Platform-dependent (PD) micro-benchmarks are not troubled by the problems specific to PI benchmarks since they do not necessitate unsafe assumptions. But it is not obvious how PD benchmarks can achieve portability. We propose that PD micro-benchmarks can achieve portability via *adaptability*—the ability to easily adapt micro-benchmarks to new platforms. The key challenge to realizing *adaptable-platform-dependent* (APD) micro-benchmarks is reducing the labor needed to adapt benchmarks to new platforms.

We demonstrate the APD approach to micro-benchmark development with Yamba—a new benchmark-development framework facilitating the development and adaptation of machine-code PD micro-benchmarks.[1] Yamba facilitates adaptation by allowing programmers to segregate the platform-specific elements of a benchmark from the re-usable platform-independent elements. This way programmers define generic "benchmark blueprints" that can later be instantiated as PD micro-benchmarks. To port benchmarks to new platforms, programmers need only adapt the platform-specific elements of benchmarks.

[1]Yamba: Yet another micro-benchmark architecture

The APD approach is promising because it stands to achieve portability without the limitations inherent to the PI approach. However, it is not without cost—namely, the labor involved in adapting PD micro-benchmarks. Though Yamba attempts to minimize the amount of labor needed to adapt a benchmark, this labor is unavoidable. To demonstrate adaptation labor, we qualitatively describe the labor required to port an example Yamba benchmark to a specific platform.

1.2 Contributions

There are several contributions of this paper:

1. Through a case study, we present a critical analysis of the platform-indepedent (PI) approach to micro-benchmarking. Our principal finding is that the PI approach suffers inherent limitations. We present this analysis in Section 3.

   It is important to understand these limitations so that researchers and engineers may develop and use PI benchmarks appropriately and with caution. Furthermore, this analysis is important for informing explorations of alternative designs.

2. We propose a new approach to achieving portability—adaptable platform-depedent (APD) micro-benchmarks. We describe this approach in Section 4.

3. We present Yamba, a new micro-benchmark-development framework that instantiates the APD philosophy. Yamba facilitates the development and adaptation of platform-specific micro-benchmarks. We give an overview of Yamba in Section 5 and evaluate it quantitatively in Section 6.

The rest of the paper is organized as follows. Section 2 presents an overview of micro-benchmarks, discussing their implementation challenges. Section 2 also establishes notation used throughout the paper. Section 3 and analyzes the PI approach to micro-benchmarking. Section 4 follows, which describes our alternative: the APD approach. This section presents Yamba, our instantiation of the APD approach. Section 6 presents our experimental evaluation of Yamba as well as a qualitative analysis of the cost of adaptation labor. We present related work in Section 7 and conclude in Section 8.

2. BACKGROUND

Before discussing the peculiarities of platform-independent and adaptable-platform-dependent micro-benchamrks, it is important to first understand the general challenges associated with micro-benchmarks. Micro-benchmarks measure the performance of primitive operations such as instructions, functions, and system calls. The quick execution-time of these operations presents challenges to micro-benchmark developers.

In this section we use an example to introduce the generic challenges that programmers face when developing micro-benchmarks. We also establish pseudo-code notation that we use throughout the paper.

2.1 Example: benchmarking an instruction

Say you wished to measure the latency (i.e. execution time) of a primitive operation such as the x86 instruction

$$\texttt{mov}\ \mathrm{eax},\ [\mathrm{immediate}].$$

In fact, Yamba's x86-32 page-fault benchmark (presented in Section 5.3) benchmarks page faults by measuring the latency of this exact instruction. This instruction treats an "immediate" value as a pointer and copies the contents of the memory-location referenced by the immediate value into eax. For example, if memory location 0x45afb630 contained the value 3, then mov eax, [0x45afb630] would copy the value 3 into the eax register.

We denote the operation of interest as $\mathcal{P}$. In this case, $\mathcal{P}$ represents mov eax, [immediate]. Let the function $T(\mathcal{P})$ be the *true* average execution time of $\mathcal{P}$. In general, it is not possible to know the true value of $T(\mathcal{P})$, but nevertheless we would like to approximate it using a micro-benchmark.

To approximate $T(\mathcal{P})$, we would like to measure the execution time of a representative "program." We denote this program $\mathbb{P}$ and denote the measurement of $\mathbb{P}$ as $M(\mathbb{P})$. In this example, we define $\mathbb{P} =$ mov eax, [0x45afb630] The measurement function $M(\mathbb{P})$ operates as follows: (1) M takes a time query, (2) executes $\mathbb{P}$, (3) takes another time query, and (4) returns the difference. For this example, let us assume that time queries are only accurate within a one-millisecond resolution.

Unfortunately, directly measuring the execution time of a single instruction will not give a good approximation for $T(\mathcal{P})$.

$$T(\mathcal{P}) \not\approx M(\mathbb{P})$$

$M(\mathbb{P})$ is a poor approximation because $\mathbb{P}$ executes far more quickly than the resolution of the clock; $M(\mathbb{P})$ drowns in the overhead of the time query and the noise introduced by the low-resolution timer. It is thus desirable to execute $\mathbb{P}$ many times and approximate $T(\mathcal{P})$ as follows:

$$T(\mathcal{P}) \approx \frac{M(\mathbb{P}_1, \mathbb{P}_2, ..., \mathbb{P}_n)}{n}$$

where $\mathbb{P}_1, \mathbb{P}_2, ..., \mathbb{P}_n$ denotes "repeating" the program $\mathbb{P}$ n times. You can repeatedly execute $\mathbb{P}$ several different ways. For example, one could naïvely use a simple conditional loop:

$$\mathbb{P}' = \texttt{for}(\mathrm{i}\ = 0;\ \mathrm{i}\ < \mathrm{n};\ \mathrm{i}{+}{+})\{\mathbb{P};\}$$
$$T(\mathcal{P}) \not\approx \frac{M(\mathbb{P}')}{n}$$

The problem here is that the time measurement will include the execution of the for-loop statements. We could estimate the overhead of this for-loop (and then subtract it from our result) by benchmarking a dummy loop that contains no body. However, the for-loop statements and $\mathbb{P}$ likely execute within the same order of magnitude—limiting the applicability of this technique.

An alternative (and still naïve) approach could do away with the execution-overhead of the loop by fully "unrolling" the loop into a sequence of consecutive statements:

$$\mathbb{P}' = \mathbb{P}_1; \mathbb{P}_2; ...; \mathbb{P}_n;$$
$$T(\mathbb{P}) \not\approx \frac{M(\mathbb{P}')}{n}$$

This time the problem is a little more subtle. The number of repetitions, n, needs to be large enough such that the execution of $\mathbb{P}_1, \mathbb{P}_2, ..., \mathbb{P}_n$ overcomes the low-clock resolution. When n is large though, the working set transcends memory pages and cache lines, likely introducing unwanted timing anomalies.

The best way to approximate $T(\mathcal{P})$ is to partially unroll the loop and account for the for-loop overhead:

$$\begin{aligned}\mathbb{P}' &= \mathbb{P}_1; \mathbb{P}_2; ...; \mathbb{P}_m;\\ \mathbb{P}'' &= \texttt{for}(\mathrm{i}\ = 0;\ \mathrm{i}\ < \mathrm{n/m};\ \mathrm{i}{+}{+})\{\mathbb{P}';\}\\ \mathbb{F} &= \texttt{for}(\mathrm{i}\ = 0;\ \mathrm{i}\ < \mathrm{n/m};\ \mathrm{i}{+}{+})\{\textit{no-op}\}\\ T(\mathbb{P}) &\approx \frac{M(\mathbb{P}'') - M(\mathbb{F})}{n}\end{aligned}$$

By intelligently choosing the number of consecutive mov statements (m) and the number of loop iterations (n/m), you can minimize loop overhead and memory timing anomalies. Finally, to account for the loop overhead, just measure (and substract) the execution time of a dummy loop that contains no loop body. We are able to account for for-loop overhead this way because the latency of $\mathbb{P}'$ dwarfs the latency of the for-loop statements.

2.2 General challenges

The above example illustrates some of the challenges micro-benchmark developers encounter. Often micro-benchmark developers cannot simply measure the execution time of a single operation; they must measure the execution time of repeated executions. Also, they must ensure that the body of the loop does not drown in the overhead of the loop—taking into account a number of factors including clock resolution. Furthermore, developers must account for potential timing anomalies introduced by low-level systems such as cache behavior.

The example hints at deeper, more fundamental challenges. Micro-benchmark developers must know exactly what they wish to measure; what exactly should $T(\mathcal{P})$ represent? Though it is simple to state,

> "$T(\mathcal{P})$ represents the true average execution time of the mov eax, [immediate] instruction,"

it is unclear exactly what this means.

Is the developer interested in the performance of *cached* mov instructions? Similar but distinct: should the benchmark measure the performance of cached memory operands? Is the developer interested in the execution time of a pipeline of mov instructions, or in the execution time of individual "randomly" placed mov instructions? Is mov eax, [0x45afb630] representative? Should each mov instruction target a different memory location? For each of these examples, $\mathcal{P}$ represents a different operation.

In general, the latency of an operation greatly depends on

$$\mathbb{P} = \text{touch}(\text{address}[\text{j}])$$
$$\mathbb{P}' = \texttt{for}(\text{j} = 0;\ \text{j} < \text{n};\ \text{j}{+}{+})\{\mathbb{P};\}$$
$$\begin{aligned}\mathbb{P}'' = \texttt{for}(\text{i} = 0;\ \text{i} < \text{m};\ \text{i}{+}{+})\{ \\ \mathbb{P}' \\ \text{munmap}();\ \text{mmap}();\ \text{msync}(); \\ \}\end{aligned}$$
$$\begin{aligned}\mathbb{F} = \texttt{for}(\text{i} = 0;\ \text{i} < \text{m};\ \text{i}{+}{+})\{ \\ \text{munmap}();\ \text{mmap}();\ \text{msync}(); \\ \}\end{aligned}$$
$$T(\mathcal{P}) \approx \frac{M(\mathbb{P}'') - M(\mathbb{F})}{m \cdot n}$$

Figure 1: lat_pagefault semantics

the state of the computer system, including the contents of memory, registers, the operating-system kernel, user-space processes, and so on. Furthermore, optimizing compilers, assemblers, and interpreters may profoundly affect micro-benchmark results. Often, subtle aspects of systems influence the execution of micro-benchmarks in seemingly unpredictable ways. Without a thorough understanding of these complex system interactions, developers are liable to make naïve assumptions that produce flawed micro-benchmarks.

Because of these inherent challenges, micro-benchmarks are notoriously difficult to engineer. Micro-benchmark developers must thoroughly understand, and take into account, all the myriad ways a computer system may influence a benchmark.

3. PLATFORM-INDEPENDENCE

The previous section describes the challenge of developing micro-benchmarks in general. Here, we discuss additional challenges associated specifically with platform-independent (PI) micro-benchmarks. Furthermore, we demonstrate that these challenges represent a fundamental limitation to the PI approach to micro-benchmarking.

We illustrate the challenges associated with the PI approach through a case study. This study critically analyzes a real-world PI micro-benchmark, drawn from the latest version of lmbench, 3.0-a9, released November 27, 2007 [10]. After presenting this case study, we conclude this section by presenting the fundamental limitation of PI micro-benchmarks.

3.1 Case study: lat_pagefault

Lmbench measures the latency of page-faults in the lat_pagefault benchmark. Here, $\mathcal{P}$ is a page fault. lat_pagefault operates by mmap-ing a sequence of pages, then causing them to page out. This way, any memory access to an mmap-ed page causes a page fault. Thus, $\mathbb{P}$ simply needs to touch a paged-out memory location. Figure 1 presents the semantics of the lat_pagefault benchmark.[2]

[2]Our description matches lmbench's source code [10]. The lat_pagefault man-page erroneously describes other semantics.

Platform	Original lat_pagefault results	Fixed lat_pagefault results
$C_{1Darwin}$	567.1450	
C_{1Linux}	1.1617	398.9608
$C_{2Darwin}$	90.0432	
C_{2Linux}	1.5016	63.0881

Table 1: Experimental results showing that lat_pagefault under-reports page-fault latency on modern Linux systems.

lmbench attempts to ensure pages are paged out using the POSIX msync() system call with the MS_INVALIDATE flag set. It then measures the execution time of page faults by accessing each page in an inner for-loop. It accesses the pages in random order by iterating over a random permutation of page pointers (address[j] references the jth random page).

An outer loop keeps repeating this procedure to make sure the benchmark executes over a sufficient length of time. On each iteration of the outer loop, the pages must be re-mmap-ed and paged out using msync().

Lmbench's approach is problematic for two reasons.

3.1.1 *Problem 1*

Lmbench assumes that msync() with the MS_INVALIDATE-flag causes page outs. This behavior is a platform-specific implementation detail; the POSIX standard does not specify that invocations of msync() cause page outs. However, lmbench's published results show that, historically, this assumption holds for many systems. When we tested lat_pagefault on several modern Linux systems, we discovered lat_pagefault severely under-reports page-fault latency.

Table 1 presents these results. It shows the mean latency from running lat_pagefault 31 times on 5 different computers using the original version of lat_pagefault. The table also presents experimental results from running a "fixed" version of lat_pagefault, which we discuss shortly. For now it is sufficient to know that $C_{1Darwin}$ and C_{1Linux} are have identical hardware and only differ by operating system, and so on for $C_{2Darwin}$ and C_{2Linux}. We detail our experimental setup in Section 6.1.

As Table 1 shows, lat_pagefault seemed to show Linux strongly outperformed Darwin. We were surprised by this and suspected that msync() was not properly causing page outs. Investigation revealed that modern Linux kernels cache the swap area and msync() does not invalidate the swap cache. Thus msync() with MS_INVALIDATE does not cause disk page-outs on Linux. Linux's implementation of msync() implements the POSIX standard, but it does not provide the platform specific implementation that lmbench expects.

To remedy this problem we modified lat_pagefault. After the call to msync() we inserted a Linux command that directly instructs the kernel to flush the swap cache.[3] Fixing lat_page-

[3]Write the value "1" to the special file `/proc/sys/vm/drop_caches`

fault this way allows lat_pagefault to cause page outs as expected. Unfortunately, the fix necessitated platform-specific functionality; there is no platform-independent mechanism to guarantee page outs using the POSIX interface.

3.1.2 Problem 2

Though we solved Problem 1 by adding platform-specific code to cause page-outs on Linux, there remains another problem. lat_pagefault still significantly under-reports page-fault latencies.

Many modern operating-system kernels (such as Linux and Darwin) employ *anticipatory paging*. Rather than just paging in the single faulting page, the kernel pages in multiple pages during a fault—hoping to reduce the chance of future page faults. The kernel uses information about the process workload to determine which pages should be paged in during a fault. Thus, the memory access patterns of lat_pagefault directly determine the behavior of the virtual-memory manager (VMM).

We cloned the lat_pagefault benchmark using Yamba and instrumented it with calls to the mincore() system call to monitor the behavior of the anticipatory pagers on Linux and Darwin (we discuss this clone in detail in Section 6). Table 2 presents our experimental results. These results show that only a small proportion of memory accesses cause page faults. This is because the anticipatory pager preemptively pages-in pages. Thus when lat_pagefault accesses a page, it is very likely already paged in from a previous fault.

More importantly though, the Linux and Darwin anticipatory pagers behave differently. The POSIX standard does not specify the behavior of the VMM. Thus there is no platform-independent workload that reliably yields the same paging behavior. However, lat_pagefault could derive desirable workloads by exploiting information gained from monitoring specific anticipatory pagers using mincore. However, mincore() is not a standard POSIX system call. In fact the semantics of mincore() differ on Linux and Darwin. Thus there is no platform-independent mechanism to produce reliable workloads for lat_pagefault.

One could argue that this problem is not a bug lat_pagefault—that lat_pagefault is designed to measure the average memory-access latency according to lat_pagefault's workload. But the man page for lat_pagefault ambiguously specifies "lat_pagefault times how fast a page of a file can be faulted in" [10]. Even if this benchmark was intended to measure this statistic, how valuable would it be? Is this workload representative of application memory accesses? It doesn't seem likely.

3.2 Fundamental limitations

Our case study demonstrates two problems with the lmbench lat_pagefault benchmark. First, there is no POSIX mechanism to cause page outs. Lmbench is thus forced to make assumptions about non-standardized implementation details of the host system. These assumptions do not hold on Linux, and lead to significantly biased latency measurements. Second, there is no platform-independent workload that reliably causes page faults since POSIX systems are free to implement arbitrary VMM behaviors.

Platform	Proportion of accesses that fault	Average page-ins per fault
$C_{1Darwin}$	0.069	14.54
C_{1Linux}	0.046	21.63
$C_{2Darwin}$	0.068	14.71
C_{2Linux}	0.046	21.51

Table 2: Anticipatory-paging behvaior for lat_pagefault's workload

These problems demonstrate that some PI micro-benchmarks are not truly portable. To remedy the first problem we were forced to adapt lat_pagefault to Linux in order to utilize a Linux-specific feature (viz. `drop_cache`). Due to the second problem we were forced to use a platform-specific feature (viz. mincore()) in order to correctly interpret lat_pagefault's results.

Here we generalize this observation to formulate a fundamental limitation to the PI approach to micro-benchmarking. But before we present the fundamental limitation we must delineate the difference between *pure* and *impure* PI micro-benchmarks.

We define *pure* PI micro-benchmarks to have stricter requirements than *impure* PI micro-benchmarks. Specifically, the pure PI approach dictates that: (1) benchmarks may only use platform-independent interfaces and (2) benchmarks may not rely on implementation details specific to the host platform. For example lmbench's bw_pipe benchmark measures the bandwidth of POSIX pipes. It is a pure PI micro-benchmark since it only uses standardized interfaces (namely POSIX and ANSI C) and does not rely on platform-specific implementation details.

On the other hand, *impure* PI micro-benchmarks are only constrained to the first requirement; that is, they may only use platform-independent interfaces. This frees them to utilize platform-specific implementation information. For example, lmbench's lat_pagefault benchmark is an impure PI micro-benchmark because it relies on non-standard implementation details of the msync() POSIX function. Though this benchmark is impure, it is still a PI micro-benchmark since it only utilizes platform-interfaces.

The PI approach to micro-benchmarking is alluring because it offers the most direct route to portability: developers simply construct benchmarks in such a way that they run on many platforms. Unfortunately PI micro-benchmarks suffer the following fundamental limitation.

FUNDAMENTAL LIMITATION 1. *Pure PI micro-benchmarks cannot safely measure the performance of platform-specific operations.*

We say a pure PI micro benchmark is *safe* if it is accurate on all systems that implement its interface.

For example, it is not possible to craft a pure PI micro-benchmark that safely measures the latency of a page fault using only POSIX and ANSI C. This is because POSIX and ANSI C do not specify the behavior of the virtual memory system. As a consequence there is no platform-independent mechanism to cause page faults.

It is certainly possible to craft a pure PI micro-benchmark that happens to measure the latency of page faults on many POSIX systems. However, such a benchmark would not be safe since it is possible for someone to implement the POSIX interface with different page-fault behavior.

Due to the fundamental limitation of the PI approach, PI micro-benchmarks that measure platform-specific operations must necessarily be *impure*. An impure PI micro-benchmark, such as lat_pagefault, may make assumptions about its execution environment. However, this limits the versatility of the benchmark; the benchmark then only applies to systems that support its assumptions. Linux, for example, violates lat_pagefault's assumptions.

Fundamental Limitation 2. *Impure PI micro-benchmarks are only accurate on systems that support their assumptions.*

Due to these limitations it is difficult to write portable PI micro-benchmarks that apply to many systems. Benchmark authors must make assumptions that limit their benchmark's portability.

4. ADAPTABLE PLATFORM-DEPENDENCE

The premise of adaptable-platform-dependent (APD) micro-benchmarks is simple. APD micro-benchmarks are platform-dependent, which overcomes the limitations inherent to the PI approach. APD micro-benchmarks must then achieve portability through adaptability.

Accuracy PD micro-benchmarks have the potential to be more accurate than equivalent PI micro-benchmarks because PD micro-benchmarks do not necessitate potentially unsound assumptions. Section 3 demonstrates why PI micro-benchmarks are inherently vulnerable to potentially unsound assumptions as well as the effect of invalid assumptions on accuracy. Additionally, PD micro-benchmarks may leverage platform-specific features (such as controlling the process scheduler) as well as platform-specific information (such as cache sizes) to further improve accuracy.[4]

Precision PD micro-benchmarks have the potential to be more precise than equivalent PI micro-benchmarks because PD micro-benchmarks may use platform-specific features to boost precision. For example, a PD micro-benchmark can use platform-specific high-precision, high-resolution timers. Also, a PD micro-benchmark can use platform-specific features to boost precision. For example, a benchmark can boost its precision by using PD mechanisms to ensure neighboring user-space processes do not interfere with the benchmark.

Straightforward development Engineering micro-benchmarks is a notoriously difficult task. Engineering PI micro-benchmarks is even more difficult because (1) developers have a smaller toolset (only PI features are available), (2) developers must hack around the limitations of PI interfaces, and (3) developers must maintain a potentially unwieldy set of assumptions. The development history of the `mhz` micro-benchmark documents many of these challenges [11]

PD micro-benchmarks do not suffer from these extra difficulties. PD micro-benchmarks have access to all the features and information a systems offer. Also, there is no need to circumvent limited PI interfaces. Lastly, developers are not burdened with the labor of maintaining potentially unsafe assumptions.

While retaining all of the benefits of PD micro-benchmarks, APD benchmarks also achieve portability—the major benefit of the PI approach. APD micro-benchmarks get the best of both worlds.

The primary impediment to using APD micro-benchmarks is the labor required to adapt micro-benchmarks to new platforms. While PI micro-benchmarks apply to many platforms "out of the box," programmers must manually port APD micro-benchmarks to new platforms. Thus it is crucially important that APD micro-benchmark frameworks (such as Yamba) ease the cost of adaptation as much as possible.

5. YAMBA

To investigate the feasibility of achieving adaptability, we developed the Yamba programming language. The Yamba language uses object-oriented features to allow programmers to segregate the platform-specific elements of a benchmark from the generic platform-independent elements. For example a page-fault benchmark (similar to lmbench's lat_pagefault) can first be coded as a platform-independent *benchmark template*. This template acts as a blue print for PD micro-benchmarks. To port the benchmark, a programmer would need only fill out the template to fully define a platform-specific micro-benchmark.

Yamba allows programmers to specify benchmark programs using a combination of machine code, assembly language, and C/C++. This gives the programmer the freedom to choose the most appropriate abstraction level for the different components of the benchmark. Additionally, programmers are free to use hardware-specific features that may only be available via assembly language or machine code.

Yamba provides a number of features that ease the development of PD micro-benchmarks in general. For example, Yamba provides language constructs to dynamically unroll loops at run time. This allows programmers to choose the most suitable degree of loop unrolling at runtime, based on platform-specific runtime values.

We describe the Yamba language through an example. Here we develop a page-fault benchmark equivalent to lmbench's lat_pagefault. We begin by describing the benchmark template, then follow by showing how to port this benchmark to a specific platform.

5.1 Y_pagefault_generic semantics

[4]**DRAFT NOTE:** To do: discuss danger of misleading results

We wish to approximate the latency of page faults.

$$\mathcal{Q} = \text{page fault}$$
$$T(\mathcal{Q}) = \text{true latency of page fault}$$

First we establish the benchmark template Y_pagefault_generic, which outlines the benchmark in a platform-independent manner. Our template is similar to lmbench's lat_pagefault benchmark. We begin by developing the inner loop. The inner loops is a hybrid loop that uses partial unrolling (à la Section 2.1):

$$\begin{aligned}
\mathbb{Q}_i &= \text{touch}(\text{address}[\text{i}]) \\
\mathbb{Q}' &= \mathbb{Q}_1; \mathbb{Q}_2; ...; \mathbb{Q}_m; \\
\mathbb{Q}'' &= \texttt{for}(\text{i} = 0;\ \text{i} < \text{n} / \text{m};\ \text{i}{+}{+}) \\
&\qquad \{\mathbb{Q}'\}
\end{aligned}$$

$\mathbb{Q}_i$ touches a paged-out memory address causing a page fault. $\mathbb{Q}''$ represents the complete inner loop, which touches every page from a sequence of n pages. The inner loop is partially unrolled by a degree of m; i.e. the inner-loop body ($\mathbb{Q}'$) is composed of m consecutive $\mathbb{Q}_i$ statements. Because there are a total of n pages, it takes n/m loop iterations to touch every page. (n is constrained to be a multiple of m.)

As our case study showed in Section 3.1, to reliably cause page faults, memory-access workloads must be tailored to specific platforms. To facilitate this Y_pagefault_generic uses an array of pointers. Instantiations of Y_pagefault_generic then encode their workload as a sequence of pointers; when accessed sequentially each pointer dereference should cause a page fault. (address[i] points to the ith memory location in the workload.)

Next, we craft the outer loop and the dummy loop:

$$\begin{aligned}
\mathbb{Q}''' &= \texttt{for}(\text{i} = 0;\ \text{i} < \text{j};\ \text{i}{+}{+})\{ \\
&\qquad \mathbb{Q}'' \\
&\qquad \text{cause\_page\_outs}(); \\
&\quad \} \\
\mathbb{G} &= \texttt{for}(\text{i} = 0;\ \text{i} < \text{m};\ \text{i}{+}{+})\{ \\
&\qquad \text{cause\_page\_outs}(); \\
&\quad \}
\end{aligned}$$

$$T(\mathcal{Q}) \approx \frac{M(\mathbb{Q}''') - M(\mathbb{G})}{n \cdot j}$$

Each iteration of the outer loop iterates over the entire sequence of pages, causing page-faults along the way. After accessing the pages, the pages are then re-paged-out so the next iteration will cause more page faults.

Y_pagefault_generic defines cause_page_outs() to use msync() with MS_INVALIDATE since lmbench's experimental results show that this works well on many platforms. Instantiations of Y_pagefault_generic redefine the cause_page_outs() function if it does not work on their platform. For example, the Linux instantiation of Y_pagefault_generic uses a combination of msync() and the `drop_cache` special file.

5.2 Y_pagefault_generic source code

Figure 2 presents the Yamba source code for Y_pagefault_generic. The source code contains one "abstract" Yamba class, Y_pagefault_generic that encapsulates $\mathbb{Q}_i$, $\mathbb{Q}'$, $\mathbb{Q}''$, two C functions (benchmark() and benchmark_dummy() that correspond to $\mathbb{Q}''$ and $\mathbb{G}$, respectively. The code for cause_page_outs() has been removed for space.

Bitstrings

In Yamba, a `bitstring` is simply a string if bits that contain executable machine code. For example, the `bitstring` prologue will later be defined to contain machine code that acts as a function prologue—setting up the function stack and so forth.

Yamba functions

Yamba functions, such as pagefault_func(), are functions that return `bitstring` values. They generate bitstrings by concatenating sequences of bitstrings. To illustrate, consider the Yamba function pagefault_func().

The Yamba function pagefault_func() composes and returns an executable `bitstring` that is composed of:

prologue + touch(0) + touch(1) + ... + touch(m) + epilogue

Dynamic loop unrolling

The `unroll` statement unrolls the body of the loop into a series of `bitstring` concatenations.

Subtypes

pagefault_func() returns a particular type of `bitstring`, whose sub-type is defined within the angle brackets. pagefault_func() returns a machine-code `bitstring` that is callable as a C function with the following signature:

```
void c_function(byte_t * first_page)
```

Essentially, pagefault_func() is as Yamba function that returns a C function. We use pagefault_func() to generate the body of the inner loop. Then the C function benchmark can call this C function to cause page outs.

Abstract classes

Abstract classes are simply partially defined Yamba classes. Yamba functions are either concrete or abstract. The definition of pagefault_func() is said to be abstract because it is defined in terms of undefined `bitstring` data-members (viz. prologue, epilogue, etc.). It is not possible to execute an abstract function, because they are ill-defined.

To make pagefault_func() concrete (i.e. fully defined), a programmer would need to create a new, concrete class that extends the Y_pagefault_generic class. This new class would need to provide concrete definitions for all the `bitstring` data-members in the Y_pagefault_generic class. Making a Yamba function concrete allows us to call the Yamba function.

We use abstract classes to define platform-independent bench-

```
yamba abstract Y_pagefault_generic
{
        int file_ref;           // file to be mmaped
        byte_t * pages;         // points to contents of file_ref in memory
        int num_pages;

        byte_t * targets;       // workload description
        int unroll_degree;
        int num_groups;         // num_pages / unroll_degree
        int num_passes;

        bitstring prologue;
        bitstring epilogue;
        bitstring touch(unsigned int page_index);

        /* actual benchmark */

        bitstring<void c_function(byte_t * first_page)> pagefault_func(unsigned int m)
        {
                prologue
                unroll(unsigned int i = 0; i < m; i++)
                {
                        touch(i)
                }
                epilogue
        }

        void benchmark ()
        {
                for (pass_i = 0; pass_i < num_passes; pass_i++)
                {
                        byte_t * this_targets = targets;
                        for (group_i = 0; group_i < num_groups; group_i++)
                        {
                                pagefault_func(targets);
                                this_targets += unroll_degree;
                        }
                        cause_page_outs_func(pages);
                }
        }

        /* dummy benchmark */

        bitstring<void c_function(byte_t * first_page)> pagefault_dummy_func(unsigned int m)
        {
                prologue
                epilogue
        }

        void benchmark_dummy ()
        {
                for (pass_i = 0; pass_i < num_passes; pass_i++)
                {
                        byte_t * this_targets = targets;
                        for (group_i = 0; group_i < num_groups; group_i++)
                        {
                                pagefault_dummy_func(targets);
                                this_targets += unroll_degree;
                        }
                        cause_page_outs_func(pages);
                }
        }

};
```

Figure 2: Source code for Y_pagefault_generic

```
yamba Y-pagefault_x86_32 extends Y-pagefault_generic
{
        import(x86_32);
        import(compiler_x86_32);

        bitstring prologue
        {
                compiler_x86_32.prologue_standard
                asm
                {
                        push ebx
                        mov ebx, [ebp + 8]
                }
        }

        bitstring epilogue
        {
                asm { pop ebx }
                compiler_x86_32.epilogue_standard
        }

        bitstring touch(int target_index)
        {
                // eax = pointer to target[i]
                // mov eax, [ebx + offset]
                0x8B83 x86_32.cast_imm_32(target_index)

                // touch target[i]
                asm {mov eax, [eax] }
        }
};
```

Figure 3: Source code for Y-pagefault-x86_32

mark templates. Then we can port an abstract class to a specific platform by extending it to a concrete class that provides appropriate platform-specific `bitstring` definitions. The comments in a benchmark template inform programmers of expected semantics for the machine code encoded in `bitstring` data-members.

Benchmark parameters

Y_pagefault_generic leaves several parameters unspecified (e.g. m, num_pages, unroll_degree, etc.). This is by design, since these values depend on platform-specific information. Instantiations of Y_pagefault_generic must define these values.

5.3 Y_pagefault_x86_32

We now demonstrate how to port Y_pagefault_generic to a specific platform. Here we port Y_pagefault_generic to the 32-bit x86 platform. Figure 3 presents the source code for Y_pagefault_x86_32.

Y_pagefault_x86_32 is a concrete class that extends Y_pagefault_generic. By providing definitions for Y_pagefault_generic's `bitstring` data-members, Y_pagefault_x86_32 inherits a fully defined version of pagefault_func(). C/C++ functions can then compile pagefault_func() for a particular degree of unrolling simply by calling pagefault_func(m).

Y_pagefault_x86_32 imports two other concrete Yamba classes, x86_32 and compiler_x86_32. These classes contain platform-specific data members, which Y_pagefault_x86_32 leverages (the complete definitions of these classes is omitted to conserve space).

x86_32 Yamba class and *cast* functions

The x86_32 Yamba class defines a number of data members and Yamba functions that are useful for x86-32 micro-benchmarks. Y_pagefault_x86_32 uses x86_32.page_size in the Yamba function touch().

The x86_32 Yamba class also defines a *cast* function. A cast function converts a C-type to an equivalent Yamba `bitstring`. For example, many x86-32 instructions take an *integer-immediate* operand, which is simply an integer encoded within the instruction. The `mov` eax, [immediate] instruction treats immediate as a pointer and moves the data pointed by immediate into eax. The immediate value is encoded in the actual instruction.

Yamba can dynamically create such instructions using platform-specific cast functions. The touch() Yamba function uses a cast to create an instance of the `mov` eax, [immediate] instruction. The x86_32.cast_immediate_32() Yamba function converts a C `int` into a 32-bit little-endian `bitstring` value—which is the exact format `mov` eax, [immediate] expects for its operand. The opcode for `mov` eax, [immediate] is `0x8B83`, hence the following `bitstring` function

`0x8B83` x86_32.cast_immediate_32(m)

generates the `mov` eax, [m] instruction.

compiler_x86_32 Yamba class

Likewise, the compiler_x86_32 Yamba class defines a number of data members and Yamba functions that are needed by x86-32 micro-benchmarks. Y_pagefault_x86_32 uses compiler_x86_32 to provide the basis for its function prologue and epilogue.

Adapting to operating systems

Benchmarks may need to be adapted for specific operating systems. For Darwin, Y_pagefault_x86_32 actually works mostly out-of-box. So we define Y_pagefault_x86_32_Darwin to simply inherit Y_pagefault_x86_32 and define the workload to produce desirable paging behavior. We describe this workload in Section **??**. Y_pagefault_x86_32_Linux almost works out-of-box. It needs to redefine cause_page_outs() to include a the `drop_cache` command.

Executing pagefault_inner_loop_body()

To complete this demonstration we illustrate the result of executing Y_pagefault_x86_32.pagefault_func():

```
func_pfilb_t loop_body =
Y_pagefault_x86_32.pagefault_func(5);
```

func_pfilb_t is a `typedef` for a function pointer with the following signature:

```
void c_function(byte_t * first_page )
```

The code below contains the disassembly for loop_body. The resulting machine code can be called as a normal C function. For example, if p points to an array of 1000 bytes, then loop_body(p) would touch the first 5 bytes in that array.

```
/* Y-pagefault_x86_32.prologue */
push ebp
mov ebp, esp
push ebx

/* Y-pagefault_x86_32.init_touch */
mov ebx, [ebp + 0x12]

/* unroll(unsigned int i = 0; i < 5; i++)
 * {
 *        Y-pagefault_x86_32.touch(i)
 * }
 */
mov eax, [ebx + 0x0]
mov eax, [ebx + 0x1000]
mov eax, [ebx + 0x2000]
mov eax, [ebx + 0x3000]
mov eax, [ebx + 0x4000]

/* Y-pagefault_x86_32.epilogue */
pop ebx
pop ebp
ret
```

6. EXPERIMENTS

We ran several experiments to evaluate the accuracy and precision of our example Yamba benchmark. At this point we do not have an operational Yamba compiler so we compiled the benchmarks by hand.

6.1 Setup

We ran experiments on two computers:

C_1 MacBook Pro Generation 2,2. 2.16 GHz Intel Core 2 Duo. This computer also employs a typical platter-based drive, so we expect page-fault latencies to be slower than C_2.

C_2 MacBook Pro Generation 5,2. 2.93 GHz Intel Core 2 Duo. This computer uses a solid-state drive—specifically a 256 GB Apple SSD TS256 hard drive. We therefore expect its page-fault latencies to be lower than C_1.

We installed OS X 10.6 (Darwin) and Ubuntu 9.04 on each computer yielding four platforms: $C_{1Darwin}$, C_{1Linux}, $C_{2Darwin}$, and $C_2 Linux$.

6.2 Workloads

To manipulate the behavior of the VMM to create page faults on every memory access we created workloads that skip over pages. For example, "scaling" the workload by a factor of 16 will lead the benchmark to only touch 1 page in every 16 pages. On all of ours systems, scaling by factors of 16, 32, and 64 led to 100% percentage of memory accesses that caused faults. Though these scales consistently caused faults, each scale on each operating system led to a different number of page-ins per faults. Table 4 summarizes these results.

As a result we can measure the latencies of three different types of page-fault operations. On Darwin we can measure the latency of page-faults that page in (on average) 8.42 page, 9.30 pages, and 11 pages. On Linux we can measure the latency of page-faults that page in (on average) 1 page, 8.55 pages, and 32 pages. O

6.3 Results

Tables 5, 6, and 7 summarize our results from measuring these page-fault latencies. Each result is the mean from executing the benchmark 31 times. We used an unroll value of 40 and used gettimeofday() for our time measurements.

Because we were able to isolate different types of page fault operations (based on how many pages they paged in), we were able observe the effect of the page-in count on page fault latency. We found that page-fault latency seems to scale linearly w.r.t the number of pages paged in. Table 8 tabulates these results along with a linear regression for each system. The table shows high coefficient-of-determination values (R^2), indicating a strong likelihood that systems do in fact exhibit linear performance. Performing more experiments with different workloads to give us more data points to experimentally evaluate this hypothesis.

Figure 9 graphically presents these results for C_2. As the graph shows, Linux and Darwin both scale linearly to the number of page-ins. Furthermore, they scale at approximately the same rate. Linux outperforms Darwin by only a small constant factor.

6.3.1 Accuracy

It is difficult to directly measure the accuracy of our benchmarks but we believe they are at least more accurate than lmbench's lat_pagefault benchmark, which we have shown is biased to under-report latencies. We have verified that each memory access causes a page fault and are able to determine the average number of pages paged in per page fault.

6.3.2 Precision

The benchmark with the highest variability is $C_{1Darwin}$ with scale = 16, with a variation of 0.0833. C_2's benchmarks tended to be an order of magnitude less variablity than C_1 (C_2 uses a solid-state drive). Since the only independent variable here is the hardware, this suggests the hardware has a significant impact on variability. This is reassuring because it suggests that the our variation is more likely representative of the latency variation and not an artifact introduced by our methodology.

Intuitively, these results make sense since we expect platter-based drives to exhibit more variability than solid-state drives (due to read-head movement, etc.) We thus believe that our benchmarks are precise since we believe most of the observed variation is the result of the operation's actual variability.

On C_1 Yamba and lmbench had comparable variability. However, on C_2 generally had two orders of magnitude less variation than lmbench. We believe this is due to lmbench's decision to randomize page access sequences. When we modified lat_pagefault to use sorted (non-random) accesses, we saw lat_pagefault's variability improve by an order of magnitude. We therefore believe that Yamba's page-fault benchmark is more precise than lat_pagefault since Yamba has better variation and Yamba uses a methodology with less nondeterminism.

Operating System	Scale	Page-ins per fault
Darwin	16	8.42
Darwin	32	9.30
Darwin	64	11.00
Linux	16	1
Linux	32	8.55
Linux	64	32

Figure 4: Effect of operating-system and scale on pages-in-per-fault

Platform	Page-fault latency μs	Variation	Page-ins per fault
$C_{1Darwin}$	2360.58	0.0833	8.42
C_{1Linux}	2542.07	0.0137	1.00
$C_{2Darwin}$	1124.16	0.0051	8.42
C_{2Linux}	676.75	0.0045	1.00

Figure 5: Yamba page-fault benchmark (scale = 16)

Platform	Page-fault latency μs	Variation	Page-ins per fault
$C_{1Darwin}$	4748.56	0.0697	9.30
C_{1Linux}	5087.88	0.0491	8.55
$C_{2Darwin}$	1176.83	0.0528	9.30
C_{2Linux}	882.51	0.0033	8.55

Figure 6: Yamba pagefault benchmark (scale = 32)

Platform	Page-fault latency μs	Variation	Page-ins per fault
$C_{1Darwin}$	9283.69	0.0673	11.00
C_{1Linux}	9955.01	0.0236	32.00
$C_{2Darwin}$	1212.67	0.0063	11.00
C_{2Linux}	1754.04	0.0016	32.00

Figure 7: Yamba pagefault benchmark (scale = 64)

Platform	Page-ins per fault	Page-fault latency μs	Linear regression
$C_{1Darwin}$	8.42	2360.58	$y = 2685x - 2025.2$
	9.30	4748.56	$R^2 = 0.99997$
	11.00	9283.69	
C_{1Linux}	1.00	2542.07	$y = 231.62x + 2653.7$
	8.55	5087.88	$R^2 = 0.98816$
	32.00	9955.01	
$C_{2Darwin}$	8.42	1124.16	$y = 32.562 + 859.45$
	9.30	1176.83	$R^2 = 0.9171$
	11.00	1212.67	
C_{2Linux}	1.00	676.75	$y = 35.325x + 615.18$
	8.55	882.51	$R^2 = 0.997$
	32.00	1754.04	

Figure 8: Effect of page-ins-per-fault on page-fault latency.

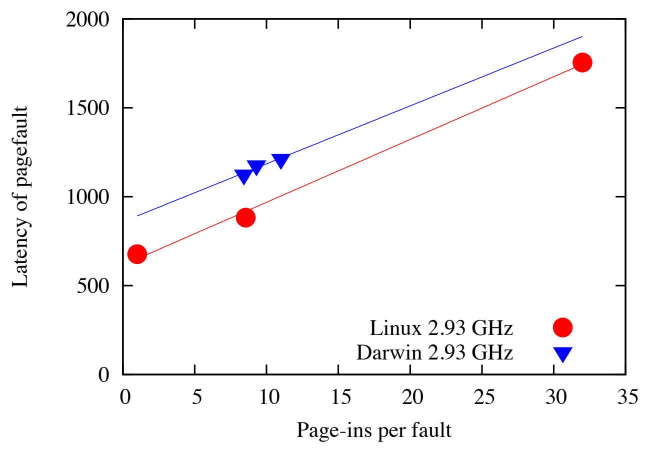


Figure 9: Page-fault latencies with linear model for C_2.

7. RELATED WORK

Related work falls into two categories (1) critical analyses of micro-benchmarks and (2) micro-benchmark development frameworks.

7.1 Critical analyses

Micro-benchmarks often contain flaws due to the inherent difficulties associated with developing micro-benchmarks. Consequently, micro-benchmarks are often criticized throughout the research literature, forums, and mailing lists. A complete survey of these critiques is beyond the scope of this paper, but we discuss a notable critique here.

Brian Goetz published notable micro-benchmark critique [4]. His critique focused on a flawed micro-benhcmark that attempted to measure the performance of Java concurrency primitives. Goetz discusses the fundamental challenges associated with developing Java micro-benchmarks and concludes that only Java experts should endeavor to develop micro-benchmarks. Similar to Goetz, we discuss the fundamental difficulties

To the best of our knowledge, no other work exists that provides a specific critical analyses of PI micro-benchmarks.

7.2 Development frameworks

We begin by reviewing the uniques aspects of the Yamba micro-benchmark-development framework. First and foremost, Yamba is the only development framework that specifically facilitates the development of APD micro-benchmarks. Yamba is also, to the best of our knowledge, the only tool that specifically facilitates the development of machine-code micro-benchmarks. Yamba is further unique through its use of a domain-specific programming language that allows programmers to succinctly express machine-code benchmarks using a variety of different abstraction levels.

Although Yamba is unique in several regards, the design of Yamba has been influenced by micro-benchmark systems developed in prior research. Many researchers have developed specialized tools to facilitate the execution and measurement of micro-benchmarks. It seems, though, that most often these tools are tied to specific micro-benchmark suites and are either unpublished or unsuitable for micro-benchmark development. There are a few notable exceptions that we are aware of: lmbench, hbench, X-Ray, and Japex.

7.3 Lmbench and hbench

In 1996, Larry McVoy and Carl Staelin first published lmbench, a portable micro-benchmark suite designed to measure the performance of operating systems [9]. Lmbench uses micro-benchmarks written in ANSI C to measure the latency and bandwidth of operating system operations. Although lmbench is primarily a micro-benchmark suite, lmbench also has built in functionality to support the development of new micro-benchmarks.

Shortly after the initial lmbench release, Brown and Seltzer fokred lmbench, creating hbench. While lmbench focused on achieving portability through platform independence, hbench primarily focused on studying operating systems on x86. When lmbench was unsuitable for certain tasks, the authors of hbench adapted it to suit their specific needs. For example they utilized x86 performance counters to measure the latency of destructive operations. This early example foreshadowed the problems with the PI approach that become more apparent as systems continued to evolve.

7.4 X-Ray

The X-Ray tool uses high-level micro-benchmarks, written in C, to automatically measure the instruction cache capacity of processors [11]. Like Yamba, X-Ray uses a micro-benchmark compiler that generates micro-benchmark code from user-defined parameters. In X-Ray, the control engine reads micro-benchmark parameters to produce micro-benchmark specifications. A micro-benchmark generator then translates specifications to C source code, which is then compiled and executed. This build process is similar to Yamba's, with the exceptions that: (1) In Yamba, users create micro-benchmark specifications directly (as Yamba classes). In X-Ray, users define parameters and specifications are generated. (2) Yamba generates machine-code-level micro-benchmark code whereas X-Ray generates high-level micro-benchmark code.

7.5 Japex

Japex is a micro-benchmark development framework written in Java, for Java [8]. Japex aims to alleviate micro-benchmark developers from the tedious aspects of Java micro-benchmark development. To accomplish this goal, Japex primarily acts as a micro-benchmark control engine. The Japex control engine performs tasks such as warming up the Java Virtual Machine (JVM), making time measurements, and forking threads. Like Yamba, Japex is designed to be extensible.

8. CONCLUSIONS

Our analysis of the PI micro-benchmark approach shows that PI micro-benchmarks suffer from two fundamental limitations.

FUNDAMENTAL LIMITATION 1. *Pure PI micro-benchmarks cannot safely measure the performance of platform-specific operations.*

FUNDAMENTAL LIMITATION 2. *Impure PI micro-benchmarks are only accurate on systems that support their assumptions.*

Our experiments demonstrate the quantitative effects of the problems that result from these fundamental limitations. We also experimentally evaluated that an APD micro-benchmark is more accurate and precise than its PI counterpart.

Ultimately this work shows that platform independence is an illusion; micro-benchmarks are inherently not platform-independent. It would not be possible for a *pure* PI micro-benchmark to generate Figure 9. Though an *impure* PI micro-benchmark could generate this graph, it would only work on systems that support the benchmark's assumptions. We propose that micro-benchmark developers accept this reality, and embrace the APD philosophy when developing portable micro-benchmarks.

9. REFERENCES

[1] Andrew Bauman, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Symposium on OS Principles*, 2009.

[2] Alexandre X. Duchateau, AlbertSidelnik, María Jesús Garzarán, and David Padua. P-Ray: A suite of micro-benchmarks for multi-core architectures. In *Proceedings of The International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[3] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Workshop on Hot Topics in Operating Systems*, 2007.

[4] Brian Goetz. Java theory and practice: Anatomy of a flawed microbenchmark. `http://www.ibm.com/developerworks/java/library/j-jtp02225.html`, February 2005.

[5] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F.H. Sumner. One-level storage system. *IEEE Transactions on Electronic Computers*, EC-11(2), 1962.

[6] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference on Advances in Cryptology*, 1996.

[7] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security*, 2009.

[8] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Workshop on Hot Topics in Operating Systems*, 1999.

[9] Carl Staelin. lmbench: an extensible micro-benchmark suite. *Software: Practice and Experience*, 35(11):1079 – 1105, May 2005.

[10] Carl Staelin. lmbench-3.0-a9. `http://sourceforge.net/projects/lmbench/`, November 2007.

[11] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX Annual Technical Conference*, 1998.

[12] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *International Conference on Middleware*, 2008.

[13] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *International Conference on the Quantitative Evaluation of Systems*, 2005.