Pitfalls of Accurately Benchmarking Thermally Adaptive Chips

Laurel Emurian[*] Arun Raghavan[§] Lei Shao[‡] Jeffrey M. Rosen [†]
Marios Papaefthymiou[†] Kevin Pipe[†‡] Thomas F. Wenisch[†] Milo Martin[*]

[*] Dept. of Computer and Information Science, University of Pennsylvania
[†] Dept. of Electrical Engineering and Computer Science, University of Michigan
[‡] Dept. of Mechanical Engineering, University of Michigan
[§] Oracle Labs

Abstract

The performance of today's chips varies over time due to active thermal management and energy conservation policies. Such changes in performance are necessary in thermally constrained systems that operate beyond sustainable thermal limits, such as Intel's second generation Turbo Boost. This varying performance creates challenges in accurately benchmarking such systems. This paper gives examples of potential pitfalls of benchmarking thermally aware systems: extrapolating steady-state performance from a short run, comparing throughput across program runs of different lengths, and failing to consider recent system activity. We analyze situations in which these pitfalls can lead to measurement errors and discuss potential mitigation strategies. We experimentally demonstrate that simple methodological mistakes can result in measurement errors of 8% or more between steady-state and instantaneous throughput on a Turbo Boost-enabled system today. We conclude by discussing the implications of a widening gap between peak and sustainable performance in future systems, such as the recently proposed computational sprinting.

1. Introduction

Benchmarking systems accurately has always been challenging due to measurement bias and system start up costs, such as variability in cache state and instruction counts [3, 5, 9, 13, 15]. The emergence of thermally aware systems has added new challenges to benchmarking, such as time-varying workload throughput due to frequency changes and warm-up periods of up to a few minutes to reach thermal steady state.

Although dynamically adapting processor operation is implemented almost universally in today's systems, its impact on measuring performance has grown in prominence with the increasing power density (watts per unit area) of more recent processors and the shift to mobile systems. Initial active thermal management policies throttled performance only in emergencies to prevent an atypical workload (*e.g.*, a power virus) from overheating the system [6]. As chip power constraints further increased with each generation of CMOS technology, multicore chips began employing schemes such as AMD's PowerNow! and Intel's first generation Turbo Boost to sustainably boost the frequency of active cores by borrowing from the headroom afforded by other idle cores. Although these early performance policies adapted to changes in system configurations or certain worst-case workloads, temperature-induced adaptations were seldom invoked during typical executions, resulting in unperturbed steady-state performance for most applications.

With the increase in chip power density, the gap between a system's peak performance configuration and its ability to vent heat has widened even further. Hence, processors expose their highest performance states only temporarily. Intel's second generation Turbo Boost boosts frequency for tens of seconds before throttling down to a thermally sustainable level. This paper discusses how these transient performance states can introduce throughput estimation errors under the common benchmarking assumptions of steady-state performance.

	Frequency Range
Non-Turbo frequency	1.8 GHz - 2.4 GHz
Turbo mode frequency	2.7 GHz - 3.0 GHz

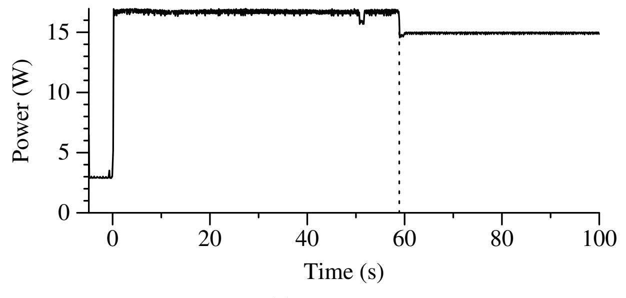
Table 1. Operating frequencies of "Haswell" core i7 4500U.

We find that even though such systems may employ peak-performance modes for only a few to several tens of seconds, the transients persist over several minutes. Such large periods are unlike the cache or branch-predictor warm-up times, which last for at most a few seconds, considered by previous work on benchmarking [7, 15]. These transients add to the subtlety of extrapolating steady-state performance from shorter runs. In addition, different systems may implement different adaptation mechanisms, *e.g.*, based on temperature or weighted average of power over a time window.

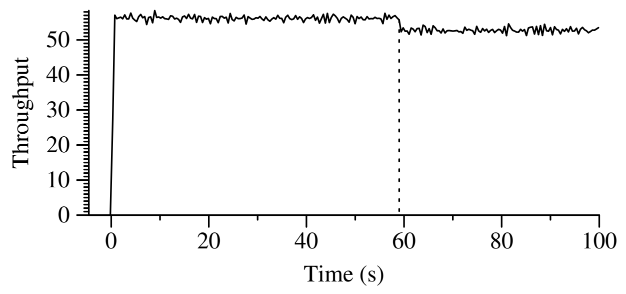
In this work, we identify and evaluate benchmarking pitfalls on a current generation processor equipped with Intel's second generation Turbo Boost. This paper enumerates the perils of benchmarking on thermally adaptive systems, making the following contributions:

1. We demonstrate how extrapolating steady state performance from a short benchmarking run leads to benchmarking errors.
2. We show that comparing benchmark throughput across runs of different lengths can lead to false results.
3. We model how long a workload must run to achieve a desired bound on such measurement error.
4. We demonstrate how failing to consider recent system activity can falsely deflate performance results and suggest approaches to ensure a consistent initial state to eliminate error arising from prior activity.

We note that these pitfalls will likely become more pronounced as thermally aware systems continue to advance. The impact of the pitfalls depends on the gap between the boosted and throttled performance. If current trends continue, systems will potentially boost further past their sustainable power limits, increasing the ratio between boosted and throttled performance. For example, recent work on computational sprinting [10, 11] proposes deliberately engineering future systems to provide sub-second bursts of even larger peak-performance (up to 10× or more by incorporating phase-change materials to buffer heat for short durations). As this burst ratio increases, the magnitude of benchmarking pitfalls increases as well.



(a) Power



(b) Throughput

Figure 1. Power consumption and throughput with Turbo Boost enabled. The dotted line indicates that the system throttles down after approximately 55 seconds.

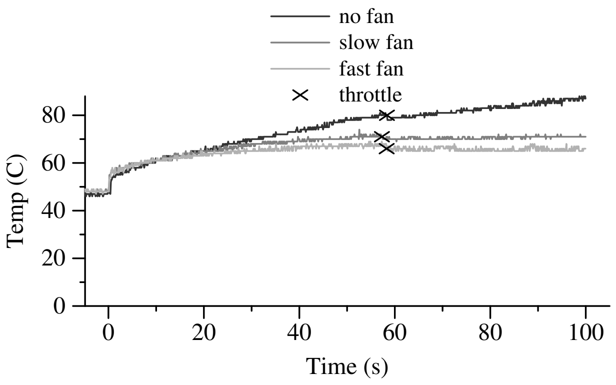


Figure 2. System temperature when run with different fan speeds. The 'x' marks when Turbo Boost throttles to a lower frequency. Because the system throttles down after about 55 seconds regardless of fan speed, we conclude that temperature does not affect when Turbo Boost throttles.

2. Background and Analysis of Turbo Boost

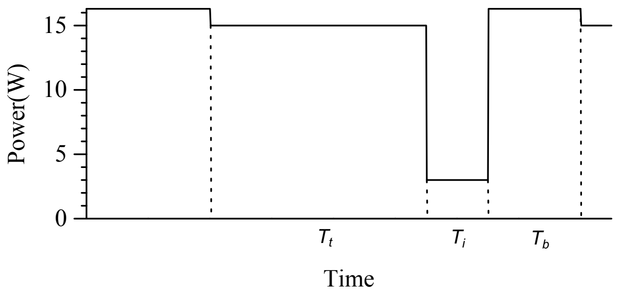
In this section, we characterize Turbo Boost behavior on our experimental machine, which we then use to demonstrate benchmarking pitfalls.

Background. Intel introduced second generation Turbo Boost with the release of the "Sandy Bridge" processor. Unlike the previous generation, which boosted frequency only within sustainable thermal power, second generation Turbo Boost allows processor power to temporarily exceed the sustainable cooling rate, relying instead on the system's ability to buffer the excess heat as component temperatures rise over time [1, 10–12]. Because such operation will eventually drive chip temperature beyond safe margins, the hardware implements a dynamic policy to throttle frequency down to sustainable limits.

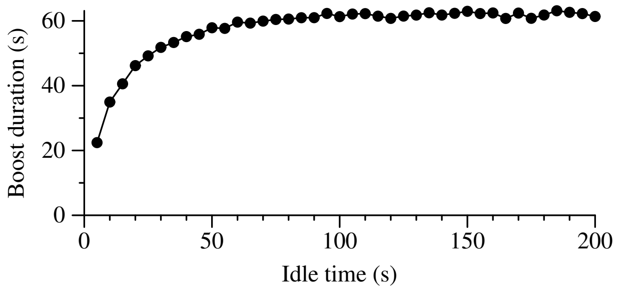
Second generation Turbo Boost operation. Figure 1(a) shows processor power over time with second generation Turbo Boost operation on a dual core "Haswell" (Intel Core i7 4500U) laptop. After prolonged idle time, activating computation at time zero causes both cores to operate at the boosted frequency of 2.7 GHz. Correspondingly, the power increases from the idle state draw of 3 W to the boosted power level of 17 W. After 55 s of execution, a control mechanism on the chip throttles system frequency, reducing power to the sustainable level of 15 W. Figure 1(b) shows the corresponding decrease in throughput resulting from the frequency throttling down to the nominal 2.4 GHz.

Sensitivity to temperature. Despite being motivated primarily by thermal concerns, we found that the Turbo Boost control policy implemented by this Intel chip does not depend directly on temperature. Figure 2 shows chip temperature over time with the system operating with three different fan speeds (including one with the fan turned off). The hardware throttles all three executions after nearly the same interval—55 s—regardless of the temperature. Although temperature-based throttling is invoked as an emergency measure, we conclude the primary control mechanism for Turbo Boost is not based on temperature.

Turbo Boost throttle policy. Intel documents that Turbo Boost throttles frequency based on running average power limit—a weighted accumulation of energy derived from several activity monitors [12]. The hardware then ensures that the average power



(a) Power



(b) Boost Duration

Figure 3. Boost duration after idling for a certain number of seconds. The idle period, T_i, occurs after a workload has saturated the power history window. To ensure that the power history window has been saturated, we allow the workload to run for two minutes during the throttle period, T_t. The boost duration, T_b, is the amount of time that the workload remains operating at boosted frequency immediately following the idle period. The power history window is about 120 seconds long, shown by the graph reaching an asymptote.

sampled over a time window remains within the expected sustainable power limits by raising and lowering frequency. The control policy is influenced by the boost frequency, duration, and averaging time window. We use a simple microbenchmark to experimentally determine these values. The power trace shown in Figure 3(a) demonstrates how we ascertained the averaging window. After the initial boost period, we operate the cores at the sustainable frequency for about 120 seconds, T_t, after which we vary the idle time, T_i, before a repeated execution. We then record how long a subsequent boost, T_b, lasts. Figure 3(b) shows the system is always able to function at the boosted frequency for the entire 55 s duration when preceeded by at least two minutes of idleness. When running workloads with Turbo Boost enabled, prior activity on the system will hence affect the workload performance.

3. Benchmarking Thermally Aware Systems

Consider a scenario in which a programmer runs a simple program to evaluate the throughput of a system. To remove noise in the data, the programmer decides to run the program ten times and take the average of the results. However, after the programmer gets the results back, the programmer realizes that the performance of the first and last run differ by about 6%. The programmer decides to repeat the experiment and run the same program for a longer period of time. Now there is an 8% difference in throughput between the first run from the initial (short) set and first run of the second (longer) set. The throughputs of the first and last run in the second set still differ by 6%. The programmer begins to wonder if something is wrong with either the program or the system under test. In fact, the real issue is that Turbo Boost is enabled on the experimental system.

In this section, we explore the perils of collecting performance results on a system that has Turbo Boost enabled. We discuss the problems of extrapolating performance from a short run, comparing throughput across program runs of different lengths, and issues with running multiple back-to-back iterations of one program. We use a simple 4-threaded saxpy workload that fits in the L1 cache in our experiments.

For all of the following pitfalls, we could mitigate the problem with Turbo Boost by disabling it. However, on our system, turning Turbo Boost off results in the system only running at about 1.8GHz, which is much less than the 2.4 GHz sustainable frequency that the system settles at with Turbo Boost enabled. Disabling Turbo Boost thus may grossly under-report the potential system performance. Running at a reduced frequency will also underestimate the impact of memory performance bottlenecks. This option is also not feasible if one is evaluating performance on a shared machine, in which a non-root user may not have permission to disable Turbo Boost.

In all of our pitfalls, we suggest solutions that will minimize the error of performance results. Several of these suggestions involve running at a constant frequency for the duration of a workload run. Running during either the boosted frequency or the throttled frequency is a viable option for running at a constant frequency. We leave the "correct" choice up to the programmer and their specific benchmarking goals.

3.1 Pitfall: Extrapolating steady state performance from a short run

A common benchmarking practice is to warm up before taking measurements in order to get accurate benchmarking results [15]. This warm up period allows microarchitectural structures (such as the cache, branch predictor, TLB), and OS file system caches to reach a steady state before performance measurements are taken.

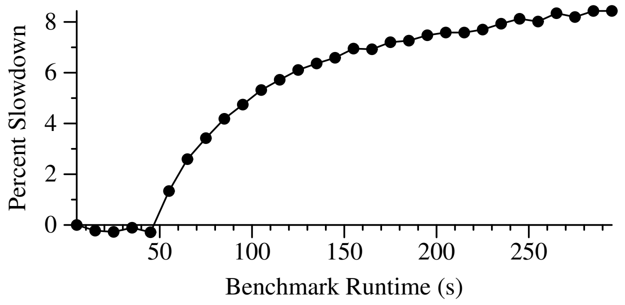


Figure 4. Percent slowdown of workloads run for increasing length. As workload length increases, a higher proportion of workload runtime is spent running at the throttled frequency resulting in greater slowdown.

Programs can execute in different phases. Therefore, it is generally ill-advised to extrapolate entire benchmark performance from truncated runs [7]. In non-thermally-adaptive systems, necessary warm up times are often on the order of seconds. However, Turbo Boost requires a warm-up period of at least a minute because the system boosts to a higher frequency for about a minute. If program performance is extrapolated from within this minute, it will be distorted. In this case, we define a short run as a program that runs for a minute or less.

Due to the long warm-up period required when Turbo Boost is enabled, programmers must be careful about extrapolating performance measurements from a run that is tens of seconds long. Often when running a benchmark, a programmer will assume that the performance of a short run is equivalent to the performance of a longer run. For example, if we run a program for 20 seconds and find that the program does about one unit of work per second (a throughput metric), we would assume that the same program would execute one unit of work per second when run for 100 seconds as well. However, throughput is not the same across these cases if Turbo Boost is enabled. Turbo Boost throttles frequency after the system has consumed a certain amount of power, so the frequency of a long benchmark run will drop during execution. The frequency of a workload whose runtime fits within the boosting period does not change, but the frequency of a workload whose runtime does not fit within the boosting period does change, so the performance of the two runs is incomparable. The differences in performance between workload lengths that fit within the boosting period and those that do not can be significant. When we state differences in performance, we mean the percentage difference between throughput of a program run on a machine that does not throttle frequency and the same program run with Turbo Boost. We will refer to this percentage in performance difference as performance error.

To experimentally confirm this pitfall, we ran a microbenchmark for increasing time intervals and recorded the performance of each run. Figure 4 illustrates the difference in performance error if a programmer assumes that performance will remain steady at the boosted frequency. Our microbenchmark runs for about 55 seconds of boosted time before throttling frequency. We can see that the benchmark instances that run for less than 55 seconds maintain a constant frequency throughout their run and therefore have stable performance. However, performance results begin to vary once workload runtime exceeds 55 seconds. At this point, Turbo Boost throttles down, causing the workload to run at a lower frequency for part of its execution. This change in frequency causes up to 8% change in reported performance from the shortest workload length to the longest. In other words, if we simply extrapolated the performance of our short run to that of a longer run, we would be overestimating performance by 8%.

Mitigation. To prevent this pitfall, the programmer should be aware of the length of the boost period and avoid extrapolating results past the boost period. A second mitigation would be to run the program for a long enough time that the effects of Turbo Boost are negligible. However, this run can take up to several minutes, which is much longer than is typically assumed to be necessary to avoid startup transients. We discuss how long is "long enough" in the following section.

3.2 Pitfall: Comparing program runs of different lengths

Next, consider the case of evaluating the performance improvement in a program as a result of an algorithmic or compiler optimization. The relative performance improvement (*i.e.*, speedup) is commonly computed as the ratio of the run times of the unoptimized and optimized versions. The optimized workload, being a shorter run, spends a larger fraction of its execution time at the boosted frequency when compared to the longer, unoptimized version. The observed speedup of a set of optimizations will likely appear larger when running with Turbo Boost enabled than when run at a constant frequency. Further, this error is largest when the optimized run fits exactly within the boost window. Below, we quantify this error in speedup as a function of peak and sustainable throughputs (*i.e.*, the ratio of boosted and sustainable frequencies).

Let t_t be the total execution time of the unoptimized program and t_b be the total execution time of the optimized program. We assume that both programs are run after the system has been idle. Because we seek to quantify the maximum error, we assume that the runtime of the optimized program is exactly equal to t_b. Let F_s be a sustainable frequency. In this case, we consider the speedup and peak throughput of a both the optimized and unoptimized program on a sustainable system running at frequency F_s. Therefore:

$$\begin{aligned} Sustained\ speedup\ &= S = \frac{t_t}{t_b} \\ Sustained\ unoptimized\ throughput &= \frac{F_s \cdot t_t}{t_t} \\ &= F_s \\ Sustained\ optimized\ throughput &= \frac{F_s \cdot t_b}{t_b} \\ &= F_s \end{aligned} \tag{1}$$

Consider the same experiment performed with the same set of optimizations, this time on a boost enabled system. Despite using the same optimized and unoptimized programs, the resulting speedup that is reported is now much higher than the speedup on the sustained system. On the boost enabled system, whereas the optimized run completes in boosted mode, (*i.e.*, at a boosted frequency F_b for the entirety of duration t_b), the unoptimized run throttles down to frequency F_s for the remainder t_t- t_b period after the initial t_b seconds of boosting. Therefore:

$$\begin{aligned} Boosted\ speedup\ &= S = \frac{t_t}{t_b} \quad \textbf{(pitfall)} \\ Boosted\ unoptimized\ throughput &= \frac{F_b \cdot t_b + F_s \cdot (t_t - t_b)}{t_t} \\ Boosted\ optimized\ throughput &= \frac{F_b \cdot t_b}{t_b} \\ &= F_b \end{aligned} \tag{2}$$

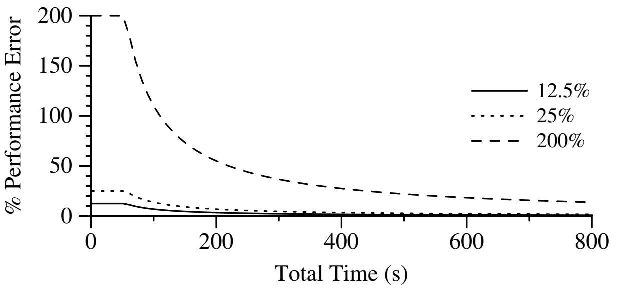


Figure 5. Modeled performance difference for a programs run for a certain duration of where t_b = 55 seconds and r_{boost} is 12.5%, 25%, and 200%. At least twenty minutes is needed for performance difference to be less than 5% when r_{boost} is 25%. This amount of time increases as r_{boost} increases.

Turbo Boost thus inflates the speedup achieved by the optimization because the unoptimized baseline on the boosted system completes some of its execution at a lower frequency than the optimized version, which completes its entire execution at the boosted frequency. This inflation error can be computed from the ratio of the relative throughputs of the two unoptimized executions (Equation 1, Equation 2), the observed speedup (S), and the ratio of boosted to steady-state frequency as:

$$\begin{aligned} Frequency\ boost\ ratio\ &= r_{boost} = \frac{F_b}{F_s} \\ Max\ Speedup\ error &= \frac{S \cdot r_{boost}}{r_{boost} + (S-1)} - 1 \end{aligned} \tag{3}$$

For the frequency values in our experimental system (r_{boost} = 12.5%), Equation 3 estimates a maximum speedup error of 5.8% as a result of enabling Turbo Boosts for workloads that nominally experience a 2× speedup. Further, the error grows with either factor (*i.e.*, S or r_{boost}). For example, doubling the boost frequency ratio for the same nominal speedup (S = 2×, r_{boost} = 25%) increases maximum performance estimate error of 11%. The same 11% error would also manifest in an alternative scenario with the original boost ratio, but a ten-fold increase in speedup at the original boost ratio (S = 10×, r_{boost} = 12.5%).

Proposals like computational sprinting suggest that future thermally constrained systems may boost frequency more aggressively, (for example r_{boost} = 3× or more) [10, 16]. Equation 3 predicts that such a 3× boost in frequency would cause an error of up to 50% for programs with a nominal speedup of 2×, and an error of 150% if the nominal speedup was 10×. Such systems would hence exacerbate the performance estimation error if the gap between sustainable and peak frequency continues to widen.

Mitigation. To mitigate this pitfall, we suggest either ensuring fixed frequency during runs, ensuring fixed workload runtimes or running a workload long enough that the performance difference from the change in frequency has subsided. We can achieve constant frequency during runs by either never running a workload longer than the boost window or by disabling Turbo Boost and pinning each active core to the same frequency. Both methods would prevent frequency from changing during runtime and would reduce measurement error.

If it is not possible to run a workload within the boost period (for example, because the workload cannot complete a run in that time period or lack of access to Turbo Boost override functions), then all workloads should be run for the same length of time when measuring throughput. Each workload run will have a similar ratio

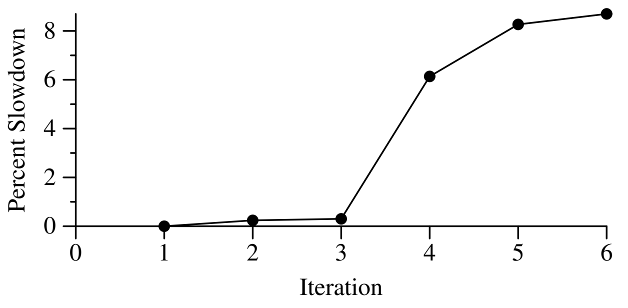


Figure 6. Percent slowdown when workloads are run back-to-back. The power history of Turbo Boost causes each subsequent iteration to have a shorter boost duration, leading to decreased performance.

of boost frequency to throttle frequency, reducing measurement error.

A third alternative is to run programs for a long time period such that the performance error is negligible. The equations relating throughput, total time, boost time, and boost ratio (Equation 1, Equation 2), can be used to determine the performance error for a given run length:

$$Speedup\ Error = \frac{r_{boost} \cdot t_b + F_s \cdot (t_t - t_b)}{t_t} - 1$$

Figure 5 illustrates the above trend of performance error decreasing with execution time for different frequency boost ratios. We fix the boost window (t_b) to 55 seconds, which is the average observed duration on our experimental system. Each line represents a specific percent difference in boost and throttle frequencies. Our current system has a difference (r_{boost}) of 12.5% between the boosted frequency and throttled frequency when both cores are active. We chose a 25% difference because it accurately reflects the difference between the boosted and throttled frequency for one active core on our system. Although we have not seen our system throttle when one core is active, it is possible that such a frequency range will throttle in the future. We also chose a 200% difference because this is the boosted frequency difference that sprinting systems such as computational sprinting may be able to sustain in the future. Other proposals suggest up to a 15× boost above sustainable thermal limits [16].

Figure 5 shows that it takes several minutes for the performance error to be negligible. The first 55 seconds of Figure 5 show no change in performance error since the total time is within the boost window and the frequency of the system does not change. After the system throttles, the frequency lowers to a sustainable level, so the performance error begins to move towards an asymptote of zero. If r_{boost} equals 12.5%, the performance error is 1% after ten minutes of workload execution. As r_{boost} increases, a workload must be run for longer periods of time to get a negligible error. For example, if we wanted to obtain less than 5% performance error when r_{boost} equals 200%, we would have to run a workload for at least 37 minutes.

3.3 Pitfall: Failing to consider recent activity

When multiple workloads are run back-to-back with Turbo Boost enabled, as may be done when running a suite of benchmarks, performance results will be inconsistent. Turbo Boost uses a history of power consumption to determine when to throttle. Therefore, the length of a boost depends on the amount of power previously consumed by the system. We expect a shorter boost period if the system was not idle immediately before a boost in frequency. In Section 2 we found that our system takes the power history of the previous 120 seconds into account. When two consecutive workloads are run within 120 seconds of one another, the repeated workload will have a shorter boost period than the initial workload, affecting performance results.

We run a microbenchmark for 15 seconds for six separate runs without pausing between each run. We repeat this experiment five times and average the results. Figure 6 shows that the first three runs have all boosted for the full 15 seconds and therefore have a 0% difference in performance. We refer to any performance difference between runs that have boosted for the entirety of their execution and runs that have not as error. The 4th run has a 6% error because it has only boosted for part of its execution. The 5th and 6th runs exhibit 8% error because these runs do not boost at all.

Due to Turbo Boost's use of power history, the runs do not have the same boost duration. The first three runs have boosted for the full 15 seconds of their execution because their cumulative runtime of 45 seconds is less than the 55 second boost window. The 4th run consumes the remainder of the boost window and the system throttles down during its execution, leaving no boost time for the 5th and 6th run. Therefore, failing to consider recent system activity can result in a falsely deflated performance result of up to 8%.

Mitigation. To prevent this pitfall we recommend either starting a workload with a "clean" power history or fully saturating the power history until a boost no longer happens. We can achieve a clean power history by idling between each run for the full length of the power history window, which is about 120 seconds on our system. An alternative is to fully saturate the power history by running a workload until it exhausts the boost period and throttles down and then immediately run the workloads to be measured.

4. Related Work

Several researchers have studied Intel's Turbo Boost. Charles et al. [4] analyzed the power and performance trade offs of a previous version of Turbo Boost that does not throttle by characterizing the system with CPU and memory intensive workloads. They found that Turbo Boost increases performance by up to 6% but also increases energy consumption by about 16%. Wamhoff et al. [14] focus on comparing Intel's Turbo Boost and AMD's Turbo CORE. They use their comparison to write and evaluate a library that helps optimize software based on frequency scaling. Lo et al. [8] analyze the impact of Turbo Boost 2.0 on metrics including system power and performance. They build a model that predicts the optimal Turbo Boost setting for each metric.

Several studies have been done on the system warm-up periods required to ensure accurate benchmarking effects. These warm-up periods generally last for a few seconds, whereas the warm-up effects we discuss last for at least a minute. Wunderlich et al. [15] propose SMARTS, a framework for accurately measuring benchmarks in simulation. They discuss that a variety of microarchitectural state, including the cache, branch predictor, and TLB, need to be warmed up before measurements are taken to ensure accurate results. Hsu et al. [7] discuss how small pieces of a benchmark cannot be representative of an entire application because applications comprise different phases.

There has also been work done on reducing measurement bias in benchmarking. However, none of these works consider the benchmarking pitfalls that arise in thermally aware systems. Mythkowitz et al. [9] describe how changing variables such as link order or UNIX environment size introduces measurement bias. They propose several solutions for reducing measurement bias, including using a large benchmark suite and randomizing experimental setup. Curtsinger et al. [5] introduce Stablizer, a tool for ensuring that performance evaluations of software are statistically significant. Alameldeen et al. [2] discuss thread scheduling bias in simulated multi-threaded workloads. In a later work, they determine that IPC does not accurately characterize workloads [3]. Tsafrir et al. [13] introduce input shaking—executing multiple simulations with random variations in each workload to determine which results contain noise artifacts.

5. Conclusion

Programmers have had to overcome challenges of measurement bias and microachitectural structure warm up time to accurately benchmark systems. Thermally constrained systems have introduced additional benchmarking pitfalls, including longer warm up times due to the relevance of prior system power history and the boosting and throttling of frequencies during workload execution.

This paper discussed several pitfalls of benchmarking thermally adaptive systems that can befall programmers when evaluating performance. Measurement error occurs when programmers extrapolate performance from runs of less than a minute. Comparing runs of different lengths may lead programmers to produce erroneous results and report that system performance has been deflated by 8%. Performance results may be deflated when a programmer fails to consider system power history of more than a few minutes.

The above performance differences may grow worse over time. If techniques such as computational sprinting become commonplace, systems will be able to boost past their sustainable power at higher ratios. When comparing programs of different lengths, higher boost ratios could lead to mispredicting performance by much larger percentages than those produced by running Turbo Boost. As boost ratios continue to increase, measurement error will grow as well. The future of sprinting systems will lead to greater performance and responsiveness, but will also introduce more challenges and greater pitfalls when benchmarking.

References

[1] Intel Turbo Boost Technology 2.0. URL `http://www.intel.com/technology/turboboost/index.htm`.

[2] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the Ninth Symposium on High-Performance Computer Architecture*, Feb. 2003.

[3] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEEMICRO*, 26(4):8–17, July 2006.

[4] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel®Core™i7 Turbo Boost feature. In *Proc. of the IEEE Int'l Symp. on Workload Characterization*, Sept. 2009.

[5] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2013.

[6] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal*, Q1 2001.

[7] W. C. Hsu, H. Chen, P. C. Yew, and H. Chen. On the predictability of program behavior using different input data sets. In *Interaction between Compilers and Computer Architectures, 2002. Proceedings. Sixth Annual Workshop on*. IEEE, 2002.

[8] D. Lo and C. Kozyrakis. Dynamic Management of TurboMode in Modern Multi-core Chips. In *Proceedings of the 17th Symposium on High-Performance Computer Architecture*, Feb. 2014.

[9] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[10] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational Sprinting on a Hardware/Software Testbed. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2013.

[11] A. Raghavan, Y. Luo, A. Chandawalla, M. C. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational Sprinting. In *Proceedings of the 17th Symposium on High-Performance Computer Architecture*, Feb. 2012.

[12] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power Management Architecture of the 2nd Generation Intel Core Microarchitecture, Formerly Codenamed Sandy Bridge. In *Hot Chips 23 Symposium*, Aug. 2011.

[13] D. Tsafrir, K. Ouaknine, and D. G. Feitelson. Reducing performance evaluation sensitivity and variability by input shaking. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS'07. 15th International Symposium on*. IEEE, 2007.

[14] J. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The Turbo Diaries: Application-controlled Frequency Scaling Explained. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Apr. 2014.

[15] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS - Accelerating Microarchitecure Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[16] H. Zhang, R. Amirtharajah, C. Nitta, M. Farrens, and V. Akella. Burst Mode Processing: An Architectural Framework for Improving Performance in Future Chip MultiProcessors. *Workshop on Managing Overprovisioned Systems (W-MOS)*, 2014.