Dynamic Management of TurboMode in Modern Multi-core Chips

David Lo and Christos Kozyrakis
Stanford University
{davidlo, kozyraki}@stanford.edu

Abstract

Dynamic overclocking of CPUs, or TurboMode, is a feature recently introduced on all x86 multi-core chips. It leverages thermal and power headroom from idle execution resources to overclock active cores to increase performance. TurboMode can accelerate CPU-bound applications at the cost of additional power consumption. Nevertheless, naive use of TurboMode can significantly increase power consumption without increasing performance. Thus far, there is no strategy for managing TurboMode to optimize its use across all workloads and efficiency metrics.

This paper analyzes the impact of TurboMode on a wide range of efficiency metrics (performance, power, cost, and combined metrics such as QPS/W and ED^2) for representative server workloads on various hardware configurations. We determine that TurboMode is generally beneficial for performance (up to +24%), cost efficiency ($QPS/\$$ up to +8%), energy-delay product (ED, up to +47%), and energy-delay-squared product (ED^2, up to +68%). However, TurboMode is inefficient for workloads that exhibit interference for shared resources. We use this information to build and validate a model that predicts the optimal TurboMode setting for each efficiency metric. We then implement autoturbo*, a background daemon that dynamically manages TurboMode in real time without any hardware changes. We demonstrate that* autoturbo *improves $QPS/\$$, ED, and ED^2 by 8%, 47%, and 68% respectively over not using TurboMode. At the same time,* autoturbo *virtually eliminates all the large drops in those same metrics (-12%, -25%, -25% for $QPS/\$$, ED, and ED^2) that occur when TurboMode is used naively (always on).*

1 Introduction

There is a constant need for higher performance in computer systems. This is particularly the case for large-scale datacenters (DCs) that host demanding popular services such as social networking, webmail, streaming video, websearch, and cloud computing. These applications demand both high throughput to support large number of users and low latency to meet Quality of Service constraints for each user. Nevertheless, performance is not the only efficiency metric. A large concern for datacenter operators is the total cost of ownership (TCO) for a certain level of service, including both capital and operational expenses. The two largest TCO components are the cost of servers and the cost to power them, which can be 53% and 19% respectively [10]. Therefore, any optimization should also target metrics that capture energy and cost efficiency such QPS/W or $QPS/\$$ for throughput workloads and energy-delay products (ED or ED^2) for latency sensitive applications.

This paper focuses on dynamic CPU overclocking (TurboMode) as a method to increase the efficiency of server computers. While our work focuses on server workloads, the techniques we present can be used in any kind of computer system that uses TurboMode. TurboMode has been recently introduced on x86 multi-core chips from Intel and AMD. It overclocks a cores by utilizing available thermal headroom from idle execution resources [12, 1]. Current chips can dynamically overclock a core by up to 40% or more, which can lead to an increase in performance by up to 40%. As the number of cores per chip increases over time, the overclocking range and the performance implications of TurboMode will also increase. TurboMode is controlled by firmware using an embedded hardware controller that sets the exact clock frequency based on the thermal headroom available and the expected performance benefits. Software has little control over TurboMode, except for the ability to enable/disable it.

Unfortunately, TurboMode is not always beneficial and deciding when to enable it is quite complex. As seen in Section 3, the optimal TurboMode setting varies across different applications, hardware platforms, and efficiency metrics. There is no clear static setting or simple heuristic for TurboMode management. Naively turning on TurboMode (i.e., relying solely on the firmware controller) can actually lead to energy waste of up to 28% and decrease cost efficiency in some cases. At the same time, disabling TurboMode all the time misses opportunities for huge gains in energy and cost efficiency. We need an automatic system that manages TurboMode intelligently, maximizing its benefits when applicable and masking its impact otherwise.

Towards this goal, we make the following contributions:

1. We characterize the impact of TurboMode on performance, power, cost and energy efficiency metrics for various server workloads on several hardware platforms (Intel and AMD). We demonstrate that turning on TurboMode can greatly improve such metrics (ED^2 up to 68%) but can also significantly degrade the same metrics in the presence of workload interference (ED^2 up to 25%). The wide variability in TurboMode's efficiency necessitates a dynamic control scheme.
2. We develop a predictive model that utilizes machine learning to predict the proper TurboMode setting. We demonstrate that our modeling technique is effective for a wide range of efficiency metrics and different hardware configurations. It can accurately predict the optimal TurboMode setting for various application configurations.
3. We implement *autoturbo*, a daemon that uses the model to dynamically control TurboMode. The daemon requires neither hardware modifications nor a priori knowledge of the workload. We demonstrate that it improves both energy efficiency (ED by 47% and ED^2 by 68%) and cost efficiency ($QPS/\$$ by 8%). More importantly, it eliminates nearly all the cases where TurboMode causes efficiency drops.

We expect the utility of *autoturbo* to be even greater with future multi-core chips. With more cores per chip, the frequency range and potential of TurboMode will be larger. At the same time, there will be increased interference for shared resources such as caches, memory channels, and on-chip or off-chip interconnects.

The rest of the paper will be organized as follows. Section 2 describes TurboMode and current implementations of it from x86 server CPU vendors. Section 3 analyzes the impact of TurboMode for various workloads, metrics, and machine configurations. Section 4 describes our implementation of *autoturbo* and Section 5 evaluates its efficacy. Section 6 presents related work and Section 7 concludes the paper.

2 TurboMode Background

TurboMode (TM) is a form of dynamic voltage frequency scaling (DVFS). Unlike prior applications of DVFS which decrease CPU clock frequency and voltage to save power, TM operates in the opposite direction. Namely, TM *increases* CPU clock frequency in the active cores when a multi-core processor has idle resources. Thus, previous DVFS control schemes ([15, 24]) are inadequate in managing TM, as they do not account for the dependency between CPU clock frequency and the presence of idle resources. The processor is able to overclock active cores because the CPU's thermal design power (TDP) is the worst case power at maximum load. When a workload idles some resources of a core, there will be thermal and power headroom available. Since in practice applications do not fully utilize all execution units of a core, TM functions even if all cores are active. TM exploits this available headroom by increasing the operating frequency of the processor [12, 1]. Increasing the CPU's frequency will then speed up the applications running on the CPU, potentially leading to better performance. The frequency boost enabled by TM is already significant on current processors. In one of our evaluation machines, TM can boost the CPU from a nominal frequency of 2.50GHz to 3.60GHz, a 44% gain that could lead to a 44% performance increase.

All current implementations of TM are controlled via a feedback controller implemented in firmware. The firmware controller monitors the CPU to determine the frequency boost that should be applied. The final operating frequency of the processor is determined by several factors that include CPU power consumption, the number of idle cores, and CPU temperature. The only way for software to manage TM for all modern x86 server CPUs is to enable/disable its use. One way to do so is to statically enable/disable TM from the BIOS. However, this approach is extremely coarse-grained, as turning on TM from a disabled state would require a machine reboot. Instead, we use an online approach by utilizing the operating system to dynamically enable or disable TM through existing ACPI power management functions. In our work, we use the Linux kernel's built-in CPU frequency scaling support to set the maximum frequency allowed for the CPU. By setting the maximum CPU frequency at the nominal frequency, or one step lower, the CPU will be prevented from entering the TM regime. However, the CPU frequency scaling facility *cannot* set the maximum allowable TM frequency. For instance, if a CPU can scale from a nominal 2.50GHz to 3.60GHz, setting the maximum CPU frequency to 3.20GHz will not prevent the firmware TM controller from setting the CPU frequency to 3.60GHz. However, for the same processor, setting the maximum frequency at 2.50GHz will prevent TM from activating. We will show later in the paper that adjusting this one simple switch can have profound performance, power, and cost implications.

Both major x86 server CPU vendors, AMD and Intel, have implemented TM in their latest generation processors. AMD's version is named Turbo CORE, while Intel's version is called TurboBoost. For this paper, we use "TM" as an umbrella term for both versions. At a high level, both implementations are quite similar. Both overclock the processor beyond its base frequency when there is thermal and power headroom, and both have a maximum TM boost frequency that is prescribed by the number of active cores. At the hardware level, there are significant differences between AMD's and Intel's TM controller. AMD's implementation reacts to current and temperature sensors on the CPU die to adjust the frequency boost. Intel's version predicts the power utilization based on the instruction mix and uses that information to determine the frequency boost. Intel's TM is also subject to thermal throttling, but unlike AMD's version, it is not the primary method to control frequency gain. Theoretically, Intel's implementation has a more repeatable TM boost, while AMD's implementation is more sensitive to external factors such as ambient temperature. However, in our experiments, we have observed that the frequency boost from AMD's TM does not experience wide variance between experiment runs. This is due to our test machines being situated in a climate controlled environment that would be found in a datacenter. Finally, we also observe that TM appears to be frequency limited and not power limited.

Another difference between Intel's and AMD's TM controllers is that Intel's version attempts to address energy efficiency. It does this by applying differing amounts of frequency boost depending on the "memory scalability" of the workload and a user configurable parameter for the aggressiveness of TM (*MSR_IA32_ENERGY_PERF_BIAS*) [28]. In practice, we find that this setting is hard to use effectively. We have observed that setting the parameter to "energy efficient" can still lead to degradation in energy efficiency metrics. Therefore, in our study, we leave this parameter at the setting for "highest performance" to avoid interference between the firmware controller and our software controller for TM. Intel also exposes a MSR that allows software to control the maximum achievable TM boost (*MSR_TURBO_RATIO_LIMIT*); however, we do not evaluate it because there is no equivalent MSR for the AMD platform.

A software TM controller is motivated further by the fact that improving the firmware TM controller does not guarantee improvements in energy efficiency. This is because it is infeasible to pre-program firmware to be aware of the impact of TM on all workloads and workload combinations. In addition, the firmware does not have information on the other aspects of the system that affect cost and power (e.g. cooling system, component costs, etc.), nor does it know which metric to optimize for. Thus, an entire system needs to be assembled and operated before it can be determined whether TM will improve efficiency metrics.

3 TurboMode Analysis

We now examine the highly variable impact of TM on performance, energy, and cost efficiency for a variety of workloads on a wide-range of hardware platforms.

3.1 Methodology

Hardware We selected several very different hardware platforms that represent a wide selection of server parameters that are seen in practice [18, 6] in order to reach conclusions that are not specific to one hardware configuration. We use real hardware for our evaluation in order to accurately capture the complex behavior of TM.

Name	Base Freq.	Max TM boost %	HW Cost	CPU TDP
SBServer (Intel)	3.20GHz	19%	$2000	130W
ILServer (AMD)	1.90GHz	59%	$1700	115W
SBMobile (Intel)	2.50GHz	44%	$1300	45W
IBServer (Intel)	3.30GHz	12%	$1500	69W
HServer (Intel)	3.10GHz	13%	$1500	80W

Table 1. Machine configuration summary.

1. Sandy Bridge Server (*SBServer*) has a Sandy Bridge EP (Core i7 3930k) processor with 6 cores that share a 12MB L3 cache. The granularity of TM control is chip-wide. This configuration is representative of a low-end Sandy Bridge server.
2. Interlagos Server (*ILServer*). This system contains an AMD Opteron 6272 processor, mimicking a low-end Interlagos based server. The processor can logically be viewed as a dual socket part, since it is composed of two dies that share a package. Each die has 4 modules, where each module has 2 integer cores that share a front-end. All modules on the same die share 8MB of L3 cache. We lower the base frequency from 2.1GHz to 1.9GHz in order to be able to dynamically control TM. Unlike Sandy Bridge and Ivy Bridge CPUs, the Interlagos architecture allows for control of TM at the module level.
3. Sandy Bridge Laptop (*SBMobile*). The laptop contains an Intel Core i7 2860QM processor with 4 cores that share a 8MB L3 cache. We use a mobile part in order to understand the implications of TM on a more energy-proportional system. In addition, the larger frequency boost of this CPU gives insight into future CPUs with a larger TM frequency swing.
4. Ivy Bridge Server (*IBServer*) and Haswell Server (*HServer*). We also performed a partial evaluation on Ivy Bridge (Xeon E3-1230v2) and Haswell (Xeon E3-1220v3) CPUs. Due to space constraints we do not present a full evaluation of TM on these platforms; however, we will demonstrate that our dynamic controller works well on these platforms.

All machines use Ubuntu 12.04 LTS as the operating system with Linux 3.2.0-26. We summarize the machine configurations used in this study in Table 1. For our experiments, we disable HyperThreading on both Intel CPUs and the second core in each Interlagos module to avoid performance counter interference.

We monitor the frequency of the CPU throughout our experiments using *turbostat* to verify that TM is functioning as expected. We measure the total system power by interposing a power meter between the power supply and the power plug. This enables us to determine the impact of TM on total system power, which is important to DC operators. In addition, we also measure the power of the system when idle in order to approximate the system active power for a completely energy-proportional system (e.g. $ActivePower \approx TotalPower - IdlePower$). We apply the energy-proportional approximation to *SBServer*, *ILServer*, and *SBMobile* and denote the resulting theoretical configurations as *SBServer*†, *ILServer*†, and *SBMobile*†. We make energy-proportional approximations because scenarios that are not energy-proportional heavily favor the use of TM. This is because high system idle power masks the power increase from TM. The idle power of *SBServer*, *ILServer*, and *SBMobile* is 83W, 60W, and 25W, respectively. Utilizing a single core is a non-energy-proportional scenario because it adds approximately 50W, 30W, and 20W to the power consumed by *SBServer*, *ILServer*, and *SBMobile*, respectively. *ActivePower* is underestimated due to the inclusion of CPU active idle power in *IdlePower*. Since TM will always increase $TotalPower$, the ratio $ActivePower_{TM}/ActivePower_{baseline}$ will be overestimated. Thus QPS/W, $QPS/\$$, ED, and ED^2 results for the energy-proportional approximation conservatively underestimate the benefits of TM.

Workloads We evaluate the performance, power, and cost implications for a variety of applications on each of the different hardware configurations. A wide ranging set of benchmarks were chosen in order to capture the behavior of TM for compute-driven datacenter applications. For our study, we focus mainly on the CPU and the memory subsystem and ignore interactions with the disk, network, and I/O subsystems. We use SPECCPU2006 applications (SPECCPU) as single-threaded workloads and mixes of SPECCPU applications to emulate multi-programmed workloads. SPECCPU application mixes are chosen similar to the process used in [29]. Applications are categorized based on cache sensitivity, and a mix is created by randomly choosing categories and then randomly choosing an application for each category. The number of applications in a mix is 4, 6, 8, 4, and 4 for *SBMobile*, *SBServer*, *ILServer*, *IBServer*, and *HServer* respectively. We generate 35 random application mixes for each hardware configuration. We use PARSEC[3] as a multi-threaded benchmark application. We evaluate all PARSEC applications for all thread counts from 1 to the number of cores on each machine.

For enterprise class workloads, we use SPECpower_ssj2008 that is run with enough threads to utilize the entire machine. We also construct websearch, a representative latency-critical web application, by using the query serving component of Nutch [19, 9] with an in-memory index. The index for websearch is generated by indexing a 250GB dump of 14.4 million pages from 2005 Wikipedia, which generates a 37GB index. We then use a 4GB portion of that index, which captures approximately 27GB and 1.6 million pages of the original data. Websearch is driven by simulated search queries that are generated by a model that uses an exponential distribution for query arrival times and a query term model that is representative of web search queries seen at search engines [30]. Websearch must also satisfy a Quality of Service (QoS) constraint that 95% of the queries must be processed in less than 500ms [22]. Nutch is configured such that at 100% utilization it will use all available CPU cores.

Single SPECCPU workloads and application mixes from SPECCPU can be considered to be either throughput-oriented or latency-oriented. These benchmarks represent throughput-oriented applications if the number of benchmarks executed per second is the primary metric. Similarly, if the metric of interest is how long each benchmark takes to execute, then SPECCPU is an analogue for latency-oriented applications. SPECpower_ssj2008 is a throughput-oriented benchmark that scores the system by how many queries per second it can handle. Thus, we do not compute latency metrics for SPECpower_ssj2008. Finally, while websearch is both sensitive to throughput and latency, it can be optimized for either one. To optimize websearch for throughput, one would measure the maximum sustained QPS that can be achieved without violating QoS. To optimize for latency, one would reduce the 95%-ile latencies for a fixed QPS rate. In addition, since SPECpower_ssj2008 and websearch are request-

	Workload	$\frac{QPS}{W}$	$\frac{QPS}{\$}$	ED	ED^2
SBServer	**SPECCPU**	On	On	On	On
	SPECCPU mixes	Off	?	Off	?
	PARSEC	Off	?	Off	?
	SPECpower_ssj2008	Off	On	N/A	N/A
	websearch	Off	On	On	On
SBServer†	**SPECCPU**	Off	On	On	On
	SPECCPU mixes	Off	Off	Off	?
	PARSEC	Off	Off	Off	?
	SPECpower_ssj2008	Off	On	N/A	N/A
	websearch	Off	On	On	On
ILServer	**SPECCPU**	On	On	On	On
	SPECCPU mixes	On	On	On	On
	PARSEC	On	On	On	On
	SPECpower_ssj2008	Off	On	N/A	N/A
	websearch	On	On	On	On
ILServer†	**SPECCPU**	Off	On	On	On
	SPECCPU mixes	Off	Off	?	On
	PARSEC	Off	On	?	On
	SPECpower_ssj2008	Off	On	N/A	N/A
	websearch	On	On	On	On
SBMobile	**SPECCPU**	Off	On	On	On
	SPECCPU mixes	Off	?	?	?
	PARSEC	Off	On	?	?
	SPECpower_ssj2008	Off	On	N/A	N/A
	websearch	Off	On	On	On
SBMobile†	**SPECCPU**	Off	On	Off	?

Table 2. Optimal TM settings for each metric/workload class/hardware configuration. On/Off indicates a static optimal TM setting of On/Off. ? indicates that there is no optimal static TM setting. The results for SPECpower˙ssj2008 are reported at 100% QPS rate achieved with TM on. The QPS/W and $QPS/\$$ for websearch are reported at 100% QPS rate with TM on, and ED and ED^2 metrics are reported at 100% QPS rate with TM off. $SBMobile^†$ only has SPECCPU listed because all other workloads are energy-proportional on $SBMobile$.

driven workloads, we sweep the input QPS rate from 10% to 100% of the maximum QPS rate on each server configuration. This is done because datacenters are not always utilized at 100%.

3.2 TurboMode for Different Metrics

There are many metrics that are useful when dealing with performance, power, and cost. All of the metrics that we examine are legitimate metrics when applied to different workload scenarios. For instance, under periods of high load, the metric of interest will be performance, but under normal load conditions, $QPS/\$$ is more important. Even the same workload can have different metrics of interest. Take websearch for example. If one has a fixed 95%-ile latency target, the metric of interest is QPS/W or $QPS/\$$. On the other hand, if one wants to optimize for a lower 95%-ile target, then ED and ED^2 are more relevant. We list the metrics that we analyze below:

1. *Performance* measures the raw performance of a workload and is useful when application throughput and latency is critical. This metric will obviously be improved by the use of TM, and we include other metrics to see the true cost of using TM.

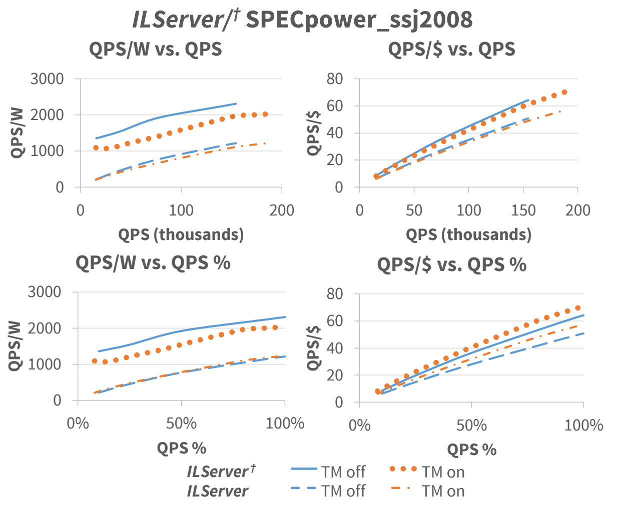

Figure 1. Impact of TM on QPS/W and $QPS/\$$ for SPECpower˙ssj2008 on $ILServer$ and $ILServer^†$. The graphs in the upper half are for absolute QPS numbers, while the graphs in the lower half are for QPS numbers relative to the maximum QPS rate that can be supported for each setting of TM. Higher numbers are better for QPS/W and $QPS/\$$.

2. *Power* measures the total system power consumed while executing the workloads. This metric is for situations where minimal power is needed, even at the expense of performance.
3. *Energy Delay Product and Energy Delay Squared Product*: Energy Delay Product is calculated as the product of the energy it takes to complete an operation with the time it takes to complete the operation, and Energy Delay Squared Product is found in a similar way. These metrics are commonly used as a way of finding the optimal tradeoff between performance and energy. This tradeoff is important for latency sensitive applications in an energy constrained environment. [4]
4. *Queries Per Second Over Power (QPS/W)*: The efficiency of throughput-oriented batch workloads running on power constrained systems are best measured by this metric. QPS/W is inversely proportional to the amount of energy needed to execute one job; thus, this metric is the same as energy efficiency which is important in a large-scale DC [2].
5. $QPS/\$$ is a direct measure of cost efficiency. The fiscally conscious system operator will find this metric to be most relevant for throughput-oriented workloads. We use a standard 3 year depreciation cycle, and we assume an electricity cost of \$0.15/kWHr. We use the cost model found in [25, 26] in our calculations. This metric is commonly used to measure the cost-efficiency of large-scale DCs [2].

We measured the changes in performance and power caused by the use of TM and calculated the above metrics the workloads on various hardware platforms. In Figure 2, we show the impact of TM on the various metrics for a subset of applications. For completeness, we show TM's impact on all metrics in Table 2.

Using TM will always have a positive impact on performance, as speeding up the clock frequency of the processor should not degrade application performance. Similarly, TM will certainly increase power consumption. As a case study of the optimal TM setting for QPS/W and $QPS/\$$ for a throughput driven workload, we examine SPECpower_ssj2008's behavior on *ILServer* and *ILServer*† as illustrated in Figure 1. SPECpower_ssj2008 is amenable to the use of TM, as enabling TM increases the maximum QPS rate by 25%. To optimize for both relative and absolute QPS/W, TM should be kept off. This is because QPS will typically scale only as much as frequency, but the power consumed scales faster than frequency. This is due to the fact that $P \propto V^2 f$ and that higher frequencies require increased supply voltage [5]. Figure 1 shows this effect when idle power is removed (*ILServer*†), as QPS/W when TM is on always falls below the QPS/W for when TM is off.

On the other hand, if TM can produce a performance gain, then it should almost always be enabled to optimize for $QPS/\$$. This effect can be explained in a straightforward manner by reviewing the hardware and energy costs associated with the TCO. For *ILServer*, the cost of the hardware is about the same as the cost of electricity and infrastructure for power delivery and cooling. Since the use of TM does not increase power consumption by an inordinately large factor, it is always more advantageous to enable TM if it enables more performance out of the same piece of hardware. In Figure 1, the $QPS/\$$ for absolute QPS numbers seems to show that TM degrades this metric. However, in reality, if each machine can handle more work, then fewer machines need to be provisioned. Thus, TM improves $QPS/\$$ not for an individual machine, but rather for the entire datacenter, as reflected in the graph for $QPS/\$$ vs. relative QPS.

Table 2 shows that our observations for QPS/W and $QPS/\$$ based on SPECpower_ssj2008 running on *ILServer* also generalizes to many other workloads and platforms. There are some exceptions to the rule. Using TM can actually improve QPS/W in situations that are not energy-proportional, such as when running only a single instance of a SPECCPU application. When the idle power is removed (e.g., *SBServer*† and *ILServer*†), then the optimal TM setting for QPS/W for the single SPECCPU application workload is to turn TM off. In addition, there are times when TM should be disabled to optimize for $QPS/\$$, such as when TM does not appreciably increase the QPS rate.

TM has a variable effect on applications for ED and ED^2 in energy-proportional scenarios. In the single application workload scenario, ED and ED^2 always benefit from the use of TM; but tradeoffs emerge when the energy-proportional approximation is applied. When multiple cores are being utilized, there is no optimal setting for ED and ED^2 across all workloads. As we will discuss in Section 3.3, the optimal TM setting for ED and ED^2 depends on whether the workload exhibits interference for shared resources. In Section 4, we will combine our observations from this section to build an effective system to select the optimal TM setting for ED, ED^2, and $QPS/\$$.

3.3 TurboMode for Different Workloads

As seen in Table 2, the optimal TM setting depends on the workload. We now focus on ED and ED^2 as a case study for how TM can affect those metrics depending on workload. In Figure 2, we show a subset of workloads that exhibit large swings for ED and ED^2 on the different machine configurations.

We study the ED and ED^2 metrics for websearch on *SBMobile*; the conclusions we arrive at also apply to *SBServer*†. *ILServer*† can mostly be explained in the same way as well, except for TM's large positive effect on QPS/W and $QPS/\$$. We sweep the QPS rate for websearch from 10% to 100% of the maximum QPS that it can support with both TM on and TM off and plot the effect of TM on various metrics in Figure 3. 95%-ile latency is improved by 30%-35% across the same QPS range that websearch can support if TM is not enabled. The reason why the 95%-ile latency is reduced by a factor larger than the QPS gain can be explained by queuing theory. A significant component of 95%-ile latency is due to a query being held up by earlier queries; thus, reducing the processing time for an individual query will have an amplified effect on reducing 95%-ile latency. Due to this effect, the ED and ED^2 metrics, which are based on 95%-ile latency, are always improved significantly through the use of TM. However, on the relative QPS graph, it may appear that enabling TM degrades ED and ED^2 when the utilization of the machine goes over 80%. This is misleading, because running websearch at that same *absolute* QPS rate with TM disabled will violate the QoS constraint. Thus, a direct comparison is invalid beyond that point. TM also provides a 24% boost to the maximum QPS, and its behavior for QPS/W and $QPS/\$$ mirrors that of SPECpower_ssj2008 and are explained in the same way. On *ILServer*, TM actually increases the maximum QPS by 180%, due to *ILServer* struggling to meet the QoS constraint when running with TM off. Because of the queuing delay effect, increasing the clock frequency leads to a large increase in maximum QPS, QPS/W, and $QPS/\$$. If the QoS constraint was relaxed to 1 second, then the large increase in maximum QPS vanishes and TM no longer benefits QPS/W or $QPS/\$$ as much.

We now turn our attention to the SPECCPU application mixes. Mix 1 is composed of lbm, xalancbmk, lbm, and milc. Mix 2 is composed of namd, hmmer, gobmk, and bzip2. These mixes were created by a random process as previously described. On *SBMobile*, using TM on Mix 1 increases the CPU frequency by 20%, but only produces an average performance increase of 0.2% with a 29% increase in power. On the other hand, enabling TM for Mix 2 boosts the core frequency by 13% for a 13% increase in performance and a 22% increase in power. The reason for the disparate behavior is due to the composition of the mixes. Mix 1 is composed of applications that either fit in the last level cache, or exhibit cache thrashing behavior, while Mix 2 is composed of applications that are classified as cache insensitive or cache friendly [29]. Mix 1 incurs interference in the LLC and is a completely memory-bound workload that cannot benefit from TM. Enabling TM increases the CPU's power consumption, even if that increase does not translate into a performance gain. For workloads not dominated by memory accesses (e.g., Mix 2), TM will speed up the workload with modest power increases, resulting in a win for $QPS/\$$, ED, and ED^2. However, interference can occur in other shared resources external to the CPU, such as disk, network, and I/O. While dealing with interference for those resources is beyond the scope of this paper, we expect that workloads that interfere in those resources will also experience large efficiency degradations when TM is enabled. TM has many benefits but also runs the risk of increasing energy consumption without realizing gains in performance. This strongly motivates the construction of an intelligent controller that determines when TM should be enabled. In Section 4, the observation that significant memory interference causes TM to burn extra energy will be leveraged to build an intelligent TM controller.

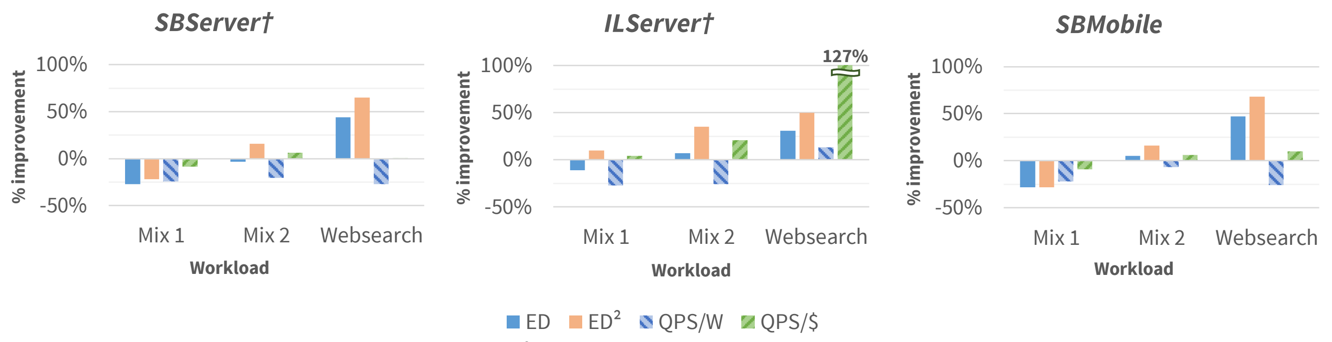

Figure 2. TM impact on ED, ED^2, QPS/W, and $QPS/\$$ metrics for a subset of workloads.

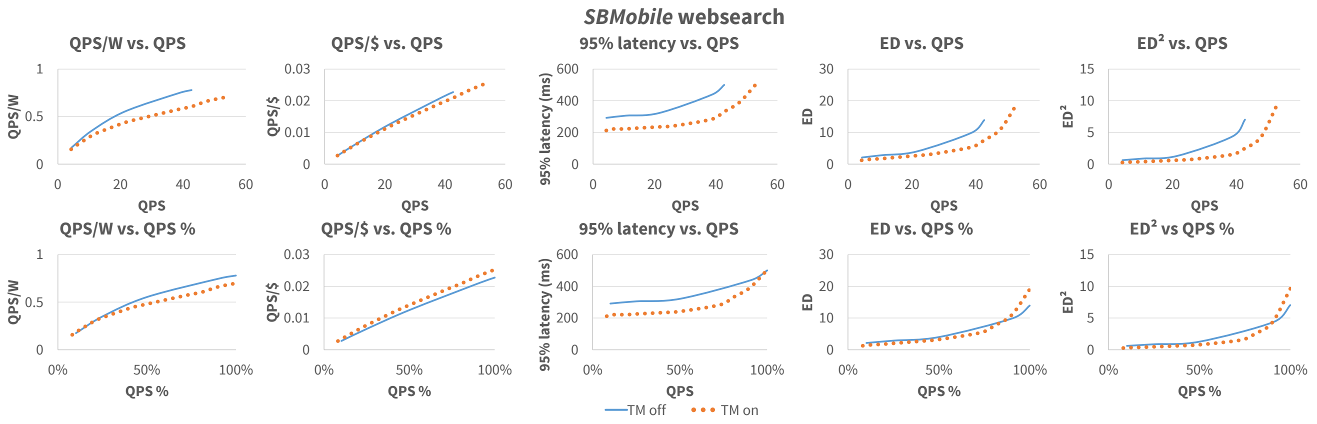

Figure 3. TM impact on QPS/W, and $QPS/\$$, 95%-ile latency, ED, and ED^2 metrics for websearch on $SBMobile$. The 95%-ile latency, ED, and ED^2 numbers are absolute, meaning that lower is better.

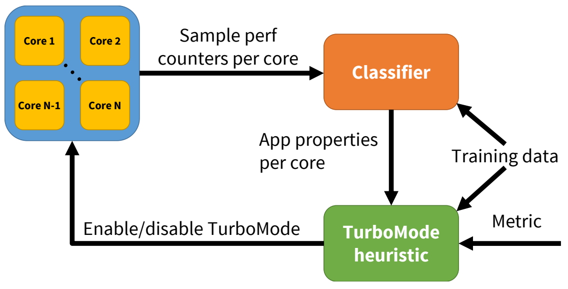

Figure 4. Block diagram of online component *autoturbo*.

4 Dynamic TurboMode Management

4.1 Overview

We have demonstrated earlier that the use of TM can significantly increase efficiency for some workloads while also causing major degradations for others. We want the best of both worlds, where the system intelligently uses TM only in situations that produce an efficiency improvement. To achieve this goal, we implement a software controller for TM named *autoturbo*.

The block diagram for *autoturbo* is shown in Figure 4. It runs as a periodic background daemon in order to minimize resource usage. *autoturbo* starts by collecting system wide per-core performance counters for a configurable fixed time period, set to 5 seconds in our experiments. After it has collected counters, it uses a machine learning classifier to predict application characteristics for the workload on each active core. These results are then used by heuristics to determine whether TM should be enabled or disabled to optimize for a certain metric. The heuristic can be as simple as enabling TM if the classifier predicts that the workload benefits from the use of TM for a certain metric, to a more complicated heuristic that is used when several workloads are running at the same time. The TM setting from the heuristic is applied, and the process repeats from the beginning.

Since TM impacts each metric differently, we build a classifier and heuristic for each metric of interest. Every metric requires a separate classifier since TM will affect each metric differently. The process to build the classifier and heuristic can be automated and is described in Section 4.2. As *autoturbo* is an open-loop controller, having accurate models is very important.

4.2 Offline Classifier Training

The online component of *autoturbo* uses models that are generated offline for the various metrics. The modeling parameters only need to be generated once per server type. The offline training can be done by the system integrator, who then provides this information to all their customers for a plug-n-play solution that works right out of the box. The offline training can also be done by the DC operator for a more customized solution, as they can use their own metrics and workloads for the training phase. Another advantage of the DC operator performing offline training is that they can provide a more accurate TCO model for their infrastructure and cooling costs.

Classifying Individual Workloads Creating a model that predicts the impact of TM on individual workloads for a specific machine is fairly straightforward. We first use SPECCPU and two other memory intensive benchmarks, *stream* and *lmbench-lat_mem*, as the training applications. *stream* and *lmbench-lat_mem* are selected because they exhibit streaming and cache

Machine	Accuracy	Metric	Classifier	Features
SBServer†	100.0%	ED^2	Naive Bayes	% cycles w/ mem. requests
SBMobile†	96.8%			

Table 3. Classifier building results for metrics of interest on various hardware configurations.

thrashing behaviors. These applications are then run on the machine that we wish to model, and performance, power, and various performance counters are measured when TM is on and when it is off. We then calculate the effect TM has on the power, performance, and cost metrics of interest.

As seen earlier, an application tends to benefit from TM if it is not memory bound. Therefore, to predict if an individual workload benefits from TM, we need only predict if it is memory bound. We build such a predictor by applying machine learning techniques on a wide array of performance counters that capture memory boundness, such as IPC, L3 loads/misses, TLB misses, memory requests, etc. We first use feature selection to find the performance counters that best capture memory boundness, and to reject performance counters that have poor signal to noise ratio or that simply don't function correctly. Then, we train a model based on those features to pick the proper parameters for the model. We fit a model to each metric and machine, due to cost/hardware variation between different configurations. We use the Orange data mining package [7] for this because manually tuning the model is both time consuming and error prone. The model is then used in an online classifier that predicts whether TM will benefit a metric. The results of the feature selection and the accuracy of the classifiers are shown in Table 3. We show the results for the ED^2 metric on *SBServer*† and *SBMobile*† only since these are the only configurations where the optimal setting for TM varies depending on application. All other metrics on other hardware configurations have a static optimal setting, e.g., TM should always be off to optimize for QPS/W on *SBServer*†. For the remaining hardware configurations and metric, we show that using a single performance counter that tracks the fraction of time there is a memory request outstanding provides excellent prediction power.

Classifying Application Mixes Predicting the optimal TM setting for a given metric on a mix of applications is more complicated. The naive approach would be to model the interactions between workloads running on every core in the system. However, this approach is unscalable for large core counts, as the number of parameters for the classifier would scale with the square of the number of cores. Training such a classifier would also require a prohibitively large training set of application mixes.

Instead, we use a heuristic approach to determine the optimal TM setting for application mixes. In Section 3.3 we observe that memory interference (L3 for *ILServer*, main memory for *SBServer* and *SBMobile*) causes TM to consume extra power without a corresponding performance gain. Thus, we developed a heuristic that dynamically detects excessive memory interference and disables TM. While we examine only memory interference for this work, extending the heuristic to detect other types of interference is straightforward. To aid in the detection of memory interference, we build classifiers using the same process previously discussed to predict if a workload is sensitive to memory interference (*Sensitive*) and if it causes memory interference (*Interfering*). This classification scheme was inspired by [17]. These two properties are independent from each other. For example, a workload that is memory latency bound but causes the prefetcher to trash the L3 cache is *Interfering*, but not *Sensitive*. To build each machine-specific classifier for *Interfering* and *Sensitive*, we run each SPECCPU application in tandem with the *stream* benchmark and measure the performance degradation for the SPECCPU application and for *stream*. Applications that degrade significantly when run in parallel with *stream* are marked as *Sensitive*, and applications that significantly slow down *stream* are marked as *Interfering*. The results of the application classification are shown in Table 5. We then build predictors using this data and show the results in Table 4. Misclassifications occur when applications with different properties have similar inputs to the classifier, which is caused by the use of indirect features (e.g., performance counters) to measure memory behavior. However, these cases are rare, as seen by the high classification accuracy.

The performance counters automatically selected by feature selection are intimately related to the architectures of the different CPUs. For *SBServer* and *SBMobile*, applications that are frequently stalled on memory (a high portion of cycles with a memory request pending and a high portion of cycles where the front end is stalled) are sensitive to memory interference. Similarly, applications that have intensive memory activity, as measured by a large fraction of cycles with pending memory requests, cause significant amounts of memory interference. For *ILServer*, since the L3 cache is exclusive of the L2, applications with high L2 MPKI are sensitive to memory interference. Similarly, applications with high L3 MPKI and memory access intensity cause memory interference. Since the best performance counters vary by processor architecture, it would have been difficult to manually pinpoint the precise performance counters to use even if the general theme of "performance counters related to memory" is known. Adding more performance counters to the classifiers causes the prediction accuracy to drop, indicating that all other performance counters have poor signal to noise ratios. In addition, measuring too many performance events would require sampling over a limited number of physical hardware event counter registers, degrading their accuracy.

The next step is to determine the maximum amount of interference before TM no longer provides a win. This process is controlled via static policy, which is determined by the maximum number of interfering workloads that can be co-located before the use of TM degrades a metric. To find the proper policy for each metric, we measure the power and performance increase caused by TM for different numbers of active cores. For example, TM increases power by 23% for a 4 core workload on *SBMobile* with a 12% frequency boost. Thus TM must significantly increase the performance of applications running on at least 2 cores in order to not degrade the ED^2 metric for *SBMobile*. This heuristic applies TM conservatively, as it pessimistically assumes that *Sensitive* workloads will not benefit at all from TM in the presence of memory interference. To determine the number of applications that will be degraded, the heuristic looks at the application properties of the workload on each core in a NUMA node. If there is more than one *Interfering* application, then the number of degraded applications will be the number of *Sensitive* applications. If there is one *Interfering* application, the number of degraded applications is the number of *Sensitive* applications

	Machine	Classifier	Counters used	Accuracy
Sensitive	**SBServer**	Logistic Regression	% cycles with outstanding memory requests % cycles front end is idle	83.9%
	ILServer	Naive Bayes	L2-load-misses / instruction	93.5%
	SBMobile	Naive Bayes	% cycles with outstanding memory requests % cycles front end is idle	87.1%
Interfering	**SBServer**	Logistic Regression	% cycles with outstanding memory requests	87.1%
	ILServer	Naive Bayes	L3-misses # requests to memory / instruction	93.5%
	SBMobile	Logistic Regression	% cycles with outstanding memory requests	87.1%

Table 4. Classifier building and validation results for *Sensitive* and *Interfering* on various hardware configurations.

Property	SPECCPU apps with property
Sensitive	mcf, milc, cactusADM, soplex, GemsFDTD, libquantum, lbm, omnetpp, astar, sphinx3, xalancbmk
Interfering	bwaves, milc, leslie3d, soplex, GemsFDTD, libquantum, lbm

Table 5. SPECCPU application properties on *SBServer* and *SBMobile*. *ILServer* is the same as *SBServer* except that leslie3d is also in *Sensitive*.

on cores other than the one hosting the *Interfering* workload, as a workload cannot interfere with itself.

4.3 Online *autoturbo* Operation

We implement the online component of *autoturbo* as a Python program to have convenient access to the Orange machine learning library. However, this portion could also be implemented in the CPU firmware to offload the computational load of the classifier. This would require the CPU firmware to expose an interface to input training data and to declare the metric that *autoturbo* should be optimizing for.

The pseudocode for *autoturbo* is given below:

```
loadModels(systemType, metric)
while (true) {
 perfCounters = sampleCounters()
 if (numCoresActive() == 1)
  setTM(singleAppModel(perfCounters))
 else if (numCoresActive() > 1)
  setTM(multipleAppModel(perfCounters))
}
```

First, the system operator sets the system type and the metric that *autoturbo* should be optimizing for. This loads the appropriate models that were generated by the training phase as described in Section 4.2. Next, *autoturbo* samples the appropriate performance counters available on the system for 5 seconds. From this data, it can determine the number of cores that were active in the sampling period. If there were no active cores, then there is nothing to optimize for. If there is one active core, then the single application classifier is used to predict the optimal TM setting for that application. If there is more than one active core, then the multiple application classifier is used to predict if there is significant workload interference that would negate the benefits of TM. The predicted optimal TM setting is then set via *cpufreq-set*, and the process repeats.

The choice of a 5 second sampling period was done after considering the tradeoff between increased reactiveness of *autoturbo* at shorter sampling periods and lower power consumption of *autoturbo* at longer sampling periods. Running the predictive classifiers, like any other application, consumes CPU resources and power. When there is a workload present, each sampling period will invoke at least one classifier, and so increased sampling increases the overhead of *autoturbo*. However, if the sampling period is too long, then changes in the workload composition or changes in the workload phase will not be quickly detected. We chose 5 seconds as a time period that is long enough to not increase CPU utilization or power by an appreciable amount, while being short enough such that the typical workload will not change drastically within most sampling windows.

As *autoturbo* runs as a background daemon, it can be easily deployed across the entire datacenter by standard application deployment procedures. *autoturbo* can be easily extended with remote management features, such as providing real-time reporting on when TM is enabled and the frequency boost of TM. The architecture of *autoturbo* also allows for real time switching of metrics to be optimized for, which is useful for large virtualized datacenters where workloads with different optimization targets can shift between physical machines. In addition, *autoturbo* can also be extended to manually disable TM on a node, which is invaluable in case of a thermal or power emergency in the datacenter. While remote management is currently not available in *autoturbo*, it is planned as future work.

While TM can be controlled at a module granularity, in practice we find that controlling TM at such a fine granularity on the Interlagos platform is not worthwhile. This is due to the Interlagos chip not being energy-proportional. For example, if two cores (on two separate dies) are active with TM on, the power draw is 53W above idle power. If TM is disabled on one core, then the power draw only drops by 3W. However, disabling TM on both cores drops the power by an additional 20W. Thus, *autoturbo* controls TM on *ILServer* at a chip-level granularity.

5 Evaluation

We evaluate *autoturbo* for all workloads on all hardware platforms across all metrics by running *autoturbo* on top of the workload. We measure the power consumption of the machine and the performance of the workload to determine how effectively *autoturbo* can use TM to optimize for a given metric. We show the results of using *autoturbo* on a subset of workloads/metrics/hardware platforms in Figure 5. Each line shows the metric of interest for the workload, normalized to the baseline case of when TM is always off. We compare *autoturbo* against

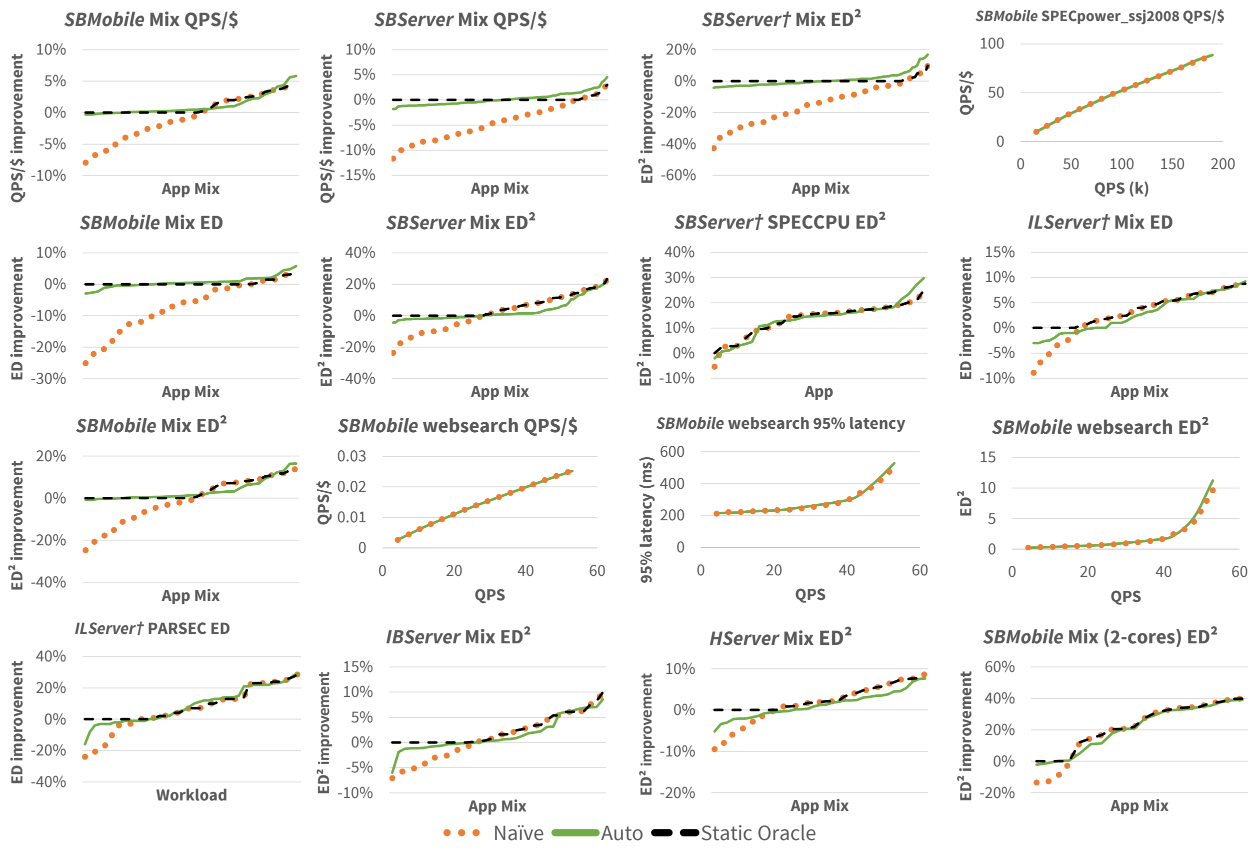

Figure 5. *autoturbo*'s **effect on various application/workload/hardware configurations.**

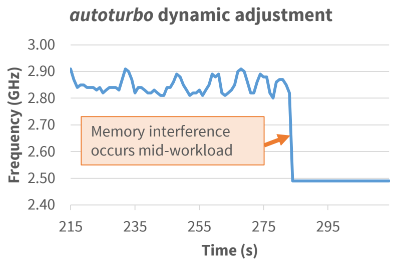

Figure 6. *autoturbo* **detecting a phase change and adjusting TM for a SPECCPU mix made of gromacs, bwaves, gromacs, and GemsFDTD.** *autoturbo* **is optimizing for** ED^2 **on** *SBMobile***.**

two other management strategies for TM. The Naive manager naively uses TM all the time. The Static Oracle has prior knowledge of which setting of TM should be applied for the *entire duration* of the workload to optimize for a given metric. The Static Oracle required several profiling runs and as such is not practical for scheduling unknown workloads. For each line, the workloads (X axis) are sorted by the improvement achieved in ascending order. The behavior of *autoturbo* on the single SPECCPU workload running on *SBMobile*† is similar to *SBServer*†. In addition, *autoturbo* also has the same behavior for websearch and SPECpower_ssj2008 for all metrics measured on all machines. We compare the performance of *autoturbo* on websearch and SPECpower_ssj2008 against naively enabling TM, as that is the optimal setting for those workloads. Even though *autoturbo* was trained on SPECCPU applications, we can still use them as test applications because *autoturbo* was trained on performance counters for the *entire run*, while the classifier uses counters obtained from 5 second windows as its inputs.

SBMobile *autoturbo* is able to use TM to intelligently improve $QPS/\$$ and ED^2 for the workload consisting of SPECCPU application mixes. We show results for when all cores are used and when half the cores are used. When all cores are used, the naive approach can improve the $QPS/\$$ ratio and ED^2 metrics for approximately 40% of the mixes compared to the baseline. However, the other 60% of the time it will cause degradations for those metrics. This degradation can be severe, up to 8% for $QPS/\$$ and 25% for ED^2. These degradations are due to applications within the mix interfering with each other, causing the naive use of TM to consume extra power to provide a frequency boost that does not translate into improved performance. *autoturbo*'s heuristic to avoid the use of TM in such interference scenarios is clearly effective, as *autoturbo* virtually eliminates all cases of metric degradation. However, the heuristic trades off a low rate of false positives for a low rate of false negatives. *autoturbo's* heuristic is conservative in managing interference, as it preemptively disables TM in the face of predicted interference even if that interference turns out to be mild. This effect is shown by *autoturbo* failing to match the same level of metric improvement as the Static Oracle. One avenue of future work is to build a regression model that predicts the amount of interference and to only disable TM if that interference is above a threshold. Interestingly, there are also cases where *autoturbo* outperforms the Static Oracle. This occurs in situations where the SPECCPU application has phases in its execution that vary between CPU-intensive and memory-intensive. The Static Oracle cannot take advantage of this dynamism, since it chooses a TM setting for the entire workload duration. On the other hand, *autoturbo* can detect and account for these phase changes, as seen in Figure 6. When half the cores are used, we see larger gains in ED^2 because TM provides a larger frequency boost. *autoturbo* functions properly in this case, and is able to enable TM when ED^2 is improved while disabling it for mixes with heavy memory interference.

For SPECpower_ssj2008, not only is *autoturbo* able to properly apply TM, but its CPU footprint is small enough such that it does not interfere with it. Even as QPS is swept from low to high, *autoturbo* does not have a detrimental effect on the efficiency of SPECpower_specssj2008, as the $QPS/\$$ for *autoturbo* closely tracks the behavior of naive TM. For websearch, the use of *autoturbo* generally does not degrade performance or efficiency metrics compared to the naive use of TM. The only exception is when websearch is run at 100% QPS and *autoturbo* slightly degrades the 95%-ile latency and the associated ED^2 metric. When *autoturbo* wakes up to classify the applications, it will cause the OS to switch out a websearch thread, degrading the response latency. If the classifier component of *autoturbo* is integrated into firmware, then this performance penalty will disappear. *autoturbo* is also able to properly optimize for the ED^2 metric for the single application SPECCPU workload for *SBMobile*†, disabling TM for the one workload (milc) that does not benefit from TM.

While not shown, we observed that *autoturbo* is able to improve $QPS/\$$ and ED^2 for PARSEC as well.

SBServer *autoturbo* is also able to optimize the SPECCPU application mixes for $QPS/\$$ and ED^2 for *SBServer* and ED^2 for *SBServer*†. While the degradation of TM for $QPS/\$$ can be -12%, *autoturbo* is able to select the proper TM setting to ensure that it only degrades $QPS/\$$ by a maximum of -3%. Likewise, *autoturbo* is also able to disable TM in cases where workload interference would degrade ED^2 significantly. *autoturbo* is quite successful at preventing significant degradation of metrics, as it has strictly better worse case performance compared to the Naive TM manager. However, like *SBMobile*, the interference heuristic in TM is overly conservative, causing *autoturbo* to miss some opportunities for improving ED^2. For ED^2 on *SBServer*†, where there is no idle power to buffer the ED^2 of workloads with interference, the conservative heuristic in *autoturbo* pays off by being strictly better than the Naive use of TM to improve ED^2. *autoturbo* does have a few minor false positives that degrade ED^2 by at most -7%. These false positives arise when *autoturbo* misclassifies applications as not being *Sensitive* or *Interfering* when they actually are. *autoturbo* also is able to successfully optimize the single application SPECCPU workload running *SBServer*† for ED^2. It successfully tracks the performance of the Static Oracle, demonstrating that the theoretical accuracy of the single workload ED^2 classifier is realized in a practical setting. While not shown, *autoturbo* successfully optimizes PARSEC for ED^2 on *SBServer*† as well. Like *SBMobile*, *autoturbo* also correctly turns on TM for SPECpower_ssj2008 and websearch to take advantage of TM improving $QPS/\$$, ED, and ED^2 while exhibiting negligible interference with those workloads.

ILServer For the SPECCPU application mix workload, we use *autoturbo* to optimize the ED metric on *ILServer*†, since all other metrics on *ILServer*† and all metrics on *ILServer* have a statically optimal TM setting. This is because an 8 application SPECCPU mix fails to saturate memory bandwidth, a result of the low clock frequency and the low IPC on the *ILServer*. Thus, running many interfering applications together still leads to a small but non-negligible speedup for SPECCPU mixes for a relatively small increase in power. Like *SBServer*†, *autoturbo* successfully eliminates major degradations in ED but also misses some opportunities to improve ED. For PARSEC on *ILServer*†, *autoturbo* is able to eliminate most of the degradations to ED while keeping the benefits of TM. The only case where it failed to do so was when *autoturbo* activated TM for *streamcluster* at low thread counts, because it could not predict that the performance gain was too small to be worthwhile. *autoturbo* is also able to successfully optimize SPECpower_ssj2008 for $QPS/\$$ and websearch for $QPS/\$$, ED, and ED^2 by turning on TM consistently and by not interfering with those workloads.

IBServer and HServer We observe the generality of *autoturbo* by demonstrating that it functions across several generations of architectures from the same CPU vendor. A surprising result for HServer is that while Haswell has independent frequency domains for each core, once TM is activated, all cores will run at the same frequency, thus *autoturbo* is still useful. Due to space constraints, we omit the results from the entire metric and workload evaluation and only show results for *autoturbo* optimizing the ED^2 metric for SPECCPU application mixes. Just like with the other hardware configurations, *autoturbo* intelligently enables TM to optimize for ED^2. However, *autoturbo* will occasionally disable the use of TM even when it is optimal to use TM. Again, this is because of the conservative heuristic used to determine if there is too much interference.

6 Related Work

While the management of TM is a developing field, there is much prior work on dynamic power management for CMPs. [15] proposes the use of a global manager that selects the proper power mode with the help of DVFS to minimize power with respect to a given performance target. However, they assume that the underlying hardware supports software managed per-core DVFS, which is not available from current implementations of TM. [20] studies the use of on-chip regulators for DVFS and conclude that CMPs can benefit greatly from its use. Per-core DVFS would allow *autoturbo* to selectively enable TM for cores hosting workloads that benefit from the extra frequency boost. [8] proposes a hardware mechanism that can be used to accurately model the profitability of DVFS at various operating points. If this hardware mechanism was available on our evaluation hardware, then we could accurately use it to predict the optimal setting of TM as opposed to using our ML model to approximate the benefits of TM. [13] also proposes using performance counters to decide when to use low-power and high-power DVFS states. We use a machine learning to extend this idea to create a more robust predictor that can handle workload interference from executing several workloads on the same CMP. A key enabler for TM is per-core power gating [21], which creates large amounts of thermal and electrical headroom that is used to boost the frequency of the remaining active cores. Another enabling technology is for the CPU to estimate its power consumption when deciding if there is sufficient headroom to activate TM [14]. Computational sprinting [27] utilizes thermal capacitance to temporarily overclock the CPU to achieve increased performance for a short period of time. This is different from TM in that TM can achieve sustained overclocking if there are idle resources on the CPU.

The field of evaluating and managing TM is nascent, as the technology was only recently introduced. [5] evaluate TM on an Intel Nehalem processor and determine that TM improves execution time at the cost of power. We extend their work by demonstrating that although QPS/W is not improved, the use of TM can still improve other metrics of interest. In addition, we also implement a software TM controller that is able to prevent the activation of TM in suboptimal scenarios.

There has been much prior work on detecting and managing workload interference caused by memory contention. [16] quantifies destructive interference between separate processes on CMPs and concludes that most of the performance penalty comes from contention for the shared memory bus, supporting our strategy for detecting workload interference. [23] propose Cross-core interference Profiling Environment (CiPE) as a way of directly detecting cross-core interference in an accurate fashion, as opposed to indirect detection of *autoturbo*. CiPE uses offline profiling, which is infeasible for DCs that can accept any workload, such as Amazon EC2. However, CiPE is a viable method for characterizing applications as *Sensitive* and/or *Interfering* during the offline training phase for *autoturbo*. [24] also deals with resource contention with an eye towards optimizing for the *ED* metric. Their focus is more on scheduling, although they also use DVFS as a fallback attempt to deal with energy inefficiency in the case of memory contention that scheduling cannot resolve. *autoturbo* complements this work, as it can optimize for energy efficiency in the newly introduced TM regime that cannot be controlled in the same way as DVFS. [29] proposes Vantage, an effective cache partitioning scheme. *autoturbo* can use Vantage to partition the last level cache to shield *Sensitive* applications from *Interfering* applications to maximize the benefit of TM. [11] manages interference between contending workloads on a CMP by using clock modulation, which is another knob *autoturbo* can use to selectively enable TM for workloads that benefit from a faster clock.

7 Conclusions

Modern servers must carefully balance performance gains against energy consumption and cost increases. We showed that TurboMode, the dynamic overclocking of multi-core chips when thermal and power headroom exists, can greatly improve performance. However, its impact on efficiency metrics that include energy and cost is not always positive and depends significantly on the characteristics of the application and underlying hardware. We developed *autoturbo*, a software daemon that utilizes predictive models to understand the impact of TurboMode on the current workload and to decide if TM should be turned on or off. *autoturbo* allows workloads to get the maximum benefit from TurboMode when possible (up to 68% efficiency increase for ED^2), while eliminating virtually all the cases where TurboMode leads to efficiency drop (up to -25% for ED^2). We also demonstrate that *autoturbo* is a general solution that works for many different hardware configurations.

Acknowledgements

We sincerely thank Adam Belay, Caroline Suen, Christina Delimitrou, Jacob Leverich, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. This work was partially supported by a Google grant and the Stanford Pervasive Parallelism Lab.

References

[1] AMD. (2011) The New AMD Opteron™ Processor Core Technology.
[2] L. A. Barroso *et al.*, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2009.
[3] C. Bienia *et al.*, "The parsec benchmark suite: characterization and architectural implications," in *PACT 2008*, 2008.
[4] D. Brooks *et al.*, "Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors," *Micro, IEEE*, vol. 20, no. 6, 2000.
[5] J. Charles *et al.*, "Evaluation of the intel ® core ™ i7 turbo boost feature," in *IISWC 2009*, 2009.
[6] B.-G. Chun *et al.*, "An energy case for hybrid datacenters," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, 2010.
[7] T. Curk *et al.*, "Microarray data mining with visual programming," *Bioinformatics*, vol. 21, 2005.
[8] S. Eyerman *et al.*, "A counter architecture for online dvfs profitability estimation," *Computers, IEEE Transactions on*, vol. 59, no. 11, 2010.
[9] M. Ferdman *et al.*, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS 2012*, 2012.
[10] J. Hamilton, "Keynote," in *ISCA 2011*, 2011.
[11] A. Herdrich *et al.*, "Rate-based qos techniques for cache/memory in cmp platforms," in *ICS-23*, 2009.
[12] Intel. (2008) Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors.
[13] C. Isci *et al.*, "Long-term workload phases: duration predictions and applications to dvfs," in *MICRO-38*, 2005.
[14] C. Isci *et al.*, "Runtime power monitoring in high-end processors: methodology and empirical data," in *MICRO-36*, 2003.
[15] C. Isci *et al.*, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO-39*, 2006.
[16] M. Jahre *et al.*, "A quantitative study of memory system interference in chip multiprocessor architectures," in *HPCC 2009*, 2009.
[17] A. Jaleel *et al.*, "Adaptive insertion policies for managing shared caches," in *PCT 2008*, 2008.
[18] V. Janapa Reddi *et al.*, "Web search using mobile cores: quantifying and mitigating the price of efficiency," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, 2010.
[19] R. Khare *et al.*, "Nutch: A flexible and scalable open-source web search engine," Tech. Rep., 2004.
[20] W. Kim *et al.*, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *HPCA 2008*, 2008.
[21] J. Leverich *et al.*, "Power management of datacenter workloads using per-core power gating," *Computer Architecture Letters*, vol. 8, no. 2, 2009.
[22] K. Lim *et al.*, "Understanding and designing new server architectures for emerging warehouse-computing environments," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, 2008.
[23] J. Mars *et al.*, "Directly characterizing cross core interference through contention synthesis," in *HiPEAC 2011*. ACM, 2011.
[24] A. Merkel *et al.*, "Resource-conscious scheduling for energy efficiency on multicore processors," in *EuroSys 2010*, 2010.
[25] C. D. Patel *et al.*, "Enterprise Power and Cooling: A Chip-to-DataCenter Perspective," *HotChips 19*, 2007.
[26] C. D. Patel *et al.*, "Cost Model for Planning, Development and Operation of a Data Center," 2005.
[27] A. Raghavan *et al.*, "Computational sprinting," in *HPCA-2012*, 2012.
[28] E. Rotem *et al.*, "Power management architecture of the 2nd generation intel® core™ microarchitecture, formerly codenamed sandy bridge," 2011.
[29] D. Sanchez *et al.*, "Scalable and Efficient Fine-Grained Cache Partitioning with Vantage," *IEEE Micro's Top Picks from the Computer Architecture Conferences*, vol. 32, no. 3, 2012.
[30] Y. Xie *et al.*, "Locality in search engine queries and its implications for caching," in *IEEE Infocom 2002*, 2002.