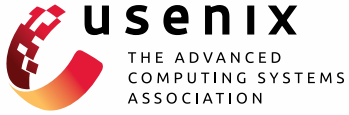

The TURBO Diaries: Application-controlled Frequency Scaling Explained

Jons-Tobias Wamhoff, Stephan Diestelhorst, and Christof Fetzer, ***Technische Universät Dresden;*** **Patrick Marlier and Pascal Felber,** ***Université de Neuchâtel;*** **Dave Dice,** ***Oracle Labs***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/wamhoff

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

The TURBO Diaries: Application-controlled Frequency Scaling Explained

Jons-Tobias Wamhoff
Stephan Diestelhorst
Christof Fetzer
Technische Universtät Dresden, Germany

Patrick Marlier
Pascal Felber
Université de Neuchâtel, Switzerland

Dave Dice
Oracle Labs, USA

Abstract

Most multi-core architectures nowadays support dynamic voltage and frequency scaling (DVFS) to adapt their speed to the system's load and save energy. Some recent architectures additionally allow cores to operate at boosted speeds exceeding the nominal base frequency but within their thermal design power.

In this paper, we propose a general-purpose library that allows selective control of DVFS from user space to accelerate multi-threaded applications and expose the potential of heterogeneous frequencies. We analyze the performance and energy trade-offs using different DVFS configuration strategies on several benchmarks and real-world workloads. With the focus on performance, we compare the latency of traditional strategies that halt or busy-wait on contended locks and show the power implications of boosting of the lock owner. We propose new strategies that assign heterogeneous and possibly boosted frequencies while all cores remain fully operational. This allows us to leverage performance gains at the application level while all threads continuously execute at different speeds. We also derive a model to help developers decide on the optimal DVFS configuration strategy, e.g, for lock implementations. Our in-depth analysis and experimental evaluation of current hardware provides insightful guidelines for the design of future hardware power management and its operating system interface.

1 Introduction

While early generations of multi-core processors were essentially homogeneous with all cores operating at the same clock speed, new generations provide finer control over the frequency and voltage of the individual cores. A major motivation for this new functionality is to maximize processor performance without exceeding the thermal design power (TDP), as well as reducing energy consumption by decelerating idle cores [4, 35].

Two main CPU manufacturers, Intel and AMD, have proposed competing yet largely similar technologies for dynamic voltage and frequency scaling (DVFS) that can exceed the processor's nominal operation frequency, respectively named *Turbo Boost* [39] and *Turbo CORE* [3]. When the majority of cores are powered down or run at a low frequency, the remaining cores can boost within the limits of the TDP. In the context of multi-threaded applications, a typical use case is the optimization of sequential bottlenecks: waiting threads halt the underlying core and allow the owner thread to speed up execution of the critical section.

Boosting is typically controlled by hardware and is completely transparent to the operating system (OS) and applications. Yet, it is sometimes desirable to be able to finely control these features from an application as needed. Examples include: accelerating the execution of key sections of code on the critical path of multi-threaded applications [9]; boosting time-critical operations or high-priority threads; or reducing the energy consumption of applications executing low-priority threads. Furthermore, workloads specifically designed to run on processors with heterogeneous cores (e.g., few fast and many slow cores) may take additional advantage of application-level frequency scaling. We argue that, in all these cases, fine-grained tuning of core speeds requires *application knowledge* and hence cannot be efficiently performed by hardware only.

Both Intel and AMD hardware implementations are constrained in several ways, e.g., some combination of frequencies are disallowed, cores must be scaled up/down in groups, or the CPU hardware might not comply with the scaling request in some circumstances. Despite the differences of both technologies, our comparative analysis derives a common abstraction for the processor performance states (Section 2). Based on the observed properties, we present the design and implementation of TURBO, a general-purpose library for application-level DVFS control that can programmatically configure the speed of the cores of CPUs with AMD's Turbo CORE and Intel's Turbo Boost technologies, while abstracting the low-level differences and complexities (Section 3).

The cost of frequency and voltage transitions is subject to important variations depending on the method used for modifying processor states and the specific change requested. The publicly available documentation is sparse, and we believe to be the first to publish an in-depth investigation on the latency, performance, and limitations of these DVFS technologies (Section 4). Unlike previous research, our goal is not energy conservation or thermal boosting [36], which is usually applied to mobile devices and interactive applications with long idle periods, but long running applications often found on servers. We target efficiency by focusing on the best performance, i.e., shorter run times or higher throughput using the available TDP. In this context, hardware is tuned in combination with the OS to use frequency scaling for boosting sequential bottlenecks on the critical path of multi-threaded applications. We use the TURBO library to measure the performance and power implications of both blocking and spinning locks (Section 4.2). Our evaluation shows that connecting knowledge of application behavior to programmatic control of DVFS confers great benefits on applications having heterogeneous load. We propose new configuration strategies that keep all cores operational and allow a manual boosting control (Section 4.3).

Based on the evaluation of manual configuration strategies

	AMD FX-8120	Intel i7-4770
Model	AMD Family 15h Model 1	Intel Core 4th generation
Codename	"Bulldozer"	"Haswell"
Design	4 modules with 2 ALUs & 1 FPU	4 cores with hyper-threading
L2 cache	4×2MB per module	4×256KB per core
L3 cache	1×8MB per package	1×8MB per package
TDP	124.95W (NB 14.23W)	84W
Frequency	3.1GHz, (1.4–4.0GHz)	3.4GHz (0.8–3.9GHz)
Stepping	ACPI P-states, 100MHz	multiplier P-states, 100MHz
Voltage	0.875–1.412V (3.41–27.68W)	0.707–1.86V (5–75W)

Table 1: Specification of the AMD and Intel processors.

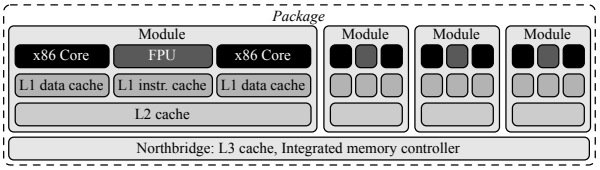

Figure 1: Organization of an AMD FX-8120 processor.

and their latencies, we derive a simplified cost model (Section 4.4) to guide developers at which size of a critical region a frequency transition pays off. Four case studies investigate the performance gains exploited by application-level frequency control based on real-world benchmarks (Section 5).

2 Hardware Support for Boosting

With both AMD's Turbo CORE and Intel's Turbo Boost, performance levels and power consumption of the processor are controlled through two types of operational states: *P-states* implement DVFS and set different frequency/voltage pairs for operation, trading off higher voltage (and thus higher power draw) with higher performance through increased operation frequency. P-states can be controlled through special machine-specific registers (MSRs) that are accessed through the `rdmsr/wrmsr` instructions. The OS can request a P-state change by modifying the respective MSR. P-state changes are also not instantaneous: the current needs to be adapted and frequencies are ramped, both taking observable time.

C-states are used to save energy when a core is idle. C0 is the normal operational state. All other C-states halt the execution of instructions and trade different levels of entry/wakeup latency for lower power draw. The OS can invoke C-states through various means such as the `hlt` and `monitor/mwait` instructions. We argue in this paper that there are benefits in keeping selected cores operational, albeit at a lower frequency, and that manipulating P-states can be more efficient in terms of latency than manipulating C-states.

We base our work on AMD's FX-8120 [1] and Intel's i7-4770 [19] CPUs, whose characteristics are listed in Table 1.

2.1 AMD's Turbo CORE

The architecture of the AMD FX-8120 processor is illustrated in Figure 1. The *cores* of a *package* are organized by pairs in *modules* that share parts of the logic between the two cores.

Our processor supports seven P-states summarized in Table 2. We introduce a TURBO naming convention to abstract from the manufacturer specifics. AMD uses P-state numbering based on the ACPI standard with P0 being the highest performance state. The two topmost are boosted P-states (#$P_{boosted}$ = 2) that are by default controlled by the hardware. The remaining five P-states can be set by the OS through the MSRs[1].

Hardware P-state	P0	P1	**P2**	P3	P4	P5	P6
TURBO naming	P_{turbo}		P_{base}				P_{slow}
Frequency (GHz)	4.0	3.4	3.1	2.8	2.3	1.9	1.4
Voltage (mV)	1412	1412	1275	1212	1087	950	875
Power 4×`nop` (W)	—	123.3	113.6	97.2	70.1	49.9	39.3
Power 4×*ALU* (W)	—	—	122.6	104.3	74.6	52.9	41.2
Power 3×P_{slow}, 1×P0..6 (W)	125.0	119.8	100.5	87.4	65.5	48.5	41.2
Power 3×`mwait`, 1×P0..6 (W)	120.1	116.5	90.9	77.6	55.5	40.5	32.8

Table 2: Default P-state configuration of AMD FX-8120.

The boosting of the frequency beyond the nominal P-state (P_{base}) is enabled by the hardware's Turbo CORE technology if operating conditions permit. The processor determines the current power consumption and will enable the first level of boosting ($P1_{HW}$) if the total power draw remains within the TDP limit and the OS requests the fastest software P-state. A multi-threaded application can boost one module to $P1_{HW}$ while others are in P_{base} if it does not use all features of the package to provide the required power headroom, e.g., no FPUs are active. The fastest boosting level (P_{turbo}) is entered automatically if some cores have furthermore reduced their power consumption by entering a deep C-state. Note that Turbo CORE is deterministic, governed only by power draw and not temperature, such that the maximum frequency is workload dependent. During a P-state transition, the processor remains active and capable of executing instructions, and the completion of a P-state transition is indicated in an MSR available to the OS.

The Turbo CORE features can be enabled or disabled altogether, i.e., no core will run above P_{base}. Selected AMD processors allow developers to control the number of hardware-reserved P-states by changing #$P_{boosted}$ through a configuration MSR. To achieve manual control over all P-states, including boosting, one can set #$P_{boosted} = 0$. The core safety mechanisms are still in effect: the hardware only enters a boosted P-state if the TDP limit has not been reached. In contrast to the processor's automatic policy, the manual control of all P-states can enable P_{turbo} with all other cores in C0 but running at P_{slow}.

Due to the pairwise organization of cores in modules, the effect of a P- and C-state change depends on the state of the sibling core. While neighboring cores can request P-states independently, the fastest selected P-state of the two cores will apply to the entire module. Since the `wrmsr` instruction can only access MSRs of the current core, it can gain full control over the frequency scaling if the other core is running at P_{slow}. A module only halts if both cores are not in C0.

The processor allows to read the current power draw (P) that it calculates based on the load. Out of the total TDP, 14.24W are reserved for the northbridge (NB) (including L3 cache) and logic external to the cores. Each of the four modules is a voltage (V) and frequency (f) domain defined by the P-state. The package requests V defined by the fastest active P-state of any module from the voltage regulator module (VRM).

[1]The numbering in software differs from the actual hardware P-states: $P_{HW} = P_{SW} + \#P_{boosted}$. With a default of $\#P_{boosted} = 2$: $P_{base} = P0_{SW} = P2_{HW}$ and $P_{turbo} = P0_{HW}$. $P0_{SW}$ is the fastest requestable software P-state.

Hardware P-state	P39	**P34**	P20	P8
TURBO naming	P_{turbo}	P_{base}		P_{slow}
Frequency (GHz)	3.9	3.4	2.0	0.8
Voltage (mV)	1860	n/a	n/a	707
Power `nop` (W)	—	39	20	11
Power *ALU* (W)	—	51	25	12
Power `mwait` (W)	25	19	11	8

Table 3: Default P-state configuration of Intel i7-4770.

Table 2 lists P with (1) all cores in the same P-state executing `nop` instructions, (2) execution of integer operations with *ALU*, (3) three modules in P_{slow} except one in the given P-state, and (4) all modules halted using `mwait` except one active core. The consumed active P depends on V, f and the capacitance (C) that varies dynamically with the workload ($P = V^2 * f * C_{dyn}$). Therefore, for the `nop` load all cores can boost to $P1_{HW}$, while for integer loads all cores can run only at P_{base}. Boosting under load can be achieved when other modules are either in P_{slow} or halted. `Mwait` provides the power headroom to automatically boost to P_{turbo}. The manual boosting control allows to run one module in P_{turbo} if the others run at P_{slow}.

2.2 Intel's Turbo Boost

Intel's DVFS implementation is largely similar to AMD's but more hardware-centric and mainly differs in the level of manual control. All cores are in the same frequency and voltage domain but can each have an individual C-state. The P-states are based on increasing multipliers for the stepping of 100MHz, non-predefined ACPI P-states in the opposite order. Our processor supports frequencies from 0.8GHz to 3.9GHz corresponding to 32 P-states that are summarized in Table 3. In TURBO terms, P_{base} corresponds to $P34_{HW}$, leaving 5 boosted P-states. All active cores in C0 symmetrically run at the highest requested frequency, even if some cores requested slower P-states. The consumed power was measured in a fashion analogous to that in Section 2.1, with hyper-threading enabled and all cores always in the same P-State.

The processor enables Turbo Boost if not all cores are in C0. The level of boosting depends on the number of active cores, estimated power consumption, and additionally the temperature of the package. This "thermal boosting" allows the processor to temporarily exceed the TDP using the thermal capacitance of the package. In contrast to AMD, the maximum achievable frequency also depends on the recent execution history, which relates to the current package temperature and makes it somewhat stateful. While boosting can be enabled or disabled altogether, the boosted P-states are always controlled automatically by the processor and no manual control by software is possible.

Intel's design choice targets to speed up critical periods of computation, e.g., boosting sequential bottlenecks by putting waiting cores to sleep using C-states or providing temporarily peak performance for interactive applications as on mobile devices or desktops. Our focus is on multi-threaded applications mostly found on servers that run for long periods without much idle time. Thermal boosting is not applicable to such workloads because on average one cannot exceed the TDP. Instead, our goal is to improve the performance within the TDP limits.

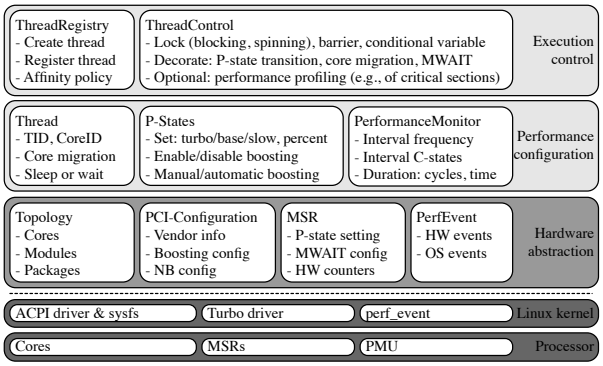

Figure 2: Overview of TURBO *library components.*

3 TURBO Library

The TURBO library, written in C++ for the Linux OS, provides components to configure, control, and profile processors from within applications. Our design goals are twofold: we want to provide a set of abstractions to (1) make it convenient to improve highly optimized software based on DVFS; and (2) set up a testbed for algorithms that explore challenges of future heterogeneous cores [2], such as schedulers. The components of the TURBO library are organized in layers with different levels of abstraction as shown in Figure 2. All components can be used individually to support existing applications that use multiple threads or processes. The layered architecture allows an easy extension to future hardware and OS revisions.

3.1 Processor and Linux Kernel Setup

The default configurations of the processors and Linux kernel manage DVFS transparently for applications: All boosted P-states are controlled by the processor and the Linux governor will adapt the non-boosted P-states based on the current processor utilization ("ondemand") or based on static settings that are enforced periodically ("performance", "userspace").

We must disable the influence of the governors and the processor' power saving features in order to gain explicit control of the P-states and boosting in user space using our library. Note that the "userspace" governor provides an alternative but inefficient P-state interface [16]. Therefore, we disable the CPU frequency driver (`cpufreq`) and turn off AMD's *Cool'n'Quiet* speed throttling technology in the BIOS. To control all available P-states in user space, we can either disable automatic boosting altogether, which is the only solution for Intel, or for AMD set $\#P_{boosted} = 0$ to enable manual boosting control (for details see Section 2). Changing the number of boosted P-states also changes the frequency of the *time stamp counter* (`tsc`) for AMD processors so we therefore disable `tsc` as a clock source for the Linux kernel and instead use the *high precision event timer* (`hpet`). Note that these tweaks can easily be applied to production systems because we only change BIOS settings and kernel parameters.

The processor additionally applies automatic frequency scaling for the integrated NB that can have a negative impact on memory access times for boosted processor cores.

Therefore, NB P-states are disabled and it always runs at the highest possible frequency.

Linux uses the `monitor` and `mwait` instructions to idle cores and change their C-state. When another core writes to the address range specified by `monitor`, then the core waiting on `mwait` wakes up. The `monitor-mwait` facility provides a "polite" busy-waiting mechanism that minimizes the resources consumed by the waiting thread. For experiments on AMD, we enable these processor instructions for user space and disable the use of `mwait` in the kernel to avoid lockouts. Similarly, we must also disable the use of the `hlt` instruction by the kernel, because otherwise we cannot guarantee that at least one core stays in C0. We restrict the C-state for the Linux kernel to C0 and use the polling idle mode. These changes are required in our prototype only for the evaluation of C-state transitions and are not necessary in a production system.

The presented setup highlights the importance of the configuration of both hardware and OS for sound benchmarking. Multi-threaded algorithms should be evaluated by enforcing P_{base} and C0 on all cores to prevent inaccuracies due to frequency scaling and transition latencies. All other sources of unpredictability should be stopped, e.g., all periodic `cron` jobs.

3.2 Performance Configuration Interface

The library must be aware of all threads even if they are managed explicitly by the application. Therefore, the *thread registry* is used first to create or register all *threads*. Next, the threads are typically assigned to distinct cores based on the processor's *topology*, which is discovered during initialization. If thread migration to another core is required at runtime, it must be performed using our library to allow an update of the core specific configuration, e.g., the P-state.

The easiest way to benefit from DVFS is to replace the application's locks with *thread control* wrappers that are decorated with implicit P-state transitions, e.g., boosting the lock owner at P_{turbo}, waiting at P_{slow}, and executing parallel code at P_{base}.

If the wrappers are not sufficient, the application can request an explicit *performance configuration* that is still independent of the underlying hardware. Threads can request the executing core to run at P_{turbo}, P_{base}, or P_{slow}, and can alternatively specify the *P-state* in percent based on the maximum frequency. The actual P-state is derived from the selected setup, e.g., if boosting is enabled and controlled manually. The current P-state configuration is cached in the library in order to save the overheads from accessing the MSRs in kernel space. If a P-state is requested that is already set or cannot be supported by the processor's policy or TDP limits, then the operation has no effect.[2] Threads can also request to temporarily migrate to a dedicated processor core that runs at the highest possible frequency and stays fully operational in C0.

[2]In practice, we write our request in MSR_P_{cmd} and can read from MSR_P_{val} what the CPU actually decided. We can either (a) wait until both MSRs match, i.e., another core makes room in the TDP, (b) return the CPU's decision, or (c) just write and provide best-effort guarantees (default). Deterministic hardware without thermal boosting does not overwrite MSR_P_{cmd}.

The lowest layer presents *hardware abstractions* for the machine specific interfaces and DVFS implementations, as well as the Linux OS. The Linux kernel provides a device driver that lets applications access MSRs as files under root privilege using `pread` and `pwrite`. We implemented a lightweight TURBO *kernel driver* for a more streamlined access to the processor's MSRs using `ioctl` calls. The driver essentially provides a wrapper for the `wrmsr/rdmsr` instructions to be executed on the current core. Additionally, it allows kernel space latency measurements, e.g., for P-state transition time, with more accuracy than from user space. We derive the *topology* from the Linux ACPI driver and use `sysfs` for AMD's package configuration using PCI functions.

3.3 Performance and Power Profiling

The TURBO library provides means to profile highly optimized applications and algorithms for heterogeneous cores. The profiling can be used to first identify sections that can benefit from frequency scaling and later to evaluate the performance and power implications of different configurations.

Again, the simplest ways to obtain statistics is to use *thread control* wrappers, which exist to replace locks, barriers, and condition variables. The wrappers can be decorated with profiling capabilities of the *performance monitor*, which uses the `aperf/mperf` and `tsc` counters of the processor [1, 19] and the `perf_event` facilities of the Linux kernel to access the processor's *performance monitoring unit* (PMU).

The performance monitor operates in intervals, e.g., defined by a lock wrapper, for which it captures the cycles, frequency, and C-state transitions. Additional counters such as the number of cache misses or stalled cycles can be activated, e.g., to analyze the properties of a critical section. The PMU also provides counters to read the *running average power limit* (RAPL) on Intel and the processor power in TDP on AMD.

4 Processor Evaluation

On top of the TURBO library presented in Section 3, we implemented a set of benchmark applications that configure and profile the underlying processor. In this section, we present (1) the static transition latencies introduced by the OS and hardware, (2) the overheads of blocking upon contended locks and when it pays off regarding speed and energy compared to spinlocks, and (3) new static and dynamic P-state transition strategies that optimize spinlocks and allow applications to expose heterogeneous frequencies.

4.1 Hardware Transition Latency

The latency for DVFS results from a combination of OS overhead to initiate a transition and hardware latency to adjust the processor's state. Therefore, we present in Tables 4 (AMD) and 5 (Intel) the overhead for system calls, P-state requests and the actual transition latencies in isolation. Throughout our evaluation, we use a Linux kernel 3.11 that is configured according to Section 3.1. We use only the x86 cores (ALU)

Operation	P-State Transition	Mean Cycles	Mean ns	Deviation Cycles	Deviation ns
System call overheads for `futex` and TURBO driver					
`syscall(futex_wait_private)`	—	1321	330	42	10
`ioctl(trb)`	—	920	230	14	3
P-state MSR read/write cost using `msr` or TURBO driver					
`pread(msr, pstate)`	—	3044	761	43	10
`ioctl(trb, pstate)`	—	2299	574	30	7
`pwrite(msr, pstate, P`$_{base}$`)`	$P_{base} \rightarrow P_{base}$	2067	741	110	27
`ioctl(trb, pstate, P`$_{base}$`)`	$P_{base} \rightarrow P_{base}$	1875	468	42	10
Hardware latencies for P-state set (`wrmsr`) and transition (`wait`) (kernel space)					
`wrmsr(pstate, P`$_{slow}$`)`	$P_{base} \rightarrow P_{slow}$	28087	7021	105	26
`wrmsr(pstate, P`$_{slow}$`) & wait`	$P_{base} \rightarrow P_{slow}$	29783	7445	120	30
`wrmsr(pstate, P`$_{turbo}$`)`	$P_{slow} \rightarrow P_{turbo}$	1884	471	35	8
`wrmsr(pstate, P`$_{turbo}$`) & wait`	$P_{slow} \rightarrow P_{turbo}$	226988	56747	84	21
`wrmsr(pstate, P`$_{base}$`) & wait`	$P_{slow} \rightarrow P_{base}$	183359	45839	130	32
`wrmsr(pstate, P`$_{turbo}$`) & wait`	$P_{base} \rightarrow P_{turbo}$	94659	23664	87	21
`wrmsr(pstate, P`$_{base}$`)`	$P_{turbo} \rightarrow P_{base}$	23203	5800	36	9
`wrmsr(pstate, P`$_{base}$`) & wait`	$P_{turbo} \rightarrow P_{base}$	24187	6046	139	34
`wrmsr(pstate, P1`$_{HW}$`)`	$P_{base} \rightarrow P1_{HW}$	974	234	132	33
`wrmsr(pstate, P1`$_{HW}$`) & wait`	$P_{base} \rightarrow P1_{HW}$	94642	23660	136	34
`wrmsr(pstate, P`$_{base}$`) & wait`	$P1_{HW} \rightarrow P_{base}$	24574	6143	138	34
Hardware latencies for C-state transitions (in user space)					
`monitor & mwait`	—	1818	454	18	4
Software and hardware latency for thread migration					
`pthread_setaffinity`	—	26728	6682	49	12

Table 4: Latency cost (AMD FX-8120, 100,000 runs).

Operation	P-State Transition	Mean Cycles	Mean ns	Deviation Cycles	Deviation ns
System call overheads for `futex` and TURBO driver					
`syscall(futex_wait_private)`	—	1431	366	32	8
`ioctl(trb)`	—	1266	324	64	16
P-state MSR read/write cost using `msr` or TURBO driver					
`pread(msr, pstate)`	—	2638	775	24	7
`ioctl(trb, pstate)`	—	2314	680	54	16
`pwrite(msr, pstate, P`$_{base}$`)`	$P_{base} \rightarrow P_{base}$	4246	1248	122	35
`ioctl(trb, pstate, P`$_{base}$`)`	$P_{base} \rightarrow P_{base}$	3729	1096	72	21
Hardware latencies for P-state set (`wrmsr`) and transition (`wait`) (kernel space)					
`wrmsr(pstate,P`$_{base}$`)`	$P_{slow} \rightarrow P_{base}$	44451	13073	131	38
`wrmsr(pstate,P`$_{base}$`) & wait`	$P_{slow} \rightarrow P_{base}$	48937	14393	86	25
`wrmsr(pstate,P`$_{slow}$`)`	$P_{base} \rightarrow P_{slow}$	2015	592	61	17
`wrmsr(pstate,P`$_{slow}$`) & wait`	$P_{base} \rightarrow P_{slow}$	58782	17288	65	19
`wrmsr(pstate,P`$_{turbo}$`)`	$P_{base} \rightarrow P_{turbo}$	2012	591	44	12
`wrmsr(pstate,P`$_{turbo}$`) & wait`	$P_{base} \rightarrow P_{turbo}$	41451	12191	78	22
Hardware latencies for C-state transitions (in kernel space)					
`monitor & mwait C1`	—	4655	1369	25	7
`monitor & mwait C2`	—	36500	10735	1223	359
`monitor&mwait C6`	—	74872	22021	672	197
Software and hardware latency for thread migration					
`pthread_setaffinity`	—	12145	3572	81	23

Table 5: Latency cost (Intel i7-4770, 100,000 runs).

and no FPU or MMX/SSE/AVX to preserve the required headroom for manual boosting.

System calls for device-specific input/output operations (`ioctl`) have a low overhead and are easily extensible using the request code parameter. The interface of the TURBO driver (`trb`) is based on `ioctl`, while the Linux MSR driver (`msr`) uses a file-based interface that can be accessed most efficiently using `pread/pwrite`. The difference in speed between `msr` and `trb` (both use `rdmsr/wrmsr` to access the MSRs) results mostly from additional security checks and indirections that we streamlined for the TURBO driver. The cost in time for system calls depends on the P-state, i.e., reading the current P-state scales with the selected frequency, here P_{base}.

Observation 1: *P-state control should be made available through platform-independent application program interfaces (APIs) or unprivileged instructions. The latter would eliminate the latency for switching into kernel space to access platform-specific MSRs but require that the OS's DVFS is disabled.*

We measured the cost of the `wrmsr` instruction that initiates a P-State transition of the current core, as well as the latency until the transition is finished, by busy `wait`ing until the frequency identifier of the P-state is set in the status MSR. Both measurements are performed in the TURBO driver, removing the inaccuracy due to system call overheads.

For AMD, requesting a P-state faster than the current one (e.g., $P_{slow} \rightarrow P_{base}$) has low overhead in itself, but the entire transition has a high latency due to the time the VRM takes to reach the target voltage. The request to switch to a slower P-state (e.g., $P_{base} \rightarrow P_{slow}$) has almost the same latency as the entire transition, i.e., the core is blocked during most of the transition. We suspect that this blocking may be caused by a slow handshake to coordinate with the other module's core to see if an actual P-state change will occur. Overall, the transition has a lower latency because the frequency can already be reduced before the voltage regulator is finished. If only switching to a slow P-state for a short period, the transition to a faster P-state will be faster if the voltage was not dropped completely.

On the Intel CPU, total latency results are very similar: A P-state transition also takes tens of microseconds but depends on the distance between the current and requested P-state. A significant difference to AMD, however, lies in the faster execution of the `wrmsr` request of a P-state transition going slower (e.g., $P_{base} \rightarrow P_{slow}$) because Intel does not need to perform additional coordination.

Observation 2: *The frequency transitions should be asynchronous, triggered by a request and not blocking, i.e., keeping the core operational. The API should include the ability to read or query P-state transition costs for building a cost model that allows DVFS-aware code to adapt at runtime.*

We additionally show costs related to the OS. In the `mwait` experiment, one core continuously updates a memory location while the other core specifies the location using `monitor` and calls `mwait`. The core will immediately return to execution because it sees the memory location changed, so the numbers represent the minimal cost of executing both instructions. Although AMD allows the use of `mwait` from user space, the feature is typically used by the OS's `futex` system call when the kernel decides to idle. The `pthread_setaffinity` function migrates a thread to a core with a different L2 Cache that is already in C0 state and returns when the migration is finished. Thread migration typically results in many cache misses but the benchmark keeps only minimal data in the cache.

Observation 3: *The OS should keep the current frequency in the thread context to better support context switches and thread migrations. Ideally, the OS would expose a new set of advisory platform-independent APIs to allow threads to set their desired DVFS-related performance targets. Furthermore, the OS kernel (and potentially a virtual machine hypervisor) would moderate potentially conflicting DVFS resource requests from independent and mutually unaware applications.*

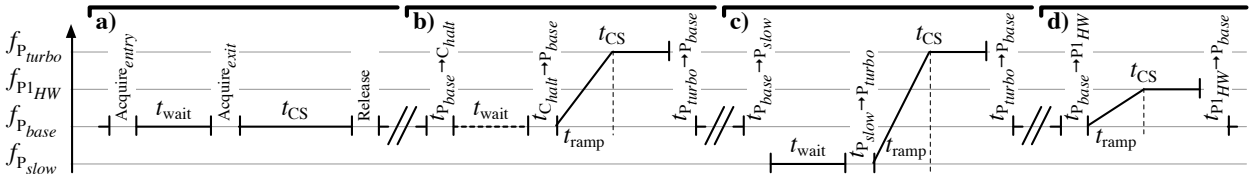

Figure 3: Frequency sequence for (a) spinning, (b) blocking, (c) frequency scaling and (d) critical regions.

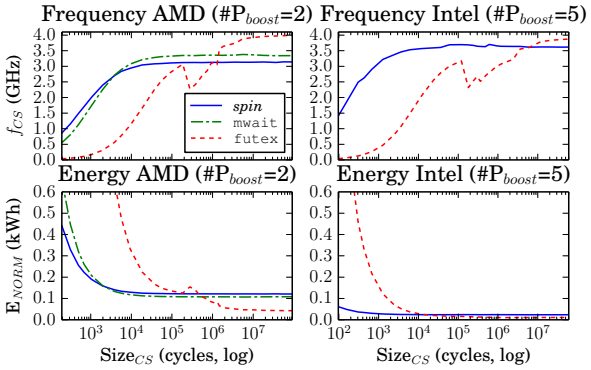

Figure 4: Characteristics of blocking and spinning.

4.2 Blocking vs. Spinning Locks

We evaluate the boosting capabilities using a thread on each core that spends all its time in critical sections (CS). The CS is protected by a single global lock implemented as an MCS queue lock [29] in the TURBO library. The lock is decorated such that upon contention, the waiting thread either *spins* or blocks using `mwait` (AMD only) or `futex`. The sequence is illustrated in Figure 3a and 3b, respectively. In all cases, the thread-local MCS node is used for the notification of a successful lock acquisition. Inside the CS, a thread-local counter is incremented for a configurable number of iterations (∼10 cycles each). While the global lock prevents any parallelism, the goal of the concurrent execution is to find the CS length that amortizes the DVFS cost.

We want to discuss when blocking is preferable over spinning, both in terms of performance and energy, using the default configuration of hardware and OS: The P-states are managed automatically by the processor and the setup from Section 3.1 is not applied. We run the application for 100 seconds and count the number of executed CS, which gives us the cycles per CS including all overheads. Separately, we measure the cycles per CS without synchronization at P_{base}, i.e., the cycles doing real work. The effective frequency inside a CS is: $f_{CS} = f_{base} * \frac{cycles_{nosync}}{cycles_{mcs}}$. The energy results are based on the processor's TDP/RAPL values, from which we take samples during another execution. We compute the energy it takes to execute 1 hour of work at P_{base} inside CS: $E = E_{sample} * \frac{cycles_{mcs}}{cycles_{nosync}}$.

The results are shown in Figure 4. The *spin* strategy runs all cores at P_{base} and is only effected by synchronization overhead, with decreasing impact for larger sizes of CS. The `mwait` and `futex` strategies are additionally effected by C-state transitions that halt the core while blocking, which allows to boost the active core. The C-state reached by `mwait` is not deep enough to enable P_{turbo}, probably because it is requested from user space. Still, CS are executed at $P1_{HW}$ and the low overhead lets `mwait` outperform *spin* already at a CS size of ∼4k cycles. Using `futex` has the highest overhead because it is a system call. The C-state reached depends on t_{wait} (see Figure 3b), which explains the performance drop: Deep C-states introduce a high latency (see Table 5) but are required to enable P_{turbo}. We verified this behavior using `aperf/mperf`, which showed that the frequency in C0 is at P_{turbo} only after the drop. The `futex` outperforms *spin* and `mwait` at ∼1.5M cycles for AMD and ∼4M cycles for Intel, which also boosts *spin* 2 steps. Note that an optimal synchronization strategy for other workloads also depends on the conflict probability and t_{wait}, but our focus is on comparing boosting initiated by the processor and on application-level.

The sampled power values do not vary for different sizes of CS (see Tables 2 and 3 for *ALU* and `mwait`), except for `futex`, which varies between 55-124W for AMD depending on the reached C-state. The reduction in energy consumption due to deeper C-states must first amortize the introduced overhead before it is more efficient than spinning. With only a single core active at a time, `futex` is the most energy efficient strategy for AMD after a CS size of ∼1M cycles, which results for 8 threads in t_{wait} =∼7M cycles because the MCS queue lock is fair. Intel is already more energy efficient after ∼10k cycles, indicating that it trades power savings against higher latencies. Boosting provides performance gains for sequential bottlenecks and halting amortizes the active cores' higher energy consumption [31]. The default automatic boosting is not energy efficient for scalable workloads because all energy is consumed only by a single core without performance benefit [12].

4.3 Application-level P-state Transition Strategies

Our goal is to enable application-level DVFS while keeping all cores active. Therefore, we enable manual P-state control with the setup described in Section 3.1 and restrict the following discussion to just AMD. For the evaluation, we use the same application as in the previous Section 4.2 but with a different set of decorations for the lock: The strategy `one` executes iterations only on a single core that sets the P-state statically during initialization to either P_{slow}, P_{base} or P_{turbo}. All other threads run idle on cores at P_{slow} in C0. This provides the baseline for different P-state configurations without P-state transition overheads but includes the synchronization. The dynamic strategies `ownr` and `wait` are illustrated in Figure 3c. For `ownr`, all threads are initially set to P_{slow} and the lock owner dynamically switches to P_{turbo} during the CS. For `wait`, all

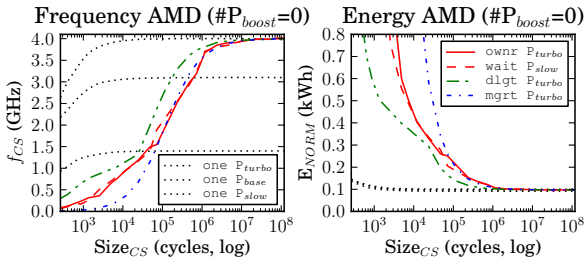

Figure 5: Characteristics of manual P-state control.

threads initially request P_{turbo} and dynamically switch to P_{slow} while waiting. The processor prevents an oversubscription and allows P_{turbo} only if 3 modules are in P_{slow}. The remaining strategies use only a subset of the cores for executing CS: `dlgt` uses only 1 thread per module and delegates the P-state transition request to the thread executing on the neighboring core. The strategy is otherwise the same as `ownr`. `mgrt` uses only 6 cores on 3 modules running at P_{slow}. The remaining module runs at P_{turbo} and the current lock owner migrates to a core of the boosted module during the CS.

The results are presented in Figure 5. The dynamic strategies `ownr` and `wait` introduce overhead in addition to the synchronization costs because two P-state transitions must be requested for each CS. This overhead is amortized when the resulting effective frequency of the CS is above `one` with P_{base}, starting at CS sizes of ∼600k cycles. Both strategies behave similarly because the application does not execute parallel code between CS. Otherwise, the idea is that `wait` hides the slow blocking transition to P_{slow} (see Section 4.1) within t_{wait}, whereas `ownr` must perform this transition after releasing the lock. To that extent, `dlgt` shifts the P-state transition cost entirely to the other core of the module and can outperform `one` already at ∼200k cycles, but only half of the processor cores can be used. The `mgrt` strategy does not include overhead from P-state transitions but costly thread migrations. Still, it outperforms `one` at ∼400k cycles. A real-world benchmark would show worse results because it suffers from more cache misses on the new processor core than our synthetic benchmark that keeps only little data in the cache [30]. Additionally, initiating a migration at P_{slow} will be executed slowly until the thread reaches the boosted core. Overall, we observe that application-level DVFS is more effective than C-state control because it allows to outweigh overheads for CS of sizes smaller than ∼1.5M cycles.

Observation 4: *The P-state transition should be as fast as possible so that short boosted sections can already amortize the transition cost. It exists hardware that can switch to arbitrary frequencies within one clock cycle [17].*

As long as one modules runs at P_{turbo}, which is the case here, the processor consumes the maximal TDP of 125W. The consumed energy solely depends on the overheads of each strategy because of the serialized execution. Note that the energy for executing `one` with a static P-state is almost identical for P_{slow}, P_{base} and P_{turbo}, indicating that the energy consumption is proportional to the P-state. In fact, we get for a single module in P_{turbo} 29% more speed using 25% more power compared to P_{base} (see Table 2). Compared to `mwait` and `futex`, application-level DVFS allows less power savings because all cores stay in C0, but it can be applied to parallel workloads, which we investigate in Section 5.

Observation 5: *Processors should support heterogeneous frequencies individually for each core to provide headroom for boosting while staying active. The design should not limit the frequency domain for a package (Intel) or module (AMD). An integrated VRM supports fine grained voltage domains to allow higher power savings at low speeds. Additionally, for some workloads it would be beneficial to efficiently set remote cores to P_{slow} in order to have local boosting control.*

4.4 Performance Cost Model

Based on our experimental results, we derive a simplified cost model for AMD's boosting implementation to guide developers when boosting pays off regarding performance. We first present a model for boosting sequential bottlenecks that formalizes the results from Section 4.3. We then specialize it for boosting CS that are not a bottleneck as well as for workloads that contain periods with heterogeneous workload distributions.

We make the following simplifying assumptions: (1) the application runs at a constant rate of instructions per cycle (IPC), regardless of the processor frequency; (2) we do not consider costs related to thread synchronization; (3) the frequency ramps linearly towards faster P-states (e.g., $f_{P_{slow}} \rightarrow f_{P_{turbo}}$); and (4) the frequency transition to a slower P-state takes as long as the P-state request. Assumption (4) is a direct result of our latency measurement, (1) and (2) allow an estimation without taking application specifics into account. We will revisit assumptions (1) and (2) when looking at actual applications that depend on memory performance and thus exhibit varying IPC with changing frequency (due to the changed ratio of memory bandwidth, latency and operation frequency).

For sequential bottlenecks, we follow the strategy `ownr` described in Section 4.3 and illustrated in Figure 3c. Boosting will pay off if we outperform the CS that runs at $f_{P_{base}}$: $t_{CS_{f_{P_{turbo}}}} \leq t_{CS_{f_{P_{base}}}}$. The minimal t_{CS} must be greater than the combined P-state request latencies and the number of cycles that are executed during the P-State transition (t_{ramp}, i.e., the difference between `wrmsr` and `wait` in Table 4) to P_{turbo}:

$$t_{CS} \geq t_{P_{slow}\rightarrow P_{turbo}} + t_{ramp} + t_{P_{tubo}\rightarrow P_{base}} + \frac{cycles_{CS} - cycles_{ramp}}{f_{P_{turbo}}}$$

Based on the P-state transition behavior that we observed in Section 4.3, we can compute the minimal t_{CS} as follows:

$$t_{CS} \geq \frac{f_{P_{turbo}}}{f_{P_{turbo}} - f_{P_{base}}} \cdot \left(t_{P_{slow}\rightarrow P_{turbo}} + t_{P_{turbo}\rightarrow P_{base}}\right) + \frac{1}{2} \cdot \frac{f_{P_{turbo}} - f_{P_{slow}}}{f_{P_{turbo}} - f_{P_{base}}} \cdot t_{ramp}$$

The minimal wait time t_{wait} to acquire the lock should simply be larger than the time to drop to $f_{P_{slow}}$: $t_{wait} \geq t_{P_{base}\rightarrow P_{slow}}$. With the results from Section 4.1, on AMD this equals to

a minimal t_{CS} of ~436,648 cycles (~109μs). Note that optimized strategies can reach the break even point already earlier (e.g., `dlgt` in Figure 5). Based on the above cost model for sequential bottlenecks, we can derive a cost model for boosting CS by one step (see Figure 3d):

$$t_{CS} \geq \frac{f_{P1_{HW}}}{f_{P1_{HW}} - f_{P_{base}}} \cdot (t_{P_{base} \to P1_{HW}} + t_{P1_{HW} \to P_{base}}) + \frac{1}{2} \cdot t_{ramp}$$

We never move below P_{base} and boosting pays off if t_{CS} is longer than ~336,072 cycles (~84μs).

Besides boosting sequential bottlenecks, another interesting target are periods of heterogeneous workload distributions. These workloads can run one thread temporarily at a higher priority than other active threads or have an asymmetric distribution of accesses to CS from threads. Typically, such critical regions are longer because they combine several CS, thus improving the chances of amortizing the transition cost. Based on the presented cost model, we compute the minimal duration of such periods instead of the CS size. We present examples in Section 5.

5 Boosting Applications

We evaluated the TURBO library using several real-world applications with user space DVFS on the AMD FX-8120. We chose these workloads to validate the results from our synthetic benchmarks and the cost model to boost sequential bottlenecks (5.1); highlight gains by using application knowledge to assign heterogeneous frequencies (5.2); show the trade-offs when the IPC depends on the core frequency, e.g., due to memory accesses (5.3); and outweigh the latency cost of switching P-states by delegating critical sections to boosted cores (5.4).

5.1 Python Global Interpreter Lock

The Python Global Interpreter Lock (GIL) is a well known sequential bottleneck based on a blocking lock. The GIL must always be owned when executing inside the interpreter. Its latest implementation holds the lock by default for a maximum of 5ms and then switches to another thread if requested. We are interested in applying some of the P-state configuration strategies presented in Section 4.3 to see if they provide practical benefits. For this evaluation, we use the `ccbench` application that is included in the Python distribution (version 3.4a).

The benchmark includes workloads that differ in the amount of time they spent holding the GIL: (1) the *Pi calculation* is implemented entirely in Python and spends all its time in the interpreter; (2) the computation of *regular expressions* (Regex) is implemented in C with a wrapper function that does not release the GIL; and (3) the *bz2 compression* and *SHA1 hashing* have wrappers for C functions that release the GIL, so most time is spent outside the interpreter. Table 6 summarizes the characteristics of the workloads.

We evaluate the following P-state configuration strategies in Figure 6. *Base* runs at P_{base} and, hence, does not incur P-state configuration overheads. *Dyn* waits for the GIL at P_{slow}, then runs at P_{turbo} while holding the GIL and switches to P_{base} after releasing it. While the workloads `Pi` and `Regex` do not scale, *Dyn* supports at least the execution at P_{turbo}. The performance and power implications are in line with our synthetic benchmark results (Section 4.3) and the cost model (python in Table 6 greater than t_{CS} in Section 4.4). For the workloads `bz2` and `SHA1`, the performance benefit reaches its maximum at 4 threads because we pin the threads such that each runs on a different module, giving the thread full P-state control. When two threads run on a module, more P-state transitions are required per package that eliminate the performance benefit at 8 threads. *Own* runs all threads at P_{base} and boosts temporarily to $P1_{HW}$ while holding the GIL. This manifests in a higher throughput when the GIL is held for long periods but for `bz2` and `SHA` the cost of requesting a P-state transition is not amortized by the higher frequency. *Wait* runs at P_{turbo} if permitted by the TDP and only switches to P_{slow} while waiting for the GIL. This strategy works well with high contention but introduces significant cost if the waiting period is too short (see Table 6).

In Figure 7 we compare Intel's results for boosting disabled (*Base*) and enabled automatically by the processor (*Auto*). Overall, the results are similar to the ones obtained on AMD and what we expect from Section 4.2: The level of boosting depends on the number of halted cores, which enables P_{turbo}

Task	1 Thread python	1 Thread native	2 Threads wait	2 Threads python	2 Threads native	4 Threads wait	4 Threads python	4 Threads native
`Pi` (P)	72694	160	4919	4933	14	14735	4958	18
`Regex` (C)	116593	160	5533	5556	18	16763	5600	18
`bz2` (C)	17	991	10	24	992	34	25	998
`SHA1` (C)	6	386	8	12	386	11	12	386

Table 6: `ccbench` characteristics: average time (μs) per iteration spent in interpreter (python), executing native code without GIL (native) and waiting for GIL acquisition (wait).

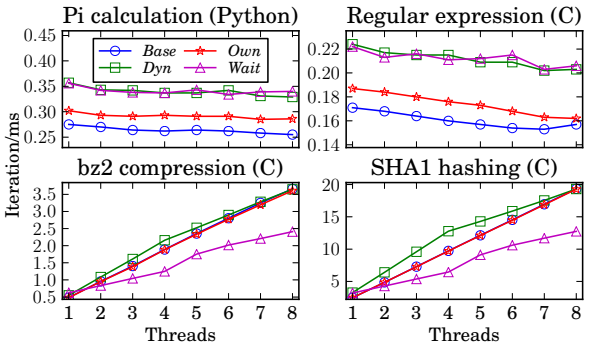

Figure 6: `ccbench` throughput (AMD FX-8120).

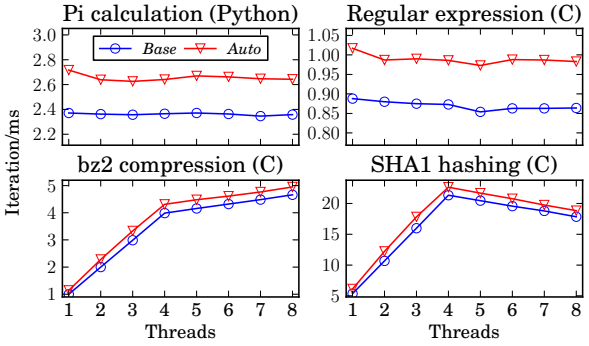

Figure 7: `ccbench` throughput (Intel i7-4770).

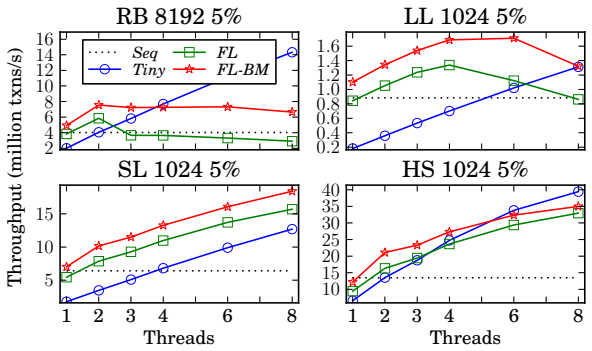

Figure 8: FastLane STM integer set benchmarks.

	RB			LL			SL			HS		
Nb. threads	*2*	*4*	*6*	*2*	*4*	*6*	*2*	*4*	*6*	*2*	*4*	*6*
FL	63	44	35	68	48	44	68	39	24	56	25	13
FL-BM	64	55	54	70	49	53	68	42	28	56	29	16

Table 7: Commit ratio of the master thread (% of all commits).

for `Pi` and `Regex`. `SHA1` and `bz2` boost slightly because not all processor features are used. The performance drop beyond 4 threads is due to hyper-threading.

5.2 Software Transactional Memory

FastLane [43] is a software transactional memory (STM) implementation that processes a workload asymmetrically. The key idea is to combine a single fast master thread that can never abort with speculative helper threads that can only commit if they are not in conflict. The master thread has a very lightweight instrumentation and runs close to the speed of an uninstrumented sequential execution. To allow helper threads to detect conflicts, the master thread must make the in-place updates of its transactions visible (by writing information in the transaction metadata). The helpers perform updates in a write-log and commit their changes after a validation at the end of the transaction. The benefit is a better performance for low thread counts compared to other state-of-the art STM implementations (e.g., TinySTM [13]) that suffer from instrumentation and bookkeeping overheads for scalability.

We used integer sets that are implemented as a *red-black tree* (RB), a *linked list* (LL), a *skip list* (SL), or a *hash set* (HS) and perform random queries and updates [13]. The parameters are the working set size and the update ratio. Either all threads run at P_{base} (*FL*) or the master statically runs at P_{turbo} (*FL-BM*) and the helpers at P_{slow}, except the helper running on the same module as the master. Note that the master thread is determined dynamically. Moreover, we compare with TinySTM (*Tiny*) and uninstrumented sequential execution (*Seq*) at P_{base}. Our evaluation on the AMD processor shows in Figure 8 that running the master and helpers at different speeds (*FL-BM*) enables high performance gains compared to running all threads at P_{base} (*FL*). The higher throughput can outweigh the higher power (50% vs. 2% for LL), thus, being more energy efficient. *Tiny* wins per design for larger thread counts. Table 7 shows that the master can asymmetrically process more transactions at P_{turbo}. While the helpers at P_{slow} can have more conflicts caused by the master, the conflict rate caused by other slow helpers does not change. Dynamically boosting the commits of the helpers did not show good results because the duration is too short.

This workload highlights the importance of making P-state configuration accessible from the user space. It allows to expose properties of the application that would otherwise not be available to the processor. For applications that contain larger amounts of non-transactional code, supporting the ability to remotely set P-states for other cores would be very helpful. When a master transaction is executed, it could slow down the other threads in order to get fully boosted for a short period.

Bulk Move	Strategy	Ops/s	Resize 10MB ms	stalled	freq	Resize 1280MB ms	stalled	freq
10k	baseline	535k	16	63%	3099	2937	67%	3099
10k	stat resizer	547k	15	82%	3999	2666	88%	4000
10k	dyn resizer	547k	15	81%	3980	2691	87%	3987
10k	dyn worker	535k	18	82%	3971	3155	88%	3982
100	baseline	529k	24	66%	3099	4021	68%	3100
100	stat resizer	540k	22	86%	3999	3647	90%	3999
100	dyn resizer	508k	30	56%	3259	4799	59%	3252
100	dyn worker	461k	48	60%	3211	7970	60%	3265
1	baseline	237k	770	72%	3099	103389	72%	3099
1	stat resizer	245k	721	94%	3999	98056	95%	4000
1	dyn resizer	209k	893	62%	3112	120430	63%	3113
1	dyn worker	90k	1886	64%	3111	252035	65%	3113

Table 8: Memcached hash table resize statistics.

5.3 Hash Table Resize in Memcached

Memcached is a high performance caching system based on a giant hash table. While for the normal operation a fine-grained locking scheme is used, the implementation switches to a single global spinlock that protects all accesses to the hash table during the period of a resizing. The resize is done by a separate maintenance thread that moves items from the old to the new hash table and processes a configurable number of buckets per iteration. Each iteration acquires the global lock and moves the items in isolation.

Our evaluation was conducted with Memcached version 1.4.15 and the `mc-crusher` workload generator. We used the default configuration with 4 worker threads that we pinned on 2 modules. The maintenance thread and `mc-crusher` run on their own modules. The workload generator sends a specified number of *set* operations with distinct keys to Memcached, which result in a lookup and insert on the hash table that will eventually trigger several resizes. The hash table is resized when it reaches a size of $2^x \times 10$MB. The cache is initially empty and we insert objects until the 7th resize of $2^7 \times 10$MB (1280MB) is finished.

For the intervals in which the maintenance thread was active, we gathered for the first (10MB) and the last (1280MB) resize interval. These are reported in Table 8: number of items that are moved during one iteration (bulk move, configurable), rate of *set* operations during the entire experiment (ops/s), length of the resize interval (ms), the number of (stalled) instructions and average frequency achieved by the maintenance thread (freq).

We applied the following strategies during the resizing period: *baseline* runs all threads at P_{base}, *stat resizer* runs the maintenance thread at P_{turbo} for the entire period, *dyn resizer*

switches to P_{turbo} only for the length of an bulk move iteration and causes additional transition overheads, *dyn worker* switches to P_{slow} while waiting for the maintenance thread's iteration to finish. The last strategy does not show a performance improvement because the cost cannot be amortized especially when the bulk move size gets smaller. The *stat resizer* shows the best performance because it reduces the resizing duration.

While the benchmark shows the benefit of assigning heterogeneous frequencies, an interesting observation is that the speedup achieved by boosting is limited because the workload is mainly memory-bound. Compared to *baseline*, *stat resizer* shows only a speedup of the resize interval between 7%–9% while it runs at a 22% higher frequency. The higher the frequency, the more instructions get stalled due to cache misses that result from the large working set. The number of stalled instructions effectively limit the number of instructions that can be executed faster at a higher frequency. On the other hand, the high cost of the P-state transitions in the dynamic strategy *dyn resizer* is hidden by an decreased number of stalled instructions but it still cannot outweigh the transition latency. Memcached's default configuration performs only a single move per iteration, which according to our results shows the worst overall duration of the experiment (ops/s). A better balance between worker latency and throughput is to set bulk move to 100. With this configuration, memcached spends 15% of its execution time for resizes, which we can boost by 10%. This reduces the total execution time by 1.5% and allows 1.5% more ops/s because the worker threads spent less time spinning. Combined, this amortizes the additional boosting energy.

5.4 Delegation of Critical Sections

We have shown that critical sections (CS) need to be relatively large to outweigh the latencies of changing P-states. Remote core locking [27] (RCL) is used to dedicate a single processor core to execute all application's CS locally. Instead of moving the lock token across the cores, the actual execution of the critical section is delegated to a designated server. We leverage this locality property by statically boosting the RCL server and eliminate the P-state transition overhead for small CS.

We experiment with three of the SPLASH-2 benchmarks [44] and the accompanying version of BerkeleyDB [33].

We report speedup for all workloads over the single-threaded baseline P-state in Figure 9, and find that we obtain only incremental performance gains for the boosted cases. We show various combinations of worker P-states (reported as "W P*x*") and P-states for the RCL server core ("R P*x*"), and contrast these with configurations where all cores run at P_{base} ("All P2") and $P4_{HW}$ ("All P4") for comparison. Note that we do show standard deviation of 30 trials, but there is hardly any noise visible. We do not reduce the P-state for the waiting workers (due to latency reasons), but it seems there is enough TDP headroom for the brief RCL invocations to run even at $P1_{HW}$ and we get speedups of 4% - 9%. As expected, the relative boost is larger if we start from a lower baseline at $P4_{HW}$.

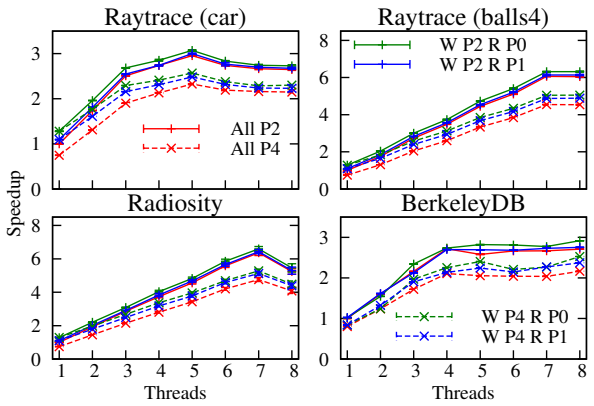

Figure 9: Throughput of SPLASH-2 and BerkeleyDB.

P-state	Intra Module			Cross Module		
Config	0:0	100:10	500:50	0:0	100:10	500:50
All P2	91	169	461	480	570	876
W P2 R P1	83	154	421	468	557	906
W P2 R P0	70	131	357	445	491	772
All P4	123	227	621	578	699	1161
W P4 R P1	83	155	421	519	636	1112
W P4 R P0	70	133	358	417	566	1010

Table 9: Core to core memory transfer latency (ns) for an average round-trip (work iterations: N_{Worker}:N_{RCL}, 0.65ns each).

Overall, scalability of the benchmarks is good, reserving one core exclusively for RCL will cap scalability at 7 (worker) threads. The authors of RCL claim, however, that reserving this single core pays off in comparison to cache coherence traffic arising from spinlock ownership migrating between cores.

Focusing our attention on the CS, we find them to be short (with a peak at ∼488ns) for the selected benchmarks. To better understand the cost of communication and its behavior under various boosting scenarios, we implemented the core of the RCL mechanism, simple cross-thread polling message passing with two threads, in a small micro-benchmark. We report results for select configurations in Table 9 for AMD, which reflect unloaded latency with no competition for communication channels. Overall we were surprised by the round-trip delay when crossing modules, 480ns, vs. 91ns when communicating inside a module (both at P_{base}). Intra-module communication benefits greatly from boosting (91ns vs. 70ns), due to both communication partners and the communication link (shared L2 cache) being boosted. Communicating cross-module, boosting has a smaller performance impact on the communication latency (480ns vs. 445ns, via L3 cache), which helps to explain the small benefit seen in our workloads with short CS.

6 Related Work

The field of DVFS is dominated by work about improving energy efficiency [23, 32, 14]. DVFS is proposed as a mid-term solution to the prediction that, in future processor generations, the scale of cores will be limited by power constraints [11, 7, 2]. In the longer term, chip designs are expected to combine few large cores for compute intensive tasks with many small cores for parallel code on a single heterogeneous chip. Not all cores can be active simultaneously due to thermal constraints [42,

22]. A similar effect is achieved by introducing heterogeneous voltages and frequencies to cores of the same ISA [10]. Energy efficiency is achieved by reducing the frequency and it was observed that the overall performance is only reduced slightly because it is dominated by memory [25] or network latencies.

Semeraro *et al.* [40] propose multiple clock domains with individual DVFS. Inter-domain synchronization is implemented using existing queues to minimize latency, and frequency can be reduced for events that are not on the application's critical path. The energy savings can be extended by profile-based reconfiguration [28, 5]. Another interesting approach to save power is to combine DVFS with inter-core prefetching of data into caches [21]. This can improve performance and energy efficiency, even on serial code, when more cores are active at a lower frequency. Choi *et al.* [8] introduce a technique to decompose programs into CPU-bound (on-chip) and memory-bound (off-chip) operations. The decomposition allows fine tuning of the energy-performance trade-off, with the frequency being scaled based on the ratio of the on-chip to off-chip latencies. The energy savings come with little performance degradation on several workloads running on a single core. Hsu *et al.* [18] propose an algorithm to save energy by reducing the frequency with HPC workloads. Authors also present and discuss transition latencies. A recent study [24] on the Cray XT architecture, which is based on AMD CPUs, demonstrates that significant power savings can be achieved with little impact on runtime performance when limiting both processor frequency and network bandwidth. The P-states are changed before the application runs. It is recommended that future platforms provide DVFS of the different system components to exploit the trade-offs between energy and performance. Our work goes in the same direction, by investigating the technical means to finely control the states of individual cores.

While energy efficiency has been widely studied, few researchers have addressed DVFS to speed up workloads [15]. Park *et al.* [34] present a detailed DVFS transition overhead model based on a simulator of real CPUs. For a large class of multi-threaded applications, an optimal scheduling of threads to cores can significantly improve performance [37]. Isci *et al.* [20] propose using a lightweight global power manager for CPUs that adapts DVFS to the workload characteristics. Suleman *et al.* [41] optimize the design of asymmetric multi-cores for critical sections. A study of Turbo Boost has shown that achievable speedups can be improved by pairing CPU intensive workloads to the same core [6]. This allows masking delays caused by memory accesses. Results show a correlation between the boosting speedup and the LLC miss rate (high for memory-intensive applications). DVFS on recent AMD processors with a memory-bound workload limits energy efficiency because of an increase of static power in lower frequencies/voltages [26]. Ren *et al.* [38] investigate workloads that can take advantage of heterogeneous processors (fast and slow) and show that throughput can be increased by up to 50% as compared with using homogeneous cores. Such workloads represent interesting use cases for DVFS.

Our TURBO library complements much of the related work discussed in this section, in that it can be used to implement the different designs and algorithms proposed in these papers.

7 Conclusion

We presented a thorough analysis of low-level costs and characteristics of DVFS on recent AMD and Intel multi-core processors and proposed a library, TURBO[3], that provides convenient programmatic access to the core's performance states. The current implementation by hardware and OS is optimized for transparent power savings and for boosting sequential bottlenecks. Our library allows developers to boost performance using properties available at application-level and gives broader control over DVFS. We studied several real-world applications for gains and limitations of automatic and manual DVFS. Manual control exposes asymmetric application characteristics that would be otherwise unavailable for a transparent optimization by the OS. Limitations arise from the communication to memory and other cores that restict the IPC. Our techniques, while useful today, also bring insights for the design of future OS and hypervisor interfaces as well as hardware DVFS facilities.

For the future, we plan to add an automatic dynamic tuning mechanism: based on decorated thread control structures, e.g., locks, we can obtain profiling information and predict the optimal frequency for each core. We also envision use cases beyond optimizing synchronization, such as DVFS for flow-based programming with operator placement (deriving the frequency from the load factor) or data routing (basing DVFS on deadlines or priorities). Finally, the TURBO library provides a research testbed to simulate future heterogeneous multi-core processors with fast/slow cores, as well as to evaluate algorithms targeting energy efficiency or undervolting.

Acknowledgements: We thank André Przywara for his help on AMD's P-states and our shepherd Emmett Witchel. This research has been funded in part by the European Community's Seventh Framework Programme under the ParaDIME Project, grant agreement no. 318693.

References

[1] AMD. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2012.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *ACM CACM*, 2011.

[3] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 2012.

[4] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE JSSC*, 2000.

[5] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.

[6] J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova. Evaluation of the intel core i7 turbo boost feature. In *IISWC*, 2009.

[3] `https://bitbucket.org/donjonsn/turbo`

[7] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *ELSEVIER PCS*, 2011.

[8] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *DATE*, 2004.

[9] D. Dice, N. Shavit, and V. J. Marathe. US Patent Application 20130047011 - Turbo Enablement, 2012.

[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.

[11] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[12] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ASPLOS*, 2011.

[13] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[14] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISPLED*, 2007.

[15] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 2008.

[16] D. Hillenbrand, Y. Furuyama, A. Hayashi, H. Mikami, K. Kimura, and H. Kasahara. Reconciling application power control and operating systems for optimal power and performance. In *ReCoSoC*, 2013.

[17] S. Hoppner, H. Eisenreich, S. Henker, D. Walter, G. Ellguth, and R. Schuffny. A Compact Clock Generator for Heterogeneous GALS MPSoCs in 65-nm CMOS Technology. *IEEE TVLSI*, 2012.

[18] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[19] Intel. Intel 64 and IA-32 Architectures Software Developers Manual, 2013.

[20] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO*, 2006.

[21] M. Kamruzzaman, S. Swanson, and D. Tullsen. Underclocked software prefetching: More cores, less energy. *IEEE Micro*, 2012.

[22] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *MICRO*, 2003.

[23] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 2005.

[24] J. H. Laros, III, K. T. Pedretti, S. M. Kelly, W. Shu, and C. T. Vaughan. Energy based performance tuning for large scale high performance computing systems. In *HPC*, 2012.

[25] M. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In *Euro-Par*, 2011.

[26] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *HotPower*, 2010.

[27] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[28] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA*, 2003.

[29] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 1991.

[30] A. Mendelson and F. Gabbay. The effect of seance communication on multiprocessing systems. *ACM TOCS*, 2001.

[31] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS*, 2002.

[32] K. Nowka, G. Carpenter, E. MacDonald, H. Ngo, B. Brock, K. Ishii, T. Nguyen, and J. Burns. A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling. *IEEE JSSC*, 2002.

[33] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley db. In *USENIX ATC*, 1999.

[34] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE TCAD*, 2012.

[35] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *MobiCom*, 2001.

[36] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational Sprinting on a Hardware/Software Testbed. In *ASPLOS*, 2013.

[37] B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu. Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In *DATE*, 2013.

[38] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity for interactive services. In *ICAC*, 2013.

[39] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 2012.

[40] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA*, 2002.

[41] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

[42] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, 2010.

[43] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. Fastlane: improving performance of software transactional memory for low thread counts. In *PPoPP*, 2013.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.