TECHNISCHE UNIVERSITÄT DRESDEN
Fakultät Informatik - Institut für Systemarchitektur

TECHNICAL REPORT

ISSN 1430-211X

TUD-FI13-02 November 2013

Selective Core Boosting: The Return of the Turbo Button

Jons-Tobias Wamhoff
Stephan Diestelhorst
Christof Fetzer

Technische Universität Dresden

Patrick Marlier
Pascal Felber

Université de Neuchâtel

Dave Dice

Oracle Labs

Technische Universität Dresden
Institut für Systemarchitektur
Lehrstuhl für Systems Engineering
Fakultät Informatik
01062 Dresden, Germany
http://se.inf.tu-dresden.de/

Selective Core Boosting: The Return of the Turbo Button

Jons-Tobias Wamhoff
Stephan Diestelhorst
Christof Fetzer
Technische Universität Dresden, Germany
first.last@tu-dresden.de

Patrick Marlier
Pascal Felber
Université de Neuchâtel, Switzerland
first.last@unine.ch

Dave Dice
Oracle Labs, USA
first.last@oracle.com

Abstract

Several modern multi-core architectures support the dynamic control of the CPU's clock rate, allowing processor cores to temporarily operate at speeds exceeding the operational base frequency. Conversely, cores can operate at a lower speed or be disabled altogether to save power. Such facilities are notably provided by Intel's Turbo Boost *and AMD's* Turbo CORE *technologies. Frequency control is typically driven by the operating system which requests changes to the performance state of the processor based on the current load of the system.*

In this paper, we investigate the use of dynamic frequency scaling from user space to speed up multi-threaded applications that must occasionally execute time-critical tasks or to solve problems that have heterogeneous computing requirements. We propose a general-purpose library that allows selective control of the frequency of the cores—subject to the limitations of the target architecture. We analyze the performance trade-offs and illustrate its benefits using several benchmarks and real-world workloads when temporarily boosting selected cores executing time-critical operations. While our study primarily focuses on AMD's architecture, we also provide a comparative evaluation of the features, limitations, and runtime overheads of both Turbo Boost and Turbo CORE technologies. Our results show that we can successful exploit these new hardware facilities to accelerate the execution of key sections of code (critical paths) improving overall performance of some multi-threaded applications [11]. Unlike prior research, we focus on performance instead of power conservation. Our results further can give guidelines for the design of hardware power management facilities and the operating system interfaces to those facilities.

1. Introduction

While early generations of multi-core processors were essentially homogeneous with all cores operating at the same clock speed, new generations of CPUs 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, 33].

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* [37] and *Turbo CORE* [3]. While the former can dynamically adjust the clock speed of individual cores and also power them down completely, the latter can only change the frequency and voltage of groups of cores at a time. Despite the differences in features and implementation of these technologies, they both allow the system to temporarily boost the performance of selected cores.

Boosting is typically controlled by hardware and is completely transparent to the operating system and applications. Yet, it is sometimes desirable to be able to finely control these features from an application as needed. Examples include: speeding up critical sections to reduce the time window during which other threads have to wait; boosting a high-priority thread that must meet a nearing deadline; minimizing the completion and waiting times of algorithms running concurrent tasks with different computing requirements; 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.

To meet these needs, we have designed and implemented a general-purpose library for programmatically controlling the speed of the cores of CPUs with AMD's Turbo CORE and Intel's Turbo Boost technologies. Both implementations are subject to several limitations (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), and the cost of switching frequencies is subject to important variations depending on the method used for modifying processor states and the specific change requested. The publicly available documentation is scarce, and we believe to be the first to publish an in-depth investigation on the behavior, performance, and limitations of these technologies. Based on these findings, we develop our library, which we named TURBO, to abstract away from the low-level differences and complexities. In a sense, this library is the software counterpart of the physical "turbo button" available in some early personal computers to manually switch the operating frequency of the processor.

Based on our findings, we provide the following recommendations for processor and operating system designers:

- The control of frequency scaling should be made available through platform-independent application program interfaces (APIs) or instructions.
- These APIs or instructions should include the ability to read or query frequency transition costs for building a cost model that allows DVFS-aware code to adapt at runtime.
- Ideally, the processor should support frequency scaling individually for each core. The Intel design allows boosted

frequencies only if some cores are inactive, while the AMD design proved to be more flexible since it allows to keep cores active at a lower frequency when others are boosted but the granularity is limited to pairs of cores. Additionally, for some workloads it would be beneficial to efficiently set the frequency for remote cores in order to have local boosting control.

- The frequency transition should be as fast as possible so that short boosted sections can already amortize the transition cost.
- The processor should support a more efficient frequency control from user space, i.e., eliminate the latency that results from the current requirement to switch into kernel space to access platform-specific power management devices. The operating system should also support a kernel parameter that disables all automatic frequency scaling.
- The frequency transitions should be asynchronous, triggered by a request and not blocking. Currently, the transition to a lower frequency blocks the AMD processor core until it is finished. When moving to a higher frequency, the core remains operational.
- The integrated voltage controller should support per-core voltage control to provide higher energy savings with lower frequencies instead of using the highest voltage for all cores.
- The operating system should keep the current frequency in the thread context to better support context switches and thread migrations.
- Ideally, the operating system would expose a new set of advisory platform-independent APIs to allow threads to set their desired DVFS-related performance targets. Furthermore, the operating system kernel (and potentially a virtual machine hypervisor) would moderate potentially conflicting DVFS resource requests from independent and mutually unaware applications.

While the hardware and operating system are currently tuned to use frequency scaling to boost sequential bottlenecks, our evaluation shows that manual frequency scaling with additional application knowledge has great benefit for applications with heterogeneous workload distributions. With automatic frequency scaling, these opportunities cannot be fully exploited and hence, the application is not able to reach the maximum available performance.

In short, the contributions of this paper are fourfold: *(i)* a comparative analysis of Intel's Turbo Boost and AMD's Turbo CORE technologies, including a description of the frequency and operational configurations of the processors used in our evaluation (Section 2); *(ii)* the design and implementation of the TURBO library that provides primitives for programmatically controlling the frequency and voltage of the cores from user space (Section 3); *(iii)* an in-depth evaluation of transition latencies and strategies for switching processor states that we use to derive a simplified model (Section 4); and *(iv)* four case studies that show the performance gains and trade-offs with real-world benchmarks (Section 5).

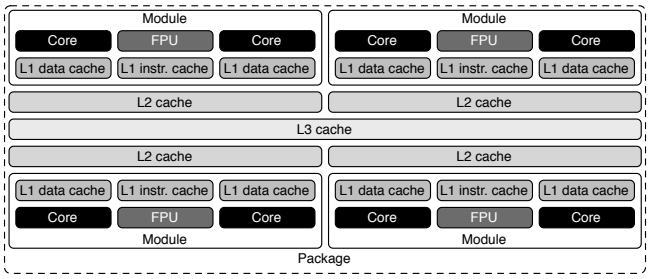

Figure 1: Organization of an AMD FX-8120 processor.

2. Hardware Support

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 dynamic voltage and frequency scaling (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. Additionally, *C-states* are used to save energy when a core is idle, with C0 being the normal, non-idle state, and C1-C*n* representing idle states with different levels of power saving and wakeup latency. C-states trade entry/wakeup latency for lower power draw while sleeping, entry and exit to deep C-states can take on the order of milliseconds. The number and nature of these states differ depending on the processor and are discussed in the rest of the section.

The operating system can invoke sleep states through various means such as the `hlt` and `monitor/mwait` instructions. P-states can be controlled from the operating system through special machine-specific registers (MSRs) that are accessed through the `rdmsr/wrmsr` instructions. The operating system can request a P-state change by modifying the respective MSR. P-state changes, also, are not instantaneous: current needs to change and frequencies are ramped, both taking observable times.

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

2.1. AMD's Hardware

The architecture of the AMD FX-8120 processor is illustrated in Figure 1. Processor cores are organized by pairs in *modules* that share parts of the logic between the two cores. Power draw of the processor can be controlled via P-states and C-states. The Turbo CORE hardware extensions are documented as part of the BIOS and kernel developer's guide [1].

Our processor supports seven P-states summarized in Table 2. The two topmost are boosted P-states that are controlled by the hardware. The remaining five P-states can be set by the operating system through the MSRs, and the numbering in software differs from the hardware P-states: $P_{HW} = P_{SW} + NumBoostStates$.

The net effect of these changes is that the operating system cannot directly set the highest available hardware P-state but instead hardware will automatically *boost* the frequency beyond the nominal P-state if operating conditions permit. The

	AMD FX-8120	Intel i7-4770
Model	AMD Family 15h Model 1, codename "Bulldozer"	Intel Core 4th generation, codename "Haswell"
Design	4 modules with 2 cores, 2 ALUs, 1 FPU	4 cores with hyper-threading (8 threads)
L1 data cache	8x 16KB, 4-way, write-through, per core	4x 32KB, 8-way, per core
L1 instruction cache	4x 64KB, 2-way, per module	4x 32KB, 8-way, per core
L2 cache	4x 2MB, 16-way, per module	4x 256KB, 8-way, per core
L3 cache	1x 8MB, 64-way, per package	1x 8MB, 16-way, per package
Cache latency	3 cycles L1, ∼18 cycles L2, ∼65 cycles L3	4 cycles L1, 11-16 cycles L2, 30-55 cycles L3
Memory	DDR3 1866 MHz, integrated memory controller	DDR3 1600 MHz, integrated memory controller
Base frequency	3.1GHz	3.4GHz
Domain frequency range	1.4–4.0GHz (100MHz stepping)	0.8–3.9GHz (100MHz stepping)
Domain voltage range	0.875–1.412V (3.412–24.577W)	1.65–1.86V
Thermal design power	125W	84W

Table 1: Specification of the AMD FX-8120 and Intel i7-4770 CPUs.

Hardware P-state	P0	P1	**P2**	P3	P4	P5	P6
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

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

processor determines the current power consumption and will switch to hardware P1 state, if the total power draw remains within the thermal design power (TDP) limit, and the operating system requests the highest (software) P-state. The highest boosted P-state, hardware P0, is entered automatically if some cores have furthermore reduced their power consumption by entering a deep C-state. During a P-state change, the processor remains active and capable of executing instructions, and the completion of a P-state transition is indicated in a MSR available to the operating system.

Due to the pairwise organization of processor cores in modules, the effect of a P- and C-state change depends on the state of the neighboring core. While neighboring cores can control P-states independently, the lowest (i.e., fastest) P-state of the two cores will apply to the entire module. Since the `wrmsr` instruction can only be executed on the current core, it can only have full control over the frequency scaling if the other core is running at the highest (i.e., slowest) P-state. The processor consumes always the voltage defined by the fastest active P-state (see Table 2).

On selected AMD processors, the number of hardware-reserved P-states can be controlled by changing *NumBoostStates* through a configuration MSR. In our work we set $NumBoostStates = 0$ and thus can achieve full control over boosting. The core safety mechanisms are still in effect: the hardware only enters a boosted P-state if the TDP limit has not been reached.

2.2. Intel's Hardware

Intel's boosting technology is largely similar to AMD's. It mainly differs in the granularity for setting frequencies of individual cores and the level of control for changing them from software. Current Intel CPUs use the same frequency for all active cores. Each core can request different P-states, i.e., frequencies but all cores will run at the highest possible requested frequency. Modern CPUs in the "Haswell" family can adjust the performance state with 100MHz steps inside the range of the operating frequencies. The i7-4770 processor used for our experiments supports frequencies from 800MHz to 3900MHz corresponding to 32 P-states, out of which 5 are boosted.

Intel's *Turbo Boost* technology supports boosting of some of the cores. While one can request a boosted P-state for an individual core, there is no guarantee that it will effectively run boosted. Indeed, boosted P-states are controlled by hardware. The maximum boosting ratio can be adjusted depending on the number of active cores using C-states set with a specific MSR, and the boost level is determined depending on the current TDP. Therefore, the more cores are sleeping, the more of the remaining cores can be boosted.

A main difference in the designs of AMD and Intel's technologies is that the granularity of C-states is at the level of a whole module for the former, whereas the latter allows powering down individual cores. Another notable difference is that Intel does not permit that only some of the cores operate at a slower frequency (i.e., they all run at the highest requested frequency). The reasoning behind this design choice is that it is in general more effective to process tasks as fast as possible, and then put some cores to sleep using C-states. 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. One should also note that the boosting ratio is higher on AMD processors, with announced plans to reach frequencies up to 5GHz for boosted cores.

3. Turbo Library

The TURBO library, written in C++, provides components for low-level and hardware-centric multi-threaded programming. Our focus is not to provide mechanisms for automatic tuning of applications but a set of abstractions that make it convenient to improve highly optimized software based on frequency scaling. The abstractions allow us to configure the underlying processor cores and to map workloads to specific cores.

3.1. Architecture and Components

Figure 2 illustrates the components of the TURBO library. The lowest level presents abstractions for the interfaces of the underlying AMD and Intel hardware, as well as the Linux

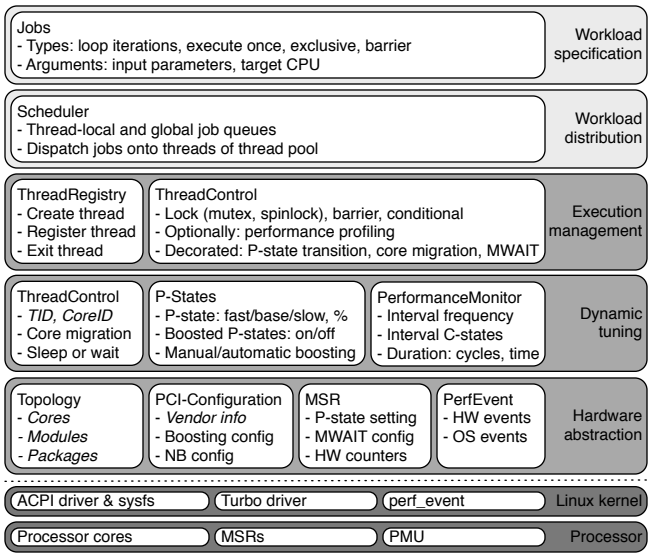

Figure 2: Library components for low-level multi-threaded programming.

operating system. 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` and `rdmsr` instructions to be executed on the current processor core. Additionally, it allows kernel space latency measurements, e.g., for P-state transition time, with more accuracy than from user space.

The TURBO library needs to be aware of all threads that are used by an application. Threads can be created or registered using the *thread registry* if the application explicitly manages them. The threads must be pinned to the processor cores according to the topology of the hardware because the TURBO library maintains the current configuration of the P-states, which is specific to a core. When the library migrates threads to other cores, the cached P-state configuration is updated.

The components for dynamic tuning provide the functionality for managing frequency scaling. The library currently provides different levels of abstraction. The lowest level captures the hardware interface and allows direct P-state control. The higher levels provide convenient abstractions that deal with P-states implicitly, e.g., mutex wrappers decorated with boosting or jobs with assigned priority. During initialization, the boosting capabilities can be configured for the entire processor (for details see Section 3.2). At runtime, the threads can request the executing processor core to run at the lowest (fastest), base or highest (slowest) P-state. Alternatively, the P-state can be specified in percent based on the fastest P-state. The actual P-state is derived from the selected boosting setup. 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 but is already set or cannot be supported by the processor policy, then the operation has no effect. Threads can also request to temporarily migrate to a dedicated processor core that runs at the highest possible frequency and always stays fully operational (C0-state).

An important task for highly optimized applications is to identify sections that can benefit from frequency scaling. Therefore, the TURBO library provides wrappers for locks, barriers, and condition variables for execution management. The wrappers can be decorated with profiling capabilities of the performance monitor, which in turn allows us to measure the frequency and C-state during an interval defined by the wrapper. The interval of the wrapper's states is measured in time and cycles, e.g., to analyze the properties of a critical sections such as the average number of cycles or stalls. The performance monitor uses the `aperf/mperf` and `tsc` counters [1] of the processor and the `perf_event` facilities of the Linux kernel to access the processor's performance monitoring unit (PMU).The wrappers can also be decorated to implicitly request P-state transitions, e.g., to run at the lowest frequency while the lock is busy and at the fastest one while holding the lock.

The workload specification and distribution provide optional components that can be used as a testbed for heterogeneous programming with dynamic frequency scaling. The basic entity of work is a job with properties such as a type, input parameters, and a target core. The supported types of jobs include: execute once, run as loop body, execute exclusive on all cores, or wait on a barrier. Our design goal was to provide an API that allows developers to communicate precisely what to execute on which processor core and at which time. This is in contrast to most parallel frameworks that try to automatically introduce parallelism without providing control on how a workload is mapped to the underlying hardware. The scheduler maintains the constraints of the specified jobs and dispatches them onto the threads for execution. For that purpose, it uses internal queues that provide the mapping of work to the processor cores.

All components for the hardware abstraction and dynamic tuning can be used individually to adapt the TURBO library to existing infrastructures that use multiple threads or processes.[1]

3.2. Processor and Linux Kernel Setup

The default configurations of the Linux kernel and AMD processors contain mechanisms that manage the frequency scaling automatically. The Linux governor will adapt the P-states based on the current processor utilization ("ondemand") or based on static settings that are enforced periodically ("performance", "powersave", "userspace"). The boosted P-states are managed by the processor itself within the TDP.

We must disable the influence of the governors and the processor in order to gain explicit control of the P-states and boosting in user space using our library. Note that he "userspace" governor provides an alternative P-state interface but introduces higher latencies and gives no control over boosted P-states. Therefore, we disable the CPU frequency driver (`cpufreq`) and turn off AMD's *PowerNow* speed throttling technology in the BIOS. We then set the number of

[1] Applications that make use of multiple processes are currently not supported but the library can be extended by keeping all configuration in memory shared between the processes.

boosted P-states to 0 so that the P-state numbering in hardware and software match ($P0_{HW} = P0_{SW}$) and disable the application power management (APM). This allows us to control all available P-states in software within the processor's enforced TDP policy. Changing the number of boosted P-states also changes the frequency of the `tsc` counter for AMD processors so we therefore disable `tsc` as a clock source for the Linux kernel and instead use `hpet`. In a production system, these tweaks will obviously become unnecessary or be performed automatically.

Each P-state sets a specific voltage and frequency identifier. Before beginning a frequency transition, the processor sends a request to the voltage regulator according to the configuration. The processor has a configurable delay before sending such requests, which we set to the minimum possible value of 32μs. During the P-state transition, the processor core remains fully operational.

The processor additionally applies automatic frequency scaling for the northbridge [9] that can have a negative impact on memory access times for boosted processor cores. Therefore, northbridge P-states are disabled and the chip always runs at the highest possible frequency.

Linux uses the `monitor` and `mwait` instructions to idle processor cores and allow them to change the 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, we enable these processor instructions for user space and disable the use of `mwait` in the kernel to avoid progress failure scenarios. 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-state. We set the maximum C-state for the Linux kernel to 0 and use the polling idle mode. Again, these changes are required in our prototype for the evaluation of `mwait`, which requires C-state control from the user space, and should not be necessary in a production system.

The presented setup highlights the importance of the configuration of both the hardware and the operating system for sound benchmarking. All modifications of the Linux kernel are done using kernel parameters and no changes to its source code are required. For experiments with our AMD processor, we used a P-state configuration of P2 for all cores to enforce the base operational frequency without latencies due to C-state transitions. We also stopped other sources of unpredictability, e.g., all periodic `cron` jobs.

The setup with the Intel processor is similar to AMD. We also disable the performance governor and the CPU frequency driver in the Linux kernel. Since Intel has not a fixed range of P-state, we adapted to match the AMD convention. P0 corresponds to the maximum boosted state (denoted by "Turbo" in experiments on Intel), P2 is the maximum non-turbo state ("Max") and P6 is the minimum frequency state ("Min"). The maximum boosted state is controlled automatically and depends on 4 criteria: number of active cores, estimated current consumption, estimated power consumption, and processor temperature. We did not run experiments with `monitor-mwait` on Intel because these instructions cannot be enabled in user-space mode.

4. Evaluation

On top of the TURBO library presented in Section 3, we implemented a set of benchmark applications that profile and configure the underlying processor. The applications measure the latency of P-state transitions and compare different strategies for the P-state configuration, both statically and dynamically.

4.1. P-State Transition Latency

We first investigate the overheads that are introduced by the dynamic control of the frequency scaling. For a better understanding, we first measure the cost of the different system calls used for configuration and then investigate the latencies of P-state transitions in isolation. We present the results for AMD in Table 3 and for Intel in Table 4.

Operation	P-State Transition	Mean Cycles	Mean ns	Deviation Cycles	Deviation ns
`syscall(getpid)`	—	939	234	68	17
`ioctl(trb)`	—	1173	293	68	17
`pread(msr,mperf)`	2 — 2	2391	597	84	21
`ioctl(trb,mperf)`	2 — 2	1475	368	98	24
`pread(msr,pstate)`	—	3448	862	114	28
`ioctl(trb,pstate)`	—	2541	635	125	31
`pwrite(msr,pstate,2)`	$2 \rightarrow 2$	3352	838	130	32
`ioctl(trb,pstate,2)`	$2 \rightarrow 2$	2077	519	108	27
`wrmsr(pstate,6)`	$2 \rightarrow 6$	28087	7021	105	26
`wrmsr(pstate,6)&wait`	$2 \rightarrow 6$	29783	7445	120	30
`wrmsr(pstate,0)`	$6 \rightarrow 0$	1884	471	35	8
`wrmsr(pstate,0)&wait`	$6 \rightarrow 0$	226988	56747	84	21
`wrmsr(pstate,2)`	$0 \rightarrow 2$	23203	5800	36	9
`wrmsr(pstate,2)&wait`	$0 \rightarrow 2$	24187	6046	139	34
`wrmsr(pstate,2)`	$6 \rightarrow 2$	1898	474	79	19
`wrmsr(pstate,2)&wait`	$6 \rightarrow 2$	183359	45839	130	32
`wrmsr(pstate,0)`	$2 \rightarrow 0$	1007	251	106	26
`wrmsr(pstate,0)&wait`	$2 \rightarrow 0$	94762	23690	138	34
`wrmsr(pstate,1)`	$2 \rightarrow 1$	1006	251	130	32
`wrmsr(pstate,1)&wait`	$2 \rightarrow 1$	95234	23808	68	17
`wrmsr(pstate,2)`	$1 \rightarrow 2$	23597	5899	33	8
`wrmsr(pstate,2)&wait`	$1 \rightarrow 2$	24574	6143	138	34
`monitor&mwait`	—	1698	424	95	23
`pthread_setaffinity`	—	26728	6682	49	12

Table 3: Overheads of system calls and latencies of P-state transitions measured during 100,000 runs (AMD FX-8120).

System calls for device-specific `ioctl` input/output operations are only slightly more expensive than regular system calls, e.g., to get the process identifier. `Ioctl` calls 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` and `pwrite`. Reading the current P-state is implemented in the Linux kernel using the `rdmsr` instruction. The difference in speed between `msr` and `trb` results mostly from additional security checks and indirections that we streamlined for the TURBO driver. The cost in time for all system calls depends on the P-state, i.e., reading the current

Operation	P-State Transition	Mean Cycles	Mean ns	Deviation Cycles	Deviation ns
`syscall(getpid)`	—	1246	366	118	34
`ioctl(trb)`	—	1266	372	98	28
`pread(msr,mperf)`	—	1602	471	113	33
`ioctl(trb,mperf)`	—	1351	397	85	25
`pread(msr,pstate)`	—	2236	657	91	26
`ioctl(trb,pstate)`	—	1975	580	92	27
`pwrite(msr,pstate,Max)`	$2 \rightarrow 2$	3635	1069	85	25
`ioctl(trb,pstate,Max)`	$2 \rightarrow 2$	3299	970	84	24
`wrmsr(pstate,Max)`	$6 \rightarrow 2$	6485	8106	103	128
`wrmsr(pstate,Max)&wait`	$6 \rightarrow 2$	63366	79207	122	152
`wrmsr(pstate,Min)`	$2 \rightarrow 6$	2008	590	50	14
`wrmsr(pstate,Min)&wait`	$2 \rightarrow 6$	71998	21175	98	28
`wrmsr(pstate,Turbo)`	$2 \rightarrow 0$	2008	590	42	12
`wrmsr(pstate,Turbo)&wait`	$2 \rightarrow 0$	311951	90571	381	112
`monitor&mwait`	—	636	163	124	32
`pthread_setaffinity`	—	15784	4642	71	20

Table 4: Overheads of system calls and latencies of P-state transitions measured during 100,000 runs (Intel i7-4770).

P-state scales with the selected frequency. The presented measurements were executed at the base operational frequency.

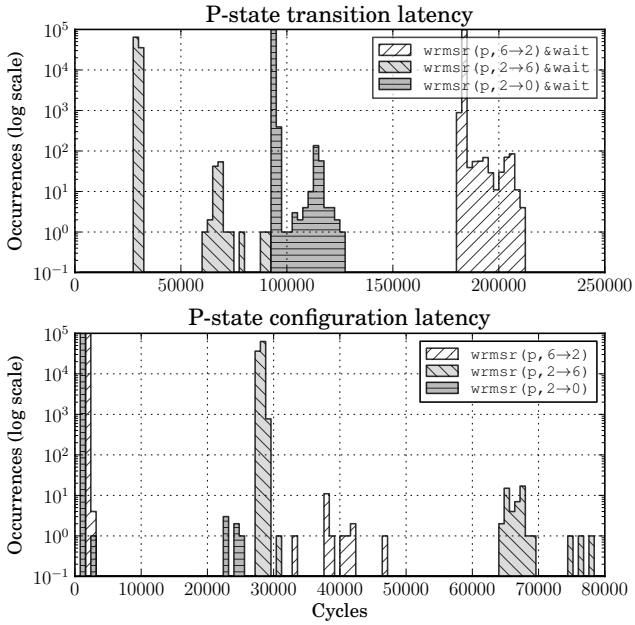

Figure 3: Latency for P-state transitions and P-state configurations measured for 100,000 executions (AMD FX-8120).

The TURBO driver also allows us to measure the latencies of P-state transitions in kernel space, removing the inaccuracy due to system call overheads. A P-state transition is initiated by writing the desired value into the P-state MSR of the current processor core using `wrmsr`. We measured the cost of the `wrmsr` instruction itself, as well as the latency until the P-state transition is finished, by busy waiting until the frequency identifier of the P-state is set in the status MSR. In addition to the averages in Table 3, Figure 3 shows the distribution for runs on the AMD processor. Requesting a P-state lower than the current one (i.e., going faster) has low overhead in itself, but the entire transition triggered by the request has a high latency due to the time the voltage regulator takes to reach the target voltage. Unfortunately, no details are given by the manufacturers. The distribution shows in log-scale that the latency is at least predictable and has only few outliers. We observe, in-line with the manuals, that the core remains fully operational during the transition.

The request to switch to a higher P-state (i.e., going slower) has almost the same latency as the entire transition, i.e., the core is blocked during most of the transition. We suspect that this may be caused by the need to coordinate with the other core in the module to see if an actual P-state change will occur. We believe that the handshake with the remote core is slow and causes the long blocking time. 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 high P-state for a short period, the transition to a lower P-state will be faster if the voltage was not dropped completely. Note that the frequency and voltage identifiers are module-wide settings, i.e., the fastest P-state of both cores determines the frequency. While a core can only move to a slower P-state if the other core has already a slower P-state, transitions to faster P-states can be driven by a single core.

On the Intel CPU, total latency results are very similar: going to a faster P-state also takes tens of microseconds, depending on distance between the current and the requested P-state. A significant different to the AMD system, however, lies in the faster execution of the `wrmsr` instruction when transitioning from a fast to a slow P-state (approx. 2000 cycles, 590 ns). The AMD system exhibited surprisingly long durations here (up to 29800 cycles, 7450 ns), and we believe the Intel numbers are lower because the CPU does not need to perform additional coordination when switching to a slow P-state.

For the evaluation of P-state configuration strategies in Section 4.2, we show the cost for `mwait` and `pthread_setaffinity`. In the `mwait` experiment, one core of a processor module 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. The `pthread_setaffinity` function migrates a thread to a core on a different processor module that is already in C0 state (no shared L2 cache) 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.

4.2. P-State Transition Strategies

We evaluate various configuration strategies for P-states using an application that spends all its time in critical sections. The critical section is protected by single global spin-lock implemented as an MCS queue lock [29]. The workload increments a local counter for a configurable number of iterations ($\sim$10 cycles each) and uses only the integer cores (shared FPU remains idle). While the global lock prevents any parallelism, the goal of the concurrent execution is to find the length of

critical sections that amortizes the P-state transition latency.

Each processor core is assigned to one thread using the TURBO library. The critical section with the counter iterations forms a job that is dispatched to each thread using the scheduler. The P-states and C-states are configured by the application according to the following static and dynamic strategies:

`stat single`: The P-states are set statically during the initialization of the application. Only a single thread executes critical sections at the given P-state ($P_{CS} \leftarrow$) while the other threads run at a possibly different P-state ($\rightarrow P_{wait}$) in C0-state. This provides the baseline for different P-state configurations without P-state change request overheads.

`stat multi`: All threads execute the critical section serialized by the global lock. The P-state is set to the same value for all threads at the base operating frequency. This shows the overhead introduced by the global lock.

`dyn owner`: All threads are initially set to the slowest P-state and dynamically switch to the fastest P-state while holding the lock, and back when releasing it.

`dyn wait`: All threads are initially set to the fastest P-state and dynamically switch to the slowest P-state while waiting for the lock, and speeding up when acquiring the lock.

`stat migrate`: Only six threads on three processor modules execute critical sections. The remaining module runs at the fastest P-state and the current lock owner migrates to a core of the boosted module until it releases the lock.

`dlgt owner`: Only one thread per processor module is executing critical sections and delegates the P-state switching to the thread executing on the neighbouring core. The strategy is otherwise the same as `dyn owner`.

`dlgt wait`: Same delegation strategy as `dlgt owner` but adapted for `dyn wait` instead of `dyn owner`.

`stat mwait`: While waiting for the MCS queue lock, the threads busy wait on a local memory location. We `monitor` this location and halt the processor using `mwait` until the lock becomes available. All processor cores run initially at the base operating frequency.

Figures 4 and 5 show the results for all strategies on AMD and Intel CPUs (except `stat mwait` that is not supported with the latter). We run the benchmark for 100 seconds and count the number of critical sections that were executed. In a second step, we measure the number of cycles per iteration in the critical section, that is, how many cycles of real work were executed. Based on the run time and cycles multiplied by the number of critical sections executed, we calculate the effective frequency achieved for doing real work inside the critical section. The frequency is affected by the P-state configuration and overheads according to the selected boosting strategy and synchronization. We ran the measurements both for automated boosting with the boosted P-states managed by the processor ($P_{auto} = 2$: $P2_{HW} = P0_{SW}$ on AMD) as well as manual boosting for AMD with all P-states managed in software using the TURBO library ($P_{auto} = 0$: $P0_{HW} = P0_{SW}$ on AMD).

The static strategies execute all critical sections on a single processor core (`stat single`). The figure shows for the two boosting setups the achieved effective frequency when (1) all other cores run at the slowest P-state and the single core is boosted, (2) all cores run at the base operating frequency and (3) all cores run at the slowest P-state. The curves reflect the optimal performance for the benchmark running at a given frequency. Overhead is only introduced by locally cached lock acquisitions. Note that automatic boosting will enter only $P1_{HW}$ as long as the majority of other cores are in C0-state. Our manual boosting setup does not suffer from this limitation. The lock acquisition consumes cycles outside the critical section so that small workload sizes cannot reach the effective frequency defined by the selected P-state.

The `stat multi` strategy shows the synchronization overhead when multiple threads try to acquire the global lock that serializes the execution of critical sections among all cores. All cores run at the base operating frequency. Larger sizes of critical sections reduce the impact of the lock acquisition and a performance similar to `stat single` is achieved.

The dynamic strategies introduce overhead in addition to the synchronization costs because two P-states transitions must be requested for each critical section. This overhead is amortized when the resulting effective frequency of the critical section is above the operational base frequency of `stat single` (starting at $\sim$100,000 cycles for manual boosting on the AMD system). On the AMD system, requesting the fastest P-state for all cores (`dyn wait`) provides best results, because threads can reacquire the lock without needing to transition to a slower P-state. Other threads trying to acquire the lock will perform their long blocking switch to the slower P-state (see Section 4.1) when they find the lock acquired before enqueuing at the MCS lock. Boosting the lock owner (`dyn owner`) must perform this transition within its execution loop before it can acquire the lock in the next iteration.

Both delegating strategies (`dlgt owner` and `dlgt wait`) provide performance close to `dyn wait` but only half of the processor cores can be used. In both cases, the slow transition to the slowest P-state does not halt the execution, in contrast to `dyn owner`.

The thread migration provided by the Linux kernel (`stat migrate`) is the strategy with the highest overhead. 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 on a slow core will be executed slowly until the thread reaches the boosted core.

Instead of adjusting the P-states, the `stat mwait` strategy performs C-state transitions. It has the benefit of enabling $P0_{HW}$ for automatic boosting because the P-states are managed by the processor. Still, the performance does not reach $P0_{HW}$ due to the high transition latencies. With manual boosting, changing the C-state leaves the P-state at its original value, which prevents the boosting.

Comparing these results to the Intel system, we find that on the Intel machine synchronization overheads are much lower (see `stat multi`). In addition, the system is able to boost already without manual intervention when threads are busy waiting on the lock variable. Hardware seems to be able to detect that data-dependent loop and allow boosting with it. Also, we find that we cannot extract the maximum performance

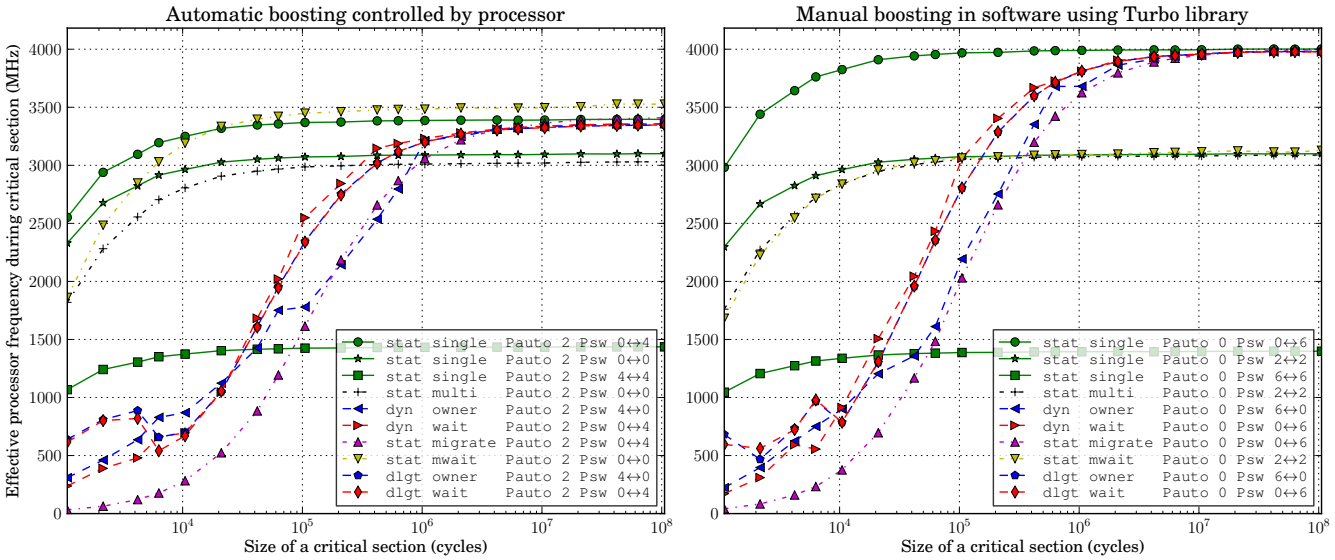

Figure 4: Strategies for P-state transitions requests with automatic boosting by the processor and manual boosting in software using the TURBO library (AMD FX-8120).

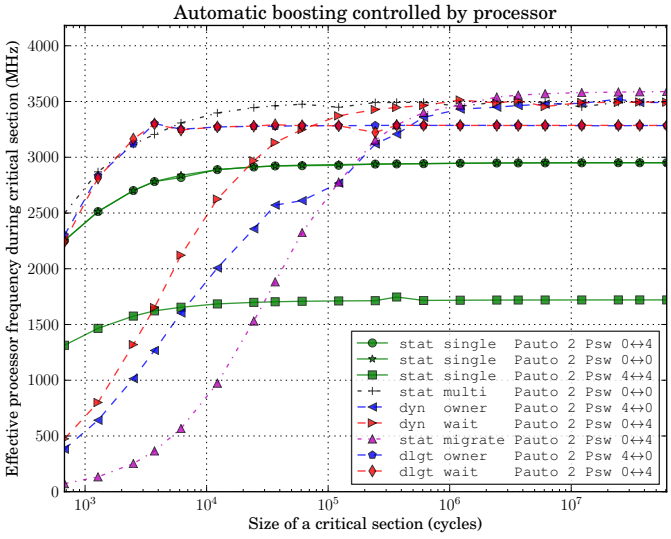

Figure 5: Strategies for P-state transitions requests with automatic boosting by the processor and manual boosting in software using the TURBO library (Intel i7-4770).

of 3.9 Ghz, likely because not enough cores are in the required C-state. The dynamic policies (`dyn owner & wait`) benefit from the shorter execution time for the fast → slow P-state transition. Finally, the delegation policies (`dlgt owner & wait`) improve further upon that because of the fully removed MSR accesses from the critical section execution and the low synchronisation overheads.

4.3. Cost Model

Based on our experimental results, we derive a simplified and pessimistic cost model for AMD's boosting implementation to guide developers when boosting pays off. We first present a cost model for boosting sequential bottlenecks that formalizes the results from Section 4.2. We then specialize it for boosting critical sections that are not a bottleneck as well as for workloads that contain periods with heterogeneous workload distributions.

We use for the model hardware P-state numbering (P_{HW}) and 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 cost related to thread synchronization; (3) the frequency ramps linearly from a higher P-state (e.g., f_{P6}) to a lower P-state (e.g., f_{P0}); and (4) the frequency transition to a higher 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 estimatation 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 and latency and operation frequency).

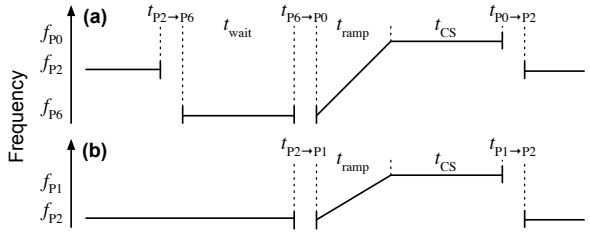

Figure 6: P-state configurations for boosting (a) sequential bottlenecks and (b) critical sections.

For sequential bottlenecks, we follow the strategy `dyn owner` described in Section 4.2 and illustrated in Figure 6: the application threads run normally at the base operating frequency (f_{P2}), threads wait to enter the critical section in f_{P6} and execute the critical section boosted at f_{P0}. Boosting will

pay off if we outperform the critical section that runs at f_{P2}:

$$t_{CS_{f_{P0}}} \leq t_{CS_{f_{P2}}}$$

The minimal length t_{CS} for critical sections must be greater than the combined P-state configuration latencies and the number of cycles that are executed during the P-State transition ($cycles_{ramp}$) to the boosted frequency (f_{P0}):

$$t_{CS} \geq t_{P6\to P0} + t_{ramp} + t_{P0\to P2} + \frac{cycles_{CS} - cycles_{ramp}}{f_{P0}}$$

Based on the P-state transition behavior that we observed in Section 4.2, we can compute the minimal critical section length as follows:

$$t_{CS} \geq \frac{f_{P0}}{f_{P0} - f_{P2}} \cdot (t_{P6\to P0} + t_{P0\to P2}) + \frac{1}{2} \cdot \frac{f_{P0} - f_{P6}}{f_{P0} - f_{P2}} \cdot t_{ramp}$$

The minimal wait time t_{wait} to acquire the lock should simply be larger than the time to drop to f_{P6}: $t_{wait} \geq t_{P2\to P6}$. With our measured results, on the AMD CPU this equals to a minimal critical section length of ∼439,369 cycles (∼110μs). Note that our cost model reflects a pessimistic result and other strategies can reach the break even point already earlier (see Figure 4), e.g., if the transition request is delegated to a thread running on the other core of the module.

Based on the above cost model for boosting sequential bottlenecks, we can derive a cost model for boosting critical sections by one step (i.e., to f_{P1}):

$$t_{CS} \geq \frac{f_{P1}}{f_{P1} - f_{P2}} \cdot (t_{P2\to P1} + t_{P1\to P2}) + \frac{1}{2} \cdot t_{ramp}$$

We never move below the base operating frequency and boosting pays off if the critical section is longer than ∼326,450 cycles (∼82μs).

On the Intel system, the cost model is very similar, but instead of the long inactive time $t_{P0\to P2}$ at the end of the critical section, this time is shorter and has an additional frequency ramp from f_{P0} to f_{P2}. The net effect is that the length for critical sections can be shorter.

So far, we looked at the possibility to boost sequential bottlenecks or critical sections. Another interesting target that is currently not exposed by means of automatic boosting are periods of heterogeneous workload distributions. We can apply our cost model to workloads where one thread temporarily runs at a higher priority than other active threads or the workload has an asymmetric distribution of accesses to critical sections from threads. Typically, such periods are longer because they combine several critical section, thus improving the chances of amortizing the cost of switching on boosting. Based on the presented cost model, we compute the minimal duration of such periods instead of the critical section length. We present examples in Section 5.

5. Boosting Applications

We evaluated the TURBO library using several real-world applications. We chose these workloads to (1) validate the results from our synthetic benchmarks and the cost model to boost sequential bottlenecks; (2) highlight gains by using application knowledge to adjust the core frequencies; (3) show the trade-offs when the IPC depends on the core frequency, e.g., due to memory accesses; and (4) outweigh the latency cost of switching P-states by delegating critical sections to boosted cores.

5.1. Python Global Interpreter Lock

The Python Global Interpreter Lock (GIL) is a well known sequential bottleneck based on a `pthread_mutex`. 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 4.2 to see if they provide practical benefits in such settings. 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 5 summarizes the characteristics of the workloads.

	1 Thread		2 Threads			4 Threads		
Task	python	native	wait	python	native	wait	python	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 5: Characteristics of the `ccbench` workload: average time in μs per iteration spent in the interpreter (python), executing native code without GIL (native) and waiting for GIL acquisition (wait) (AMD FX-8120).

We evaluate the following P-state configuration strategies in Figures 7 (AMD) and 8 (Intel). *Base* runs at the base operating frequency and, hence, does not incur P-state configuration overheads. *Dyn* waits for the GIL at the slowest P-state, then runs at the highest P-state while holding the GIL and switches back to the base operating frequency after releasing the lock. While workloads `Pi` and `Regex` do not scale, *Dyn* supports at least execution of the workloads at the boosted frequency. These results are in line with our synthetic benchmark results and the cost model in Section 4. For 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 the base operating frequency and boosts temporarily one step 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 the lowest possible P-state within the TDP and only switches to the

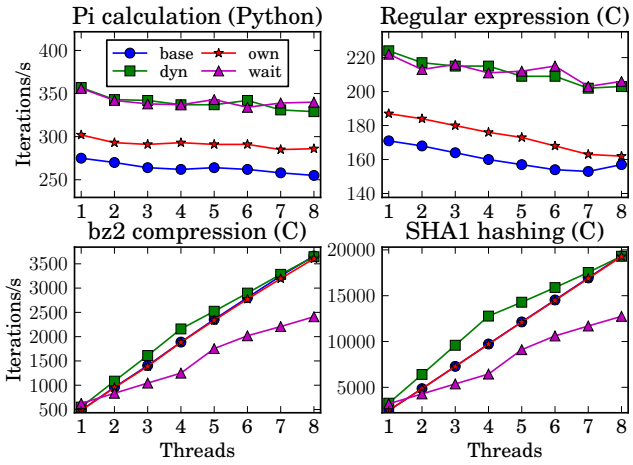

Figure 7: ccbench throughput (AMD FX-8120).

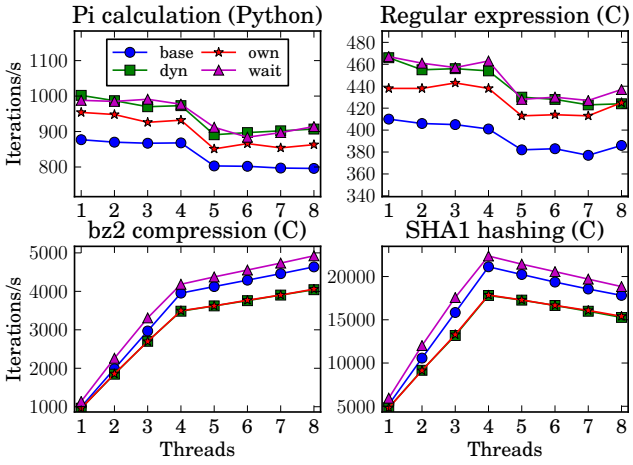

Figure 8: ccbench throughput (Intel i7-4770).

slowest P-state 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 5).

The overall results for `Pi` and `Regex` on the Intel CPU, but we see a performance drop beyond four threads due to performance crosstalk between software threads running on hardware threads of the same core. Boosting results are similar to the ones obtained on AMD.

The same performance interaction is seen for `bz2` and `SHA1`, but boosting results for these scalable applications are different: as with our synthetic benchmarks, the Intel CPU can gain already a lot performance by the automatic boosting mechanisms. Still, we can improve upon this with application knowledge with the *Wait* strategy. On the Intel machine, the policy does not exhibit the long MSR access latencies and the associated performance issues as on the AMD CPU.

5.2. Software Transactional Memory

FastLane [40] 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

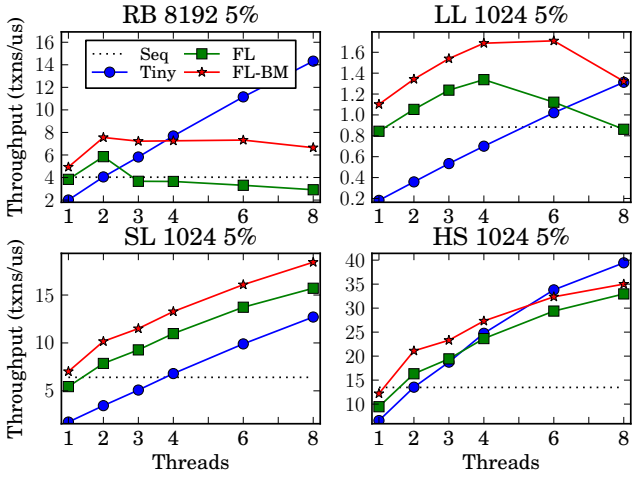

Figure 9: FastLane STM integer set benchmarks (AMD FX-8120).

abort with speculative helper threads that can only commit if they are not in conflict. The master thread has only 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 performed by 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., TL2 [10] or TinySTM [15]) that typically suffer from the high instrumentation and bookkeeping overhead.

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, as detailed in [15]. The parameters are the working set size and the update ratio. Either all threads run at the base operating frequency (*FL*) or the master always runs at the fastest P-state (*FL-BM*) and the helpers at the slowest, except for the helper that runs on the same module as the master. Moreover, we compare with TinySTM (*Tiny*) and uninstrumented sequential execution (*Seq*) at the base operating frequency. Our evaluation on the AMD processor shows in Figure 9 that running the master and helpers at different speeds (*FL-BM*) enables high performance gains compared running all threads at the base operating frequency (*FL*). Table 6 shows that the master can asymmetrically process more transactions when being boosted. While the slow helpers 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.

We chose this workload to highlight the importance to make the P-state configuration accessible from the user space. It allows developers to expose properties of the application that would otherwise not be available to the processor. For applications that contains larger amounts of non-transactional code, the support to remotely set P-states for other cores would be very helpful. When a master transaction is executed, it could

	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 6: Commit ratio of the total commits by the FastLane master (%) for 2, 4, and 6 threads (AMD FX-8120).

slow down the other threads in order to get fully boosted for a short period.

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 lock that protects all accesses to the hash table during the period of a resizing. The lock is implemented by spinning using `pthread_mutex_trylock` because `pthread_mutex_lock` would introduce too much latency when the lock is contended. The resize is done by a separate maintenance thread that move 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 two 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 is active, we gathered for the first (10MB) and the last (1280MB) resize interval the statistics shown in Table 7: the number of items that are moved during one iteration (bulk move, configurable), the rate of set operations during the entire experiment (ops/s), the length of the resize interval (ms), the number of (stalled) instructions and the average frequency achieved by the maintenance thread (freq).

We applied the following strategies during the resizing period: *baseline* runs all threads at the base operating frequency, *stat resizer* runs the maintenance thread at the fastest P-state for the entire period, *dyn resizer* switches to the fastest frequency only the length of an bulk move iteration and causes additional transition overheads, *dyn worker* dynamically reduce the frequency 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

Bulk			Resize 10MB			Resize 1280MB		
Move	Strategy	Ops/s	ms	stalled	freq	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 7: Memcached hash table resize (AMD FX-8120).

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.

5.4. Delegation of Critical Sections

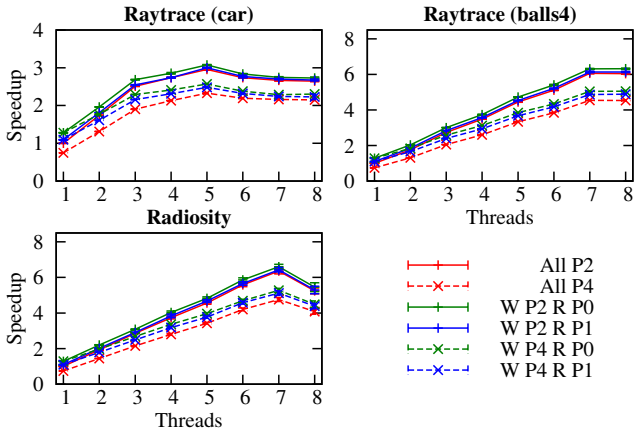

Figure 10: Normalized throughput of SPLASH-2 using RCL under various static boosting configurations (AMD FX-8120).

We have shown that critical sections 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 critical sections locally. Instead of moving the token for permission to enter the critical section

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 that is responsible for executing all critical sections. That way, we can also boost small critical sections because instead of changing P-states, we move execution to the core with the fast P-state.

We extended the software provided by the original authors of the paper and experiment with three of the SPLASH-2 benchmarks [41] that the authors identify as containing a significant amount of critical sections.

We report speedup over the single-threaded baseline P-state in Figure 10, and find that there is a clear performance advantage for the boosted case. 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 nominal ("All P2") frequency and slower frequency ("All P4") for comparison. Note that we do show standard deviation of 30 trials, but there is hardly any noise visible.

Even though critical sections are short (thousands of cycles) for the selected benchmarks, and we do not reduce the P-state for the waiting workers (due to latency reasons), there is enough TDP headroom for the brief RCL invocations to run even at the hardware P0 state and we get speedups ranging from 28% (low thread counts) to 4% at higher thread counts. As expected, the relative boost is larger if we start from a lower baseline at P4.

Overall, scalability of the benchmarks is good, reserving one core exclusively for RCL will cap scalability at 7 (worker) threads. The authors of [27] claim, however, that reserving this single core pays off in comparison to bouncing spin-locks.

6. Related Work

There exists a large body of related work in the field of dynamic voltage and frequency scaling (DVFS) that aim at reducing the power consumption and improving energy efficiency [23, 32]. DVFS is proposed as a mid-term solution to the prediction that, in future generations of microprocessors, the scale of cores will be limited by power constraints [13, 6, 16, 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 [39, 22]. As thermal constraints will prevent powering all cores simultaneously, only the large or small cores will be powered depending on the current workload. A similar effect is achieved by introducing heterogeneous voltages and frequencies to cores of the same ISA [12].

The idea underlying most related work on energy efficiency using DVFS [17] is that reducing the frequency a little gives already good energy savings, but the overall performance is only reduced slightly because it is dominated by memory [25] or network latencies.

Semeraro *et al.* [38] propose to use multiple clock domains, each of with can independently perform voltage and frequency scaling. Inter-domain synchronization is implemented using existing queues to minimize latency, and frequency can be reduced for events that are not on application's critical path. The approach has shown to provide non-negligible energy savings, and was later extended by profile-based reconfiguration [28].

Another interesting approach to save power is to combine DVFS with inter-core prefetching [21]. By using cores to prefetch data into caches, one can improve performance and energy efficiency, even on serial code, when more cores are active at a lower frequency.

Choi *et al.* [7, 8] introduce a technique to decompose programs into CPU-bound (on-chip) and memory-bound (off-chip) operations. This workload decomposition method allows fine tuning of the energy-performance trade-off, with the voltage/frequency being scaled based on the ratio of the on-chip to off-chip latencies. Authors have observed important energy savings with little performance degradation on several workloads running on a single core.

Hsu *et al.* [19] 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 and provides fine power measurement capabilities, demonstrates that significant power savings can be achieved with little impact on runtime performance when limiting both processor frequency and network bandwidth. In the proposed framework, CPU scaling is performed statically by changing the P-states before the application runs. Authors conclude by recommending that next-generation platforms provide fine control over scaling of the different components of the system 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 in modern processors.

While energy efficiency has been widely studied, few researchers have investigated the use of DVFS to speed up workloads [18]. It has been shown for a large class of multi-threaded applications that an optimal scheduling of threads to cores can significantly improve performance [35]. Isci *et al.* [20] propose using a lightweight global power manager for CPUs with per-core control over voltage and frequency. The manager can adapt the power levels according to workload characteristics and, according to authors' analysis, can perform as well as an ideal oracle and much better than static scheduling.

Raghavan *et al.* [34] propose computational sprinting that allows temporary boosting beyond the TDP based on thermal capacitance and the assumptions that idle periods for cooling follow. This technique is intended for interactive applications with short periods with high computing demands whereas our focus is on multi-threaded applications mostly found on servers that run for long periods without much idle time. Here, thermal boosting is not applicable because on average one cannot exceed the TDP.

An in-depth evaluation of Intel processors [14] showed that Turbo Boost provides performance gains only for non-scalable benchmarks because it does not become fully enabled unless only a single core is active. This highlights a major design limitation of Intel's design for multi-threaded applications. Further, Turbo Boost is not energy efficient for scalable workloads due to the high increase in power consumption without performance benefit. Miyoshi *et al.* have also observed

that older Intel Pentium-based CPUs were more energy efficient when running at the fastest performance state [31]. A study of Turbo Boost has shown that the achievable speedup can be improved by pairing CPU intensive workloads to the same core [5]. This allows masking delays caused by memory accesses. In particular, authors exhibit a correlation between the speedup obtained by boosting and the LLC miss rate (which is high for memory-intensive applications).

An analysis of DVFS on recent AMD processors with a memory-bound workload has also shown some limitations. In particular, the benefits of scaling down frequency for energy effectiveness is diminished by the increase of static power in lower voltages [26].

Ren *et al.* [36] 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 dynamic scaling.

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

Recent generations of multi-core processors support fine-grained dynamic voltage and frequency scaling (DVFS). These technologies are mainly driven by hardware and the operating system, and previous research has primarily focused on exploiting core scaling for energy savings. In this paper, we have studied the benefits of DVFS for improving performance of application specific workloads, and also include a thorough analysis of the low-level costs and characteristics of the DVFS mechanisms. We have proposed, implemented, and evaluated a library, named TURBO, that provides programmatic access to performance states of the cores on AMD and Intel processors. Our library allows developers to tune the performance of the system to the properties of their applications (e.g., depending on the number and duration of critical sections or on the asymmetry of the workloads), which are not known by hardware or the operating system.

In our study, we made several interesting findings that we expect to provide valuable insights to processor and operating system designers. In particular, we observed that the frequency transition is sometimes too slow to be amortized with short boosted sections, frequency control is not sufficiently accessible from user space, and all frequency transitions should be asynchronous to let the cores remain operational during the changes. We have also derived a cost model for reasoning about when boosting pays off.

Despite the limitations we encountered with DVFS, evaluation of our TURBO library on several real-world applications has shown promising results in terms of performance boost. As part of future work, we will publicly release our library and applications so that they can be extended to other processors and workloads, as well as used by developers to enable frequency scaling for their applications. We also plan to add profiling mechanisms for critical sections to identify boosting targets, and provide better support for asymmetric workloads.

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. *Commun. ACM*, 2011.

[3] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 2012.

[4] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *Solid-State Circuits, IEEE Journal of*, 35(11):1571–1580, 2000.

[5] J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova. Evaluation of the intel core i7 turbo boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[6] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science*, 2011.

[7] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 international symposium on Low power electronics and design*, 2004.

[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 *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, 2004.

[9] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, Mar. 2007.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06: Proceedings of the 20th international conference on Distributed Computing*, pages 194–208, 2006.

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

[12] 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 *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[13] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011.

[14] 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 *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011.

[15] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 2011.

[17] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, 2007.

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

[19] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[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 *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006.

[21] M. Kamruzzaman, S. Swanson, and D. Tullsen. Underclocked software prefetching: More cores, less energy. *Micro, IEEE*, 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 *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[23] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *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 *Proceedings of the 2012 Symposium on High Performance Computing*, 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 Parallel Processing*. Springer Berlin Heidelberg, 2011.

[26] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, 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 *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[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 *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

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

[30] A. Mendelson and F. Gabbay. The effect of seance communication on multiprocessing systems. *ACM Trans. Comput. Syst.*, 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 *Proceedings of the 16th international conference on Supercomputing*, 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. *Solid-State Circuits, IEEE Journal of*, 2002.

[33] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259. ACM, 2001.

[34] 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 *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2013.

[35] B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu. Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.

[36] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity for interactive services. *10th International Conference on Autonomic Computing*, 2013.

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

[38] 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 *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, 2002.

[39] 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 *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, 2010.

[40] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. Fastlane: improving performance of software transactional memory for low thread counts. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.