Bolt: Faster Reconfiguration in Operating Systems

Sankaralingam Panneerselvam[1], Michael M. Swift[1], and Nam Sung Kim[2]

[1]Department of Computer Sciences
[2]Department of Electrical and Computer Engineering
University of Wisconsin-Madison
[1]{sankarp,swift}@cs.wisc.edu
[2]nskim3@wisc.edu

Abstract

Dynamic resource scaling enables provisioning extra resources during peak loads and saving energy by reclaiming those resources during off-peak times. Scaling the number of CPU cores is particularly valuable as it allows power savings during low-usage periods. Current systems perform scaling with a slow *hotplug* mechanism, which was primarily designed to remove or replace faulty cores. The high cost of scaling is reflected in power management policies that perform scaling at coarser time scales to amortize the high reconfiguration latency.

We describe *Bolt*, a new mechanism built on existing hotplug infrastructure to reduce scaling latency. Bolt also supports a new bulk interface to add or remove multiple cores at once. We implemented Bolt for x86 and ARM architectures. Our evaluation shows that Bolt can achieve over 20x speedup for entering offline state. While turning on CPUs, Bolt achieves speedup upto 10x and 21x for x86 and ARM respectively.

1 Introduction

Most operating system policies focus on improving performance given a *fixed* set of resources. For example, schedulers assume a fixed set of processors to which assign threads, and memory managers assume a fixed pool of memory. However, this assumption is increasingly violated in the current computing landscape, where resources can be added or removed dynamically during runtime for reasons of energy reduction, cost savings, virtual-machine scaling or hardware heterogeneity [12].

We refer to changing the set of resources as *resource scaling* and our work focuses on scaling the set of processors available to the operating system. This scaling may be helpful in virtualized settings such as cloud computing [8] and disaggregated servers [15, 2] where it is possible to add or remove CPUs at anytime within a system. Scaling can improve performance during peak loads and minimize energy during off-peak loads [9]. Dynamically reconfigurable processors (e.g., [17]), which allow processing resources to be reconfigured at runtime to meet application demands, can also benefit from scaling the set of available CPUs [22].

Current operating systems assume that the processor cores available to them are essentially static and almost never change over runtime of the system. These systems support scaling through a hotplug mechanism that was primarily designed to remove faulty cores from the system [11]. This mechanism is slow, bulky and halts the entire machine for a few milliseconds while the OS reconfigures. The current Linux hotplug mechanism takes tens of milliseconds to reconfigure the OS [22]. In contrast, processor vendors are aiming for transitions to and from sleep states in the order of microseconds [26, 24], making OS the bottleneck in scaling. In spite of these drawbacks, hotplug is being widely used by mobile vendors as a means to scale the number of processor cores to save energy [23] due to the lack of better alternatives.

We propose a new mechanism, Bolt, that builds on the Linux hotplug infrastructure with the assumption that scaling events are frequent and a goal of low latency scaling mechanism. Bolt classifies every operation carried out during hotplug as critical or non-critical operations. The critical operations are performed immediately for proper functioning of the system whereas non-critical operations are done lazily. Bolt also supports a *new bulk interface* to turn on/off multiple cores simultaneously, which is not supported by the hotplug mechanism.

We implemented Bolt for both x86 and ARM processors, and find that Bolt can achieve upto 20x better latency over native hotplug offline mechanism. For getting a CPU to online state, Bolt offers a speedup of 10x for modern x86 processors and 21x for Exynos (ARM) processor. The bulk interface supported in Bolt achieves speedup of 18x-67x when adding or removing 3 cores in x86 processors compared to doing the same operations over native hotplug mechanism.

2 Motivation

2.1 Processor Scaling

Many current and future systems will support a dynamic changing set of cores for three major reasons.

Energy Proportionality. This property dictates that energy consumption should be proportional to the amount

of work done and is getting a major focus in all forms on computing from cloud to mobile devices. Low power (P-states) and sleep state (C-states) support from processors help achieve energy proportionality by reducing energy consumption when the processor is not fully utilized. However, in many processors further power savings could be achieved by turning off cores to a deeper sleep state, turning off an entire socket or allowing to enter package level sleep state. These deeper states require OS intervention, as the core is logically turned off and not available for scheduling threads or processing interrupts. For example, most mobile systems turn off cores during low system utilization to conserve battery capacity by reducing static power consumption. Some processors (e.g. Exynos [19]) provide a package-level deep sleep state that is enabled when all but one core is switched off. Operating systems may use core parking [13] to consolidate tasks in a single socket to switch off other sockets completely. Quick scaling support by the OS can allow rapid transitions into and out of deep sleep states.

Heterogeneity. Many processors support heterogeneity either statically [1] via different core designs or dynamically [17] through reconfiguration. On dark silicon systems [6], not all processors could be used at full performance together, and hence the OS must decide which processors to enable based on the application characteristics. Processors like Exynos [4] and Tegra [21] employ ARM's big.LITTLE architecture, with a mix of high performance and high efficiency cores. In systems with these processors, the OS must choose the type of processor core based on the performance need. The OS must change the processor set when switching between different CPU types.

VM Scaling. Virtual machines are widely used in cloud environment, and many hypervisors provide support for scaling the number of virtual CPUs in a virtual machine (VM) [20]. IBM supports VM scaling through DLPAR (dynamic logical partitioning [18]) and VMware supports them through hot add/remove interfaces. Some application like databases benefit from scaling up of virtual machines by provisioning more resources rather than scaling out where more virtual machines are spawned. These techniques can be used at finer scale if the guest OS provides quick processor scaling.

2.2 OS Support

There are several mechanisms an OS can use to scale the number of CPUs in use.

Virtualization. An extra layer of indirection through virtualization decouples the physical execution layer from rest of the operating system and exposes only virtual CPUs to the OS. To scale down, multiple virtual CPUs (VCPU) can be multiplexed on a single physical CPU (PCPU). In terms of latency, virtualization could provide an ideal support where it could switch VCPUs from a PCPU that is being switched off to a different physical CPU instantly by saving and restoring context of those virtual CPUs.

However, the drawbacks of using virtualization-based techniques are two-fold. First, virtualization adds overhead in the common case, particularly for memory access [7]. Second, multiplexing VCPUs on a physical CPU can hurt performance due to context switches during critical sections [27].

Power Management. OS support for idle sleep states (C-states) can be used to move unused cores to a sleep state and wake up when needed. The latency of entering and exiting such sleep state is very low when compared to the hotplug mechanism. However, OS power management support requires that cores can still respond to interrupts, which is not the case for all deep sleep states or for non-power uses of scaling. Furthermore, the OS may accidentally wake up a sleeping core unnecessarily to involve them in regular activities like scheduling or TLB shootdowns [25].

Processor Proxies. Chameleon [22] proposed a new alternative to hotplug called processor proxies that is several times faster than hotplug. A proxy represents an offline CPU and runs on an active CPU making the system believe that the offline CPU is still active. However, proxies can only be used for a short period because they handle interrupts and Read-Copy-Update (RCU) operations only, and do not reschedule threads from a CPU in offline state.

Scalability-Aware Kernel. Ideally, an OS kernel could natively support changing the set of CPUs at low latency and with low overhead. Rather than assuming that scaling events are rare, a scalability-aware kernel would spend little time freeing resources during a scale-down event when they are likely to be re-allocated during an upcoming scale-up event.

2.3 Hotplug

Hotplug is a widely used mechanism available in Linux to support processor set scaling. It offers interfaces for any kernel subsystem to subscribe to notifications for processor set changes. However, handling of notifications by every subsystem follow the assumption that hotplug events are rare. The shortcomings of the mechanism are discussed below.

Repeat Execution. A direct implication of the above assumption is that most kernel subsystems free or reinitialize the software structures during hotplug event. Out of the 50 subscriptions to the hotplug from various subsystems, 8 of them remove or initialize `sysfs` structures and 14 of them to free and create software structures

needed for the subsystem. However, all these operations become redundant if the CPU set changes frequently.

Synchronous. All operations performed in response to the notifications are synchronous. For example, subsystems like slab allocator frees the slab memory from its per-CPU queue when the CPU is moved to offline state, per-CPU statistics values are aggregated into a global structure, and the hotplug operation is blocked until system threads move into sleep state. However, these operations need not be synchronous for the correct execution of the system.

Hotplug Prevention. Hotplug events can be prevented by disabling preemption or interrupts on any CPU, similar to grabbing a lock. So, the hotplug mechanism ensures that preemption and interrupts are enabled on all CPUs by scheduling a special kernel thread on *every CPU* in the system. This special form of locking (through preemption) avoids the overhead of acquiring a lock and releasing during normal execution.

As a result of these properties, Linux's current hotplug mechanism is too slow for rapid scaling. As we show in Section 4, it takes orders of milliseconds to reconfigure, while current hardware can transition from sleep states in the order of *microseconds* [26].

3 Bolt

Bolt is a reconfiguration mechanism that can be used as a replacement for hotplug. The functionality of Bolt is similar to that of hotplug in getting the system from one stable state to another after processor scaling. However, Bolt aims to offer stability at very low latency and is built by refactoring the existing hotplug infrastructure.

Bolt achieves low latency by separating hotplug notifications into *critical* and *non-critical* operations. The former needs to be handled synchronously to ensure correctness of the system whereas the latter could be removed from the critical path and performed after the CPU goes online/offline.

3.1 Critical Operations

Every action taken by hotplug, including handling of notifications by kernel subsystems is classified based on its criticality. Bolt defines critical operations as those that need to be executed immediately for correct running of the system.

State Migration. In the event of CPU removal, important software state associated with that CPU has to be migrated to another active CPU. Such software states include softirq or bottom halves and threads in the CPU's runqueue. The softirqs are queued in a per-CPU structure that are moved to a different active CPU for them to be processed. Similarly, threads from the runqueue are migrated synchronously to avoid performance degradation for running programs.

Hardware Management Functions. Certain hardware dependent features need to be disabled or enabled during scaling (hotplug) events. For example, machine check has to be disabled before the CPU is put to offline state since any fault in the offline CPU should not affect the remaining system. Similarly when a CPU is started, it's microcode need to be updated and MTRR registers need to be initialized for proper functioning of the system.

Bitmask Updates. Linux maintains a few important global bitmasks of CPU state. These include `cpu_online_mask` and `cpu_active_mask`, which are accessed frequently across the system. These masks should be updated immediately during scaling events for correct functioning of the system.

We consider subscriptions from a few subsystems like workqueue and perf as critical since we are still in the process of adapting those subsystems to Bolt.

3.2 Non-Critical Operations

Bolt defines non-critical operations as those that can either be performed lazily or not performed at all. Bolt makes a best effort to push many of the non-critical operations out of the critical path and thus reduce latency.

Interrupt Migration. Handling of interrupts is a time critical event and it might be surprising to see interrupt handling as a non-critical operation. Interrupts affinitized to a CPU are moved to a different CPU when the CPU is removed. However, from our observation on a Nexus mobile device, all I/O interrupts are always delivered to the base CPU (CPU 0). This was not the case on a desktop machine where the network interrupts were distributed across multiple CPUs.

Bolt currently affinitizes all interrupts to the base CPU to avoid interrupt migration during processor scaling event. However, this will result in performance degradation for servers receiving high number of interrupts. On such systems, the high-traffic interrupts (e.g., network or SSD) can by distributed across multiple CPUs through the `irq_migration` daemon, while lightly loaded interrupts can be handled by the base CPU to avoid migration during scaling. But Bolt does not support this optimization and implementing it is part of the future work.

Memory. Many kernel subsystems, upon receiving notifications of scaling events, free or allocate memory structures such as per-CPU structures or buffer queues. Bolt elides these operations and instead saves memory to be re-used if and when the CPU comes back online. Most kernel subsystems support a master thread that is invoked during memory pressure to release all per-CPU structures. Bolt uses this master thread to avoid any memory leak if a CPU stays offline for an extended time period.

Sysfs. Volatile filesystems like `sysfs` and `procfs` expose kernel settings and metrics through virtual file sys-

tem interface in Linux. Processor-core based file or directory nodes are removed or created during processor set scaling. However, Bolt does not perform these operations but instead it prevents access to CPU dependent files by verifying if the CPU is online during file open.

Thread Operations. Earlier versions of Linux destroyed and spawned system threads during hotplug. More recent versions of Linux use thread parking [10], which suspends threads indefinitely. System threads like watchdog threads are moved to a parked state during processor scaling. Parking the thread involves waking the thread, even if it is in sleep state, invoking a registered function pointer to perform cleanup, and then moving the thread to the sleep state synchronously. Bolt instead employs an asynchronous approach and it does *not* wait for the thread to enter sleep state after sending the parking message. The benefits are that it improves latency and parking could be avoided if the CPU returns back online before the thread wakes up and sees the parking message.

3.3 Bulk Interface

Bolt adds a new bulk interface support to allow scaling multiple CPUs simultaneously; existing APIs only support adding or removing a single CPU. The API takes a cpumask argument, indicating which CPUs to add or remove rather than the index of an individual CPU. To accomplish a bulk offline with native hotplug, each CPU is moved to offline state sequentially and with no overlap; the online case is similar. Bolt leverages the fact that certain operations can be done once even if multiple CPUs change state. Bolt makes two optimizations. First, updates to global structures are made as a single operation. For example, Bolt reorganizes the scheduling domain related structures once for all CPUs. Second, Bolt performs some operations in parallel. For example, during CPU online it clears caches and register sets concurrently on all CPUs.

4 Evaluation

Our evaluation focuses on the performance of Bolt, but we speculate on the potential energy benefits as well.

Experimental Platform. We performed our experiments on two different processor architectures. First, an x86 based machine with an Intel i5-2500K (Sandy bridge) processor running Linux kernel 3.17.1. Second, an Odroid development board [14] with Exynos 5410 processor running Android 4.4 with Linux kernel 3.4. The Exynos is a big.LITTLE architecture provisioned with A15 and A7 4-core clusters. We disable Turbo Boost in the x86 machine to avoid any performance variability. All experiments were performed in an idle system without any active workloads. For all the experiments, we ran the processors at highest frequency: x86 at 3.3 GHz and A15 at 1.6 GHz.

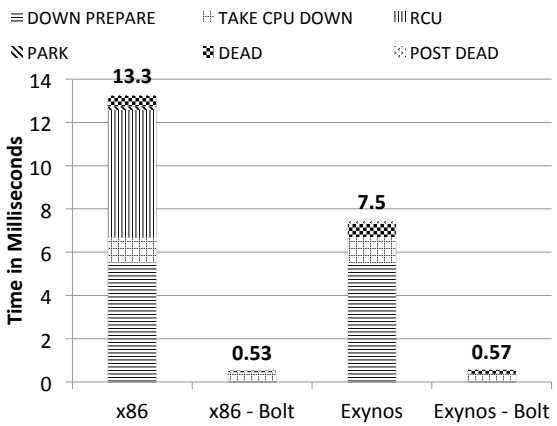

Figure 1: Native hotplug vs. Bolt during offline.

End-End Latency. The CPU state (online/offline) is accessed through the sysfs file `/sys/devices/system/cpu/cpu*/online`—writing '0' initiates the offline process and '1' brings the cpu to an online state. We measure latency as the time taken from the write to when it returns, at which point the CPU becomes invisible to the OS or it is actively available to the OS.

Figure 1 show the end-to-end latency for an offline operation. The legend represents the individual components of the hotplug operation. (a) Down prepare, dead and post dead are different notifications sent to the kernel subsystems. The down prepare is costly due to a synchronous thread creation by workqueue subsystem in the critical path. Bolt avoids this behavior by reusing threads. (b) Park refers to the thread parking operation that is classified by Bolt as non-critical and handled appropriately. (c) Reduction in the latency of `take_cpu_down` is achieved by avoiding interrupt migration and thus, saving 0.7ms. The remaining overhead is caused by `stop_machine` interface, which is not optimized by Bolt. (d) RCU denotes the protection of `cpu_active_mask` bitmask through RCU synchronization and this is costly since read-copy-update (RCU) has to wait till all CPUs undergo a context switch. Bolt replaces RCU-based synchronization with regular locks. The impact of this change is limited to very few interfaces as can be seen in this commit log [28].

Figure 2 show the latency breakdown for an online operation. Interestingly, the major source of latency in Exynos is software, and in x86, hardware. In the Exynos system, thread creation causes overhead similar to the offline case and Bolt avoids this by parking threads and re-using them during the online operation. However, the x86 incurs substantial delay when the init IPI (to start the core) is sent. Intel documentation [16] specifies that OS should wait for a period of 10ms after the init message is sent to perform hardware initialization. The speedup of Bolt over the native system is thus limited to 1.3x. How-

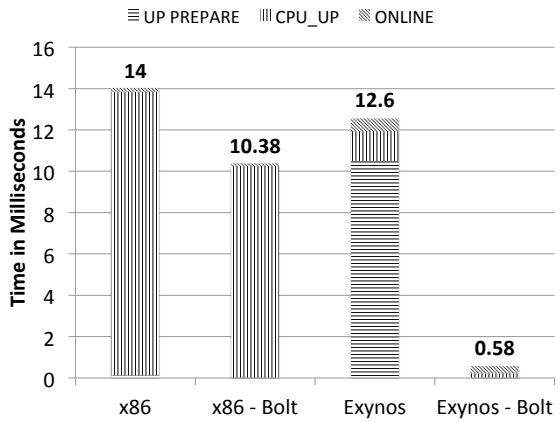

Figure 2: Native hotplug vs. Bolt during online.

ever, the initialization delay is not required for modern multi-core x86 processors [3].

Software Entry/Exit Latency. The entry latency refers to the time taken for the CPU core to be switched off during offline operation. This gives an idea on how soon the energy savings begin. The native system takes around 12.5ms (x86) and 6.7ms (Exynos) whereas Bolt takes 0.45ms (x86) with a speedup of 27.8x and 0.38ms Exynos) with a speedup over native being 17.6x. The exit latency is the time taken for the core to schedule the first thread after it is woken up. This gives an idea of the interactivity of the system. The native system takes around 13.8ms (x86) and 12ms (Exynos) whereas Bolt takes 10.27ms (x86) and 0.22ms (Exynos).

Energy Savings. Bolt does not offer a new power management policy but it relies on the processor sleep states for any energy savings. In the Exynos 5410, the offline state and deep sleep state—C2— consume almost same power: 97mW when all cores are in sleep state. However, the processor cluster is allowed to enter C3 (package-level deep sleep state), it consumes 55mW, which is a 43% reduction in power compared to C2. The constraint is that all on-chip cores but one should be in hotplug offline state to enable C3 state.

The default power management policy used for Exynos employs a conservative approach that ensures a minimum of two CPU cores is online for better interactivity. We believe that Bolt could help in implementing a more aggressive policy for more energy savings while preserving interactivity by retaining only one online core and entering C3 in the remainder. On the other hand, hotplug offline on an Intel i5-2500K puts the CPU core to deep sleep state (C6). In this case, hotplug does not result in additional power savings than Linux's cpuidle subsystem [5] but hotplug can still be beneficial by avoiding interrupt handling and scheduling threads on idle cores and extending their idle period.

Bulk Interface. The current bulk interface implementation is available only for x86 architecture and not for Exynos. We specify an input cpumask marked with three CPUs. On the native system, we simulate bulk operations by performing scaling operations sequentially. The native system took 39.2ms and 43.1ms for offline and online operations. Bolt using the same simulation technique took 1.6ms and 31.2ms, while the bulk interface in Bolt took 0.58ms and 10.84ms. The new interface executes the `take_cpu_down` concurrently and avoids multiple time re-organization of schedule domain structures during offline. Overlapping hardware initialization and processing notification messages while waiting for the hardware initialization speed the online operation.

Modern Hardware. The high latency of processor initialization (10ms delay) is not required in modern x86 multi-core processors as noted in this patchset [3]. So, the processor initialization cost becomes zero. To analyze the impact of Bolt on these processors where the software overhead dominates hotplug operations, we ran experiments by commenting the delay in the kernel. The change is valid since our x86 platform does not require the delay [3]. Bolt finishes online operation in 0.38ms compared to 4ms for native providing a speedup of 10.5x, and the online bulk interface takes 0.71ms compared to 13.1ms providing a speedup of 18x. The native latency values – 4ms and 13.1ms – are achieved by removing the hardware initialization delay of 10ms from original latency values. These numbers show that hotplug latency will be dominated by software overhead in future processors and Bolt makes significant reduction in the software overhead.

5 Conclusion

Processor set scaling is important for energy efficiency, throughput improvement, cost savings and VM scaling. However, the current hotplug mechanism is slow and cannot support frequent changes. We propose Bolt, which classifies hotplug operations as critical and non-critical. This separation helps Bolt achieve low latency by removing non-critical operations from the critical path. Bolt also supports a bulk interface that allows scaling at granularity of multiple cores. The low latency mechanism and the new interface support through Bolt enable future systems to scale at much finer time-scales.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grant CNS-1302260. We would like to thank our shepherd, Dan Tsafrir, and the anonymous reviewers for their invaluable feedback. We would also like to thank Venkat, Sanketh and Thanu for their comments on the earlier draft of the paper. Swift has a significant financial interest in Microsoft, and Nam Sung Kim has financial interests in Samsung Electronics and AMD.

References

[1] ARM big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[2] AMD - SeaMicro, Inc. Seamicro sm15000 fabric compute systems. http://www.seamicro.com/sites/default/files/SM_DS06_v2.1.pdf, 2012.

[3] L. Brown. x86/smp/boot: Remove 10ms delay from cpu_up() on modern processors. https://lkml.org/lkml/2015/5/12/102.

[4] H. Chung, M. Kang, and H.-D. Cho. Heterogeneous multi-processing solution of exynos 5 octa with arm® big.little technology. http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf.

[5] J. Corbet. The cpuidle subsystem. http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/cpuquiet.pd://lwn.net/Articles/384146/, April 2010.

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. ISCA*, June 2011.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[8] Gartner. Case studies in cloud computing. https://www.gartner.com/doc/1761616/case-studies-cloud-computing.

[9] H. R. Ghasemi and N. S. Kim. Rcs: Runtime resource and core scaling for power-constrained multi-core processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 251–262, 2014.

[10] T. Gleixner. Kthread: Implement park/unpark facility. https://lwn.net/Articles/500338/, 2012.

[11] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning up linux's cpu hotplug for real time and energy management. *SIGBED Rev.*, pages 49–52, Nov. 2012.

[12] A. Gupta, E. Ababneh, R. Han, and E. Keller. Towards elastic operating systems. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, 2013.

[13] C. Hameed. Windows 7 and windows server 2008 r2: Core parking, intelligent timer tick and timer coalescing. http://blogs.technet.com/b/askperf/archive/2009/10/03/windows-7-windows-server-2008-r2-core-parking-intelligent-timer-tick-timer-coalescing.aspx, 2009.

[14] HardKernel co., Ltd. Odroid-xu+e. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079, 2013.

[15] Hewlett-Packard Development Company. Hp moonshot system. http://www8.hp.com/us/en/products/servers/moonshot/index.html.

[16] Intel Corporation. Intel multiprocessor specification, v1.4. http://www.intel.com/design/pentium/datashts/24201606.pdf, 1997.

[17] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accomodating software diversity in chip multiprocessors. In *Proc. of the 34th ISCA*, June 2007.

[18] J. Jann. Dynamic logical partitioning for power systems. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 587–592. Springer US, 2011.

[19] B. Klug. Samsung announces exynos 5 octa soc. http://www.anandtech.com/show/6602/samsung-announces-exynos-5-octa-soc-4-cortex-a7s-4-cortex-a15s, 2013.

[20] M. Nishikiori. Server virtualization with vmware vsphere 4. *Fujitsu Scientific and Technical Journal*, pages 356–361, 2011.

[21] NVIDIA. Nvidia tegra x1 nvidia's new mobile superchip. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, 2015.

[22] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 99–110, 2012.

[23] Schrijver, Peter De and Miettinen, Antti P. Cpuquiet: A framework to manage cpus. http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/cpuquiet.pdf.

[24] A. L. Shimpi. Intel's haswell architecture analyzed: Building a new pc and a new intel. http://www.anandtech.com/show/6355/intels-haswell-architecture/3, 2012.

[25] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[26] A. V. D. Ven. Absolute power. https://software.intel.com/sites/default/files/absolute_power.pdf.

[27] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *ACM SIGOPS Operating Systems Review*, 43(2), Apr. 2009.

[28] P. Zijlstra. Sched: Remove get_online_cpus usage. https://github.com/torvalds/linux/commit/6acce3ef84520537f8a09a12c9ddbe814a584dd2, 2013.