When Slower is Faster: On Heterogeneous Multicores for Reliable Systems

Tomas Hruby Herbert Bos Andrew S. Tanenbaum
The Network Institute, VU University Amsterdam
{*thruby,herbertb,ast*}*@few.vu.nl*

Abstract

Breaking up the OS in many small components is attractive from a dependability point of view. If one of the components crashes or needs an update, we can replace it on the fly without taking down the system. The question is how to achieve this without sacrificing performance and without wasting resources unnecessarily. In this paper, we show that heterogeneous multicore architectures allow us to run OS code efficiently by executing each of the OS components on the most suitable core. Thus, components that require high single-thread performance run on (expensive) high-performance cores, while components that are less performance critical run on wimpy cores. Moreover, as current trends suggest that there will be no shortage of cores, we can give each component its own dedicated core when performance is of the essence, and consolidate multiple functions on a single core (saving power and resources) when performance is less critical for these components. Using frequency scaling to emulate different x86 cores, we evaluate our design on the most demanding subsystem of our operating system—the network stack. We show that *less is sometimes more* and that we can deliver better throughput with slower and, likely, less power hungry cores. For instance, we support network processing at close to 10 Gbps (the maximum speed of our NIC), while using an average of just 60% of the core speeds. Moreover, even if we scale all the cores of the network stack down to as little as 200 MHz, we still achieve 1.8 Gbps, which may be enough for many applications.

1 Introduction

More and more hardware vendors are developing heterogeneous multicore architectures. Well known examples include the so-called *big.LITTLE* [1] ARM, the NVIDIA Tegra-3 [2], its recently announced successor Tegra 4, and the x86-compatible Xeon Phi [4]. The *big.LITTLE* ARM combines two big Cortex-A15 cores with two little Cortex-A7 on the same die, and Samsung recently announced a 4 + 4 version [5]. The Tegra-3 is a Cortex-A9-based quad-core CPU that includes a fifth ”companion” Cortex-A9 that is slower (capped at 500MHz) and less power hungry. For sheer number of cores, the 50+ core x86-compatible Intel Xeon Phi processor is especially impressive. It serves as an extension of many little cores to accompany the host's big cores and lives on a separate PCIe card.

In all three cases, the different cores share a large subset of the instruction set architecture (ISA), so that the same code can easily run on any of the cores in the system. The main difference of the cores is their microarchitecture which is designed for different optimal operation points. This means that the *LITTLE* slower, simpler, and in-order cores (designed for power efficiency at low frequencies) cannot deliver performance equal to the *big* ones which are out-of-order and operate at higher frequencies. The same is true for the Tegra and Xeon Phi. For instance, the host x86 processors feature out-of-order cores with a deep pipeline while the Xeon Phi cores are much simpler, in-order Pentium cores with shallow pipelines to allow for efficient 4-way hyper-threading. In addition, they feature new vector instructions to support scientific workloads.

The research community has advocated such heterogeneity for many years [15] to make processing more efficient, in terms of both performance and power. However, the focus was primarily on applications, leaving the operating system by the wayside. This is remarkable, because the operating system performs a significant amount of work on behalf of the applications [23, 21].

Moreover, the changes to the system remain mostly limited to making execution on different cores possible and to finding the best schedule. Exceptions include the proposal by Strong et al. [28] to migrate long-running system calls to *system cores*—cores more suitable for running OS workloads. FlexSC[26], meanwhile, aims to remove the overhead of switching between applications and the system by running each on different cores. As a side effect, the system can run on core(s) that differ from those that host applications.

The NewtOS operating system described in this paper is a UNIX-like multiserver system that offers these major benefits:

High reliability For instance, our operating system supports dynamic updates without any downtime and survives crashes of core OS components. We described these aspects in [11, 7], and [10], and will not discuss them further in this paper.

High performance Building on a design described in [11], we show that we support network processing at 10 Gbps on COTS hardware (this paper).

The unique features of multiserver systems, composed of independent user space servers, typically trade reliability for performance. With many processes involved, the communication and context switching to share the processor constitute a significant overhead. NewtOS [11], a high-performance derivative of MINIX 3, shows that it is possible to mitigate this overhead by dedicating cores to system servers, which communicate through asynchronous channels. With their own cores, the system servers can run whenever needed from a warm cache, and without having to compete with other processes or wait for the kernel to schedule them. Moreover, the system's asynchronous communication channels allow the system servers to work independently and thus increase the parallelism within the system and streamline the processing. As a result, we were able to support TCP-based network processing at up to 5 Gbps [11]. Since then, we have further improved our design. We will show in Section 4 that we now support TCP at close to 10 Gbps—the maximum speed of our network card.

The cores of common platforms are designed for generic usage and over-provisioned [20] for running OS code. Dedicating cores results in a very coarse grained resource assignment, which leads to inefficient use of the available hardware. Looking at current trends, we anticipate more designs in the *big.LITTLE* fashion, which will have plenty of smaller, slower, in-order cores with a higher number of threads, accompanied by big, fast cores that can efficiently use the instruction level parallelism of application code. However, the big cores will become a minority.

In this paper, we explore how such architectures can help to balance performance and resource consumption. Specifically, we show that we can run the OS components on multiple slower cores, while still achieving high performance. Alternatively, the system can consolidate components on a few cores (saving power and resources) and still achieve reasonable performance.

Our contributions are:

1. We explore the hardware design space by emulating the future platforms on current hardware using frequency scaling to find out how fast the processors should be and what type of cores would suit systems the best.
2. We show that our system can deliver the same or better performance with smaller, simpler and slower cores—without compromising reliability. Our case study shows it is suitable for high-speed networking.
3. The system has a potential to dynamically reconfigure itself to use the most appropriate resources and free resources it does not need for a particular workload.

In the rest of the paper we discuss our motivations and the background in Section 2. We present details of the NewtOS design in Section 3. We explore the design space and evaluate various setups of our system in Section 4 and we put it in perspective of related work in Section 5. Finally, we conclude in Section 6.

2 Big cores, little cores and combinations

Heterogeneous processor architectures are rapidly becoming popular. In this section, we focus on Intel products and sketch some of the properties of the architectures and analyze some trends in this field.

2.1 BOCs and SICs

We start our discussion with a comparison of fast cores and slow cores. Specifically, the first two columns of Table 1 compares the Intel Core i7 "Bloomfield" with the Knights Ferry processor. The Core i7 is a prime example of a big out-of-order core (BOC) with a design that is geared for high single threaded throughput and produced by 45nm technology. In contrast, the cores on the Knights Ferry (45nm) are much simpler in-order cores (SICs) that provide only a fraction of the i7's performance.

Given the estimated die size of the Knights Ferry, the table shows the space reduction of the simple cores compared to the big i7 cores. Compared to the i7, the Knights Ferry die hosts 8× the number of cores and 16× the number of threads. It is worth noting that the difference in die size per core is 3× (and 6× per thread). While the cache size per core is obviously smaller, threaded cores can compensate for this [22]. Finally, the difference in the peak clock speed is equally remarkable.

The last column of Table 1 shows data for the successor of Knights Ferry, a recently released product called Xeon Phi. Its core count is even larger, but its die size is not public. Intel markets it as a "50+ core beast" and released up to 62 cores on a single die. With each core hosting

	Core i7 Bloomfield (45nm)	Knights Ferry (45nm)	Xeon Phi (22nm)
Die size	$263mm^2$	est. $700mm^2$	not released
Cores / threads	4 / 8	32 / 128	50+ / 200+
Die area per core / thread	$65.7/32.9mm^2$	$21/5.5mm^2$	not released
Clock speed	max 3.33 GHz	1.2 GHz	1 GHz
LL cache size	8 MB	8 MB	25+ MB
	out-of-order	in-order	in-order

Table 1: Comparison of Core i7, Knights Ferry and Xeon Phi

	Transistor count	Die size
4-core i7 + GPU	1.4 bil.	$160mm^2$
62-core Xeon Phi	5 bil.	not released

(a) Transistor count

# i7 cores	# i7 threads	# Phi cores	# Phi threads
4	8	44	176
8	16	27	108
12	24	10	40

(b) Core i7 and Xeon Phi configurations

Table 2: Given the known transistor counts shown in (a) and a 22nm production process, we can roughly project the options for different configurations that merge Core i7 and Xeon Phi cores (b)

4 threads of execution, this amounts to 200+ threads on a single chip. Interestingly, Xeon Phi is still a cache coherent design, unlike one of its research predecessors—the single chip cloud (SCC).

2.2 Configurations of BOCs and SICs

It is likely that future designs will see interesting new combinations of BOCs and SICs. For instance, rather than keep it as a separate co-processor, Intel may well merge its Xeon Phi with other Intel cores on a single die [17], in the same way that GPUs and general purpose cores have merged on a single die. What sort of processor should we expect? Clearly, there are many options. In this section, we explore possible combinations in an approximate manner.

Table 2 shows different combinations of Core i7 "Sandy Bridge" and Xeon Phi cores, taking into account the number of transistors for a quad-core Core i7 in 22nm technology as well as the number of transistors of a 62-core Xeon Phi produced by the same technology. The Core i7 die may also contain an integrated GPU. Given these transistor counts, Table 2b shows different configurations that would fit on a die with mixed cores. The simple division also accounts for each core's cache share as caches take up a big portion of the die size. Each line represents a configuration with the 5 billion transistor budget of Xeon Phi die where some of the 62 cores are replaced by i7 cores.

Finally, for the sake of completeness, Table 2b also shows the resulting number of threads. The number of threads matters, because we will see that for OS functionality it is often not needed to dedicate a full core to each component. Instead, a simple container for a process' context is good enough, as long as we can let the hardware do the context switching (and not the software) and we can suspend and resume efficiently with instructions like MWAIT and MONITOR. A hardware thread is well-suited to serve as a container. It has a set of replicated registers and, depending on the architectures, hardware switches the threads automatically when the active one stalls or it tries to schedule a different thread every cycle.

As consolidating multiple components on a single core saves resources, more threads are attractive. Moreover, many platforms today still do not offer enough cores for us to be able to dedicate one to each component—although the number of cores per die is growing steadily. By using threads instead, we can implement our design even on today's platforms.

For instance, NewtOS has about 30 system processes in its default installation out of which about 10 are important for performance. These include the process manager, the memory manager, the storage stack, the network stack, the file systems, the disk and the network drivers. The calculation in Table 2b shows that even with twelve i7 cores, there would be enough threads to dedicate one to each of our system's processes. Note that based on previous research [20, 18], the Xeon Phi cores are most likely a better match for the system processes than the i7 cores. The platform would therefore still offer plenty of big cores for applications, while the small multithreaded cores would optimize the resources for the system.

An alternative to chips preconfigured with a fixed num-

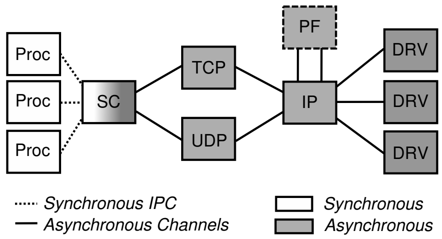


Figure 1: Design of the NewtOS network stack

ber of BOCs and SICs, are architectures that can merge smaller cores into bigger ones [12]. Another example is MorphCore [13], an architecture which can switch a core type between a i7-like BOC and a heavily threaded Phi-like SIC in run time.

3 NewtOS

The crux of this paper is the following: while evaluating the performance of the stack, we realized that it actually delivered higher throughput when we scaled the frequency of some cores down. In addition, we found that the performance of fairly slow cores is good enough for many use cases. We present an OS that explicitly exploits these properties.

Specifically, NewtOS is a high-performance clone of MINIX 3 that provides the same reliability with much reduced overhead. For instance, we completely redesigned the network stack based on new communication principles that allow different components to execute independent request simultaneously. This distinguishes NewtOS from other multiserver operating systems and increases the network throughput from hundreds of megabits per second to gigabits.

3.1 The NewtOS network stack

The heart of the NewtOS network stack is LwIP [8], a simple and portable network stack for embedded systems used by many research projects. Note that this stack is not designed for high performance but rather for its small memory footprint. As a result, its performance is not directly comparable to highly tuned stacks of commodity systems. Nevertheless, we support network processing at 10 Gbps even though we use slow cores.

One of the main design goals of NewtOS is reliability. Thus, we allow even core components of the operating systems to be replaced on the fly, without taking the system down (and often with no noticeable disruption at all) [9]. For instance, we can replace our implementation of IP or the network driver while keeping all existing network connections. Similarly, if one of its components crashes, the OS recovers automatically and often transparently [11].

To make this possible, we split the stack into several components (TCP, UDP, IP, drivers and packet filters) to reduce the chance that an error in the stack may lead to a crash of the entire stack. Likewise, we isolated functions that are easy to restart from those which are not due to large dynamic state. Besides IP, TCP, and UDP, the network stack supports an optional BSD packet filter (PF). The syscall server is the component that provides a POSIX interface to user processes. Figure 1 shows individual parts of the stack.

All shaded components in Figure 1 are fully asynchronous, while the syscall server translates synchronous system calls from user processes to asynchronous messages within the stack. The syscall server is the only process of the stack which frequently uses traditional rendez-vous based communication provided by the kernel. All other components communicate using point-to-point channels, which are shared user space memory queues accompanied by fast signaling. This mechanism is located almost purely in user space to take the kernel out of the loop (removing all overhead due to context switches, and pollution of TLBs, caches, and branch predictors).

We take advantage of the x86-specific MWAIT instruction to suspend execution of cores. Thus, we need not send high-overhead interprocessor interrupts, but wake up a waiting core by a mere memory write. Unfortunately, MWAIT is a privileged instruction in Intel chips[1]. If it were not, there would be no need for the kernel for normal mode of operation. We see it as a hardware deficiency.

Our most efficient communication model runs each component on its own dedicated core, so scheduling is not needed and the component can run at anytime out of a warm cache. However, we also allow components to share a core with other processes. In such a case, the scheduler informs the components and they transparently fallback to *notifications*, a standard method provided by microkernels. It delivers special void messages in a similar way to how devices send interrupts to the processor.

It is useful to emphasize that the key performance problems that plagued multiserver systems in the past have been the high overheads due to context switching and scheduling. While the research community heavily optimized the interprocess communication on microkernels like L4 [19] to achieve much better performance, neither of these bottlenecks could ever be eliminated on a uniprocessor. However, dedicating a core to each component fixes both. Further details of the design of the network stack and the fast communication are discussed in [11].

3.2 Dynamic reconfiguration

In contrast to monolithic systems, NewtOS resembles a distributed system. Such systems can embrace diversity and accommodate to a changing environment. This is also true for NewtOS. Each system component can run on a dedicated core or share it with other process and the core can be either big or little. Although we dedicate cores for peak performance, we can consolidate processes on a single core or even a thread if they are not used heavily.

For instance, most of today's traffic uses the TCP protocol, and dedicating a core to the UDP component is probably overkill. On the other hand, when UDP is used heavily (e.g., for video streaming), NewtOS can migrate UDP to its own core. Similarly, the network stack is not used at all times in many deployments, or at least not at its peak throughput. Thus, we can pin all its components to a single dedicated core, or even have it share a core with other processes most of the time. When the workload changes and the system detects an overload of some cores, it can redistribute itself to find the best configuration.

We argue that in many cases SICs are best for the operating system, while BOCs are better suited for applications. However, this is not true always and depends on the situation. For instance, for a web server-like workload, running the server processes on threaded SICs probably delivers higher throughput than using a small number of threads on BOCs.

Even if it allows the stack to run at high speeds, dedicating the BOCs to the network stack is probably not a good idea and the resources can be used more efficiently, unless we run, for example, a complicated intrusion detection algorithm in the packet filter. Likewise, it seems unwise to sacrifice the BOCs to the storage stack. Storage needs to do many unpredictable lookups with little instruction level parallelism while briskly delivering data to the applications and writing them back to a disk. It is likely that we can do so on slower cores, saving the BOCs for the applications. Phrased differently, the components of the system should get the resources they need and no more.

Besides good performance, power consumption is also important. Here also, we should provision a system for its peak performance, while using no more resources than needed during quieter times. The system on a heterogeneous platform can find its sweet spot using only a handful of cores. On platforms with fine-grained power gating, the system can turn off the unused cores and thus save power. Likewise, picking the right type of cores is crucial to balance the performance per Watt ratio. As we show in the evaluation in Section 4, slower cores frequently result in only small drop in performance whereas the potential for power savings is significant.

We do not consider the scheduling in this paper. As there is a lot of work on scheduling in such environments (discussed in Section 5), we are solely interested in the performance and efficiency of different configurations and designing the scheduler for NewtOS is future work.

3.3 Non-overlapping ISA

At this moment, we limit ourselves to heterogeneous architectures with an overlapping ISA [15]. In this section, we argue that by virtue of its design our system has the potential to embrace architectures with different ISAs too. We do not currently have a machine with non-overlapping ISAs on the same processor to evaluate our solution, but we briefly sketch how we can use existing features to support such platforms.

Specifically, we can use NewtOS' *live update* functionality to change the version of a component to run on a different architecture. We originally developed live update to allow us to fix buggy components with new, patched versions without the need of shutting the system down. Doing so greatly reduces maintenance of the system, disruption of its operation, and the time between diagnosing a bug and application of the fix. However, we can also replace a component with the same component compiled for a different ISA.

The update is fairly straightforward since both versions are based on the same code. Mere recompilation with different compiler settings produces the desired version. Moreover, the transition from one ISA type to another is simple because it is done only when the state is stable and the memory layout of data structures on both architectures is likely the same. Finally, we initiate the transition only at the top of the component's main loop, so that we can mostly forget about different layouts of the stack. In case of a discrepancy between the memory layout for each of the ISA versions, we provide an automatically generated transition function [10]. In practice, changing a component to a new architecture is simpler than updating a component to a new version. In contrast to a proper update, both versions for the different ISAs use identical data structures which may differ by offsets and alignment, but not by different items in structures. The system may provide a version for different ISAs when installed or use just-in-time compilation to generate one when need. If migration between ISAs is frequent, the system can cache a version for each to speed up the migration.

We can use the same mechanism as an optimization for overlapping ISAs too. Some cores may have a feature which allows the system code to run faster. For instance, file systems can take advantage of checksum instructions to verify data read from a disk or advanced instructions to encrypt the data. In such cases, the system component does not rely on the extra instructions for its correct operation, but can benefit from them if available.

4 Evaluation on a high-performance stack

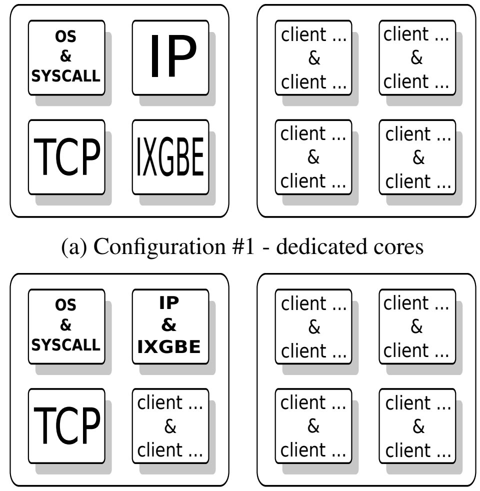
We now evaluate the network stack of NewtOS on a dual socket quad-core Intel Xeon E5520 with hyper-threading.

The peak clock speed of the chips is 2267 MHz and it is possible to scale it down to 1600 MHz in steps of 133 MHz. According to ACPI, power consumption of each chip at its maximal frequency may be as much as 80W and at the lowest frequency 34W. Unfortunately, it is not possible to scale frequencies of the cores independently and all cores of each chip run at the same speed.

However, modern Intel processors like the E5520 can still scale each core independently using thermal throttling[2] to allow further scaling in steps of 12.5% of the clock speed. Thermal throttling means that by setting the chip to 1600 MHz, it is possible to scale down to 200 MHz in the same-sized steps. Although the core still runs at the base frequency (1600 MHz), some cycles are "thrown away" and the execution slows down proportionally. We can do this for each core individually, however both threads on the same core are throttled equally. Thus, the Xeon E5520 allows us to explore both threading, and high/low frequency trade-offs. While we cannot compare in-order versus out-of-order microarchitectures, we believe that a 200 MHz core is slow enough to match the performance of wimpy cores.

To remove bandwidth limitations, and to show that a multiserver system can scale to multigigabit range, we implemented a driver for the i82599 10G Intel network chip. The driver is fairly simplistic but has standard offloading features for the outgoing traffic. We connect our machine to a Linux 3.7.1 system running on a 12-core AMD Opteron 6168 at 1.9 GHz.

Our test case is the same as in [11] which we used to stress the system when demonstrating its reliability. We run an `iperf` server on the Linux machine and connect from NewtOS. `Iperf` is a standard tool for measuring and tuning network performance. The clients send data as fast as possible, trying to saturate the network hardware, the buses, the memory, or the CPU. We verified that the Linux machine is able to receive at 10G by connecting from Linux running on the same machine as we use to run NewtOS. We use multiple streams to get the best performance. LwIP does not support TCP window scaling, and is therefore not able to have enough data in transition to saturate the 10G link on a single stream.



(a) Configuration #1 - dedicated cores

(b) Configuration #2 - hyper-threading

Figure 2: Test configurations - large squares represent the quad-core chips, small squares individual cores and '&' separates processes running on different hyper-treads.

4.1 Test configurations

We experimentally evaluated several configurations of the network stack to determine the most demanding components of the stack. Not surprisingly, TCP ranked highly. Based on these experiments, in performance-critical scenarios, the OS must choose between the two basic setups shown in Figure 2. We will evaluate them across a range of clock settings.

In both cases we place all processes of the core system (OS) on the first CPU and the network stack components involved in processing TCP traffic on the remaining 3 cores of the same chip. These components are TCP, IP and the 10G ethernet driver (IXGBE). Communication between components on different chips is slower as they do not share cache.

The syscall server shares its core with the rest of the system (OS), but runs in a different thread (denoted by the '**&**' symbol). It extensively uses kernel communication, but uses the CPU lightly. Nevertheless, it needs its own thread to use the fast signaling when translating synchronous messages from the clients to the TCP component and back. Otherwise, TCP would need to use *notifications* for correct operation when replying to the syscall server, resulting in a serious performance hit.

In both configurations, TCP has its own dedicated core, the spare hyper-thread is idle. The two configurations differ in only one thing. The first configuration (Figure 2a) also dedicates a full core to IP and the driver while the respective other threads are idle. The second configuration (Figure 2b) places both IP and the driver on the same core, but runs them in different threads. The rationale behind this choice is that TCP is the most demanding component while IP and the driver have similar CPU utilization as we demonstrate in the remainder of this section. We denote the second configuration with HT for employment of hyper-threading.

The test clients run on the remaining cores and threads. In other words, in configuration #1 (Sections 4.3 and 4.4), they all run on the other quad-core chip, while in configuration #2 (Section 4.5), they also run on one of the cores of the first chip. The scheduler distributes them equally.

MHz	drop	**Mbps**	drop	**TDP(W)**	drop
2267	–	8641	–	80	–
1867	18%	8152	6%	48	40%
1600	30%	7840	9%	34	57%

Table 3: Performance loss versus potentially saved power

4.2 Methodology of measurements

Throughout our experiments, we measure two basic values: (1) CPU utilization for each component, and (2) bitrate. We use time stamp counters to time the events. Intel guarantees that they are synchronized across all cores and tick at a constant rate regardless of frequency scaling. Each component has its log for events which we process after a test run finishes.

To measure the CPU utilization, we mark an event right before the kernel call which suspends the core and right after the call returns. In fact, this measures time actively spent in each component, so the actual CPU utilization is higher. Measuring the time this way is closer to using a single long latency instruction instead of a kernel call.

4.3 Frequency scaling 2267–1600 MHz

The first experiment is to explore how configuration #1 behaves when we change the frequency of the chip. We present the measurements in Table 3. The first line represents the baseline: all the cores run at the peak clock speed and the chip draws maximum power. As expected, we see that the bitrate drops when the clock speed goes down. As the drop is fairly small, we show only one intermediate value. The last line stands for the lowest frequency and power consumption.

Observe that scaling the cores down to the lowest frequency can save up to 57% of power, but the drop in throughput is not nearly as significant, a mere 9%. There are many cases in which 7.8 Gbps is more than enough while the opportunity to save 46 Watts is important. Also note that at maximum power the throughput is 8.6 Gbps. Later in this section we show that it is possible to throttle the cores even more and deliver better throughput than at the peak clock speed.

The CPU utilization measurements show that running the stack on high frequencies is probably suboptimal. The TCP component uses the core at approximately 70% while IP and the driver use their cores below 40%. The components spend much of their time polling the communication channels. If there is no work to do, they poll for a little while longer and eventually call the kernel to block them on MWAIT.

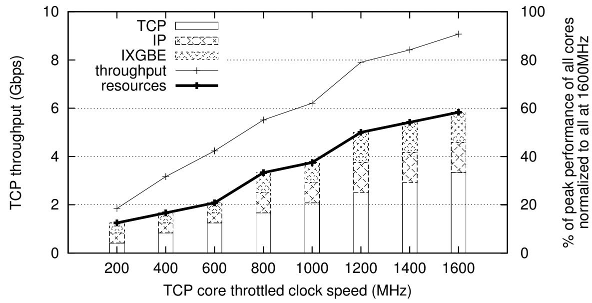


Figure 3: Throughput compared to the combined performance of the resources in use — the best combinations

4.4 Throttling below 1600 MHz

To demonstrate how our system behaves on much slower cores we use thermal throttling to artificially slow the cores down. We start our measurements by scaling all 3 cores to the minimum. Since TCP is the component which uses its core the most, we scale it up by one step for each new measurement and we try to match it with the best setting for the other 2 cores. Our experience is that if we increase the speed of the TCP core and the bitrate does not improve proportionally, we must speed up the other cores by one step too. Adding more does not help and may even lead to throughput degradation. We present our results for the best configurations in Figure 3, which compares the bitrate (the thin line) and the performance of the cores we need (the bars). In this case, 100% is the combined performance of all 3 cores running unthrottled at 1600 MHz. The thick line connects tops of the bars to highlight how the throughput scales with the added resources.

The important observations in Figure 3 are :

- Scaling the 3 cores to 12.5% of their total performance (200 MHz) delivers 1.8 Gbps which is enough for many applications like video streaming, web browsing or online gaming.
- The stack achieves approximately the same or higher throughput (7.9 Gbps) at 50% of resource utilization (bar 1200 MHz) than when all cores run unthrottled (7.8 Gbps as we reported in Table 3).
- Using TCP core clocked at 1600 MHz and the other two at 600 MHz is just 60% of performance of all of them running at 1600 MHz and only 40% of all running at 2267 MHz. This “low-power” configuration exceeds the performance of all cores at 1600 MHz as well as at 2267 MHz.

We emphasize that results are *average* bitrates of each test run. The average throughput at 60% of the combined

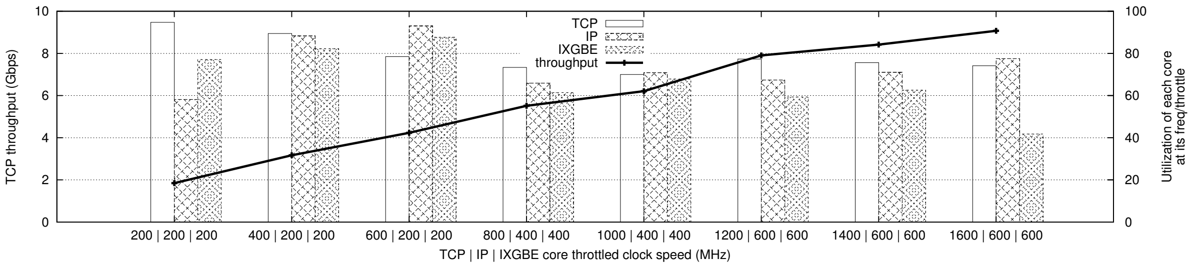


Figure 4: Configuration #1 – CPU utilization of each core throttled to % of 1600 MHz.

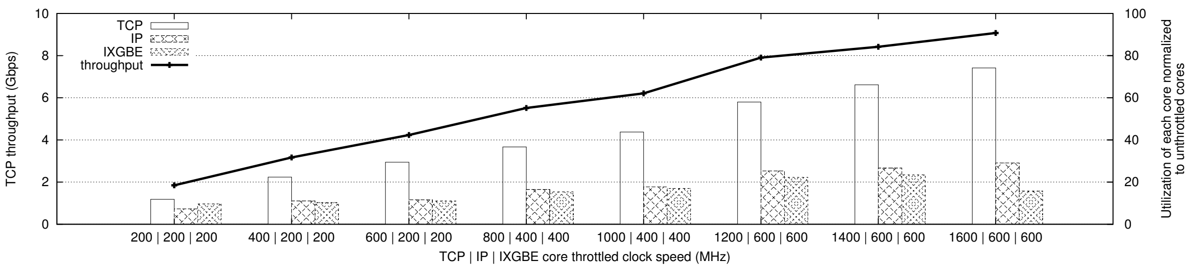


Figure 5: Configuration #1 – CPU utilization normalized to cores at 1600 MHz unthrottled.

resources (the rightmost result in Figure 3) reaches up to 9.1 Gbps with peaks approximating 10 Gbps. Slower sometimes really *is* faster!

Figure 4 presents CPU utilization of each core. The utilization is with respect to each core's throttling and each set of bars stands for one configuration. The sets form three clusters determined by the speed of the slower cores. The three sets in the first cluster show how the utilization of the IP and driver cores increases as the higher frequency permits TCP to process more data. In the third set the utilization of the slower cores approaches 100% and exceeds utilization of the TCP core. Therefore we must match speeding up TCP by speeding up the others too, if the current throughput is not enough. The same pattern repeats in each of the clusters.

Although the relative utilization of the TCP core drops slightly, Figure 5 clearly shows that the CPU time obtained by TCP directly determines the final throughput. Figure 5 presents the same data as Figure 4, but normalized to a core running unthrottled. The important observation is that the utilization of the other two cores does not grow equally fast. The main reason is that unlike TCP, IP and the driver do not touch the TCP payload. TCP must copy all the data from the client applications to the address space of the stack. It is a lengthy operation which thrashes caches and makes the core stall while the time is reported as used. The copy overhead can be significant, between 60 and 70%.

Interestingly, the faster the core, the higher the overhead. The explanation is that the difference between CPU speed and memory speed grows leading to more stalls. Without the copy overhead, CPU utilization would be comparable to the other two components. For completeness, we mention that some of this overhead can be reduced by letting the network devices transfer data directly from and to the user space buffers.

We did not measure throttling for higher clock speeds than 1600 MHz, because the results show that increasing the speed does not yield significant benefits, and because we want to make the point that lower clock speeds are good enough for even the most demanding OS components. Although we can only guess how much energy would our emulated low power cores use, for example, Intel reports that the thermal design power is less or equal to 3.5W for its Atom N2600 processors at 1.6GHz.

4.5 Hyper-threading

The same set of experiments for configuration #2 evaluates the effect of threaded cores. Threads are not equal to full cores as they share the same pipeline. Their advantage is that they allow the core to use cycles which would be otherwise wasted when the pipeline stalls due to slow memory. If the code running on one of the threads has a high cache hit rate and good branch prediction, execution of additional threads has diminishing returns. However, we do not expect system code to behave optimally. Mes-

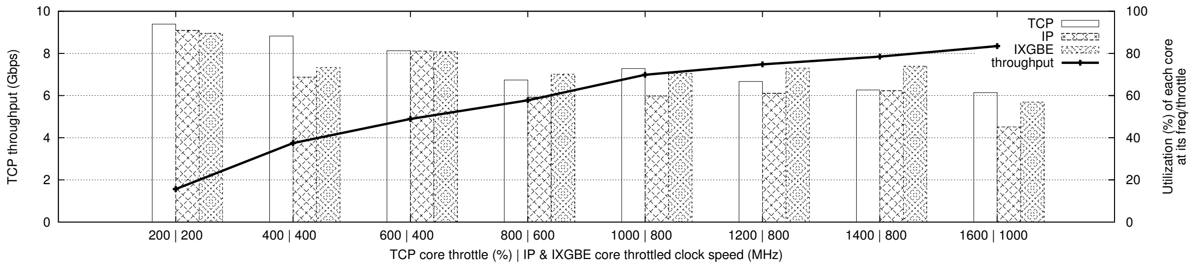


Figure 6: Configuration #2 (HT) – CPU utilization of each core throttled to % of 1600 MHz.

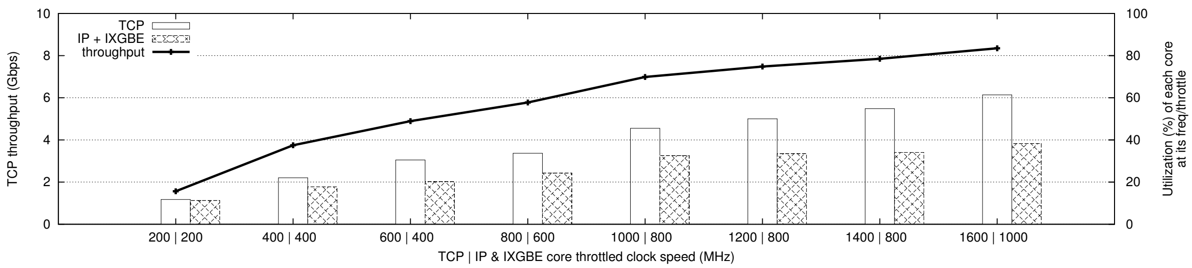


Figure 7: Configuration #2 (HT) – CPU utilization of each core normalized to 1600 MHz.

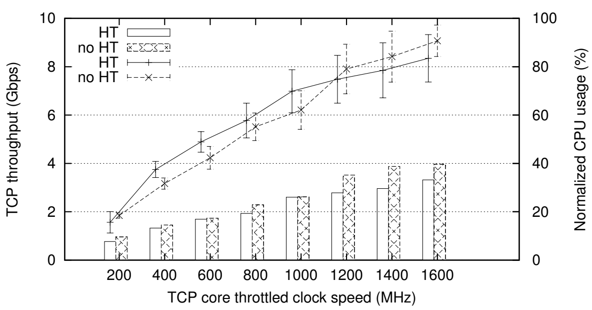


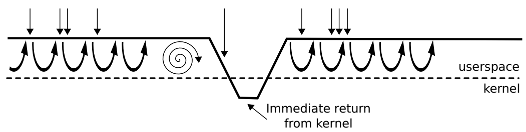
Figure 8: Comparison of configuration #1 (no HT) and #2 (HT). Lines represent bitrate, bars represent CPU utilization normalized to 3 cores at 1600 MHz

sages created on different cores are not in the remote cache until read for the first time and the CPU can hardly guess which execution path the code takes in the next loop. Therefore system code should benefit from threading.

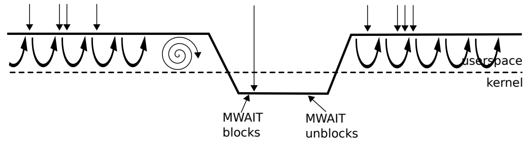
Based on the previous measurements of configuration #1 and the fact that a thread is not a full-blown core, we expected that the core which hosts IP and the driver should run at least at double the speed of a core hosting either IP or the driver to deliver the same throughput. Figure 6 shows that the actual clock speed we require is sometimes equal to, but mostly less than what we expected. Figure 7 shows the normalized values. The crosshatched bar represents the utilization of IP and the driver running on the same core. Since each component also accounts the time when their threads are not active the utilization of a single core could exceed 100%. Therefore, the bar represents mean value of both threads.

The main reason why we could run at lower clock speeds than we originally expected, is that running more threads on the same core, uses the cycles of the core more efficiently and reduces the amount of sleep time. Since the execution of both processes is interleaved, there is a higher probability that while a processes' thread is inactive, the other processes of the stack create some new jobs. Thus, when the thread activates again, instead of finding the work queues empty, the process can carry on. The benefits of sharing a core between IP and the driver is the easiest to observe when comparing the experiments with the slowest cores. Although using two cores at 200 MHz is just 66% of the resources of 3 dedicated cores at the same speed, the throughput is 77% or 1.4 Gbps.

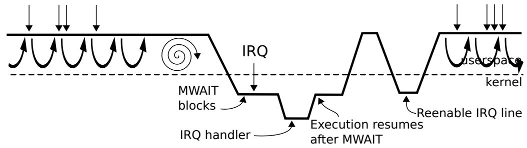
Figure 8 compares the performance of configurations #1 and #2. As long as the variance in the bitrate is low, using the threaded core outperforms configuration #1 with an extra core. The bars present the combined CPU utilization of both configurations normalized to all 3 cores running unthrottled at 1600 MHz. In all cases the normalized utilization is lower for configuration #2 while the performance is higher when TCP core runs at or below 1000 MHz. As the transmission of data gets more bursty, the ability to use more cycles per time unit on the dedicated cores to get the work done quicker, outweighs the



(a) Job arrives between deciding to sleep and halting the core



(b) Job arrives when execution is suspended on MWAIT



(c) Driver - an IRQ arrives when execution is suspended

Figure 9: Sleep overhead in different situations. The thick line represents transition between execution in user space and kernel in time. Arrows mark job arrivals, loops denote execution of the main loop and spirals denote polling.

Clock speed (MHz)	Bitrate (Gbps)
2267	4.3
1600	3.4
200	0.4

Table 4: Bitrate vs. CPU clock speed on a single core

benefits of threading. Higher bitrate leads then to higher CPU utilization.

4.6 Why is slower faster?

When a job queue is empty, we can either keep on polling which is fine only when jobs arrive frequently. Otherwise the cores use energy while not doing any useful work. The common (and probably right) thing to do is to put the cores to sleep when there is no work to do. Idly waiting for new jobs to arrive is costly when we sleep at the wrong moments, because we reduce the time we could have used for processing until the overhead due to sleeping becomes so high that we observe the “slower is faster” effect.

As we cannot predict the future, we may put a core to sleep just before a new job arrives (Figure 9a). As we still use a kernel call for sleeping, the call introduces some latency, even though the execution does not block and returns to user space immediately. The way back through the kernel is not free. It is even more expensive when a job arrives just after suspending the core (Figure 9b) as MWAIT has a relatively long latency, in the order of microseconds. Nevertheless, blocking on MWAIT is much faster than using traditional kernel IPC. Especially, since such IPC would slow down the *sending* core too.

The worst case is when an interrupt wakes up a driver core. The interrupt handling routine adds up to the MWAIT latency. In addition, the interrupt line should stay disabled until the driver masks the interrupt in the device. At that point, it has to reenable the interrupt line, which incurs another trip to the kernel (Figure 9c). Thus, drivers are the most sensitive to frequent sleeping. The solution may be to run the driver in the privileged mode of a *virtual machine*.

Polling harder eliminates some of the “slower is faster“ effect. However, designing a polling algorithm which adapts to unpredictable conditions is complicated. The ideal solution is to use an efficient *sleep* instruction in user space. On the other hand, sleeping will always have some latency. The take-away message is the following: to avoid the *expensive* idle time, the scheduler should pick the cores and hyper-threads on which it places the components carefully and scale them so that they are always highly used—with little opportunity to sleep.

4.7 Stack on a single core

In case of shortage of cores due to high demand from applications, or when cores are turned off to save power, the entire network stack of NewtOS can keep operating on a single core. Table 4 presents measurements of the stack’s performance as a function of the core speed. The stack has a throughput of up to 4.3 Gbps on a big fast core and 400 Mbps on a 200 MHz wimpy core. The throughput of the slow core is good enough for many common activities, but the fast core cannot scale further. More importantly, a network stack running on a single core has a much higher latency. If a process has work to do, it hogs the core until it exhausts its time quantum while others are on hold. Then the scheduler is free to pick any runnable process of the stack which increases non-determinism in the execution. Running the stack on dedicated cores removes these deficiencies and the throughput of a single fast core is similar to the configuration with a TCP core at 600MHz and IP and driver cores at 200MHz.

5 Related work

Kumar et al. proposed single-ISA heterogeneous multicores for power reduction [15] and to improve performance of multithreaded workloads [16]. They demonstrated that applications need a good mix of single-threaded performance and high throughput. Due to the diversity in application code, heterogeneous platforms outperform homogeneous ones with the same die size. This makes the heterogeneity promising for the future. Although we do not focus primarily on applications but on the operating system code, the similarity is that we also exploit the fact that each system component has its own requirements for optimal execution. Moreover, the system components are user space processes like applications. However, the system code differs from applications, thus its optimal requirements are different too.

Operating systems are key to leveraging the heterogeneous platforms as only the scheduler can make decisions where each application runs, therefore the schedulers got the most attention. Kumar et al. [16] proposed a whole range of sampling heuristics that permute threads on cores to find the best assignment. As the execution of applications changes during different phases, Becchi et al. [6] proposed a dynamic algorithm which constantly measures the IPC ratio of threads and tries to run on the big cores those threads that would benefit the most.

Permuting the threads and sampling them on all types of cores is an overhead. Therefore Koufaty et al. [14] designed a scheduler which monitors execution of each thread on its current core only. It uses existing low overhead performance monitoring counters to collect performance related data which the system can use to estimate what type of a core is the most suitable for the given thread. This algorithm relies on a model which translates the performance statistics to the bias of each thread to a certain type of a core.

Most heterogeneous scheduling algorithms use the speed up factor, the ratio between how fast an application runs on a small and a big core. Saez et al. [25] suggest a more comprehensive utility factor of how effectively the whole mix of running application uses the machine.

Instead of using available performance counters to feed data into the models which predict the performance of the threads on different types of cores, hardware monitoring and prediction engines [27] and performance impact estimators [29] were proposed as hardware extensions. The hardware estimates the possible speed-up on its own and the scheduler can use this direct feedback to decide which applications would benefit from running on the big cores and which can run on the small ones.

In contrast to this work, we do not focus on the performance of applications, instead our main focus is on the system. First of all, our system can use all the different heuristics or hardware estimations to schedule application on the cores which are not dedicated to the system. Second, our system is a collection of user space processes and the scheduler can use the same or similar techniques to find their optimal placement. On the other hand, execution of the system differs in several aspects. Each component is responsible for a small subset of all the system tasks, therefore they have little variance during their execution as the requests they serve are similar. The system code follows the same patterns which differ from application code. The scheduler's goal is not to let the system finish as quickly as possible, but to deliver optimal service to the changing mix of applications and workloads using the available resources. In contrast to the applications, which are opaque for the system, the system designer knows more about the system components and the components themselves can help the scheduler by providing various hints. For instance, a component can detect and signal when the recipient of its messages cannot keep up and thus may benefit from a faster core. Similarly, applications can give hints to the system, for instance, when the estimated time of downloading a file is in minutes, the application can tell the system, that it is not a sudden spike in the load and reconfiguring is worthwhile.

Mogul et al. proposed operating system friendly cores in [20], primarily to save power. They argue that many features which the operating systems do not use draw a lot of power while not contributing to performance of the operating system. They propose that the system should run on the *optimized* cores and the execution should transfer from the application cores to the system cores when necessary. The migration is a bottleneck which they address in [28]. Migrating the execution means that the cache locality is poor. In contrast to their experiments with Linux, we have a system which is more suitable for heterogeneous platforms. NewtOS moves execution only by sending a message to another core and benefits from cache locality of the code and data of the component running on the core. Of course, locality of the user data passed between the cores is poor, however, in many cases the components do not need to touch the data until the DMA of a device transfers them. Cache locality of the messages is also poor, but this data should not be cached after the message is sent in the first place. Unfortunately, the current hardware does not allow us such a fine-grained control over cache and data transfers. Strong et al. [28] also use networking for evaluation. They model the power usage of the hypothetical cores while we use frequency scaling to approximate performance of such a hardware.

Netmap [24] and OpenOnload [3] projects demonstrated high bandwidth networking in user space. In contrast to NewtOS, both need a driver in a monolithic kernel. Although most of the faults crash only the application, there is still a chance that a bug in the driver can bring the whole system to a halt. Netmap shows that a 900 MHz core is good enough to transfer 10 Gbps of small packets between the device and the user space application, however, netmap only deals with routing and does not offer a generic networking support to applications. On the other hand, OpenOnload transparently intercepts any application requests and uses a library with custom made hardware to transfer data directly between applications and devices. We endorse this approach as it would remove the copying overhead between applications and our stack.

6 Conclusions

We have demonstrated that a processor's fast cores may not be ideal for system workloads and that less can be more in some situations. We presented a network stack evaluation of a reliable and dependable system. The results support our claim that it is possible for such a system to perform well, using much more constrained resources than usually available. We use current hardware to approximate future processors and we show the potential benefits. Unlike many other researchers, we do not focus on the applications. The operating system plays a key role in the execution of applications and we should give it equal attention. However, performance should not be the only criterion, the system is also responsible for security, reliable execution and easy maintenance. NewtOS design recovers from crashes and allows administrators to update its components while it is running. Although our case study covers only one part of a generic operating system, we are confident that the findings apply to other parts and to other systems as well.

Acknowledgments

This work has been supported by the ERC Advanced Grant 227874 and EU FP7 SysSec project. We would like to thank Valentin Priescu for implementing the frequency scaling driver. Likewise, we would like to thank Dirk Vogt for implementing the IXGBE driver for MINIX 3.

References

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

[2] NVIDIA - Variable SMP architecture. `http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf`.

[3] OpenOnload. `http://www.openonload.org/`.

[4] The Intel Xeon Phi Coprocessor. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[5] Samsung to outline 8-core big.LITTLE ARM processor in February. `http://www.engadget.com/2012/11/20/samsung-to-outline-8-core-big-little-arm-processor-in-february/`, Nov. 2012.

[6] Becchi, M., and Crowley, P. Dynamic Thread Assignment on Meterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd conference on Computing frontiers* (2006), CF '06.

[7] Cristiano Giuffrida, L. C., and Tanenbaum, A. S. We Crashed, Now What? In *Proceedings of the 6th International Workshop on Hot Topics in System Dependability* (2010).

[8] Dunkels, A. Full TCP/IP for 8-bit architectures. In *International Conference on Mobile Systems, Applications, and Services* (2003).

[9] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of ASPLOS-XVIII* (2013).

[10] Giuffrida, C., and Tanenbaum, A. S. Safe and Automated State Transfer for Secure and Reliable Live Update. In *Proceedings of the Fourth International Workshop on Hot Topics in Software Upgrades* (2012).

[11] Hruby, T., Vogt, D., Bos, H., and Tanenbaum, A. S. Keep Net Working - On a Dependable and Fast Networking Stack. In *Proceedings of Dependable Systems and Networks (DSN 2012)* (Boston, MA, June 2012).

[12] Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture* (2007).

[13] Khubaib, Suleman, M. A., Hashemi, M., Wilkerson, C., and Patt, Y. N. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.

[14] Koufaty, D., Reddy, D., and Hahn, S. Bias Scheduling in Heterogeneous Multi-Core Architectures. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10.

[15] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (2003).

[16] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st annual international symposium on Computer architecture* (2004), ISCA '04.

[17] Latif, L. IDF: Intel is looking at ARM's Big Little architecture. `http://www.theinquirer.net/inquirer/news/2205764/idf-intel-is-looking-at-arms-big-little-architecture`.

[18] Li, T., and John, L. K. Operating system power minimization through run-time processor resource adaptation. *Microprocessors and Microsystems 30*, 4 (2006).

[19] Liedtke, J., Elphinstone, K., Schönberg, S., Hrtig, H., Heiser, G., Islam, N., and Jaeger, T. Achieved IPC Performance (Still The Foundation For Extensibility), 1997.

[20] Mogul, J. C., Mudigonda, J., Binkert, N., Ranganathan, P., and Talwar, V. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro 28*, 3 (May 2008).

[21] Nellans, D., Balasubramonian, R., and Brunv, E. A Case for Increased Operating System Support in Chip Multiprocessors. In *In Proc. of 2nd IBM Watson P=ac 2* (2005).

[22] Olukotun, K., Hammond, L., and Laudon, J. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. 2007.

[23] Redstone, J. A., Eggers, S. J., and Levy, H. M. An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proceedings of ASPLOS-IX* (New York, NY, USA, 2000).

[24] Rizzo, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012), USENIX ATC'12.

[25] Saez, J. C., Fedorova, A., Koufaty, D., and Prieto, M. Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems. *ACM Trans. Comput. Syst. 30* (Apr. 2012).

[26] Soares, L., and Stumm, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. of Symp. on Oper. Sys. Des. and Impl.* (2010).

[27] Srinivasan, S., Zhao, L., Illikkal, R., and Iyer, R. Efficient Interaction Between OS and Architecture in Heterogeneous Platforms. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 62–72.

[28] Strong, R., Mudigonda, J., Mogul, J. C., Binkert, N., and Tullsen, D. Fast Switching of Threads Between Cores. *SIGOPS Oper. Syst. Rev. 43* (April 2009).

[29] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12.

Notes

[1] MWAIT is optionally unprivileged in AMD chips starting with family 10h, but we use Intel due to hyper-threading and better scaling.

[2] It is usually possible to scale AMD chips to lower speeds than Intel ones (e.g., from 1.9 GHz to 800 MHz). However, the Intel-specific mechanism of thermal throttling allows us to emulate much lower speeds