Heterogeneous Multicores: When Slower is Faster

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

ABSTRACT

It is well-known that 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. 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 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 speeds of the cores. 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. The *big.LITTLE* ARM combines two big Cortex-A15 cores with two little Cortex-A7 on the same die. The Tegra-3 is a Cortex-A9-based quad-core CPU that includes a fifth "companion" Cortex-A9 that is slower and less power hungry. For sheer number of cores, the 50+ core x86-compatible Intel Xeon Phi processor is especially impressive.

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.

The research community has advocated such heterogeneity for many years [8] 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 with a few exceptions like [12, 18]. This is remarkable, because the operating system performs a significant amount of work on behalf of the applications [15, 13].

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

High reliability For instance, our operating system supports dynamic updates without any downtime and survives crashes of key OS components. We described these aspects in [6, 3], and [5], and will not discuss them further in this paper.

High performance Building on a design described in [6], we show that we support network processing at 10 Gbps on COTS hardware (this paper),

Tailored resource utilization We map the OS components to suitable cores and show that even wimpy cores deliver competitive performance (this paper).

Multiserver systems, composed of many independent processes (servers) that implement various OS functions, typically trade reliability for performance. NewtOS [6], a high-performance derivative of MINIX 3 with a completely redesigned network stack, shows that it is possible to mitigate the overhead by dedicating cores to system servers, which communicate through asynchronous user space channels without kernel involvement. With their own cores the system servers can run whenever needed from a warm cache, without having to compete with other processes or waiting for the kernel to schedule them. Moreover, the asynchrony allows 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 improve TCP throughput from hundreds of megabits per second to gigabits.

The cores of common platforms are designed for generic usage and over-provisioned for running OS code. 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.

# i7 cores	# i7 threads	# Phi cores	# Phi threads
4	8	44	176
8	18	27	108
12	24	10	40

Table 1: Estimation of options for different configurations that merge Core i7 and Xeon Phi cores

In this paper, we explore the hardware design space in an appropriate manner by emulating the future platforms on current hardware using frequency scaling. We show that our system can deliver the same or better performance with smaller, simpler and slower cores—without compromising reliability. Our case study is high bandwidth networking.

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

2. BIG CORES AND LITTLE CORES

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.

We start our discussion with a comparison of fast cores and slow cores. Specifically, Intel Core i7 with the Knights Ferry processor. The quad-core Core i7 "Bloomfield" is a prime example of a big out-of-order core with a design that is geared for high single threaded throughput. It is produced by 45nm technology and die size of 263 mm^2. In contrast, the 32 cores on the Knights Ferry (45nm, dies size est. 700 mm^2) are much simpler in-order cores that provide only a fraction of the Core i7's performance.

Compared to the Core i7, the Knights Ferry die hosts 8× the number of cores and 16× the number of threads while the difference in die size per core is 3× (and 6× per thread). The cache size per core is obviously smaller, but threaded cores can compensate for this [14]. The difference in the peak clock speed is equally remarkable (3.3 GHz vs. 1.2 GHz).

The successor of Knights Ferry, a recently released product called Xeon Phi, has even a larger number of cores. Intel markets it as a "50+ core beast" and released up to 62 cores on a single die. With each core hosting 4 threads of execution, this amounts to 200+ threads on a single chip. It is likely that future designs will see interesting new combinations as Intel may well merge its Xeon Phi with other Intel cores on a single die [10].

Tab. 1 shows different combinations of Core i7 and Xeon Phi cores, taking into account the number of transistors for a quad-core Core i7 in 22nm technology (1.4 billion) as well as the number of transistors of a 62-core Xeon Phi produced by the same technology (5 billion). Given these transistor counts, Tab. 1 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.

As consolidating multiple components on a single core saves resources when peak performance is not needed, more threads are attractive as containers for context of processes. Hardware does the context switching (and not the software) and we can suspend and resume efficiently with instructions like MWAIT. NewtOS has about 30 system processes in its default installation out of which about 10 are important for performance. Tab. 1 shows that even with twelve i7 cores, there would be enough threads to dedicate one to each of our system's processes and plenty of big cores for applications.

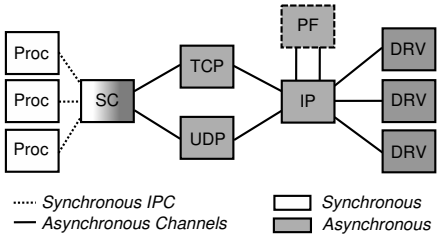

Figure 1: Design of the NewtOS network stack

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.

3.1 The NewtOS network stack

The heart of the NewtOS network stack is LwIP [4], a simple and portable network stack for embedded systems used by many research projects.

One of the main design goals of NewtOS is reliability. Thus, we allow even core components of the operating system to be replaced on the fly, without taking the system down (and often with no noticeable disruption at all) [5]. 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).

Fig. 1 shows individual parts of the stack. All shaded components in the figure are fully asynchronous, while the syscall (SC) 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 almost entirely (removing all overhead due to context switches, and pollution of TLBs, caches, and branch predictors). We only use a kernel call to execute the MWAIT instruction when a component becomes idle.

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, we transparently fallback to *notifications*, a standard method provided by microkernels.

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 [11], 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 it uses are discussed in [6].

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. Likewise, the core can be either big or little.

Although we claim that we need a dedicated core or at least a thread for the important processes, it is possible to consolidate processes on a single core (or even a thread) if they are not in heavy use. 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 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 dedicate a core to all processes of the stack, or even have it share a core with other processes most of the time. When the workload changes the system can redistribute itself to find the best configuration. 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 Sec. 4, slower cores frequently result in only small drop in performance whereas the power savings are significant.

We do not consider the scheduling as there is a lot of related work on heterogeneous scheduling. We are solely interested in the performance and efficiency of different configurations.

3.3 Non-overlapping ISA

At this moment, we limit ourselves to heterogeneous architectures with an overlapping ISA [8]. On the other hand, our system has the potential to embrace architectures with different ISAs too. Specifically, we can use NewtOS' *live update* functionality to 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, the data structures are the same and we can provide an automatically generated transition function [5] to deal with the architectural differences.

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

4. EVALUATION ON A NETWORK STACK

We 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. It is not possible to scale the frequency of the cores of one chip independently. However, modern Intel processors can still scale each core independently using thermal throttling to allow further scaling in steps of 12.5% of the clock speed[2]. This means that by setting the chip to 1600 MHz, it is possible to scale down to 200 MHz in the same-sized steps. While we cannot compare in-order versus out-of-order microarchitectures, we believe that 200 MHz is slow enough to match the performance of slow in-order cores.

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 [6] 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 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.

4.1 Test configurations

We experimentally evaluated several configurations of the network stack to determine the most demanding components. Not surprisingly, TCP ranked highly. Based on these experiments, we opted for two basic setups, which we evaluate across a range of clock settings.

In both cases we place all processes of the core system 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). The syscall server shares its core with the rest of the system since it uses the CPU lightly. It runs in a different hardware thread as it needs it to use the fast signaling when translating synchronous messages from the clients to the TCP component and back.

In both configurations, TCP has its own dedicated core and its second thread is idle. The first configuration (#1) also dedicates a full core to IP and the driver while the

[2]It is usually possible to scale AMD chips to lower speeds than Intel ones, however, the Intel-specific mechanism of thermal throttling allows us to go as low as 200 MHz.

Freq	Mbps	Mbps drop	Watts	Power saving
2267	8641	–	80	–
1867	8152	6%	48	40%
1600	7840	9%	34	57%

Table 2: Performance loss compared to saved power

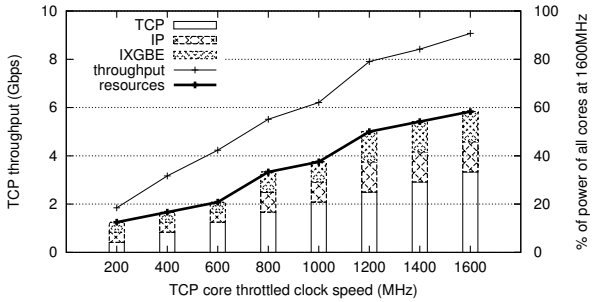

Figure 2: Throughput compared to the resources in use

respective other threads are idle. The second configuration (#2) places both IP and the driver in different threads of one core. TCP is the most demanding component while IP and the driver have similar CPU utilization as we demonstrate in the remainder of this section. The scheduler distributes the test clients equally on the threads of the remaining cores.

4.2 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 Tab. 2. 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. Scaling the cores to the lowest frequency can save 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 enough while saving 46 Watts is important.

The TCP component uses the core at approximately 70% while IP and the driver use their cores below 40%, which is suboptimal. Waking a core up from a sleep has long latency. Even so, blocking MWAIT is much faster than using traditional kernel IPC. Especially since traditional IPC would slow down the *sending* core too. To avoid the *expensive* idle time, the scheduler should scale the cores on which it places the components so that they are always highly used—with little opportunity to sleep.

4.3 Throttling below 1600 MHz

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. We present our results for the best configurations in Fig. 2, which compares the bitrate and the power of the cores we need (the bars). In this case, 100% is the combined power of all 3 cores running unthrottled at 1600 MHz. The important observations in Fig. 2 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 higher throughput (7.9 Gbps) at 50% of resource utilization (bar 1200 MHz) than when all cores run unthrottled at 1600 MHz (7.8 Gbps as we reported in Tab. 2).
- 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 while this low-power configuration exceeds the performance of both.

We emphasize that results are *average* bitrates of each test run. The throughput at 60% of the combined resources (the rightmost result in Fig. 2) reaches up to 9.1 Gbps with peaks approximating 10 Gbps. Slower sometimes really *is* faster!

Fig. 3 presents the CPU utilization of each core. The utilization is with respect to each core's throttling and each set of bars stands for one setting of the cores' speeds. 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 it is necessary to speed them up if the current throughput is not enough. The same pattern repeats in each of the clusters.

Utilization of the TCP core grows faster than of the other two because IP and the driver do not touch the TCP payload. TCP must copy all the data from the sending clients to the address space of the stack. The copy overhead is between 60 and 70%, however, there are ways to reduce it [1].

4.4 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.

Besides using the core's cycles more efficiently, hardware threads reduce the amount of expensive 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, the process can carry on. Comparing the experiments with the slowest cores show that using two cores at 200 MHz is just 66% of the resources of 3 dedicated cores at the same speed, but the throughput is 77% or 1.4 Gbps. This is still plenty for many applications, especially in the embedded world where such low-powered cores are common.

Fig. 5 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. Note that in all the cases, the clock speed

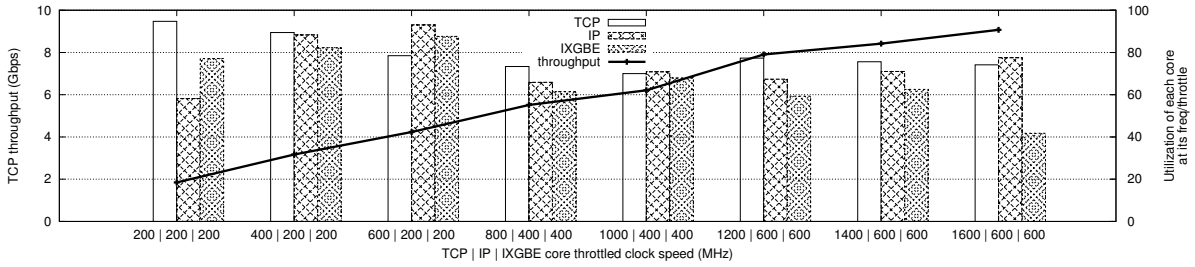

Figure 3: Configuration #1 – CPU utilization of each core throttled to % of 1600 MHz.

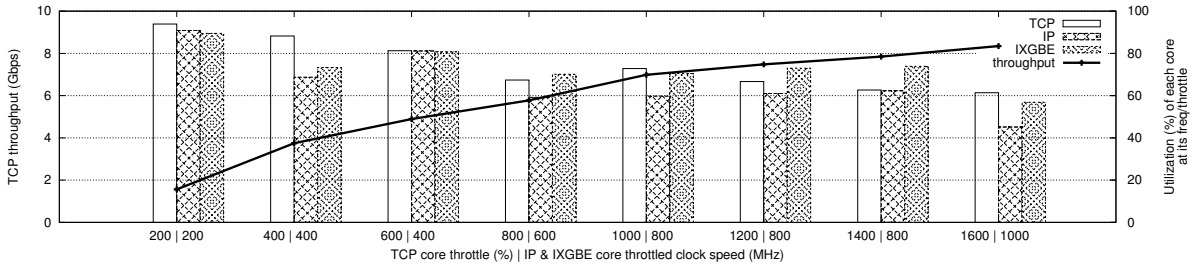

Figure 4: Configuration #2 (HT) – CPU utilization of each core throttled to % 1600 MHz.

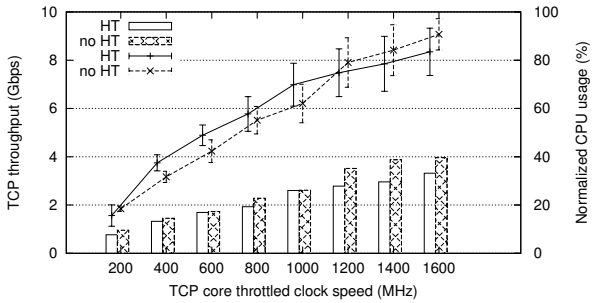

Figure 5: Comparison of configuration #1 (no HT) and #2 (HT). Lines represent bitrate, bars represent CPU utilization of each configuration normalized to 3 cores at 1600 MHz

of the threaded core is less than or equal to double the speed of the respective dedicated cores. The bars in Fig. 5 present the combined CPU utilization of both configurations normalized to the power of 3 cores running unthrottled at 1600 MHz. In all cases the normalized utilization is lower for configuration #2 while the performance is higher. As the transmission gets more bursty, faster clock speed on the dedicated cores (to get the work done quicker) outweighs the benefits of threading. Higher bitrate leads then to higher CPU utilization.

4.5 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. The stack has throughput of up to 4.3 Gbps on a big fast core clocked at 2267MHz, 3.4 Gbps at 1600 MHz 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 multi-cores for power reduction [8] and to improve performance of multithreaded workloads [9]. 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.

Heterogeneous platforms inspired many novel scheduling algorithms. Kumar et al. [9] proposed a whole range of sampling heuristics that permute threads on cores to find the best assignment. Becchi et al. [2] 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. As permuting the threads and sampling them on all types of cores is an overhead, Koufaty et al. [7] 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 data and a model which translates the performance statistics to the bias of each thread to a certain type of a core. The algorithms mostly use the speed up factor, the ratio between how fast an application runs on a small and a big core. Saez et al. [17] suggest a more comprehensive utility factor of how effectively

the whole mix of running application uses the machine.

Instead of using available performance counters as input of the models which predict the performance on different types of cores, hardware monitoring and prediction engines [19] and performance impact estimators [21] were proposed as hardware extensions. The hardware estimates the possible speed-up and the scheduler can uses this feedback to decide which applications would benefit from running on the big cores and which can run on the small ones.

Our system can use the different heuristics or hardware estimations to schedule applications. Second, our system is a collection of user space processes and the scheduler can use the same techniques to find their optimal placement. On the other hand, scheduler's goal is not to let the system finish as quickly as possible, but to deliver optimal service to the changing mix of workloads using the available resources. In addition, the system components can themselves actively 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 may benefit from a faster core.

Mogul et al. proposed operating system friendly cores in [12], 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 [20]. FlexSC [18], 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. 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.

Netmap [16] and OpenOnload [1] projects demonstrated high bandwidth networking in user space. In contrast to NewtOS, both need a driver in a monolithic kernel, hence there is still a chance that a bug can bring the whole system to a halt. Netmap shows that a 900 MHz core is good enough to transfer 10 Gbps between the device and the user space, however, it does not offer a generic networking support to applications. OpenOnload transparently intercepts any application requests and uses custom made hardware to transfer data directly between applications and a device. We endorse this approach as it would remove the copying overhead in TCP between the 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 an evaluation of network stack 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. However, performance should not be the only criterion, the system is also responsible for security, reliable execution and easy maintenance. NewtOS 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 European Research Council Advanced Grant 227874. We would like to thank Valentin Priescu for implementing the frequency scaling driver. Likewise, we would like to thank Dirk Vogt for implementing the MINIX 3 version of the IXGBE driver.

7. REFERENCES

[1] OpenOnload. http://www.openonload.org/.

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

[3] 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).

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

[5] 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).

[6] 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* (2012).

[7] Koufaty, D., Reddy, D., and Hahn, S. Bias Scheduling in Heterogeneous Multi-Core Architectures. EuroSys '10.

[8] 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).

[9] 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 *Proc. of the 31st Annual Int. Sym. on Computer Arch.* (2004).

[10] 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.

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

[12] 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).

[13] 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).

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

[15] 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* (2000).

[16] 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.

[17] 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).

[18] 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).

[19] 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.

[20] 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).

[21] 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 Int. Symp. on Computer Architecture* (2012).