Towards Optimal Scheduling of Multiserver System Components

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

Abstract

Until recently, microkernel-based multiserver systems could not match the performance of monolithic designs due to their architectural choices which prefer high reliability over high performance. With the advent of multicore processors, heterogeneous and over-provisioned architectures, it is possible to employ multiple cores to run individual components of the system, avoid expensive context switching and streamline the system's operations. Thus, multiserver systems can overcome their performance issues without compromising reliability. However, while resources are becoming abundant, it is important to use them efficiently and to select and tune the resources carefully for the best performance and energy efficiency depending on the current workload. Most of the prior work focused solely on scheduling and placement of the applications. *In multiserver systems, the operating system itself needs to be scheduled in time and space as the demand changes. Therefore the system servers must no longer be opaque processes and the scheduler must understand the system's workload to make good decisions.*

1. Introduction

Multiserver systems composed of user space processes running on top of a microkernel and performing operating system functions can easily embrace heterogeneous multicore architectures. Moreover, they can run their components on the best available cores and can reconfigure themselves to deliver the best performance. Unfortunately the execution resources are limited, most of all by power constraints. Essentially all prior work focused on scheduling *applications* on such architectures, usually running on top of a monolithic system like Linux, Windows or a BSD. In contrast, a multiserver system does not schedule only applications, but also itself—as the servers are user space processes that need to share the CPU core(s) with applications and with each other. Systems like MINIX 3 [1] or QNX Neutrino [2] make little difference between scheduling applications and OS servers. For instance in the case of MINIX 3, the servers only have a higher priority and a shorter time quantum. However, as completing a single application request often involves multiple servers, finding the right order in which to interleave their execution to deliver optimal performance is difficult.

Unfortunately, the complexity of the problem is exacerbated by the trend toward ever more components. Multiservers systems keep splitting their servers into finer-grained components in order to isolate runtime failures and allow for simpler crash recovery and live updates[5]. In our previous work on NewtOS [7], an experimental high-performance fork of MINIX 3, we tackled the performance issues of multiserver systems by running the performance-critical servers of the network stack as individual components on dedicated cores. Similarly, the Loris [3] storage stack replaced the single file system server with many processes for enhanced reliability and modularity.

Dedicating cores to servers avoids scheduling on such cores. Each server has its own core and it can run any time it needs to. On the other hand, it limits the number of cores left for the applications. As some parts of the system will be more heavily used than others and not all parts of the system experience the same load in all situations, blindly dedicating cores to any system server is not optimal. In our previous work [6], we proposed to use emerging heterogeneous multicores as they could offer a higher number of cores on a die of the same area with a smaller energy budget, and reduce the amount of dark silicon as demonstrated in [10]. Meanwhile, the cores have different capabilities and performance characteristics and running the servers on the slower cores has the potential to deliver good (or even better) performance [6] than using the big fast cores for the OS components. Previous work [11, 12, 13] demonstrated that applications often deliver the best performance (or performance to energy ratio) when they do *not* use the fastest processors in the system. The same holds for the system servers—they may have a special role, but in the end, they are just user space processes. Even so, this leaves open the question of how to schedule what components under specific workloads.

Although the code of monolithic systems likely has completely different performance requirements than the applications [9], the system and the applications share the cores. In contrast, multiserver systems can pick the right choice independently for both the applications and the system servers. For example, they can easily spread components across multiple cores of different types. However, doing so complicates scheduling. Prior work on schedulers in monolithic systems is looking for the best distribution of applications among the CPUs. The system runs on all of the cores and uses the voltage and frequency settings selected for the applications. In contrast, we need to place the OS servers in the same way as the applications since they are also user space processes. However,

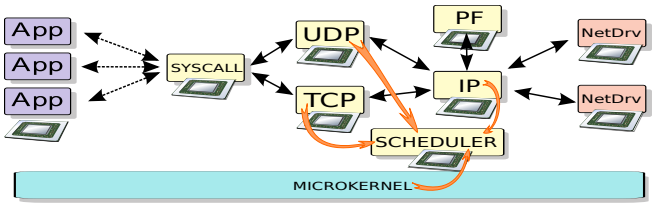

Figure 1: NewtOS - network stack and the scheduler's inputs

unlike the mostly independent applications, the servers cooperate heavily, so the scheduler must consider them together and find a combination of the cores and their settings that is optimal for groups of servers.

In this paper we discuss our experience with finding optimal configurations in NewtOS [7] operating system and discuss what information the servers must provide to allow designing new schedulers specifically for multiserver systems running on multicore processors.

2. The NewtOS Scheduler and Network Stack

NewtOS is a high performance fork of MINIX 3 that marries reliability and high performance. In particular, we use its network stack to prototype fast inter-process communication in extreme situations. We present architecture of the network stack in Figure 1. NewtOS runs performance-critical components on dedicated cores. It allows them to communicate asynchronously in user space without the overhead of using the microkernel. In fact, the kernel rarely runs on the dedicated cores as discussed in detail in [7].

The fact that the applications run on different cores than the system makes it possible to extended the user space communication from the stack to the applications. This allows us to avoid many system calls and memory copying. We expose the socket buffers to the applications and we let the application poll the buffers blocking only if there is no work or a transmission buffer is full. Since the cost of system calls (especially non-blocking and select-like calls) dramatically drops, it makes performance of NewtOS competitive with the state of the art network stack of Linux. For example, Figure 2 shows performance of lighttpd serving small files, 10 requests on a single persistent connection both for NewtOS and Linux. Although the NewtOS cannot compete with Linux where latency matters since many components are involved in processing the packets, when one of the servers fails, it is much easier to repair it while the system stays online. However, where throughput is more important than latency, NewtOS can even outperform Linux.

NewtOS inherited the user space scheduler of MINIX 3 which is inspired by L4-based systems. The scheduler is a process which is consulted by the in-kernel priority round-robin scheduler every time a process runs out of its quantum. The kernel sends a message to the scheduler updating it on basic statistics like the process' CPU usage, how often it was blocked, the cache hit rate, etc., while it removes the process from the run queue. Once the scheduler decides on the pro-

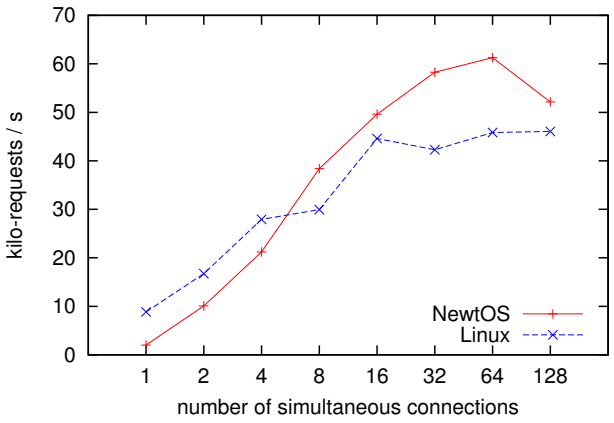

Figure 2: NewtOS vs Linux performance. Requesting 10 times a 10-byte file using a persistent connection

cesses' new scheduling parameters like quantum size, priority and the CPU where it runs during its next period, it tells the kernel by the means of a kernel call, an equivalent of a system call of monolithic systems, that the process can run again. In the mean time, the kernel lets other ready processes run. Besides immediate decisions, the scheduler also contiguously monitors the system. It gathers information from the kernel and other sources as depicted in Figure 1 and changes the scheduling parameters pro-actively when it finds that an adjustment is needed. For instance, it redistributes processes when a core is under- or overloaded. In the case of a dedicated core, the kernel lets the only process run and only periodically updates the scheduler on the runtime statistics.

As full cores are expensive in die space and energy, we also exploit hardware multithreading. For instance, the initial implementation of Intel's Hyperthreading [8] used only 5% of extra die area which is a relatively cheap way of over-provisioning. The threads serve as containers for the process' state, allowing it to use instructions like `MWAIT` to monitor memory writes which we require for our fast communication. The application thus halts its thread when it knows that there is no work and the hardware scheduler efficiently interleaves execution of all runnable processes on the core, together effectively avoiding the overhead of the system's scheduler.

All processes are opaque for the kernel, however, unlike in other similar systems, the user space scheduler of NewtOS knows that some processes are special—that a certain process is a system server, part of the network stack or a driver and if so, whether it drives a network card or a disk controller. It also knows that a process is part of the network or storage stack. The scheduler uses this knowledge to place each component of the network stack on its own core for the peak performance, but when the network traffic is low and the stack is mostly idle, it co-locates all of its processes on a single core to save resources. Depending on the architecture, it also scales the frequency of such a core down (e.g., on AMD Opterons). We know from experience that drivers often do not need powerful cores and can co-exit on different hardware threads of scaled

down Intel cores, thus providing some thermal headroom for the Intel Turbo Boost.

In [6], we evaluated our stack and showed that the system has potential to deliver the same performance when running on less powerful cores and with careful tuning may deliver even higher performance than on fast cores running at full throttle. However, the scheduler requires much richer information about the state of the system to make its decisions.

3. Network Traffic Indicators

Current schedulers derive the nature of the applications, for example whether they are interactive or long running jobs, based on their CPU usage and other runtime factors. Similarly, to schedule the components of the network stack optimally, the scheduler needs to know the nature of the current traffic and it needs to guess whether the current workload is request-response oriented, how many connections open and close or whether it is a long running bulk transfer. Different types of load stress the system differently and the scheduler needs to react accordingly. Although the programmer may be able to provide some additional performance related information, for instance in the case of server applications, the actual runtime requirements depend highly on external factors.

The scheduler of a multiserver system is isolated and its only way of gathering information is explicit communication with other servers. While the kernel provides basic profiling information like CPU load or cache misses, the rest must come directly from the system servers—in the case of the network stack in a form of traffic class indicators.

Collecting the information as well as sending the information to the scheduler must be cheap so it does not impact performance. We opt for a set of indicators similar to the CPU performance counters. Based on their readings the user space scheduler can make an educated guess about the workload. Each of the components exports periodically the values in a message to the scheduler. The indicators exported by individual components are summarized in Table 1.

TCP	RX / TX bytes and segments; new outbound connections; accepted and closed connections
UDP	RX / TX bytes; new sockets; closed sockets
IP	RX / TX sum of bytes, protocol including headers, ARP and ICMP protocols

Table 1: Network Traffic Indicators

The number of received (RX) and transmitted (TX) bytes for UDP and TCP includes only the size of the payload while IP reports a sum of all bytes including transport and link layer headers and protocols handled by IP only. Although the number of TX/RX bytes is a key indicator, it is by no means the only metric which determines the load of the stack. In the case of TCP, it is extremely important to know whether the bytes were sent in many small segments or in large ones as the per-byte overhead is inversely proportional to the segment size. Since the stack hardly touches the outgoing data, preparation of the headers, computation of their checksums (assuming that the network interface can checksum the data) routing of individual packets, splitting TCP data into the segments which fit in the allowed window and assigning data to individual sockets is what matters. In addition, small segments may indicate interactive or request-response traffic which is latency sensitive. Large segments are a sign for bulk transfers and elephant flows, which put much less stress on the system, especially when TCP offloading is used, the per-byte overhead drops and much slower, simpler and less power hungry cores can do the job. In contrast, request-response type of workloads take advantage of fast cores as it reduces the latency.

TCP receives and sends many segments which do not contain any payload and serve only as acknowledgment for the payload transfers. This overhead needs to be accounted for as the overhead is significant when sending or receiving only a few bytes, while the bitrate is in the order of a few tens or hundreds of megabits per second. Looking at the bitrate only may suggest that the load is low if we consider multigigabit network speeds. Large transfers of many gigabits per second usually do not stress the cores much. On the other hand, the stack copies received data to the right socket buffers which puts high load on memory. Therefore placement of the stack close to the application (in terms of memory communication) is critical.

Another indicator is the frequency and number of connections the system sets up. Setting up TCP connections requires an exchange of packets which do not carry any data. However, their processing requires resources which may be significant, especially in the case of short-lived connections. Handling such connections on high rate (for example in the test in Figure 2) overloads the application as well as the stack due to many management system calls which do not result in sending or receiving any payload. Hence the connection rate, rather than bitrate, is the indicator which tells the scheduler what is the root cause of the problem. Similarly, it is also important to know whether the connections are mostly initiated from this system (it is a client) or whether it is a server

It is the task of the user space scheduler to gather the data and generate richer statistical information. For example, the mean overhead for transmitted or received byte, the segments needed for setting up connections or the mean life-time of connections. The frequency of updating the scheduler decides how quickly and how precisely the system is able to respond, however, high rate of updates introduces communication overhead and the updates which come from different sources may not be synchronized.

4. Web Server Profiles

We have conducted several measurements of the *lighttpd* web server running on NewtOS with the focus on stressing the net-

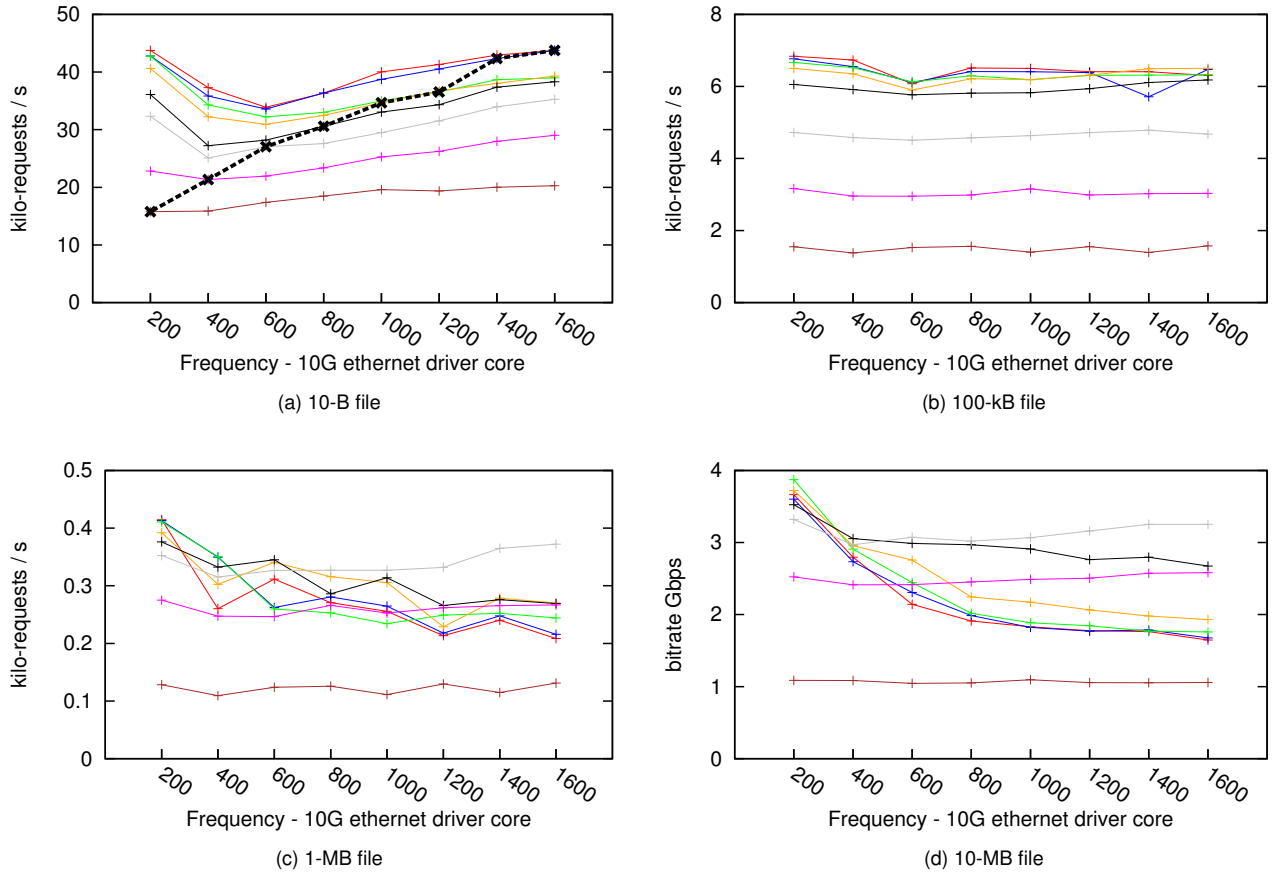

Figure 3: Requesting a file of a size between 10-B and 10-MB using 8 simultaneous connections. Fixed frequency of TCP and IP cores.

work stack by various workloads to demonstrate that selecting the right configuration is not a trivial task. Web servers are a common case which demands high performance and spans across a large range of workloads. We use a dual-socket quad-core Xeon E5520 machine. One of the chips hosts the system while the other one is left for the applications. We use a single lighttpd process. We set the speed of the system chip to 160-MHz and we let the other chip run at the peak 2267 MHz. We use thermal throttling of the individual cores to emulate high and low performing heterogeneous cores for the system with frequency ranging from 1600 MHz to 200 MHz. Our goal is to explore the range in which various cores may become a bottleneck. For simplicity, we excluded the packet filter (and UDP is obviously not used). This leaves TCP, IP and a 10G Ethernet driver.

Figure 3 presents a subset of measurements in which we scale the IP and TCP cores together (individual lines—legend in Figure 4b) and the driver core separately (the X axis) as our experiments suggest that the speed of the driver influences the performance the most. TCP and IP have similar CPU usage. The second set of figures in Figure 4 presents the same data. However, we now fix the frequency of the driver's core instead. We carried out all the measurements requesting a file of various sizes using 8 simultaneous connections issuing 10 requests per a single connection. This is enough to overload the stack cores when they run at low clock speed.

Some of the workload patterns show the scaling we would expect. For instance requesting a 100-kB file scales almost linearly depending on the performance of both IP and TCP cores. On the other hand, the speed of the driver's core has no impact. However, when requesting a tiny 10-B file, the speed of the driver's core matters. The lowest speed of 200 MHz delivers the same performance as if the core were running at 1600 MHz. Comparing the results to the same test when all the cores run at the peak clock frequency of 2267 MHz (Figure 2, data point 8), it shows the request rate may even drop again when we use even faster cores. The potential energy savings make the choice obvious, however, we would also need to have a good understanding of the energy consumption profiles of individual cores' types and their dynamic voltage and frequency scaling (DVFS) states. Part of this information is provided by a firmware like ACPI, however, it is important to have accurate online energy measurements provided by the cores themselves rather than letting the scheduler estimate the energy consumption indirectly, for instance by using the existing performance counters [4].

Counterintuitively, when the response size grows into the order of mega bytes, running TCP and IP on fast cores may

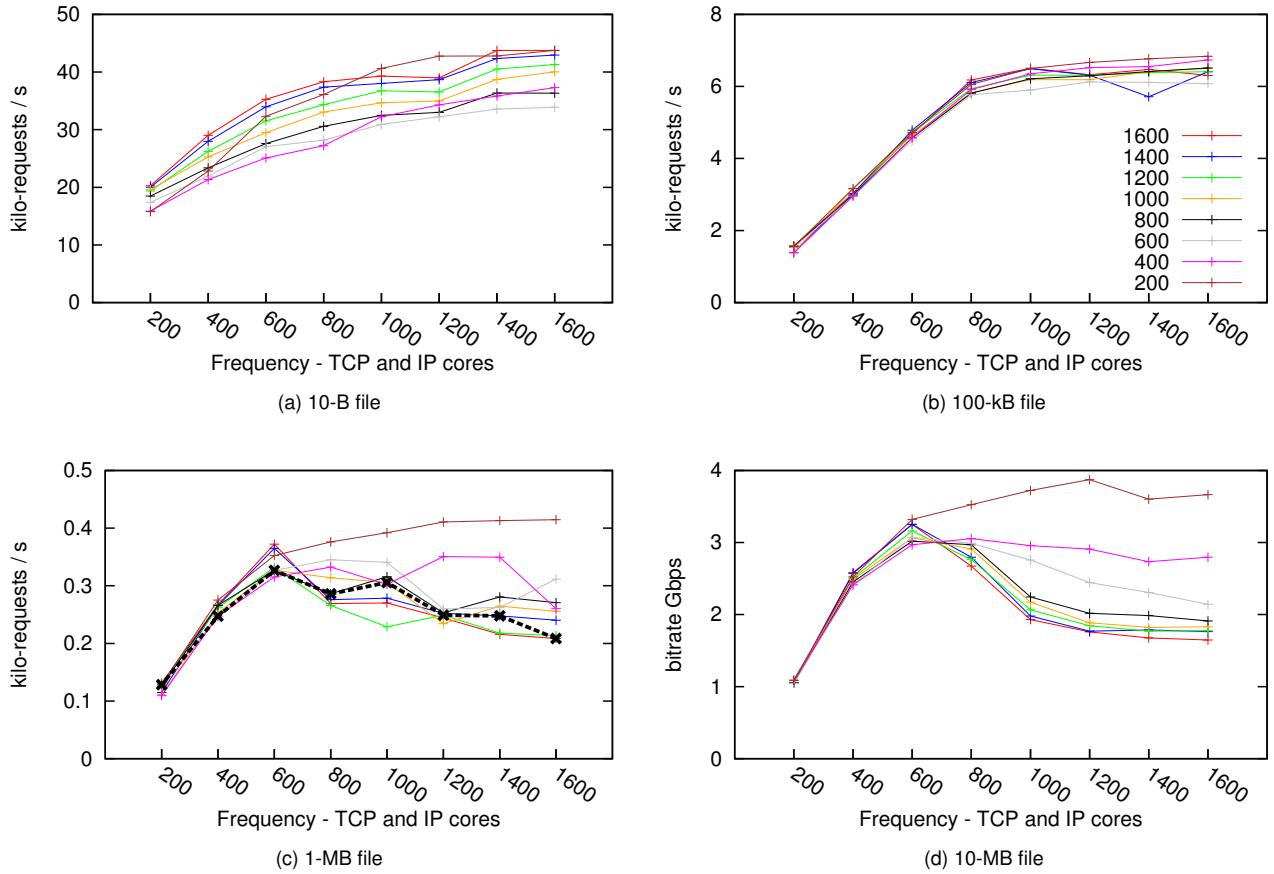

Figure 4: Requesting a file of a size between 10-B and 10-MB using 8 simultaneous connections. Fixed frequency of the driver's core.

have negative impact on the performance. This depends on the frequency of the driver's core, that is, whether it is below or above 400 MHz. The difference is the amount of sleep time of the underutilized fast cores and how much latency adds waking them up. In this case, saving energy on a fast core significantly impacts performance and a more utilized slower core can deliver the best of both measures.

An important observation is that even in situations which scale monotonously (10-B and 100-kB case), increasing the speed of the cores beyond a certain point does not deliver any increase in request or bit rate. In such a situation, the application core is usually (close to being) overloaded. It is pointless to scale the stack's cores up or move it to faster cores if the application is the bottleneck. In contrast, the scheduler needs to find such cores and settings for the stack that it frees up resources for the applications. For instance, by moving the stack to a different chip to let the application use Turbo Boost on the Intel processors or letting to power on faster cores.

5. Profile Based Scheduling

Since the profiling shows that the system's behavior is not linear, finding an optimal solution is a hard problem. Many schedulers use a trivial algorithm which would scale all the cores of the network stack simultaneously as long as the bitrate increases. Doing so would find the optimal setting for the performance in the case of 10-byte files. However, it would be far from the energy optimum (dashed line in Figure 3a). In addition, the same algorithm would find neither the performance nor the energy optimum in the case of 1 or 10-megabyte files (Figure 4c). Moreover, many schedulers scale up or down only when a certain CPU usage threshold is reached. It is tricky to pick such thresholds which would suite any deployment and workload.

A solution which would find the optimum dynamically is further complicated by the fact that after selecting a different configuration the performance is worse or better either due to the new configuration or because the load of the system has changed due to external events. Unlike in the case of local storage (for example), in the case of networking, the remote machines as well as the network itself react to the changed timings. Therefore we need the richer indicators from which we can conclude that, say, the mix of requested files did not change, but only the service rate is higher or lower. In our model we assume that the workload does not change abruptly and that although the load of the system fluctuates, a recent history of the readings of the traffic indicators allow us to observe a trend. If the scheduler's decision significantly

worsens the trend, it is likely a wrong decision.

The solution we propose at this moment is a *Profile Based Scheduling* which assumes that we can estimate the expected workloads, benchmark them and create their profiles. Based on the current indicators' readings, the scheduler decides which of the profiles is the closest match and applies the best settings for the given profile.

For example, in the case of a web server, the workload can temporarily change either in the number of connections or requests per second or in the mix of requests. Therefore, to approximate the optimal behavior requires many profiles along several dimensions. The profiles cannot capture all possible scenarios and we select a configuration based on an approximation of the observed load and the profiled ones. Therefore each of the load and configuration pairs can serve as a new data point in an online profile which makes approximation finer and further increases precision of our future selections.

A large number of profiles is not always required. A node in high-performance computing may just switch between handful of its roles. For example, when the node receives data for its computation, when it sends out the results and when it computes. In the last case, it can power off the networking cores as it needs the energy elsewhere. Alternatively, if the cores employ hardware threading, we can co-locate the network stack on the same cores as other parts of the system which are mutually exclusive in different periods of the operation of such a node. Since we would not need to power off a set of cores to power on another set, this decreases the latency of switching between different modes of operation. Although the load of the network stack's cores can hint whether the node uses the network or not, the extra indicators can detect whether it sends or receives data and allow the scheduler to act accordingly. For example, sending data is typically less demanding, therefore we can pick a profile which scales the network stack's cores down, while in the case of receiving the profile suggest to use faster cores.

The need for the profiles should not prohibit deployment of multiserver systems as it is already a good practice to carefully evaluate and tune current commodity systems and applications before production deployment. Generating the required profiles is easily scriptable and can be part of the initial evaluation process. The added value of our approach is that the hardware and the system is not only tuned for the expected peak load, but the system can adapt as the workload changes and pick the most appropriate resources. Heterogeneous and over-provisioned architectures are newly emerging products with different characteristics and we can apply profiling prior deployment equally to any of them.

6. Conclusions and Future Work

We presented an operating system which is designed for high reliability and dependability. Running on multicore processors allows the system to reduce the overheads of its design and even surpass performance of the state of the art systems. This system is able to embrace a heterogeneous over-provisioned machine and reconfigure itself based on the changes in the workload. To further improve its performance it can use more resources and the most appropriate resources and free the resources when they are not needed. We use current commodity hardware to emulate heterogeneous environment and we show that trivial solutions cannot reach the optimal performance to energy ratio. We show the potential of the system and we present a basic method which allows the system to adapt to changing scenarios. For now, evaluation of more diverse architectures and a truly comprehensive algorithm that adapts the system to any situation remains work in progress.

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 and Teodor Crivat for his work on extending the high performance asynchronous user space communication to the applications.

References

[1] MINIX3. http://www.minix3.org, 2006.

[2] QNX Neutrino RTOS System Architecture. http://support7.qnx.com/download/download/14695/sys_arch.pdf, 2006.

[3] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Loris - A Dependable, Modular File-Based Storage Stack. In *PRDC*. IEEE Computer Society, 2010.

[4] Gilberto Contreras and Margaret Martonosi. Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, 2005.

[5] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[6] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When Slower is Faster: On Heterogeneous Multicores for Reliable Systems. In *Proceedings of USENIX ATC*, 2013.

[7] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. Keep Net Working - On a Dependable and Fast Networking Stack. In *Proceedings of Dependable Systems and Networks*, 2012.

[8] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[9] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*.

[10] Sourav Roy, Xiaomin Lu, Edmund Gieske, Peng Yang, and Jim Holt. Asymmetric Scaling on Network Packet Processors in the Dark Silicon Era. In *Proc. of the Symp. on Arch. for Net. and Comm. Sys.*, 2013.

[11] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green Governors: A Framework for Continuously Adaptive DVFS. In *Proceedings of the International Green Computing Conference and Workshops*, 2011.

[12] Xiao Zhang, Kai Shen, Sandhya Dwarkadas, and Rongrong Zhong. An Evaluation of Per-chip Nonuniform Frequency Scaling on Multicores. In *Proc. of the USENIX Annual Technical Conf.*, 2010.

[13] Xinghui Zhao and Nadeem Jamali. Fine-Grained Per-Core Frequency Scheduling for Power Efficient-Multicore Execution. In *IGCC*, 2011.