Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor

Alexandra Fedorova, Viren Kumar, Vahid Kazempour,
Suprio Ray and Pouya Alagheband

School of Computing Science
Simon Fraser University
Vancouver, Canada

ABSTRACT

In this position paper, we present our vision for the scheduling infrastructure in a many-core hypervisor – the hypervisor targeted for many-core platforms. The key objectives of our system are *scalability* and *heterogeneity-awareness.* We see these as first-order objectives, because future many-core processors will consist of thousands of cores and those cores will be heterogeneous. Since existing hypervisors were not designed to handle the scale and heterogeneity of many-core hardware, our design will differ from that of existing hypervisors in many important ways. The design of our experimental many-core hypervisor, Cypress, is based on three principles: *partitioning, localization,* and *customization*. Together, these principles facilitate scalability, by minimizing the sharing of scheduling runqueues, and manage heterogeneity, by assigning to each VM the cores most suitable for its workload. In this paper we motivate our design, present its key components, discuss challenges in our research, and report on its status.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design, Process Management

General Terms

Algorithms, Design, Experimentation, Management, Measurement.

Keywords

Many-core processors, scheduling algorithms, hypervisor, virtual machine monitor, heterogeneous multi-core systems, scalability, guest operating system.

1. INTRODUCTION

Future many-core processors will consist of hundreds or thousands of heterogeneous processing cores [7,11,25]. Schedulers in existing hypervisors were not designed to scale at this level, and they are not heterogeneity aware. The focus of our work is to design a scalable and heterogeneity-aware scheduling infrastructure in a virtual machine hypervisor. In particular we expect to address two research problems:

(1) *Evaluate scalability of schedulers in existing hypervisors* on processors with hundreds or thousands of cores, and (if necessary) investigate how to design a scheduler that scales.

(2) *Design hypervisor scheduling algorithms that allocate heterogeneous cores to VMs in the optimal manner*. We expect that this will be done in one of two ways: (1) via explicit communication with heterogeneity-aware guest operating systems or (2) via hints from legacy (heterogeneity-unaware) guests. In the former case, we will have to design the interface by which the guest will learn about the kinds of CPU cores available on the system and by which it will communicate to the hypervisor its resource requirements. We expect our contribution to be new scheduling algorithms that make optimal decisions in terms of performance, energy consumption and fairness, and that are, at the same time, light-weight and scalable.

Addressing scheduling in the hypervisor is an important problem, because dependence on hypervisor technology will increase as many-core hardware matures. Since hardware scales at a faster rate than software [7], utilizing a many-core system by a single workload will be challenging. Therefore, those systems will often run multiple independent workloads. Those diverse workloads will run on various guest operating systems and will have to be isolated from one another for faults and performance effects. Since hypervisor technology addresses both of these concerns[1] [9,26], its use will be widespread in many-core environments.

Since our design will introduce new guest/hypervisor interfaces, some of our work concerns with implementing a heterogeneity-aware guest OS that works synergistically with the Cypress hypervisor. At the same time, our hypervisor will support legacy (heterogeneity-unaware) guests.

The rest of the paper is organized as follows. In Section 2 we motivate our work by discussing the challenges posed by many-core hardware. In Section 3 we present the design of Cypress. In Section 4 we discuss the status of our project. In Section 5 we

This research is supported by NSERC Strategic Project Grant No. STPSC/356815-2007 and by Sun Microsystems.

[1] While microkernels and container operating systems (such as Linux VServer, Solaris Zones, and IBM AIX 6.1) also provide isolation, they offer less flexibility as far as diversity of operating systems.

compare and contrast our approach with related work, and in Section 6 we summarize.

2. MANY-CORE CHALLENGES

We begin with identifying our assumptions about future many-core processors and about the software that will be running on them. We then discuss the challenges that scale and heterogeneity of many-core processors pose for system software.

2.1 System model

We envision that future many-core systems will consists of three kinds of CPU cores: (1) low-power simple in-order cores, i.e., *slow* cores, (2) powerful complex, i.e., *fast,* cores, and, finally, (3) *accelerator* cores, such as vector processors (as in IBM Cell engine) or general-purpose GPUs. Slow cores will be abundant. Fast cores will be less plentiful. They will expose the same ISA as slow cores but offer better performance for high-ILP programs. Finally, accelerator cores will expose distinct ISA and there will be only a handful of them on the chip. It is projected that the total number of cores per chip will be on the order of hundreds or thousands, as early as a decade from the time of this writing [7,11].

We expect that on many-core processors the prevalent execution environment will be based on a hypervisor. The workload consolidation enabled by the hypervisor will allow keeping the many-core machine busy. Despite ongoing parallelization of many applications, there will still be algorithms that are fundamentally serial, so workload consolidation will be an important means of achieving full utilization of many-core hardware. Aggressive workload consolidation will create very diverse VM workloads running on the same processor, e.g., servers, desktop applications, scientific applications and soft real-time workloads, and thus we expect to see diverse guest operating systems. There will be operating systems that are customized for heterogeneous hardware. Those guests will be able to give the hypervisor hints with respect to the kind of cores that are needed by their workloads. There will also be legacy operating systems that will provide no such information. The hypervisor must be designed to provide efficient allocation of resources in both of these scenarios.

2.2 Scale

The first goal of our research is to *evaluate scalability of existing hypervisors on many-core processors*. While hypervisor scalability was evaluated on small-scale multiprocessor systems [2,3,22], scalability on systems with more than four cores has not been measured. If we find that scalability has limitations, we will investigate *how the hypervisor must be designed so that it scales.* While scalability of operating systems is a well-studied area [6,12,16], it is not clear whether hypervisor will be subject to same issues. Furthermore, existing research on OS scalability addressed this problem in the context of multi-processor hardware. Understanding the nature of scalability limitations on *many-core hardware* is still an open question.

Nevertheless, existing research on OS scalability allows us to speculate about potential scalability bottlenecks and project how the hypervisor might be designed to avoid them. Earlier research showed that scalability of system software is usually limited by (1) synchronized access to shared data structures [16,17] and (2) reduced cache affinity due to interrupt processing [2,3]. We now elaborate on these problems.

2.2.1 Shared data structures

When several cores simultaneously read and write shared data they must synchronize their access and transfer modified data from one core to another. This limits concurrency and increases access latency.

In schedulers, most frequently shared data structures are per-CPU (or per-core) runqueues. Most commercial schedulers (e.g., in Linux, Solaris, Xen) access shared runqueues during context switches and during load balancing [5,17,21]. A recent study of the Linux kernel [17] revealed that the majority of scalability bottlenecks came from accessing those shared runqueues, with roughly 22% of highest-latency cache misses coming from that code. Although the Linux study was done on a *multi-chip* multicore system and thus observed higher inter-core communication latencies than those that would be seen on a *single-chip* multicore processor, it showed that sharing of runqueues can be a barrier to scalability[2]. While the impact of runqueue sharing in many-core systems is yet to be understood, in our design we minimize access to shared runqueues. As Section 3.1.1 explains, we partition the cores among VMs rather than time-share them, when possible.

2.2.2 Interrupts

Many scalability problems in existing hypervisors and operating systems stem from the fact that interrupts are delivered to a different processor than that which last ran the thread expecting the interrupt [2,17]. As a result, the thread is awaken on a different processor and loses its cache affinity. We are interested in experimenting with the hardware that supports targeted interrupt delivery, i.e., *message signalled interrupts,* and in evaluating its implications for the scalability of many-core hypervisors.

2.3 Heterogeneity

Heterogeneous architectures are expected to be in wide use in future many-core processors [7,11]. The reason is that heterogeneous architectures can potentially achieve a higher performance per watt than homogeneous architectures [19,20]. To realize this potential, however, applications must be scheduled to run on the cores that best fit to their architectural properties [10,20]. Existing hypervisors are agnostic of heterogeneous hardware and thus they do not reap the energy-saving potential of heterogeneous processors. Our research will address the *design of heterogeneity-aware scheduling algorithms in a hypervisor* that will allow realizing this potential. This will be the main contribution of our work.

We now provide an example demonstrating why heterogeneity-aware scheduling is crucial. Consider, a single-ISA heterogeneous system. On this system, a scientific application with high instruction-level parallelism (ILP) would run much faster on a "fast" core, as opposed to a "slow" core. On the other hand, a memory-bound database application would experience comparable performance on slow and fast cores. The best performance/energy trade off is achieved by assigning the

[2] We are not aware of any recent studies examining scalability of schedulers in other operating systems and in VM hypervisors. Existing scalability of Xen and VMWare focused mostly on scalability of I/O and were done on hardware with at most four cores

scientific application to the fast core and the database application to the slow core. As another example, an application with high thread-level parallelism (TLP) could perform well on many slow cores, while a low-TLP application that cannot take advantage of hardware parallelism must be run on a few fast cores.

These examples demonstrate that the hypervisor (and the guest OS) must be *heterogeneity aware*. That is, the hypervisor must *expose* heterogeneous properties of the hardware to the guest, and the guest must schedule tasks to cores in the most efficient manner. While heterogeneity-awareness in the guest was partially addressed in the research community [10,15,19,20], OS-level solutions are not directly applicable to the hypervisor and design heterogeneity-aware algorithms for the hypervisor remains an open question.

In addition to building a heterogeneity-aware hypervisor we are also exploring heterogeneous aware designs of the guest OS. Although several heterogeneity-aware scheduling algorithms for OS were proposed in the past [10,19,20], they were not meant to scale to many cores and assumed long-lived threads. In particular, the two most prominent algorithms by Becchi [10] and Kumar [20] relied on continuous monitoring of performance of each thread (or thread co-schedules) on each heterogeneous core to determine the best core for each thread. When the number of different core types is large (as may be the case on many-core systems), this task becomes infeasible. Furthermore, these algorithms (while not implemented in a real OS) implied frequent examination of scheduler's runqueues, which, as we know [17], may limit scalability. Finally, these algorithms assumed that threads were long-lived, since short-lived threads may terminate before the scheduler learns the optimal core placement for them. Our goal is to experiment with new scheduling algorithms that address these shortcomings.

Studies of heterogeneous multicore systems showed that they achieve superior performance/watt in comparison to homogeneous systems, but only when CPU cores are assigned to match the properties of applications [10,19,20]. This means that system software *must* be heterogeneity-aware in order to leverage the power-efficiency of those systems. This also means that if heterogeneity-aware system software is not designed before heterogeneous hardware enters the mainstream, hardware designers will have little incentive to build heterogeneous systems, a so-called *chicken-and-egg* problem. By developing heterogeneity-aware system software early on, before heterogeneous processor designs are finalized, we have a unique opportunity to influence future hardware trends by facilitating the adoption of heterogeneous processors, and to enable systems that are both energy efficient and fast.

3. OUR DESIGN

In this section, we present the design of Cypress (Section 3.1), discuss our research on heterogeneity-aware operating systems (Section 3.2), and present some experimental results showing the benefits of heterogeneity-aware system software.

Cypress is novel in two respects:

(1) It will be the first hypervisor designed for heterogeneous hardware
(2) Its will be designed, from the start, to scale to hundreds/thousands of cores.

While implementation of this vision will, in some cases, require application of existing techniques in the new domain (e.g., ideas from research on scalable OS), it will also necessitate completely new design ideas and new algorithms, in particular related to implementing support for heterogeneous cores.

3.1 Design of Cypress

Our experimental design is based on three principles: *partitioning, localization* and *customization (PLC)*. Partitioning and localization address scalability (although partitioning must be heterogeneity-aware). Customization addresses heterogeneity.

3.1.1 Partitioning

Partitioning is a decentralized framework for scheduling of VMs that minimizes the overhead of runqueue management in the hypervisor. With partitioning, cores are partitioned among VMs (VMs are pinned to subsets of cores), and so cores are not time-shared among VMs. This reduces the overhead of runqueue management in the hypervisor. Although the idea of partitioning is not new (as far as we know it is often used in existing hypervisors to avoid the overhead), in existing systems partitioning does not take into account the fact that cores are heterogeneous. In Cypress, partitioning is heterogeneity-aware. We now describe how it works.

As we mentioned, on many-core systems small, simple and low-power cores will be abundant. Because of abundance, there will be no need to time-share those cores among VMs, so the cores can be simply partitioned among them. Therefore, Cypress maps a group of cores to each virtual machine and lets the guest OS assign those cores to applications, performing no load-balancing or queue management itself.

Fast cores will be less numerous, and so it may be necessary to time-share them among VMs. In Cypress, fast cores are assigned to a special *anonymous* partition. They are given to VMs on short-term leases. Short-term leases are similar to time-sharing, but the scheduling quantum is longer (on the order of seconds), so there should be less overhead due to runqueue management. When a VM's lease for a fast core expires, the virtual CPU that was mapped to the fast core is re-mapped to a slow core. So while some of the VM's applications run slower in-between leases, they still make progress.

Accelerator cores cannot be substituted with other (abundant) cores, because they expose a distinct ISA. In Cypress, those cores are time-shared among VMs in a traditional fashion. Since the number of these cores will be small, the overhead of runqueue maintenance should be low.

Partitions will be created, modified and destroyed by a *Global Partition Manager (GPM)*. Since partition changes will be rare, the GPM should not be a bottleneck.

Partitioning should reduce the sharing of runqueues and thus promote scalability. In addition, partitioning will give the guest a better control over CPU resources, since most of the virtual CPUs will be mapped directly to physical CPUs. As a result, partitioning may offer better support for real-time guest domains. (In this case, it may be necessary to notify the guest if the core being is mapped to a dedicated physical core or if it is being time-shared.)

While partitioning in itself is already used in existing VM installations to reduce hypervisor overhead, it is configured manually by the system administrator. In Cypress, partitioning

will be built into the hypervisor itself and, most importantly, it will support heterogeneous CPU cores.

3.1.2 Localization

Each VM partition will be managed by a *Local Partition Manager (LPM)*. LPM will perform virtualization-related housekeeping tasks for its VM, such as page table updates and processing of interrupts – those housekeeping tasks are usually done globally for all VMs in existing hypervisors. LPM will also notify the GPM if the partition size must be changed due to a change in the CPU utilization inside the VM.

LPM will run on a dedicated core (or cores) within the partition. This will prevent CPU bottlenecks related to interrupt processing reported in previous work [2,3]. LPM will access only the memory objects belonging to the VM running in that partition, hence we have localization. Localization will reduce memory latency and avoid synchronization bottlenecks. Localization is inspired by Tornado and K42 operating systems [6,16].

3.1.3 Customization

Customization is about making sure that each VM schedules its applications on the "right" kinds of heterogeneous cores, so as to maximize the system's performance per watt. This, in contrast to partitioning and localization, is an entirely new concept in the realm of hypervisor design.

There are three things that the hypervisor must do to offer good support for processors with heterogeneous cores: (1) it must export the cores' features to heterogeneity-aware guests, (2) it must optimally allocate heterogeneous cores to heterogeneity-unaware (legacy) guests, and (3) it must properly account for utilization of the cores of different types.

A heterogeneity-aware guest (as in [10,20]) must be able to discover the features of heterogeneous cores present on the physical machine, so it can map its heterogeneous virtual cores to the heterogeneous physical cores and then assign threads to run on the "right" cores. To enable this, Cypress will provide support for dynamic hardware discovery. In existing systems, an operating system may discover features of the processor by reading its model-specific registers (MSR). Cypress will provide support for reading MSR registers of heterogeneous processors. The key challenge is to provide the guest enough information about the features of the hardware without overwhelming it with microarchitectural details.

If the guest is *not* heterogeneity aware, Cypress will take hints from the guest in order to map the guest's virtual cores to the machine's physical heterogeneous cores in the optimal way. Those hints may come, for example, from power management policies used in existing (heterogeneity-unaware) operating systems [14,23,24]. According to these policies, the OS brings cores into lower power states (via DVFS) when applications running on them do not benefit from high clock frequencies. In Cypress, when the legacy guest lowers the power state of the core, the hypervisor maps that virtual core to a slow physical core, and vice versa.

Some mechanism must be used in the hypervisor to encourage the use of power-efficient scheduling policies in the guest. One solution is to "price" the processing time on each core in accordance with the core's power consumption. As a result, a VM that uses power-intensive cores will be charged more per unit of processing time than a VM that uses low-power cores.

Another solution is to give each VM a certain power budget. Initially, each VM would be allocated a certain number of "cheap" low-power cores that fit within its budget. (The budget is decided by a system administrator.) Then, as the guest OS learns about the properties of its workload, it may "exchange" low-power cores for higher-power cores. Exchange can be driven by DVFS hints from heterogeneity-unaware guests or by explicit communication with heterogeneity-aware guests. The hypervisor ensures fair sharing of cores. For example, a core drawing 80 watts can be exchanged for two cores drawing 40 watts each, etc. As a result, a VM with a high thread-level parallelism (TLP) will probably get many low-power cores, while a VM with a low TLP will get few high-power cores. (This method of trading the number of cores and their complexity in known to maximize performance per watt [19,20].)

3.2 Heterogeneity awareness in the guest

In this section we outline our research on heterogeneity-aware scheduling algorithms in the guest OS. As we explained above, existing heterogeneity-aware OS schedulers assume long-lived threads and are not meant to scale to hundreds of cores. The reason for these drawbacks is reliance on dynamic performance monitoring to determine the suitability of each heterogeneous core for each thread [10,20]. The goal of our work is to eliminate reliance on dynamic monitoring from the scheduling algorithms.

Our main idea is that the application itself should provide information that helps the scheduler determine the best suitable core for it. This information is encoded in an *architectural signature*, a set of microarchitecture-independent characteristics of the application [18]. These characteristics may describe the memory access patterns of the application, its available ILP, the properties of its instruction mix, etc., thus helping predict its sensitivity to the variation in certain processor features, such as cache size, issue width, instruction scheduling architecture, clock frequency, etc. An architectural signature is generated offline, via profiling or binary analysis, and embedded in the application binary.

Use of architectural signatures eliminates the need to perform online profiling in the scheduler and facilitates efficiency and scalability in the scheduling algorithm.

Our early results indicate that it is possible to construct architectural signatures that help effectively determine allocation of CPU cores that differ in terms of clock frequency. We are still evaluating whether signatures can be constructed for other kinds of heterogeneous cores.

3.3 Benefits of heterogeneity awareness

We now present experiments demonstrating the benefits of heterogeneity-awareness in the hypervisor and in the guest. Since our hypervisor implementation is not complete, we illustrate its future potential via an emulated environment.

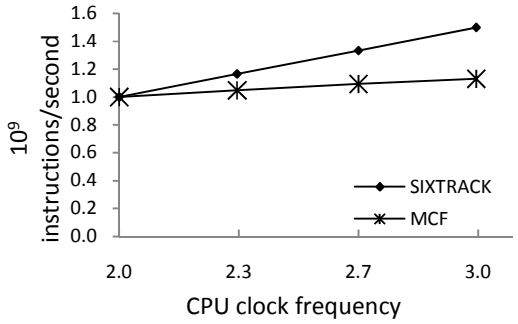

Figure 1. Performance of *sixtrack* and *mcf* with varying clock frequencies. *Sixtrack* is CPU bound (0.002 misses per 1000 cycles), and *mcf* is memory bound (4.62 misses per 1000 cycles). This explains the differences in performance sensitivities.

We use UNIX tasksets to represent a virtual machine, a user-level *taskset manager* to emulate a guest OS, and a user-level *CPU-affinity manager* to emulate the hypervisor. The taskset manager launches the applications belonging to its taskset and tells the CPU-affinity manager what types of cores it needs for its applications. (We assume that optimal core assignments are known to the taskset manager *a priori.*) The CPU-affinity manager then assigns cores to tasksets (emulating the partitioning done in the hypervisor), and the taskset manager assigns tasks to cores (emulating the scheduling done in the guest).

We run experiments on an Intel system with two X5365 quad-core processors. There are a total of four chips, and there is a pair of cores on each chip. Each pair of cores shares the L2 cache and each core has a private L1 cache. We use only one core per chip in our experiments to eliminate L2 cache interference and the associated performance effects.

We configure the system to be heterogeneous via dynamic voltage/frequency scaling (DVFS). DVFS allows us to selectively set frequencies on different cores, creating cores with different performance. We run one core at 3GHz, one core at 2.7GHz, one core at 2.3GHz, and one core at 2GHz. The operating system used in our experiments is OpenSolaris 11.

We picked two benchmarks from the SPEC CPU2000 suite [1], *sixtrack* and *mcf*. *Sixtrack* is a CPU-bound benchmark, so it achieves a significantly better performance when it runs on a high-frequency core vs. a low-frequency core (see Figure 1). *Mcf* is a memory-bound benchmark, so its performance is virtually insensitive to the changes in processor frequency. Therefore, if *sixtrack* and *mcf* are running together on our heterogeneous system, *sixtrack* should be assigned to higher-frequency cores, and *mcf* to lower-frequency cores.

We experiment with the following tasksets:

Taskset 1: *sixtrack, sixtrack*
Taskset 2: *mcf, mcf,*

and the following scheduler configurations:

Default: All tasks are assigned to cores randomly. This is equivalent to having a heterogeneity-unaware hypervisor and a heterogeneity unaware guest.

Basic: We emulate a heterogeneity-aware hypervisor, but a heterogeneity-unaware guest. The hypervisor ensures fair sharing of the cores according to their power consumption. (Power consumption is not measured directly but rather estimated based on the core's frequency). That is, the hypervisor will give to one taskset the cores running at 3GHz and 2GHz, and to the other taskset the cores running at 2.7 GHz and 2.3 GHz. The hypervisor cannot ensure optimal allocation of cores to tasks, because the guest is heterogeneity-unaware.

Het-aware: Each taskset gets the cores best suited for its workload. That is, Taskset 1 gets the cores running at 3GHz and 2.7GHz, and Taskset 2 gets the cores running at 2.3GHz and 2GHz. This is equivalent to having *both* a heterogeneity aware hypervisor and a heterogeneity-aware guest.

The experiment consists running the two tasksets simultaneously with each scheduler. We run the experiment with each scheduler five times and report the following data for each taskset: (1) average *energy per instruction* (EPI) normalized to the highest EPI observed during all experiments, and (2) average *instructions per second* (IPS) normalized to the highest IPS observed during all experiments. We expect to see better performance and lower energy consumption when both the hypervisor and guest are heterogeneity-aware (Het-Aware scenario).

Figures 2 and 3 show the results. The first thing to note is the overall reduction in energy consumption in Het-Aware scenario. Despite the increase in *sixtrack*'s EPI (by 39%), the dramatic reduction in *mcf*'s EPI (50%) results in overall energy savings.

The second thing to note is performance. In Het-Aware scenario, the IPS for *sixtrack* rises by 20%, as compared with Default and Basic scenarios, while the IPS loss for *mcf* is modest (only 5%).

This experiment demonstrates that heterogeneity awareness is essential for achieving optimal performance/energy trade-off on heterogeneous many-core systems. Further, it demonstrates the importance of heterogeneity-awareness in *both* guest and hypervisor and of cooperation between them.

4. STATUS

We are in the process of implementing and evaluating Cypress using Xen hypervisor [9] as base, as well as refining and improving its design. One challenge in our research is to find a suitable evaluation platform. Since real many-core processors are not yet available, we have to use existing multiprocessor/multi-core systems, software simulators, or FPGA-accelerated simulators. Unfortunately none of these alternatives suits our needs perfectly. Large-scale multiprocessor systems are expensive and do not accurately reflect the architectural features of future many-core processors. Software simulators, while extremely flexible, are extremely slow. FPGA-accelerated simulators, such as RAMP [4], offer the best potential, but none of the existing RAMP systems simulating 100s or 1000s of cores supports running a single-image OS or a hypervisor. We are currently using multiprocessor systems and software simulators for evaluation and are on the look-out for any new developments in the RAMP community.

5. RELATED WORK

VT-ASOS [8], a hypervisor for application-specific system customization, bears many similarities to Cypress. VT-ASOS is, essentially, an enhanced version of Xen that gives guest domains

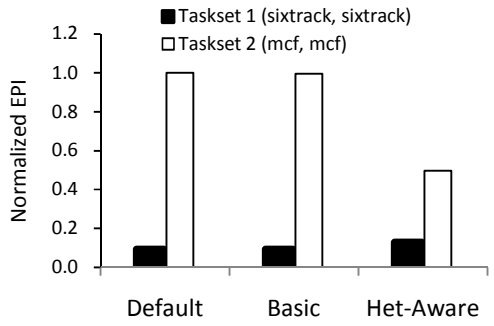

Figure 2. Normalized EPI with the three schedulers.

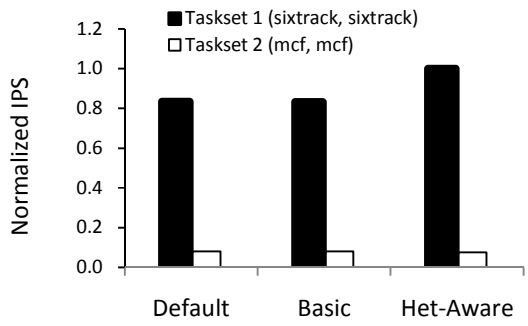

Figure 3. Normalized IPS with the three schedulers.

Note (in Figure 2) that even though Basic scheduler assures fair power distribution, each *mcf*'s instruction takes much longer than *sixtrack*'s instruction, so the energy (kilowatt hour) per instruction for *mcf* is much higher.

a higher control over CPU resources. For example, in VT-ASOS guest domains can explicitly tell the hypervisor how many virtual CPUs they need and whether these virtual CPUs can co-exist with other CPUs on the same core, socket, or node. While our hypervisor also provides enhanced control over CPU resources (via customization), it differs in several ways.

The most important difference of our hypervisor is heterogeneity awareness. While VT-ASOS provides no support for heterogeneous hardware, Cypress will provide proper virtualization of heterogeneous hardware. Cypress will also provide heterogeneity-aware partitioning (even though VT-ASOS also uses partitioning, no account is taken of heterogeneity) and enforce fair sharing of cores based on their power consumption.

Another difference of our hypervisor is the way in which guest domains communicate their preferences to the hypervisor. While VT-ASOS explicitly targets "intelligent" guests, i.e., guests that can specify *a priori* how many virtual CPUs they need, our framework allows a greater flexibility by letting guests exchange cores as they learn about properties of their workload. Furthermore, our framework accommodates legacy guests by using their power configuration as hints for optimal allocation of CPU cores.

Finally, our hypervisor design specifically targets scalability. Therefore, our philosophy is to minimize the amount of work done in the hypervisor and to allow only the minimal amount of (performance-critical) interactions between guest and hypervisor. VT-ASOS pursues a different philosophy: it allows rather high-volume interactions between guest and hypervisor, even introducing some application awareness into the hypervisor. (For instance, VT-ASOS hypervisor will monitor performance of individual applications using hardware counters. Furthermore, it will use a configurable scheduler that is controlled by guest domains.) We explicitly avoid heavy-weight interactions between guest and hypervisor, because we believe that they might limit scalability on many-core machines.

Another area of related work is on scalable OS design [6,12,16], with Tornado [16] and K42 [6] operating systems being some of the most prominent contributions. We will study applicability of those designs to our problem domain and use them as appropriate.

Finally, we discuss specialized execution environments [13]. A specialized execution environment (SEE) is created by configuring the features of OS/runtime uniquely for each workload. In SEE, it is possible to turn off some features of the OS, such as scheduling or memory management. By "getting the OS out of the way", SEEs can improve performance. We believe that our hypervisor will facilitate SEEs for high-performance applications. Partitioning and heterogeneity awareness give the guest a higher control over CPU resources than that which would be provided by existing hypervisors. Our hypervisor, therefore, ideally suits high-performance SEEs where such control is needed.

6. SUMMARY

Scale and heterogeneity of future many-core processors motivate us to reconsider the design of the scheduling infrastructure in the hypervisor. We argued that the scheduler must be designed to scale to thousands of cores and to provide effective utilization of heterogeneous hardware. We presented the design of Cypress, our experimental hypervisor, with which we hope to learn how to achieve these goals.

Cypress is novel in two respects: It will be the first hypervisor with explicit support for heterogeneous hardware, and it will be explicitly designed to scale on machines with hundreds or thousands or processing cores. Implementation, evaluation and refinement of design ideas in Cypress are currently under way and we hope to share our results with the community in the near future.

7. REFERENCES

[1] SPEC CPU2000 web site. *http://www.spec.org*

[2] Multi-NIC Networking Performance in ESX 3.0.1 and XenEnterprise 3.2.0. *VMWare Technical Note, http://www.vmware.com/pdf/Multi-NIC_Performance.pdf*

[3] Networking performance in multiple virtual machines. *Technical Note, VMWare, http://www.vmware.com/pdf/Multi-VM_Network_Performance.pdf*

[4] RAMP - Research Accelerator for Multiple Processors. *http://ramp.eecs.berkeley.edu*

[5] E. Ackaouy. The Xen Credit CPU Scheduler. *Xen Summit, http://xen.org/files/summit_3/sched.pdf*

[6] J. Appavoo, M. Auslander, M. Burtico, D. Da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. Wisniewski, and J. Xenidis. Experience With K42: an Open-Source Linux-Compatible Scalable Operating

System Kernel. *IBM Systems Journal*, 44(2):427-440, 2005

[7] K. Asanovic and et al. The Landscape of Parallel Computing Research: A View From Berkeley. *UC Berkeley Technical Report UCB/EECS-2006-183,* 2006

[8] G. Back and D. Nikolopoulos. Application-Specific System Customization on Many-Core Platforms: The VT-ASOS Framework. In *Proceedings of Second Workshop on Software Tools for Multi-Core Systems,* 2007

[9] P. Barham, B. Dragovic, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of SOSP,* 2003

[10] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of Conference on Computing Frontiers,* 2006

[11] S. Borkar. Thousand Core Chips—A Technology Perspective. In *Proceedings of DAC,* 2007

[12] R. Bryant, J. Hawkes, and J. Steiner. Scaling Linux to the Extreme: From 64 to 512 Processors. In *Proceedings of Ottawa Linux Symposium,* 2004

[13] M. Burtico, D. Da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, E. Hensbergen, R. Wisniewski, and J. Xenidis. Specialized Execution Environments. *ACM SIGOPS Operating Systems Review*, 42(1):106-107, 2008

[14] G. Dhiman and T. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of International Symposium on Low Power Electronics and Design,* 2007

[15] Alexandra Fedorova, D. Vengerov, and Daniel Doucette. Operating System Scheduling On Heterogeneous Multicore Systems. In *Proceedings of the PACT'07 Workshop on Operating System Support for Heterogeneous Multicore Architectures,* 2007

[16] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of USENIX Annual Technical Conference,* 1999

[17] C. Gough, S. Siddha, and K. Chen. Kernel Scalability -- Explanding the Hoizon Beyond Fine Grain Locks. In *Proceedings of Ottawa Linux Symposium,* 2007

[18] K. Hoste and L. Eechhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro Hot Tutorials*, 27(3):63-72, 2007

[19] R. Kumar, K. Farkas, N. Jouppi, R. Parthasarathy, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of 36th annual IEEE/ACM International Symposium on Microarchitecture,* 2003

[20] R. Kumar, Dean M. Tullsen, R. Parthasarathy, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. In *Proceedings of 31st Annual International Symposium on Computer Architecture,* 2004

[21] Richard McDougall and Jim Mauro. Solaris™ Internals: Solaris 10 and OpenSolaris Kernel Architecture. *Prentice Hall*, 2006

[22] B. Quetier, V. Neri, and F. Cappello. Scalability Comparison of Four Host Virtualization Tools. *Journal of Grid Computing*, 5(1):83-98, March 2007

[23] K. Rajamani, H. Hanson, J. Rubio, S. Ghiasi, and F. Rawson. Application-Aware Power Management. In *Proceedings of IEEE International Symposium on Workload Characterization,* 2006

[24] Suresh Siddha and Venkatesh Pallipadi. Chip Multi Processing Aware Linux Kernel Scheduler. In *Proceedings of Linux Symposium,* 2005

[25] Burton Smith. Many-Core Operating Systems. In *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-34, Keynote Speech,* 2007

[26] VMWare White Paper. VMware ESX Server 2 Architecture and Performance Implications. *http://www.vmware.com/pdf/esx2_performance_implications.pdf*