
MULTICORE RESOURCE MANAGEMENT

CURRENT RESOURCE MANAGEMENT MECHANISMS AND POLICIES ARE INADEQUATE FOR FUTURE MULTICORE SYSTEMS. INSTEAD, A HARDWARE/SOFTWARE INTERFACE BASED ON THE VIRTUAL PRIVATE MACHINE ABSTRACTION WOULD ALLOW SOFTWARE POLICIES TO EXPLICITLY MANAGE MICROARCHITECTURE RESOURCES. VPM POLICIES, IMPLEMENTED PRIMARILY IN SOFTWARE, TRANSLATE APPLICATION AND SYSTEM OBJECTIVES INTO VPM RESOURCE ASSIGNMENTS. THEN, VPM MECHANISMS SECURELY MULTIPLEX, ARBITRATE, OR DISTRIBUTE HARDWARE RESOURCES TO SATISFY THE VPM ASSIGNMENTS.

Kyle J. Nesbit

James E. Smith

University of Wisconsin

—Madison

Miquel Moreto

Polytechnic University of

Catalonia

Francisco J. Cazorla

Barcelona Supercomputing

Center

Alex Ramirez

Mateo Valero

Barcelona Supercomputing

Center and Polytechnic

University of Catalonia

..... Continuing the long-term trend of increasing integration, the number of cores per chip is projected to increase with each successive technology generation. These chips yield increasingly powerful systems with reduced cost and improved efficiency. At the same time, general-purpose computing is moving off desktops and onto diverse devices such as cell phones, digital entertainment centers, and data-center servers.¹ These computers must have the key features of today's general-purpose systems (high performance and programmability) while satisfying increasingly diverse and stringent cost, power, and real-time performance constraints.

An important aspect of multicore chips is improved hardware resource utilization. On a multicore chip, concurrently executing threads can share costly microarchitecture resources that would otherwise be underutilized, such as off-chip bandwidth. Higher resource utilization improves aggregate performance and enables lower-cost design alternatives, such as smaller die area or less exotic battery technology. However, increased resource sharing presents a number of new design challenges. In particular, greater hardware resource sharing among concurrently executing threads can cause

individual thread performance to become unpredictable and might lead to violations of the individual applications' performance requirements.^{2,3}

Traditionally, operating systems are responsible for managing shared hardware resources—processor(s), memory, and I/O. This works well in systems where processors are independent entities, each with its own microarchitecture resources. However, in multicore chips, processors are concurrently executing threads that compete with each other for fine-grain microarchitecture resources. Hence, conventional operating system policies don't have adequate control over hardware resource management. To make matters worse, the operating system's software policies and the hardware policies in the independently developed microarchitecture might conflict. Consequently, this compromises operating system policies directed at overall system priorities and real-time performance objectives.

In the context of future applications, this poses a serious system design problem, making current resource management mechanisms and policies no longer adequate for future multicore systems. Policies, mechanisms, and the interfaces between them must change to fit the multicore era.

In this article, our vision for resource management in future multicore systems involves enriched interaction between system software and hardware. Our goal is for the application and system software to manage all the shared hardware resources in a multicore system. For example, a developer or end user will specify an application's quality-of-service (QoS) objectives, and the developer or system software stack will translate these objectives into hardware resource assignments. Because QoS objectives are often application specific, the envisioned multicore architecture provides an efficient and general interface that can satisfy QoS objectives over a range of applications. By enriching the interaction between hardware and software, the envisioned resource management framework facilitates a more efficient, better performing platform design.

Designing general-purpose systems requires a clear separation of policies and mechanisms.⁴ *Policies* provide solutions; for flexibility, we strive for policies implemented in software. *Mechanisms* provide the primitives for constructing policies. Because primitives are universal, system designers can implement mechanisms in both hardware and software. In general, mechanisms that interact directly with fine-grain hardware resources should be implemented in hardware; to reduce hardware cost, mechanisms that manage coarse-grain resources should be implemented in software.

Virtual private machines

In a traditional multiprogrammed system, the operating system assigns each application (program) a portion of the physical resources—for example, physical memory and processor time slices. From the application's perspective, each application has its own private machine with a corresponding amount of physical memory and processing capabilities. With multicore chips containing shared microarchitecture-level resources, however, an application's machine is no longer private, so resource usage by other, independent applications can affect its resources.

Therefore, we introduce the virtual private machine (VPM) framework as a

means for resource management in systems based on multicore chips.³ VPMs are similar in principle to classical virtual machines. However, classical virtual machines virtualize a system's functionality⁵ (ignoring implementation features), while VPMs virtualize a system's performance and power characteristics, which are implementation specific. A VPM consists of a complete set of virtual hardware resources, both spatial (physical) resources and temporal resources (time slices). These include the shared microarchitecture-level resources. By definition, a VPM has the same performance and power characteristics as a real machine with an equivalent set of hardware resources.

The VPM abstraction provides the conceptual interface between policies and mechanisms. VPM policies, implemented primarily in software, translate application and system objectives into VPM resource assignments, thereby managing system resources. Then, VPM mechanisms securely multiplex, arbitrate, or distribute hardware resources to satisfy the VPM assignments.

Spatial component

A VPM's spatial component specifies the fractions of the system's physical resources that are dedicated to the VPM during the time(s) that it executes a thread. For example, consider a baseline system containing four processors, each with a private L1 cache. The processors share an L2 cache, main memory, and supporting interconnection structures. Figure 1 shows that the policy has distributed these resources among three VPMs. VPM 1 contains two processors, and VPMs 2 and 3 each contain a single processor. The policy assigns VPM 1 a significant fraction (50 percent) of the shared resources to support a demanding multithreaded multimedia application and assigns the other two VPMs only 10 percent of the resources. These assignments leave 30 percent of the cache and memory resources unallocated; these resources are called *excess service*. Excess service policies distribute excess service to improve overall resource utilization and optimize secondary performance objectives.

In our example, we focus on shared memory hierarchy resources, but VPM concepts also apply to internal processor

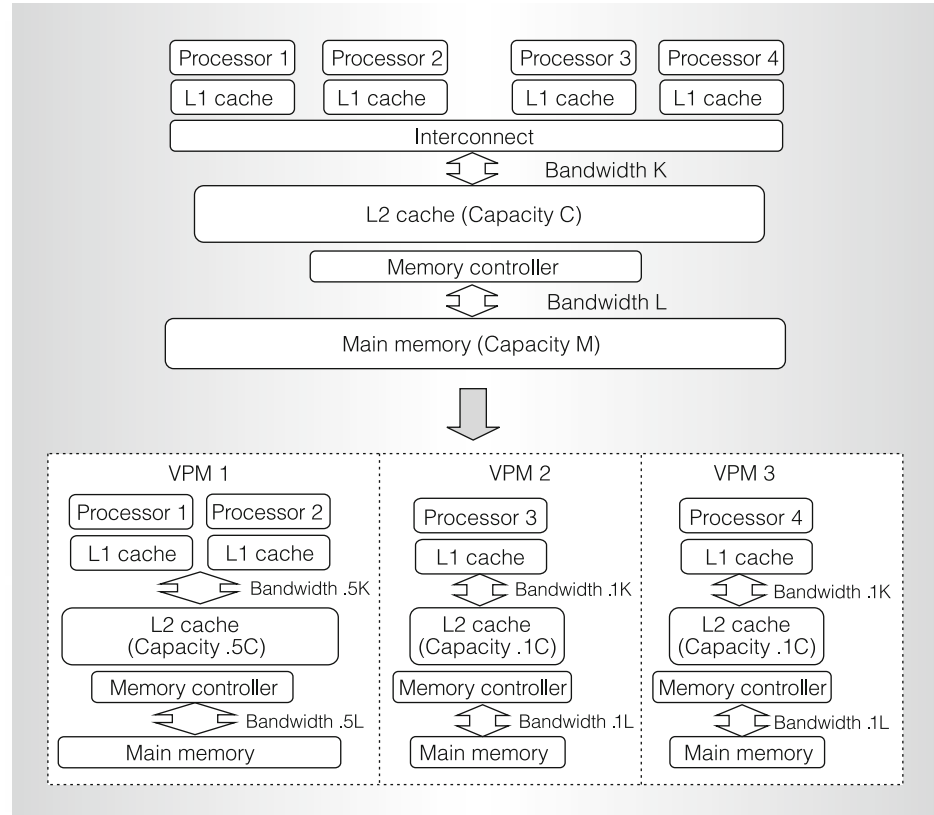


Figure 1. Virtual private machine (VPM) spatial component. The policy has distributed the multicore chip's resources among three VPMs. After assigning VPM 1 50 percent of the shared resources and VPMs 2 and 3 each 10 percent, it leaves 30 percent of the cache and memory resources unallocated for excess service.

resources, such as issue buffers, load/store queue entries, and instruction decode and execution bandwidth.² Furthermore, we can apply the same concepts to architected resources such as memory capacity and I/O.

As Figure 1 illustrates, a VPM's spatial component might contain multiple processors. Multiprocessor VPMs are a natural extension of gang scheduling and support hierarchical resource management.^{6,7} For example, schedulers and resource management policies running within a multiprocessor VPM can schedule and manage the VPM's processors and resources as if the VPM were a real multiprocessor machine. That is, multiprocessor VPMs can support *recursive virtualization*.⁵

Temporal component

A VPM's temporal component is based on the well-established concept of ideal

proportional sharing.⁸ It specifies the fraction of processor time (processor time slices) that a VPM's spatial resources are dedicated to the VPM (see Figure 2). As with spatial VPM resources, a VPM's temporal component naturally lends itself to recursive virtualization and hierarchical resources management, and excess temporal service might exist.

Minimum and maximum VPMs

To satisfy objectives, policies might assign applications minimum or maximum VPMs, or both, depending on the objectives. Mechanisms ensure an application is offered a VPM that is greater than or equal to the application's assigned minimum VPM. (We precisely define the greater than or equal to VPM ordering relationship elsewhere.³) Informally, the mechanisms offer an application at least the resources

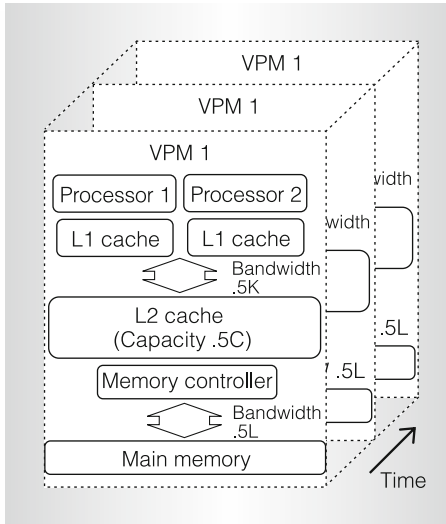


Figure 2. VPMs consist of a spatial component and a temporal component. The temporal component specifies the fraction of processor time that a VPM's spatial resources are dedicated to the VPM.

of its assigned minimum VPM. When combined with the assumption that an application will only perform better if it is offered additional resources (performance monotonicity⁹), ensuring a minimum VPM assignment leads to desirable *performance isolation*; that is, the application running on the VPM performs at least as well as it would if it were executing on a real machine with a configuration equivalent to the application's assigned VPM. This performance level is assured, regardless of the other applications in the system.

Mechanisms can also ensure an application receives no more than its maximum VPM resources. Policies can use maximum VPM assignments to control applications' power consumption, which is based on the assumption that an application's power consumption is a monotonically increasing function of its resource usage (power monotonicity). For maximum VPM assignments to improve power savings significantly, they should be supported by mechanisms that power down unused resources. Furthermore, because temperature and transistor wear-out strongly depend on power consumption, policies can use maximum

VPMs to control temperature and lifetime reliability.¹⁰ Lastly, application developers can use maximum VPMs to test whether a minimum VPM configuration satisfies an application's real-time performance requirements.⁹

Policies

VPM assignments satisfy real-time performance, aggregate performance, power, temperature, and lifetime reliability objectives. Overall, the VPM policy design space is enormous and is a fertile area for future research. Here, we begin with a high-level overview of the policy design space and then discuss the basic types of policies and how they interact in our envisioned system architecture.

In general, we envision two basic types of policies. *Application-level policies* satisfy an application's QoS objectives by translating the QoS objectives into a VPM configuration as well as scheduling and managing VPM resources assigned to the application. *System policies* satisfy system objectives by controlling the distribution of VPM resources among applications. System policies control resource distribution by granting and rejecting requests for VPM resources and revoking VPM resources when the system is overloaded.

The policy architecture's main feature is its extensibility (see Figure 3, next page): we clearly separate policies and mechanisms,⁴ and policies are easy to replace or modify on a per system and per application basis.¹¹ For example, in Figure 3, a system is running five applications, each within its own VPM (not shown). The first application (on the left) is a real-time recognition application. To satisfy its real-time requirements, the application is running with an application-specific VPM that the application's developer computed offline. The second and third applications (mining and synthesis) are written in a domain-specific, concurrent programming language. The language includes a runtime that computes VPM configurations online and optimizes applications' concurrency in coordination with the applications' VPM resources. The mining and synthesis applications are isolated from each other, but they share the

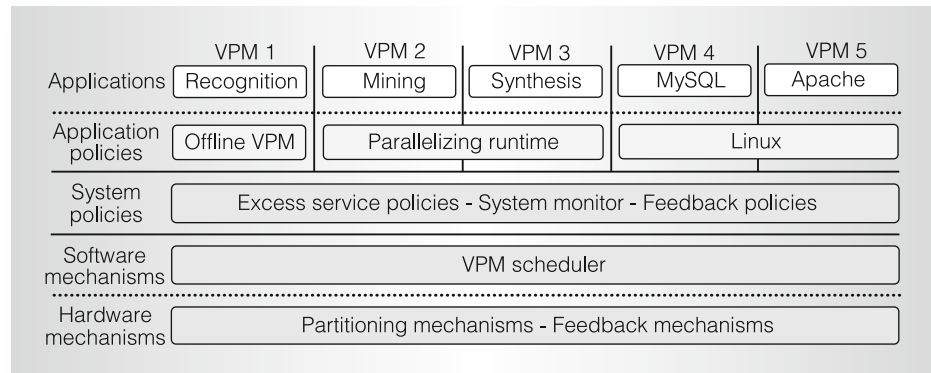


Figure 3. The VPM system architecture consists of application-level policies, system policies, software mechanisms, and hardware mechanisms. The extensible policy architecture lets policy builders modify policies on a per system and per application basis.

library code that implements the language's runtime. The last two applications (MySQL and Apache) are standard Linux applications that are oblivious to the underlying system's VPM support. The applications are running on a paravirtualized version of Linux, which in turn is running on a thin software layer that monitors the applications' workload characteristics and roughly

computes VPM configurations using heuristics and ad hoc techniques. Most importantly, application-level policies allow application developers and runtime libraries to customize a system's behavior to satisfy a range of applications' requirements.

Application-level policies

In general, there are two logical steps for determining VPM assignments: modeling and translation. We describe the steps as separate phases, although in practice they may be combined. As we described earlier, application policies compute VPM configurations either online (automated) or offline (manually). Online policies are based primarily on runtime analysis (for example, by using performance counters), while offline policies require an application developer to perform most of the program analysis. Online/offline hybrid policies are also feasible.

In addition, application policies can use standardized application-level abstraction layers that provide developers with abstract machine models. Such models are often implementation independent and have performance characteristics that are easier for developers to reason about (see Figure 4).

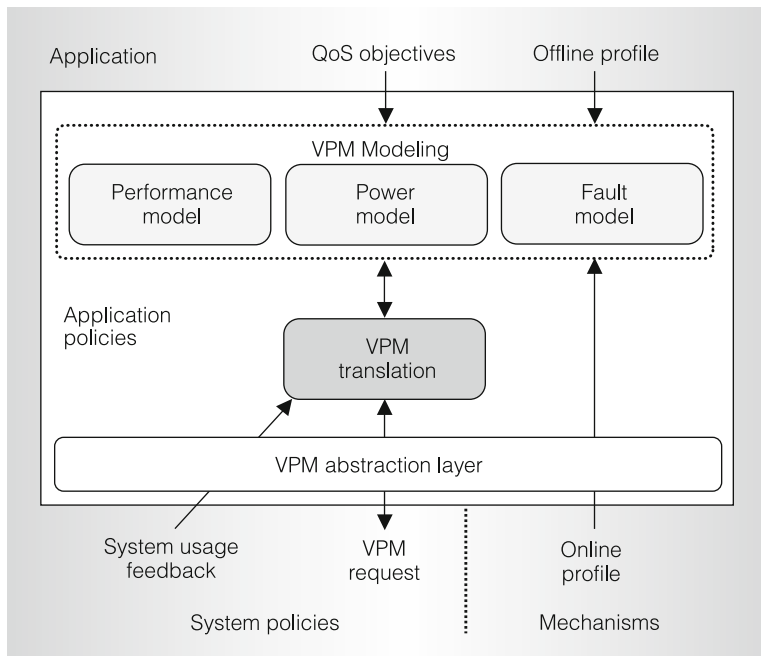


Figure 4. Application policies compute VPM configurations in two logical steps: VPM modeling and translation. Standardized application-level abstractions can be used to abstract away irrelevant implementation-specific VPM details.

VPM modeling. The VPM modeling step maps application QoS objectives, such as minimum performance and maximum power, to VPM configurations. The sophistication of VPM modeling techniques spans a fairly wide range. At one end are simple,

general-purpose analytical models that use generic online profiling information to roughly predict an application's performance and power when running on different VPM configurations. At the other end are models specialized to a specific application (or a domain of applications) through offline profiling and characterization. Such offline models can precisely capture an application's performance and power on different VPM configurations.

Application- or domain-specific VPM modeling can provide more precise predictions but might require more developer effort—for example, to determine suitable VPM configurations offline. Applications with critical QoS objectives (such as real-time applications) will generally require more precise VPM modeling.

VPM translation. The VPM translation step uses the VPM models to find VPM configurations that satisfy an application's QoS objectives. Multiple VPM configurations can satisfy the same objective. For example, multiple minimum VPM configurations can satisfy a single real-time performance objective; that is, one suitable VPM configuration might have a larger spatial component and smaller temporal component, while another suitable VPM might have a larger temporal component and smaller spatial component. Or there might be different combinations of resources within the spatial component.

Furthermore, applications might have multiple objectives that a policy can use to prune the number of suitable VPM configurations. For example, an application policy can search for a VPM configuration that satisfies a real-time performance requirement and minimizes power consumption. Moreover, a policy can assign an application a maximum VPM to bound the application's power consumption. In many practical situations, finding the optimal VPM configuration is NP-hard. Consequently, online translation generally must use approximate and heuristic-based optimization techniques. For applications with critical QoS objectives, the application developer can do a portion of the VPM translation offline. Moreover, the developer can com-

bine the VPM modeling and translation steps into a single step. For example, an application developer might use maximum VPM assignments and trial and error to compute a minimum VPM configuration that satisfies their application's real-time performance objective.⁹

Once the policy has found a suitable VPM configuration, it initiates a request to the system's policies for the VPM resources. If the VPM resources are available, the system policies securely bind the VPM resources to the application.¹¹ When a system is heavily loaded, the system policies might reject an application's VPM request or revoke a previous VPM resource binding.

An application policy must implement a procedure for handling VPM rejections and revocations. When a rejection or revocation occurs, an application policy can find another suitable VPM configuration or reconfigure the application to reduce the application's resource requirements. For example, a real-time media player can downgrade its video quality or simply return an insufficient resource error message and exit.

To help with global (systemwide) resource optimization, VPM system policies can provide feedback to the applications' policies. The feedback informs the applications of global resource usage. An application's policies can use the system feedback information to further prune the suitable VPM configurations and find a VPM that is amenable to systemwide resource constraints. In addition, an application's policies can use VPM modeling and online profiling information to dynamically respond to changing workload characteristics.

VPM abstraction layer. In our discussion so far, we've assumed a relatively high-level VPM abstraction—for example, VPMs that consist of shares of cache bandwidth, cache storage, and memory system bandwidth. However, real hardware resources are more complex. For example, a physical cache implementation consists of banks, sets, and ways. VPM mechanisms don't abstract hardware resources; that is, the VPM mechanisms convey implementation specific details to the application policies. Expos-

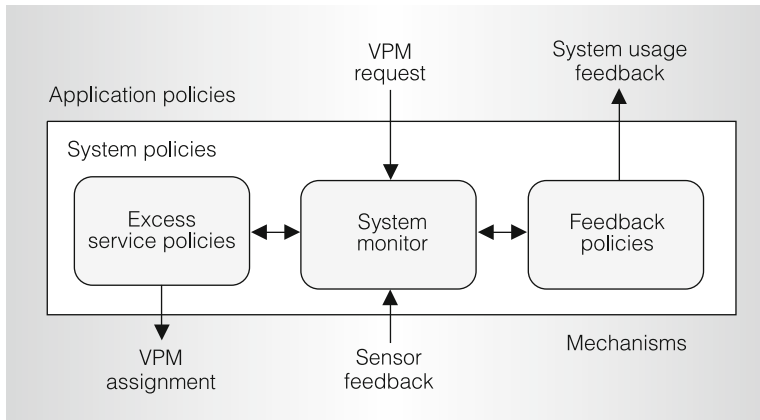


Figure 5. System policies broker VPM requests and monitor application resource usage to ensure that the system does not become overloaded.

ing implementation details creates an interface that's more efficient to implement and grants more latitude to implementers of higher-level abstractions.¹¹

However, exposing implementation details to applications can make applications implementation dependent. Moreover, many application policies don't need low-level implementation details to satisfy their QoS objectives. To improve application compatibility, many application policies can use standardized VPM abstraction layers. A VPM abstraction layer provides a high-level VPM that abstracts away a system's implementation-specific details while capturing the system's first-order performance and power characteristics. At runtime, the abstraction layer efficiently translates high-level VPM configurations into low-level VPM assignments that the system's mechanisms support.

System policies

System policies control the distribution of VPM resources by brokering applications' VPM requests. System policies have three basic components: a system monitor, feedback policies, and excess service policies (see Figure 5).

System monitor. The system monitor tracks the load on the system's resources and detects system overload. A system is overloaded if it doesn't have enough resources to satisfy its applications' VPM

assignments. In general, it's best to detect overload as early as possible, such as when an application initiates a VPM request that causes the system to become overloaded. To detect system overload, the system must monitor any shared resource that can become overloaded, such as cache storage, execution bandwidth, cooling capacity, and power delivery. We can detect that a VPM request overloads a system's physical resources with an admission control test. For example, the system monitor can compare the set of dedicated VPM resources with the capacity of the system's physical resources.

Detecting that a VPM request overloads a system's less tangible resource constraints (such as a power-delivery system, cooling capacity, or transistor lifetime reliability) is more difficult. The system monitor must use VPM models to translate between less tangible resource constraints and VPM configurations. The VPM models should be accurate enough to detect most overload conditions at the time an application initiates a VPM request; however, they don't have to be completely accurate. The system monitor's VPM models are used to supplement conventional hardware sensors, such as voltage and temperature sensors. Hardware sensors can detect anomalous overload conditions (such as overheating) in real time and prevent catastrophic failures. If a sensor detects an overloaded resource, the system policies have accepted more VPM requests than the system's physical resources can satisfy. In this case, the system policies must revoke some applications' assigned VPM resources.

Feedback policies. Feedback policies determine which applications' VPM resources should be rejected or revoked when the system is overloaded. For example, when a system is overloaded, a feedback policy might reject any incoming VPM request or revoke the VPM resources of the application with the lowest user-assigned priority. Feedback policies also inform applications of systemwide resource utilization and the cause(s) of rejected or revoked VPM requests—for example, the system might have insufficient power resources available.

As we described earlier, applications' policies can use the feedback to compute a suitable VPM configuration that's amenable to systemwide resource usage. The feedback policies should also provide applications with up-to-date systemwide resource usage information regardless of VPM rejections and revocations. This way the feedback policies can direct global optimization of a system's resource usage.

Excess service policies. After application and system policies have settled on a suitable set of VPM assignments, there might be excess service available. Excess service is service that is unassigned or assigned but unused by the application to which it's assigned. Excess service policies distribute excess service to optimize system objectives. For example, these policies can distribute service to improve response time or throughput averaged over all applications, to conserve power or transistor lifetime, or a combination of such objectives. To optimize power or transistor lifetime objectives, excess service policies prevent applications from consuming the excess service. That is, these policies assign tasks maximum VPMs, thus causing the excess resources to be powered off. Excess service policies must also ensure that distributing excess service doesn't violate applications' maximum VPM assignments.

In most cases, excess service policies transparently adjust applications' VPM assignments. For example, they can transparently increase a minimum VPM assignment or decrease a maximum VPM assignment without violating the application policy's VPM assignments. For some resources, the policies must notify an application's policies when excess resources are added—for example, they must notify an application's thread scheduler when processors are added to a VPM.

For some resources, excess service becomes available and must be distributed at a fine time granularity, such as with SDRAM memory system bandwidth. In such cases, a portion of the excess service policy must be implemented in hardware. However, a policy's hardware portion should be simple, parameterized, and general; that is, it should

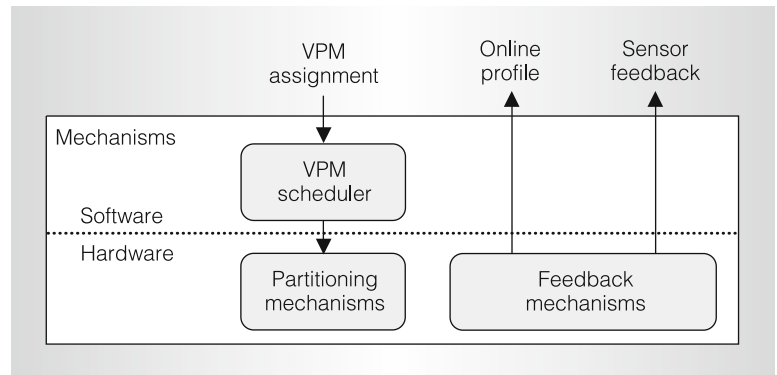


Figure 6. VPM mechanisms are implemented in hardware and software. The mechanisms satisfy VPM resource assignments and provide feedback regarding individual application and systemwide resource usage.

work with multiple software excess-service policies.

Mechanisms

There are three basic types of mechanisms needed to support the VPM framework: a VPM scheduler, partitioning mechanisms, and feedback mechanisms (see Figure 6). The first two types securely multiplex, arbitrate, or distribute hardware resources to satisfy VPM assignments. The third provides feedback to application and system policies. Because mechanisms are universal, system builders can implement mechanisms in both hardware and software. Generally, the VPM scheduler is implemented in software (in a microkernel¹¹ or a virtual machine monitor¹²), while the partitioning mechanisms and feedback mechanisms are primarily implemented in hardware. Although a basic set of VPM mechanisms are available,^{2,3} many research opportunities remain to develop more efficient and robust VPM mechanisms.

VPM scheduler

The VPM scheduler satisfies applications' temporal VPM assignments by time-slicing hardware threads.³ The VPM scheduler is a proportional-fair (p-fair) scheduler,¹³ but it must also ensure that coscheduled applications' spatial resource assignments don't conflict—that is, that the set of coscheduled threads' spatial resource assignments match the physical resources available and don't oversubscribe any microarchitecture re-

sources. VPM scheduling in its full generality, satisfying proportional fairness without spatial conflicts, is an open research problem.³

When the VPM scheduler context switches an application onto a processor (or a group of processors), the scheduler communicates the application's spatial VPM assignment to the hardware partitioning mechanisms through privileged control registers. Once the control registers are configured, the VPM resources are securely bound to the application. Secure binding decouples the authorization from resource usage¹¹—that is, once a resource is securely bound to an application, the application's policies can schedule and manage its VPM resources without reauthorization. This way VPMs can efficiently support hierarchical scheduling.⁶

The VPM scheduler we describe here is a first-level scheduler (or root scheduler), and application-level schedulers are second-level schedulers. Hierarchical scheduling is useful for satisfying different classes of QoS requirements.⁶ Furthermore, precise application-level schedulers will play an important role in future parallel programming models.

Partitioning mechanisms

To satisfy the spatial component of VPM assignments, each shared microarchitecture resource must be under the control of a partitioning mechanism that can enforce minimum and maximum resource assignments. As we described earlier, the resource assignments are stored in privileged control registers that the VPM scheduler configures. In general, each shared microarchitecture resource is one of three basic types of resources: a memory storage, buffer, or bandwidth resource. Each type of resource has a basic type of partitioning mechanism. For example, thread-aware replacement algorithms partition storage resources (main memory and cache storage^{3,14}), upstream flow control mechanisms partition buffer resources (issue queue and miss status handling registers²), and fair-queuing and traffic-shaping arbiters partition bandwidth resources (execution and memory ports³). These basic techniques combined with the

proper control logic can sufficiently enforce minimum and maximum resource assignments.

For maximum VPM assignments to be useful, the partitioning mechanisms must be accompanied by mechanisms to power down resources during periods of inactivity. An example would be, mechanisms that clock-gate unused pipeline stages and transition inactive memory storage resources into a low-power state. As we mentioned earlier, by controlling resources' power consumption, policies can control other important characteristics such as die temperature and transistor wear out.⁵

Feedback mechanisms

Mechanisms also provide application and system policies with feedback regarding physical resource capacity and usage. Feedback mechanisms communicate to system policies the capacity of the system's resources and the available VPM partitioning mechanisms. They also provide application policies with information regarding individual applications' resource usage.

Application resource usage information should be independent of the system architecture and the application's VPM assignments. For example, a mechanism that measures a stack distance histogram can predict cache storage and memory bandwidth usage for many different cache sizes.¹⁴

Lastly, feedback mechanisms provide information regarding overall resource utilization. For example, a system's mechanisms should provide system policies with die-temperature and power-consumption information.

Overall, the VPM framework provides a solid foundation for future architecture research, but many challenges remain in evaluating the VPM framework. First, the framework targets system-level metrics that occur over time granularities that preclude the use of simulation. Second, many of the applications we're interested in (such as smart phone or cloud computer applications) are unavailable or unknown. To address these problems, we plan to develop most of the framework mathematically. (The basis of the mathematical

framework is available elsewhere.³⁾ We plan to validate the mathematical framework and assumptions using statistical simulation. In turn, we plan to validate the statistical methods by prototyping the envisioned system architecture and policies in FPGAs.

MICRO

References

1. S. Lohr and M. Helft, "Google Gets Ready to Rumble with Microsoft," *New York Times*, 16 Dec. 2007; www.nytimes.com/2007/12/16/technology/16goog.html.
2. F.J. Cazorla et al., "Predictable Performance in SMT Processors: Synergy between the OS and SMTs," *IEEE Trans. Computers*, vol. 55, no. 7, Jul. 2006, pp. 785-799.
3. K.J. Nesbit, J. Laudon, and J.E. Smith, *Virtual Private Machines: A Resource Abstraction for Multicore Computer Systems*, tech. report 07-08, Electrical and Computer Engineering Dept., University of Wisconsin-Madison, Dec. 2007.
4. R. Levin et al., "Policy/Mechanism Separation in Hydra," *Proc. 5th ACM Symp. Operating Systems Principles (SOSP 75)*, ACM Press, 1975, pp. 132-140.
5. G.J. Popek and R.P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Comm. ACM*, vol. 74, no. 7, Jul. 1974, pp. 412-421.
6. P. Goyal, X. Guo, and H.M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *SIGOPS Operating Systems Rev.*, vol. 30, no. SI, Oct. 1996, pp. 107-121.
7. J.K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS 82)*, IEEE CS Press, 1982, pp. 22-30.
8. A.K. Parekh and R.G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Trans. Networks*, vol. 1, no. 3, Jun. 1993, pp. 344-357.
9. J.W. Lee and K. Asanovic, "METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors," *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS 06)*, IEEE CS Press, 2006, pp. 135-147.
10. A.S. Sedra and K.C. Smith, *Microelectronic Circuits*, 5th ed., Oxford Univ. Press, 2004.
11. D.R. Engler, M.F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP 95)*, ACM Press, Dec. 1995, pp. 251-266.
12. J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, 2005.
13. S.K. Baruah et al., "Proportionate Progress: A Notion of Fairness in Resource Allocation," *Proc. 25th ACM Symp. Theory of Computing (STOC 93)*, ACM Press, 1993, pp. 345-354.
14. G.E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," *Proc. 8th Int'l Symp. High-Performance Computer Architecture (HPCA 02)*, IEEE CS Press, 2002, pp. 117-128.

Kyle J. Nesbit is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include computer architecture and virtual machines, particularly multicore architectures and resource management for emerging platforms. Nesbit received his BS in computer engineering from the University of Wisconsin-Madison.

Miquel Moreto is a PhD candidate in the Computer Architecture Department at the Polytechnic University of Catalonia, Spain. His research interests include modeling parallel computers and resource sharing in multithreaded architectures. Moreto received his MS in both mathematics and electrical engineering from the Polytechnic University of Catalonia.

Francisco J. Cazorla is the leader of the Operating System/Computer Architecture Interface group at the Barcelona Supercomputing Center. His research focuses on multithreaded architectures for both high-performance and real-time computing systems. Cazorla received his PhD in Computer Architecture from the Polytechnic University of Catalonia.

Alex Ramirez is an associate professor at the Polytechnic University of Catalonia and research manager at the Barcelona Supercomputing Center. Research interests include heterogeneous multicore architectures, hardware support for programming models, and simulation techniques. He received his PhD in computer science from the Polytechnic University of Catalonia.

Mateo Valero is a professor at the Polytechnic University of Catalonia and director of the Barcelona Supercomputer Center. His research interests include high-performance architectures. Valero has a PhD in telecommunications from the Polytechnic University of Catalonia. He is a recipient of the Eckert-Mauchly Award and a founding member of the Royal Spanish Academy of Engineering. He is an IEEE Fellow, an Intel Distinguished Research Fellow, and an ACM Fellow.

James E. Smith is a professor emeritus in the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison and a member of the technical staff at Google. His current research interests include high-performance and power-efficient processor implementations, processor performance modeling, and virtual machines. Smith has a PhD in computer science from the University of Illinois.

Direct questions and comments about this article to Kyle J. Nesbit, Univ. of Wisconsin–Madison, 2420 Engineering Hall, 1415 Engineering Dr., Madison, WI 53706-1691; kjnesbit@wisc.edu.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.

Sign Up Today



For the IEEE Computer Society Digital Library E-Mail Newsletter

- Monthly updates highlight the latest additions to the digital library from all 23 peer-reviewed Computer Society periodicals.
- New links access recent Computer Society conference publications.
- Sponsors offer readers special deals on products and events.

Available for FREE to members, students, and computing professionals.

Visit http://www.computer.org/services/csdl_subscribe