A Multilevel Introspective Dynamic Optimization System For Holistic Power-Aware Computing

Vasanth Venkatachalam[1], Christian W. Probst[2] and Michael Franz[1]

[1] Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA, 92697, USA
{vvenkata,franz}@uci.edu
[2] Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kongens Lyngby, Denmark
probst@imm.dtu.dk

Abstract. Power consumption is rapidly becoming the dominant limiting factor for further improvements in computer design. Curiously, this applies both at the "high-end" of workstations and servers and the "low end" of handheld devices and embedded computers. At the high-end, the challenge lies in dealing with exponentially growing power densities. At the low-end, there is a demand to make mobile devices more powerful and longer lasting, but battery technology is not improving at the same rate that power consumption is rising. Traditional power-management research is fragmented; techniques are being developed at specific levels, without fully exploring their synergy with other levels. Most software techniques target either operating systems or compilers but do not explore the interaction between the two layers. These techniques also have not fully explored the potential of virtual machines for power management.
In contrast, we are developing a system that integrates information from multiple levels of software and hardware, connecting these levels through a communication channel. At the heart of this system are a virtual machine that compiles and dynamically profiles code, and an optimizer that reoptimizes all code, including that of applications and the virtual machine itself. We believe this introspective, holistic approach enables more informed power-management decisions.

Keywords. Power-aware Computing, Virtual Machines, Dynamic Optimization

1 Introduction

Power consumption has become the main obstacle for improving computer performance. As computers become faster and more versatile, they consume more

Dagstuhl Seminar Proceedings 05141
Power-aware Computing Systems
http://drops.dagstuhl.de/opus/volltexte/2005/309

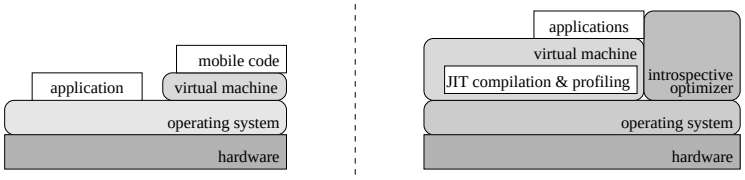

Fig. 1. High-level overview of traditional system architectures (left), and our architecture (right). The virtual machine layer is essential for compiling and executing applications: all application code executes above this layer, and applications as well as the virtual machine are profiled and reoptimized. This holistic approach allows power-management decisions to adapt to variations in resource levels and the execution behavior of the virtual machine and diverse applications.

power and become hotter. Given the rate at which chip temperature is rising, it is likely to approach "nuclear reactor" levels within ten years [29]. The main negative impacts of high chip temperatures are impaired processor reliability and life expectancy along with increased cooling costs. For battery-powered devices, power consumption poses an additional, severe problem: these devices are extremely limited in functionality, because fast processors and large memories drain batteries quickly. Without cost-effective solutions to the power problem, improvements in processor technology will eventually reach a standstill.

Encouragingly, there has been a substantial amount of research into "power-aware computing". However, traditional power-management techniques are limited in two ways. First, power management is a multi-dimensional problem, while most of the existing research addresses only a single dimension. Currently, techniques for reducing power are applied exclusively at the hardware level, operating system level, compiler level, or application level. However, each of these levels has limitations that make it non-ideal to base any decision involving program optimization or resource management solely on information available to it. Moreover, existing techniques have not fully explored the potential of a software level that already exists in many devices, the virtual-machine level.

While the currently prevailing solutions are one-dimensional in that they only target *one* system layer, we are developing a system that integrates *all* layers into a virtual-machine-based power-management system. Our system integrates information about the structure of an individual program as seen by the compiler, about the tasks currently running in the system as seen by the operating system, about the actual runtime behavior of the whole system as seen by profilers, and hardware-specific information, such as temperature variations, resource levels, and memory and I/O bottlenecks as reported by hardware sensors.

The main components of our system (Figure 1) are a *virtual-machine layer* that compiles programs and profiles their execution behavior, and a separate *dynamic optimizer* that reoptimizes application code as well as VM code, to

adapt to varying runtime conditions. For this reason, we call this system *holistic* and *introspective*.

Our research leverages the growing body of results in software techniques for power reduction [11,13,14,22], dynamic optimization [4,5,6,7,12,23,24,25,30,41], and extensible operating systems [9,15,26,27,32,33,36,37,38]. It also extends prior work in introspective computers [17] to the power-management setting.

The remainder of this paper is organized as follows. Section 2 describes the limitations of traditional power-management approaches in more detail and explains why it is beneficial to use a holistic approach exploiting virtual machines for power management. Section 3 describes our holistic approach in more detail, and is followed by a discussion of recent related work on collaborative power management in Section 4. Section 5 describes the current status of our framework and gives an outlook on future work, and Section 6 concludes.

2 The Case For A Virtual-Machine-Based Holistic Approach

Power-management techniques, as we have noted, typically target either the hardware, operating system, compiler or application level. Each of these levels allows different kinds of techniques to be used for reducing power. The hardware level allows one to choose a power-efficient circuit structure. However, hardware contains limited information about program behavior, and it is costly to implement dedicated hardware that analyzes and optimizes programs. The software level has more information about program behavior and allows ways of reducing power without changing the underlying hardware. Yet, different levels of system software differ in their capabilities.

Operating systems have a wealth of information about the runtime environment. They know, for example, what tasks have been released into the system, where the tasks have been placed in memory, what resources the tasks are consuming, and in what order the tasks are being scheduled. Using this information, they can apply a number of techniques to reduce power. These include slowing down the processor, make paging decisions that reduce cache contention, and powering down idle peripheral devices. However, operating systems lack information about an individual program's control flow and data structures. This is because this information is lost when a program is compiled into its binary representation, and it is nontrivial to recover this information from the binary. As a result, operating systems lack forward-looking information about individual programs, and are forced to extrapolate future program behavior from past behavior. This prevents operating system heuristics for power management from adapting well to irregular programs. It also prevents operating systems from being able to modify the execution behavior of individual programs. Resource inefficient tasks may be released into the system, but standard operating systems will be unable to optimize these programs to better adapt to the runtime environment.

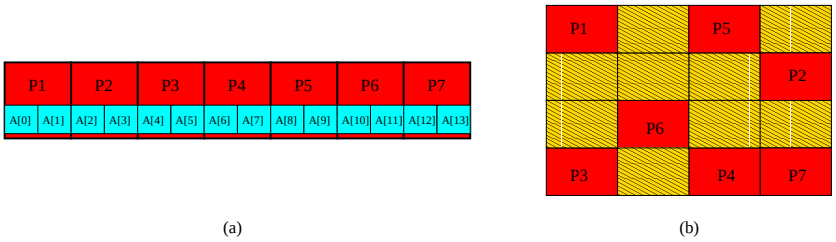

Fig. 2. An example of what can go wrong when programs are optimized in isolation from information about system-level events. A compiler assigns the successive elements of a large array to pages (P1-P7) that are contiguous in logical memory (a). However, physical memory may be internally fragmented and lack sufficient room for storing these pages contiguously. In this case, the operating system would have to assign these pages noncontiguous locations (b). If the array is traversed from beginning to end, every two array accesses will result in a random I/O, and thus may require a costly memory access to bring array elements into the cache.

Compilers, however, have forward looking information about individual program structure. As a result, compilers are in a better position to predict the control flow of a program, and to optimize a program's code prior to execution. However, static compilers lack runtime information that is relevant for making power-management decisions, information such as what paths in a program's control-flow graph are actually being executed. Hence, statically optimizing compilers often rely on laborious simulations of a program collected *prior to execution*, to determine what optimizations should be applied in different program regions. A program's actual runtime behavior may diverge from its behavior in these "simulation runs", thus leading to suboptimal decisions. Compiler optimizations also tend to treat programs as if they exist in a vacuum. For example, compiler techniques for memory allocation typically assign the elements of an array contiguously in logical memory. Consider a large array that has been assigned in this manner across several pages (Figure 2(a)). Physical memory may not have sufficient room for storing these pages in contiguous locations, since the pages of other applications may already be occupying these locations. Hence, the operating system may have to assign these pages noncontiguous memory locations (b). As a result, the array elements would be dispersed across these locations. If the array is traversed from beginning to end, poor cache utilization would result, since every two array element accesses induces a random I/O rather than a sequential I/O.

Dynamically optimizing virtual machines like ours offer a number of advantages over operating systems and static compilers. Unlike pure interpreters, these virtual machines can compile programs, monitor their execution behavior, and reoptimize them as execution patterns and resource levels change. The input

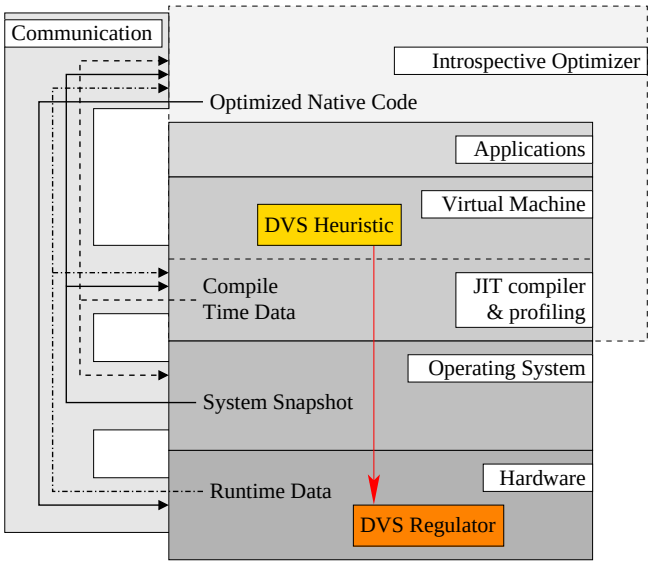

Fig. 3. Schematic overview of our system architecture, with the virtual machine layer and the JIT compiler running on top of the operating system. The system is introspective: the code of applications and the virtual machine is profiled and reoptimized continuously.

programs may consist of source code, binary code, or platform independent code such as Java or .Net bytecode. Like compilers, virtual machines that compile source code or bytecode know about a program's internal structure. But they also have recent runtime information that can more accurately guide dynamic optimizations than data gathered through offline simulations. In addition, they have high-level information, such as the library calls made in different program regions. Using this information, they can predict the resources that will be used and power down idle resources.

Because of these advantages, virtual machines are widely used in real systems, and can thus be readily exploited for power management. For example, the very popular Transmeta Crusoe processor [39,40] leverages the virtual machine concept for power management. Its code morphing software monitors applications and adjusts the clock frequency to match their performance requirements. But unlike the system we are developing, Transmeta aims to emulate x86 functionality on a VLIW platform. Its operating system runs above the code morphing software, and does not directly communicate with the underlying physical platform. Other examples of devices that come equipped with virtual machines are PDAs, pagers, and set-top boxes.

However, even virtual machines do not have a complete picture of events occurring at runtime. Even though they have profiling data, such as the frequencies

with which different program paths are taken, they do not precisely know what other layers of software may be doing at any given time. For example, context switches may occur sporadically as in the case of interrupts, or be scheduled ahead of time by the operating system or by users. In the middle of execution, a process might call a kernel routine or request memory, thereby spawning a series of events between the time execution leaves the process to the time it enters it again. All these events affect execution behavior and the dynamic optimizations that are likely to be effective. However, information about these events is spread out across multiple layers of software and hardware. Our goal is to integrate this information, so that dynamic optimizers can make better decisions.

3 Holistic Approach

For these reasons, we depart from the traditional notion of operating systems, compilers, virtual machines, applications and hardware as entities that work orthogonally, oblivious to the activities of one other. In our system these entities collaborate on functions normally assigned to only one of them. As a result, traditional compiler phases and optimizations, including runtime optimizations in an adaptive virtual machine, become *system-aware*, while traditional operating system functions (e.g., scheduling, memory management) become sensitive to internal program structure, dynamic data structures and runtime behavior. Surprisingly, this solution does not require adding extra levels to the software layers present in existing systems, since a growing number of devices already have virtual machines.

3.1 Compilation And Execution

The foundation of our system is a dynamically optimizing virtual machine that executes directly on the OS and manages the execution of all other programs running on a device. This virtual machine compiles code at *runtime* into the native instructions of the target. As programs execute, the VM collects information about their execution behavior and reoptimizes them on the fly as execution patterns change.

Our system preserves and integrates essential information throughout the compilation and execution process. The most fundamental information that needs to be preserved is information about what code is executing at any time and what its function is. For example, the system needs to differentiate between code belonging to different components of an application. It also needs to preserve more fine-grained information about a program's internal structure, e.g. loops. We use annotations in the meta data to preserve this information.

All applications are initially compiled from source code or platform independent bytecode into native code before execution. At compile time, our virtual machine inserts annotations into a program's internal control-flow representation to preserve high-level information such as what software module is being compiled, to which application it belongs, and what libraries are being used in

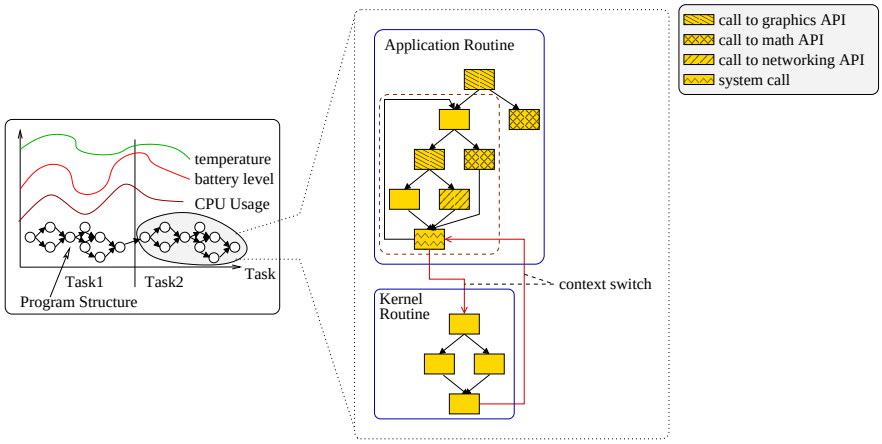

Fig. 4. Different levels of information used in the dynamic voltage scaling (DVS) heuristic. Each task is represented by a control-flow graph that represents application code as well as intentional context switches. The graph includes information about calls to API and kernel routines to enable fine grained power management decisions.

different code regions. It inserts extra annotations to more easily identify program constructs such as loops and branches. This information can provide insight into what resources are being used in different program regions. Regions using math libraries may be computationally intensive and stress the CPU, while regions using I/O and networking libraries may stress peripheral devices. Regions using graphical libraries may make heavy use of the GUI controller. By monitoring the control flow through different regions, one can determine the optimal power modes for different resources. For example, during the execution of I/O intensive regions, the processor is slowed down if it is not doing any useful work. When control flows out of a region using the network interface, the network interface can power down. Similarly, when it flows out of a region using graphical libraries, the GUI controller can power down. To make it easier to exploit this information, the dynamic optimizer can cluster successive accesses to the same libraries into contiguous program regions.

In the control-flow graph shown at the top of Figure 4, calls that have been annotated are shaded according to the API they call. For example, different regions have been labeled as *graphics*, *math*, or *networking* based on the libraries they use. The loop inside the application method periodically calls a kernel method using a system call. This high-level information will be useful when making power-management decisions. For example, if the operating system knows that control is about to enter library code in an I/O bound graphics API, it can apply a power-management heuristic optimized for this API.

When programs execute, a profiler monitors their runtime behavior and collects statistical information such as path execution frequencies and loop iterations. It also records hardware information, such as temperature and process variations and resource levels. The decision of what code to profile is based on requests from other software components, past profiling data indicating "hotspots," as well as the profiling overhead. This run-time information is integrated with program information (provided by the runtime compiler) and system information (provided by the operating system). These three levels of information drive heuristics for deciding how to apply power-management techniques (e.g., dynamic voltage scaling) and when to reoptimize executing code.

A unique feature of this architecture is that it is *introspective*: all executing programs, including the VM code, are continuously profiled and reoptimized within the contexts in which they execute. These optimizations, including power-management decisions, extend past component boundaries. The compiler infrastructure is connected to the runtime system and application layer via a communication channel. Through these channels, all layers exchange messages: applications send hints to the runtime compiler and operating system, the operating system itself sends "system snapshots" to the dynamic optimizer to broaden the optimizer's view past individual programs. In turn, the compiler and dynamic optimizer send the operating system fine-grained program information that is useful for power-management decisions as well as for standard operating system functions such as memory management and CPU scheduling.

3.2 Example: VM-driven CPU Scheduling

We illustrate the use of this architecture by explaining how VM/OS interaction during CPU scheduling can result in better power-management decisions. As we have noted, a limitation of compilers is that the standard control-flow representation does not capture interprocess control flow arising from context switches. To make the compiler aware of these switches, we need an efficient way of representing them in a program's control-flow graph. However, the exact program point where a context switch will occur is unknown because some switches (arising from external interrupts) are unpredictable while others (arising from CPU scheduling decisions) are scheduled by the operating system without any information about internal program structure.

We can improve this situation by making CPU scheduling more deterministic with virtual-machine support. Results from prior operating-systems research [35] suggest that context switches are least costly when they occur at program points where a program's working set is small, and thus very little context needs to be saved. Hence, it makes sense for a virtual machine to profile the program's working set at runtime. When the profiling data stabilizes, it can insert *context switch annotations* (c.f. Figure 4) to indicate where in its execution a program can be interrupted. These context switch points can be varied when the program's execution patterns change noticeably.

The operating system, on the other hand, can provide to the virtual machine a "system snapshot" containing information about what tasks will preempt

the task currently being compiled, and the order in which these tasks will preempt each other. Using this information, the VM can construct an *inter-program control-flow graph* with special edges indicating context switches between different tasks. This will allow more informed power-management decisions. If the VM knows, ahead of time, that a task will be interrupted, and that execution will continue with a different task that can be slowed down considerably (to save CPU power), it can inform the operating system about this ahead of time. The operating system can then slow down the processor before the context switch actually occurs.

4 Related Work In Collaborative Power Management

Recently, researchers have begun to attack the problem of collaborative power management, but up to now no definite breakthroughs have been achieved. Shin et al. [34] introduce an approach called *cooperative voltage scaling*. Under this approach, applications are divided into slices and code is inserted at the start of each slice to set the CPU speed. The speed chosen depends on the difference between the slice's worst case runtime and the remaining time to deadline. The worst case time is determined statically and the time to deadline is determined through a system call that subtracts the current time from the deadline. Thus, when a slice is completed before its worst case time, the processor can slow down and still meet the deadline. Two issues that that need to be addressed are how to divide applications into timeslices and how to prevent frequency thrashing. If the CPU speed is adjusted at every timeslice, it may be adjusted too frequently, causing power and performance overheads. Moreover, there are many other kinds of information that a DVS algorithm could consider, information such as program structure, runtime events and resource levels.

Heath et al. [20] propose application transformations for reducing disk drive energy. These transformations increase idle periods where the disk can be spun-down by clustering disk accesses. Using system calls, the application asks the operating system how much memory is available, buffers disk accesses to fill up the available memory, and informs the operating system the expected runtime of the clustered accesses. The OS serves the requests in a single batch and powers down the disk. However, expected runtimes are not easily predictable when multiple tasks are executing. The disk access patterns of different tasks can collide, causing frequent power-mode transitions. Realizing this, Hom and Kremer [21] extend the previous work to explore the benefits of compiling multiple programs at once and synchronizing them at compile time. They propose an optimization, *the inverse barrier*, that regulates access to shared resources. When a task is ready to access a resource such as a disk drive it informs the other tasks to access the resource so that the collective requests can be clustered. The approach is promising but can be developed further. Compiling sets of programs is more expensive than compiling individual programs. The question of how compilers can perform analyses on multiple programs efficiently and dynamically reoptimize multiple programs in response to runtime events needs to be addressed

Aboughazaleh et al. [1,2,3] attempt to communicate path specific information about a program's progress to the operating system. Namely, the compiler instruments each application with code that saves the remaining worst case cycles into a register. The operating system periodically reads this register and adjusts the processor speed to fill up slack. While this approach has a desirable long term goal, it could be developed further. Since the operating system's decisions consider only the worst case cycles of a *single* program, they may be suboptimal in the presence of context switches: The worst case cycles estimate does not include the cycles for context switching to other programs and back. Also, the approach could provide the OS more information. For example, it could send snapshots of entire program regions to the OS on an as-needed basis. These snapshots could include control-flow information, runtime profiling data and the most recent DVS decisions. The OS could use these snapshots to make DVS decisions across multiple programs.

Pereira et al. [28] propose a software architecture allowing applications to communicate with the operating system, and the operating system to communicate with hardware. It allows programmers to make system calls to inform the OS about the start times, end times and expected duration of tasks. An advantage of this approach is that it allows programmers to provide the operating system information that might be relevant for power management. However, this architecture could be extended to include a software runtime layer that profiles and reoptimizes programs, and integrates fine-grained information about program structure into the power-management decisions.

5 Status and Future Work

We have developed a set of heuristics [19] for doing virtual machine based power management at runtime. We have also developed an ahead-of-time SSA-based optimizing compiler [10,31,42] that embeds virtual machine functionality into the generated code.

Currently, we are migrating this solution to an architecture based on Linux and the Jikes virtual machine [4], and incorporating our existing compiler infrastructure into the introspective optimizer. We are also developing heuristics for efficiently exchanging information between different levels and for applying optimizations based on this information, and are investigating what their tradeoffs are. This architecture will allow us to manage the power consumption of resource constrained devices, taking into account all of the information needed to manage power effectively. Being virtual machine based, this framework allows portability and can be used as a testbed for new power management heuristics.

However, the impacts of our framework extend beyond power management alone. We envision future consumer devices containing dynamic compilers that intimately interact with the operating system, and where all of the traditional compiler functions (e.g., register allocation, instruction scheduling) extend beyond the boundaries of individual programs and integrate system-level information [18]. Recent research suggests that this direction is promising. For example,

register allocation is normally done for individual programs in isolation from other programs executing in the system. However, programs that do not preempt each other can use the same registers. If a compiler is aware of preemption patterns (with help from the operating system) it can minimize the total number of registers used across different tasks. As Barthelmann [8] shows, a technique like this saves memory in embedded systems. Other inter-program optimizations that have recently been shown to reduce energy include transformations at the process level [16], transformations that, for example, merge processes to reduce interprocess communication, split processes to increase concurrency, or migrate computations from one process to another.

6 Conclusion

Continued progress in system design requires cost-effective techniques for managing power. However, power management has been a highly compartmentalized discipline: techniques are being developed along one or more dimensions without fully exploring their synergy and tradeoffs along other dimensions. In particular, software techniques for power reduction are being applied at just a single one of the operating system and compiler levels. Each of these levels lacks relevant information that the others have, and thus alone offers only part of the picture needed to manage power effectively.

Our system manages power at runtime, integrating all of the intelligence offered by different layers of system software, applications and hardware. The system allows these layers to communicate efficiently, and incorporates heuristics for driving power-management decisions based on all of this information. All of the executing software, including the code of the virtual machine, is monitored continuously and reoptimized with respect to profiling data. Thus, optimizations extend beyond program boundaries and consider the system at large.

With systems becoming increasingly complex, this holistic approach to power management is likely to be the most cost effective way of shattering the limits imposed by growing power densities and enabling continued progress in system design.

References

1. N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy Management for Real-Time Embedded Applications with Compiler Support. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 284–293. ACM Press, 2003.
2. N. AbouGhazaleh, D. Mosse, B. Childers, and R. Melhem. Toward the Placement of Power Management Points in Real Time Applications. In *Workshop on Compilers and Operating Systems for Low Power*. ACM Press, 2001.
3. N. AbouGhazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven. Collaborative Operating System and Compiler Power Management for Real-Time Applications. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–143. IEEE Computer Society Press, 2003.

4. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V.Litvinov, M. F. Mergen, T.Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. In *IBM System Journal*, 2000.
5. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeno JVM: The Controller's Analytical Model. In *The 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.
6. A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-Aware Just-In-Time Complilation System. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 142–151. ACM Press, 1999.
7. V. Bala, E. Duesterwald, and S. Banerjia. *Transparent Dynamic Optimization: The Design and Implementation of Dynamo.* Technical Report HPL-1999-78, Hewlett Packard Laboratories, June 1999.
8. V. Barthelmann. Inter-Task Register-Allocation for Static Operating Systems. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 149–154. ACM Press, 2002.
9. B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283. ACM Press, 1995.
10. D. Chandra, C. Fensch, W.-K. Hong, L. Wang, E. Yardimci, and M. Franz. Code Generation at the Proxy: An Infrastructure-Based Approach to Ubiquitous Mobile Code. In *Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems*, 2002.
11. G. Chen, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Adaptive Garbage Collection for Battery-Operated Environments. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 1–12, San Francisco, CA, Aug. 2002.
12. M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, October 2000.
13. V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 138–147. ACM Press, 2000.
14. F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spindown Policies for Mobile Computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, April 1995.
15. D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266. ACM Press, 1995.
16. Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-Optimizing Source Code Transformations For OS-Driven Embedded Software. In *Proceedings of the IEEE International Conference On VLSI Design*, 2003.
17. K. Flautner and T. Mudge. Introspective computers. In *ASPLOS 1998 Wild and Crazy Ideas Session*, October 1998.
18. M. Franz. Run-Time Code Generation As A Central System Service. In *Proceedings of the 6th Workshop On Hot Topics In Operating Systems*, 1997.

19. V. Haldar, C. W. Probst, V. Venkatachalam, and M. Franz. Virtual Machine Driven Dynamic Voltage Scaling. Technical Report 03-21, University of California, Irvine, School of Information and Computer Science, October 2003.
20. T. Heath, E. Pinheiro, and R. Bianchini. Application-Supported Device Management for Energy and Performance. In *Proceedings of the Workshop on Power-Aware Computer Systems PACS'02*, Feb. 2002.
21. J. Hom and U. Kremer. Inter-program Compilation For Disk Energy Reduction. In *Proceedings of the Workshop on Power-Aware Computer Systems*, 2003.
22. M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of Compiler Optimizations on System Power. In *Proceedings of the 37th Conference on Design Automation*, pages 304–307. ACM Press, 2000.
23. T. Kistler and M. Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.
24. C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization*, Mar. 2003.
25. C. Krintz and B. Calder. Using annotation to reduce dynamic optimization time. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 156–167, N.Y., June 20–22 2001. ACMPress.
26. D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the second USENIX Symposium on Operating Systems Design and Implementation*, pages 153–167. ACM Press, 1996.
27. D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 73–84. ACM Press, 1996.
28. C. Pereira, R. Gupta, and M. Srivastava. PASA: A Software Architecture for building Power aware Embedded Systems. In *Proceedings of the IEEE CAS Workshop on Wireless Communications and Networking - Power Efficient Wireless Ad Hoc Networks*, September 2002.
29. F. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In *Proceedings of the 32nd Annual Symposium on ACM/IEEE International Symposium on Microarchitecture*, 1999.
30. P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, page 8. IBM Press, 2000.
31. C. W. Probst, A. Gal, and M. Franz. Code Generating Routers: A Network-Centric Approach to Mobile Code. In *Proceedings of the 2003 IEEE 18th Annual Workshop on Computer Communications*, 2003.
32. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 314–321. ACM Press, 1995.
33. W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, and I. Shavit-Lottem. Profile-Directed Restructuring Of Operating System Code. *IBM Systems Journal*, 37(2), 1998.
34. Y. Shin, H. Kawaguchi, and T. Sakurai. Cooperative Voltage Scaling (CVS) Between OS And Applications For Low-Power Real-Time Systems. In *Proceedings of the CICC'01*, 2001.

35. J. Simonson and J. Patel. Use Of Preferred Preemption Points in Cache-Based Real-Time Systems. In *International Computer Performance And Dependability Symposium*, 1995.
36. A. Tamches. *Fine-Grained Dynamic Instrumentation Of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin, Madison, 2001.
37. A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the third Symposium on Operating Systems Design and Implementation*, pages 117–130. USENIX Association, 1999.
38. J. Torrellas, C. Xia, and R. Daigle. Optimizing the Instruction Cache Performance of the Operating System. *IEEE Transactions On Computers*, 47(12), 1998.
39. Transmeta Corporation. *LongRun Power Management: Dynamic Power Management for Crusoe Processors*, 2001.
40. Transmeta Corporation. *Crusoe Processor Product Brief: Model tm5800*, 2003.
41. P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic Compilation for Energy Adaptation. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 158–163. ACM Press, 2002.
42. V. Venkatachalam, L. Wang, A. Gal, C. W. Probst, and M. Franz. ProxyVM: A Network-Based Compilation Infrastructure for Resource-Constrained Devices. Technical Report 03-13, University of California, Irvine, School of Information and Computer Science, March 2003.