Using Phase Behavior in Scientific Application to Guide Linux Operating System Customization [*]

Chandra Krintz Rich Wolski
Computer Science Department
University of California, Santa Barbara
{ckrintz,wolski}@cs.ucsb.edu

Abstract

In this paper, we present the design of a system that automatically generates application-specific Linux images for scientific applications that execute using batched cluster resources. Key to our approach is the use of recurring patterns in program performance, i.e., phase-behavior, that can be exploited potentially to guide automatic Linux customization and to enable significantly higher levels in program performance. We overview project and present a set of preliminary results that show the potential of our approach.

1 Introduction

Recent advances in high-performance processor and network technologies are making clusters of workstation-class computers cost-effective platforms that can support the next generation of scientific applications. Low per-unit cost, advances in computing and communication power, and the availability of Linux as a free, easy-to-use, and nearly standard operating system, make high-end computing with these systems accessible both to a very large developer base and to a wide range of users. As part of this evolution, Linux has emerged as a nearly ubiquitous, open-source operating system with a wide-range of readily available programming support tools and specialized libraries. It is currently the system-of-choice in academic and production scientific computing settings and as a result, many, if not the majority of, scientific programmers being trained today are familiar with Linux as a development platform.

Moreover, Linux runs both on individual workstations and in clustered commodity systems, e.g., Beowulf [4] systems. Thus, scientific programmers can use a local (possibly networked) workstation environment to develop, debug, and tune their programs and then transfer them to a production environment with little or no porting effort. The ability to use the same operating system in both a locally-controlled, highly responsive environment for development and a batch-controlled production environment for repeated large-scale execution greatly increases programmer productivity and lowers the "overhead" associated with the use of high-end computing to advance science.

A key limitation to the use of Linux for high-end cluster computing however, is its potential performance impact on application execution. Linux, like other general-purpose operating systems (OSs) with commercial application, continues to evolve to support an enormous range of user requirements and preferences, application domains, and devices (everything from supercomputers to hand-helds). In particular, its popularity as a web-server hosting environment has placed even greater demands on its ability to support quick response times for many competing small, relatively short-lived, and potentially difficult to predict computations (e.g. web transactions). In contrast, scientific applications executing in clustered settings are frequently large, resource intensive, long-running, and use space-sharing to gain exclusive access to the machines they use through a batch system. They do not compete dynamically for processor and I/O resources since the batch system ensures that any processors allocated to an application are not time-shared by other applications.

As a consequence of this tension between application requirements, the Linux OS includes many features and built-in policies that do not promote the performance of high-end scientific applications. In particular, scientific applications typically do not require the extensive support for fair resource sharing (since they execute in production space-shared, and not time-shared, environments) or quick response time (since they may not be interactive) that Linux includes. Much of the functionality built into Linux is included to support applications with radically different performance needs than most scientific codes, and these features can retard the performance of scien-

[*]This work was sponsored, in part, by a grant from the National Science Foundation numbered ST-HEC-0444412.

tific programs in high-end computing settings. None the less, the portability that Linux affords combined with the familiarity that its wide-spread popularity has bred make it a de facto standard operating system for clustered architectures.

The goal of our research is to investigate techniques that maintain the ease-of-use and cost benefits of Linux while enhancing the performance achievable by high-end scientific applications executing in large-scale cluster computing settings. This scenario has been explicitly identified as vital to the future of High-End computing at both the *2003 Workshop on The Road Map for the Revitalization of High-End Computing* [10] and more recently in the *2004 Federal Plan for High-End Computing* [30]. To enable this, we will study ways to automatically customize the Linux instance an application uses when it is running in a "production" (i.e., non-development or debugging) setting based on the specific needs of the application itself. We will also exploit the exclusive processor access that batch scheduling implements to relax or eliminate unneeded mechanisms that are designed to facilitate effective time-sharing, but which introduce unnecessary overhead in a space-sharing context.

We plan to explore both runtime and compile-time approaches to customizing the Linux instance used by each application. Each of which will allow the scientific programmer to use unmodified Linux as a local development and debugging environment and then to apply our techniques as an additional compilation step before initiating high-end cluster execution.

In the runtime approach, we will fit the installed Linux with a *kernel adaptation call* that exports a generalized system call for the application to use to "tune" the kernel it is using. To prevent the programmer from assuming the burden of becoming a kernel tuning expert, we will also develop the compiler analysis and code-generation technology necessary to automatically insert kernel adaptation calls into an application program to enhance performance. Thus, the compiler will direct runtime customization of the kernel by modifying the application on behalf of the programmer.

Our approach relies on two key observations about high-end scientific applications. First, scientific applications commonly exhibit regular and course-grained execution "phases". We will develop static and dynamic off-line profiling techniques to identify key patterns in resource use that we will use to guide customization. Secondly, application execution instances do not require extensive time-sharing support when run in production cluster settings (e.g., TeraGrid). Since the OS instance is not shared between competing applications, each receiving a small time slice from the processors, it can be customized to the needs of each individual application. We are currently investigating compiler and runtime techniques that exploit these observations through (1) specialization of critical paths through the OS, and (2) dynamic component-wise performance adaptation of both the OS and the application.

Much prior work in the area of application-specific operating systems (OSs) has thoroughly studied extensibility, specialization, and minimization of the OS in general [12, 36, 3, 5, 38]. In addition, many of the techniques for dynamic adaptation come from our extensive experience with dynamic and adaptive optimization for Internet computing [22, 23, 29, 45, 43]. However, our system is novel in that it combines and extends these efforts into a system that automatically customizes the Linux operating system for a *single application run* and is specifically focused on the application domain of scientific computing using high-performance clusters. At the same time, our use of Linux ensures that the environment for scientific applications developers remains familiar and unified across the development and production platforms they use thereby promoting ease-of-use, programmer productivity, and efficient program management.

In the sections that follow, we describe our system in which we specialize the Linux kernel for a specific instance of a scientific-computing application for execution using batched cluster resources. We then overview our phase-detection system and present preliminary evidence of phase behavior in scientific computing programs (Section 3). In Section 4, we present preliminary results that indicate the potential of phase-based Linux specialization. We then present related work in Section 5 and conclude in Section 6.

2 Application-Specific Linux

Currently, batched, distributed-memory, cluster-based systems for high-end computing, e.g., TeraGrid clusters [32], Red Storm [37], Tungsten [47], Lightning [24], Lone Star [25], and others, operate as follows. The system administrator installs and maintains an identical version and configuration of Linux on all of the machines in the cluster. The administrator also installs a batch system for job execution that implements space-sharing of cluster processors. User jobs can only gain access to the processors of the cluster via batch job submission. Most clusters do provide a small number (often only 1) of interactive "head" nodes that can be used for program compilation and job submission. Long-term execution on the head node or nodes, however, is typically prohibited.

A job is submitted to the batch system typically specifies the resources required (number of machines), the priority of the job (in the form of a queue name or identifier) and any necessary accounting information. When the job is selected for execution, the required number of proces-

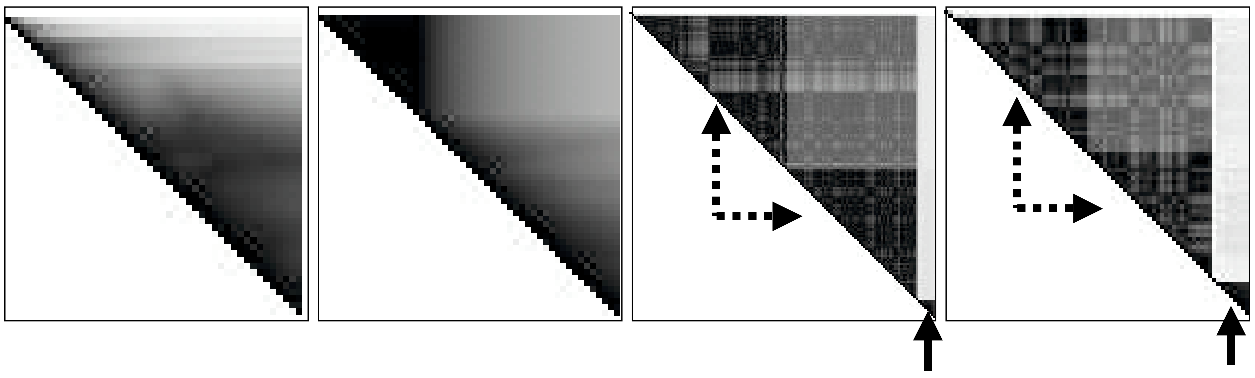

Figure 1: Phase behavior in scientific computing codes. We show the upper triangle of the similarity matrix (black = similar, white = dissimilar) for each program. Each point represents the similarity (Manhattan distance) between two snapshots of program execution. The behavior in these programs varies little (large dark sections) over the lifetime of the programs. The last two matrices are the same program executing different inputs; the phase behavior is very similar across inputs. The dotted arrows point to computational phases; the solid arrows to I/O bound phases.

sors is allocated exclusively to the job from a pool of available processors. No other job is assigned to those processors until the job voluntarily terminates, or the batch-queuing software forcibly terminates the job because it has exceeded a pre-determined time limit.

We exploit this batch model in an effort to extract high-performance from scientific applications. That is, since only a single application executes at once, we can customize the Linux kernel to remove (unload) support for any service not used by the application of interest. Moreover, we can specialize the kernel according to the usage patterns of the application. We currently rebuild the kernel for each application. However, we are also investigating the use of a virtual machine monitor (VMM) for the batched machines. e.g., a minimal Linux kernel that runs a specialized user-mode Linux [49] kernel or the Adeos nanokernel [21] that runs a specialized actual Linux kernel. Using a VMM, the batch system can install the specialized kernel (submitted with the application) upon installing the application itself.

The specialized Linux kernel employs an extended interface that implements a *kernel adaptation call*. The kernel adaptation call exposes kernel internals to the application. Our system implements and controls customization of the application and the OS using a modified gcc compiler which inserts kernel adaptation calls into the application automatically. The kernel adaptation call is a software device interface that does not support access to an actual hardware component, i.e., it is a "pseudo-device". The Linux device driver implementation is well-understood and relatively standardized making it an attractive method for adding a system call to arbitrary versions of Linux. To perform effective Linux customization, we must collect dynamic program behavior that identifies specialization opportunities. To enable this, we collect program phase behavior.

3 Phase Behavior in Scientific Applications

Research by ourselves and others [29, 39, 40, 14] has shown that general-purpose and Internet computing programs commonly do not behave randomly but instead, execute as a series of repeated behaviors termed *phases*. During a particular phase, the behavior of the program is relatively stable for some amount of time, after which the behavior may change significantly. Furthermore, these same phases may then re-occur at some point later in time. This phase behavior is architecture-independent but accurately reflects how the program consumes the available hardware resources [39]. Although Internet programs typically exhibit a large number of distinct phases, identification of phase behavior has the potential for significantly reducing the complexity of analysis and the efficacy of profile-guided program optimization.

Anecdotally, it has long been maintained by programmers in the scientific computing community, however, that scientific programs contain large-scale, course-grained program phases. To confirm this postulation, we applied a phase profiling and analysis tool [29] that we developed for Internet applications to a set scientific benchmark programs.

A visualization of the phase data for the scientific codes is shown in Figure 1. For each program, we show a *similarity matrix* which is an easily-rendered, gray-scale representation of the phase behavior in a program. The tool partitions the execution of each program into a set of

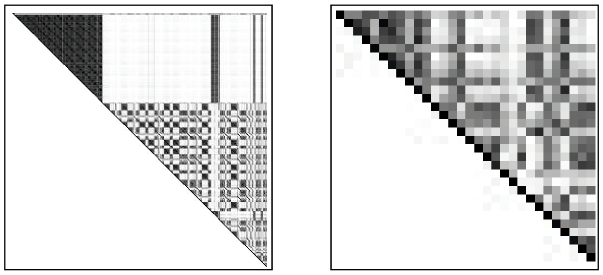

Figure 2: Phase behavior in a general-purpose (non-scientific) application using two different inputs.

user-specified *intervals* (100 billion instructions each in this experiment). Each interval contains the basic block frequencies weighted by the number of instructions in the basic block normalized by the length of the interval. The tool then computes the Manhattan distance [34] between each and renders it into the gray-scale range (0-64K) for display. In the visualization, black indicates two intervals are completely similar (the Manhattan distance is small) and white shows that that they are dissimilar. The x and y axes order interval identifiers chronologically from the first interval executed by the program to the last. Therefore, by moving from left to right across a row, we can visualize the similarity between the interval (represented by the row) and every interval that follows it in the program's execution. We omit the symmetric lower triangle to clarify the visualization.

The programs in Figure 1 include (a) A benchmark for dense linear algebra computations [18] that performs factorizations (LU, Cholesky, QR, SVD) and symmetric/non-symmetric eigenvalue operations; (b) A collection of low-level kernels, including Euler, Monte Carlo, search, molecular dynamics, and ray tracing algorithms [19] (each described in detail here [20]); and (c) PatternHunter [26], a state-of-the-art, large-scale application for fast DNA homology searches from Bioinformatics Solutions [6]. Subfigure (d) shows the phase behavior for PatternHunter using a different input (mouse DNA as opposed to fruitfly DNA used in (c)).

The figure illustrates two important conjectures regarding scientific applications that we can exploit via optimization and specialization. The first is that scientific programs contain a small number of large phases. In each subfigure, the large contiguous dark regions support this conjecture in the three applications depicted. Secondly, we conjecture that the phase behavior of scientific programs does not vary dramatically based on program input making profiling techniques especially attractive. Subfigures (c) and (d) support this supposition. Moreover, the compute and I/O phases of PatternHunter are readily visible. The dotted arrows indicate compute intensive phases of execution and the solid arrows point to I/O phases.

Compare these results to the similarity matrices shown in Figure 2 showing the phase behavior for a commonly-used Internet application benchmark (a Java compiler) [44] on two different program inputs. In subfigure (a) the phase behavior varies widely and there is only one extended period of repeated behavior (during which time the program is performing I/O). Moreover, subfigures (a) and (b) differ dramatically indicating that the program has a highly input-dependent execution profile. Thus, off-line profiling is likely to be significantly more effective for the scientific programs depicted in Figure 1 than the Internet benchmark shown in Figure 2.

A key benefit to this approach (using phase behavior) is that a single interval (and its statistics) represents the entire phase (all other intervals to which it is similar). As such, we need only analyze and specialize for the “hot” intervals, those that occur in the largest, most common phases. Moreover, we can use information about infrequently executed intervals (and their interaction with hot intervals) to identify opportunities for optimizations that allow us to avoid memory hierarchy pollution. Using phase behavior for scientific programs significantly simplifies our optimization framework since it inherently identifies code regions (and thus a small set of important optimization opportunities) on which our system should focus.

4 Linux Specialization Potential

To implement automatic run-time tuning of the kernel, we added *kernel adaptation call* to the Linux implementation that is installed on the high-end system (either by the system administrator or as part of an on-demand installation supported by a VMM). The compilation system, guided by phase behavior and analysis, inserts kernel adaptation calls into the application instance before it is launched on the high-end resource. For example, for multi-threaded and computational steering applications, the compiler inserts this system call, into the code at appropriate points to adjust the quantum automatically so that the application achieves the best performance possible. Notice that with respect to the application programmer, the code remains unchanged.

Using our phase profile information, we identify system calls that are frequently executed and use the compiler to specialize these calls according to how they are used by the program. In particular, in addition to scheduling opportunities, we are interested in specializing disk and socket I/O calls. For disk I/O, we can specialize file manipulation routines for a particular inode in use (bypassing much of the general-purpose code along the critical path) and customize file access (sequential or using access patterns indicated by the profile) to improve performance. We use similar specialization techniques to reduce the overhead of socket communication. We use our compiler to insert kernel adaptation calls that employ the specializations into the application when it is built for the production system.

In the following subsections, we present preliminary evidence of the efficacy of using Linux specialization techniques for improving the performance of scientific-computing applications. For these studies, we employed User-Mode Linux (UML) [49]. UML is one option for enabling efficient installation of a customized Linux kernel on batched, cluster resources, as mentioned previously. In this scenario, the cluster machines implement a minimal Linux kernel that is specialized for UML execution. When a user submits a job to the cluster, she submits the customized UML image with the application (both generated automatically by our specialization toolkit).

4.1 Example: Customized Time Quanta

To evaluate the potential of application-specific Linux on scientific- program performance, we evaluated the efficacy of specializing the Linux time quantum in the operating system scheduler. We found that by extending the CPU time quantum beyond what the Linux `nice` interface does, we can dramatically improve compute intensive program performance. We modified a version UML to support a 5-second CPU occupancy quantum and compare the observed performance (in terms of execution time and context switches) to default `nice 0` execution of simple Fibonacci and LINPACK [17] LU decomposition. (without ATLAS [50] optimization).

	Default	
Benchmark	Total Time (s)	Context Switches
Fibonacci	85.52	35562
LINPACK LU	87.31	60844

	5s Quantum	
	Total Time (s)	Context Switches
Fibonacci	60.69	11.8
LINPACK LU	72.23	29.5

Table 1: Performance impact of extending the quantum for a computationally intensive process.

The results from these experiments, averaged over the last 10 of 11 executions (we omit the first to elide any paging effects), are shown in Table 1. We ran each of these experiments on a quiescent machine under the identical conditions. Columns two and three show the execution time (in seconds) and number of context switches, respectively, for the default quantum setting. Columns four and five show the same statistics when we give the single-threaded process a quantum of 5 seconds. The data indicates that by extending the quantum for a computationally bound process, we can eliminate most of the context switches and thus significantly improve performance, by 29% for the toy Fibonacci program and 17% for LINPACK LU.

Quanta specialization, however, is undesirable in a time-shared setting where CPU-bound applications can lock out response-time sensitive applications (editors, email clients, etc.) for long periods. Moreover, long quanta may not be compatible with all configured devices, particularly those with tight real-time requirements (such as streaming media). Indeed, to affect this experiment, we had explicitly to disable sanity checks and assertions in the Linux kernel designed to prevent a kernel-programming bug or overly aggressive device driver from modifying the maximum allowable quantum. However, this simple experiment illustrates the degree to which even the basic mechanisms included in Linux to support time-sharing (which could be substantially modified in a batch-controlled setting) can retard scientific application performance. We also understand that these results are pertinent to UML only. However, it may be possible to build a Linux-specific VMM for the UML (as described above). As such, we are investigating the performance characteristics of UML initially.

Simply tuning the resident version of Linux for long-running, in-core execution may reduce the performance of other more complex scientific applications. In particular, multi-threaded codes that use threads to overlap communication with execution (a common parallel programming optimization) or codes that stream data while they execute to an external visualization system (e.g., for the purposes of computational steering [8, 9]) extending the quantum may degrade performance by starving the threads assigned to I/O. Under Linux, context-switchable threads (such as POSIX threads [33]) use the same CPU scheduling algorithm as competing, separate processes. Thus, our customizations must be tailored to the characteristics of the application rather than implemented as a blanket set of shared tuning enhancements. More specifically, our techniques must be adaptive given the dynamic behavior of the program. The program phase information provides us with such information. Our compiler uses this behavior to place kernel adaptation calls so that the quanta is set appropriately for compute-intensive and non-compute-intensive (e.g. interactive or I/O intensive) execution intervals.

4.2 Example: Customized Socket Sends

We also investigated the efficacy of customizing socket I/O within a specialized version of Linux. Scientific pro-

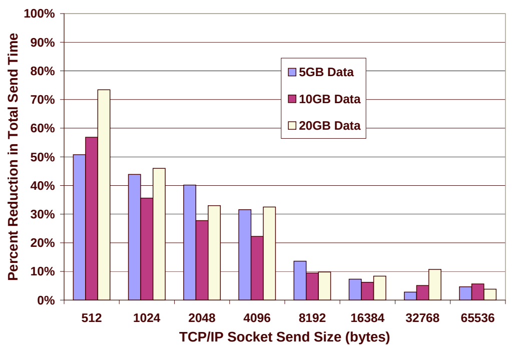

Figure 3: Percent improvement in socket send time we specialize send calls for a particular send size. The performance improvements result since we avoid crossing the kernel-user space boundary.

grams commonly perform communication between cluster nodes. Our goal with this experiment was to evaluate the efficacy of exploiting communication patterns. In particular, we generated customized TCP/IP socket send calls for a particular hand-coded program that performed a single socket send of parameterized by different send sizes.

The customized send calls (invoked by the application as kernel adaptation calls) exploit the communication patterns of the program to avoid crossing the kernel boundary. For each program instance, we employed a customized send routine that sent all of the data at once thereby avoiding repeated send calls. This experiment is a very simple and naive test; however, it lends insight into its potential.

Figure 3 shows the total time for send calls using different customized send calls (one for each amount of data sent). The x-axis is the size of each send call – the size dictates the number of system calls required for the user space version. The y-axis is the percent reduction in total send time for each of the three transfer sizes (5GB, 10GB, 20GB). The time for kernel send calls is relatively constant for each send size (13s, 25s, 50s). The data confirms other studies [5, 41] (as well as our study on application-specific quanta) that indicate that there is significant overhead imposed by crossing the user-kernel boundary. For small send sizes, the gains range from 30-70%; using a large send size, there are many fewer system calls, resulting in gains of 2-10%

5 Related Work

Our research builds upon an extensive body of research on extensible operating system kernels [12, 35, 15, 5] Extant approaches to such systems assume that extensibility is the common case. By enabling general and safe extensibility, such systems necessarily introduce additional runtime overhead. The bulk of the optimization effort then centers on minimizing this overhead [36, 3, 5, 38]. In contrast, our approach relies on the batch system to ensure safety by preventing competing applications from time-sharing processors.

Most of these systems also require applications to be written for the specific OS implementation at hand (requiring a new programming methodology) and to *select* the versions of services that will provide them the most performance benefit. An alternative approach shared by our work, is to implement an OS that *automatically* customizes the OS interface for the application [36, 3, 42, 28]. Specialization is performed using partial evaluation of system call parameters to reduce the length of critical paths through the kernel. Such systems systems focus on *specializing* existing OS code and automatically infer when specialized version should be employed. In addition, as different applications execute, specialized code is either selected using a template mechanism [11, 27, 31, 2, 16] or *dynamically replaced* with new versions [36, 42].

These latter systems, and in particular the techniques in [36] and [28], are most similar to those that we describe herein. The primary difference is that our goal is to implement tools for the automatic customization of a ubiquitous, popular, open-source OS (Linux); all prior work has either developed entirely new systems or have extended upon proprietary OSs (AIX [3], HP-UX [36], and Solaris [46]). In addition, we plan to investigate such OS specialization for scientific codes, an area that has been of focus very few, limited studies, e.g., [46, 1]. Moreover, studies that focus on scientific code performance have shown that other types of specialization, e.g., of library routine, have significant potential [48, 1, 7, 46]. Our work also differs in that we plan to consider the automation of application-specific OS customization, profile-guided compiler optimization of application code, and injection application tasks into the kernel in combination.

Since, our focus is batched cluster systems, our research is also unique in that, our customization can be specific for a *single application* since a new OS can be installed as part of the execution of a batch job. Consequently, the resulting system need not be concerned with safety and fairness issues associated with a multi-program environment. All prior work has focused on OS customization for wide range of applications and a large portion of the research has focused on trying to solve the safety and dynamic (re-)linking problems inherent in extensible OS design [13]. To enable our research, we plan to incorporate, combine, and extend this extensive body of prior work in the area of extensible operating systems as well as adaptive compiler and runtime optimization research from other application domains, e.g., Internet computing [29, 39, 40, 14], (as described in Section 3).

6 Conclusions

In this paper, we describe the design of our system for application-specific Linux customization for scientific programs. The goal of our work is to enable significantly higher application performance for scientific programs that execute using batched, cluster resources (a commonly used methodology for high-end applications). However, we intend to do so for the Linux operating system, a free, easy-to-use, popular, and nearly standard operating system. The key limitation of using Linux for high-end computing is that it is general-purpose and continues to evolve to support an enormous range of user requirements and preferences, application domains, and devices. By customizing Linux according to the needs of a *single* application (executing alone in a batched environment), we have the potential of significantly improving performance while maintaining the familiarity, ease-of-use, and cross-platform program portability that Linux offers.

To enable this, our system couples and extends extant compiler and runtime techniques that exploit specialization opportunities in both the application and Linux operating system. In particular, we employ program phase behavior to expose frequently executed behavior patterns in programs. Our compilation system uses phases to guide generation of customized Linux system call implementations that execute specific behavioral patterns very efficiently. We overview phase behavior and provide evidence and characteristics of phase behavior in scientific programs. Moreover, we present two studies that provide preliminary evidence into the potential of Linux customization: specialized scheduling quanta and socket send calls that exploit specific program behaviors.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the Performance of I/O-Intensive Parallel Applications. In *Workshop on I/O in Parallel and Distributed Systems*, 1996.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Schedular Activations: Effective Kernel Support for User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] A. Banerji and D. Cohn. An Infrastructure for Application-Specific Customization. In *ACM European SIGOPS Workshop*, September 1994.

[4] D. Becker, T. Sterling, D. Savarese, J. Dorband, U. Ranawak, and C. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *International Conference on Parallel Processing*, August 1995.

[5] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance of the SPIN Operating System. In *Symposium on Operating System Principles*, December 1995.

[6] Bioinformatics Solutions Inc. `http://www.bioinformaticssolutions.com/`.

[7] J. Bruck, D. Dolev, C. Ho, M. Rosu, and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing: Special issue on workstation clusters and network-based computing*, 40(1), 1997.

[8] Computational Steering Overview. `http://www.cwi.nl/projects/cse/anrep.html`.

[9] Computational Steering Scientific Applications. `http://discov.cs.kent.edu/resources/doc/steering/`.

[10] Computing Research Association Workshop on The Roadmap for the Revitalization of High-End Computing, D. A. Reed, Editor, June 2003. `http://www.cra.org/Activities/workshops/nitrd/`.

[11] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E. Volanschi, J. Lawall, and J. Noye. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3), 1998.

[12] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, 2002.

[13] P. Druschel, V. Pai, and W. Zwaenepoel. Extensible Kernels are Leading OS Research Astray. In *Hot OS*, May 1997.

[14] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architecture and Compilation Techniques*, September 2003.

[15] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating System Principles*, December 1995.

[16] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[17] Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`.

[18] Jama: A java matrix package. http://math.nist.gov/javanumerics/jama/.

[19] Java Grande Forum. http://www.javagrande.org/.

[20] Java Grande Forum section 3 benchmarks. http://www.epcc.ed.ac.uk/javagrande/threads/s3contents.html.

[21] K. Yaghmour, OpenSys Inc. Adaptive Domain Environment for Operating Systems. http://home.gna.org/adeos/.

[22] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)*, March 2003.

[23] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[24] Los Alamos National Laboratory Lightning Linux Cluster. http://www.acl.lanl.gov/source/orgs/ccs/ccs1/cluster/index.shtml.

[25] Texas Advanced Computing Center Lone Star Linux Cluster. http://www.tacc.utexas.edu/resources/hpcsystems/.

[26] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(3):440–445, 2002.

[27] R. Marlet, C. Counsel, and P. Boinot. Efficient Incremental Run-Time Specialization for Free. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[28] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, and T. Proebsting. Scout: A Communications-Oriented Operating System. In *Hot OS*, May 1995.

[29] P. Nagpurkar and C. Krintz. Visualization and Analysis of Phased Behavior in Java Programs. In *ACM Conference on the Principles and Practice of Programming in Java*, June 2004.

[30] NITRD Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force (HECRTF), May 2004. http://www.itrd.gov/pubs/.

[31] F. Noel, L. Hornof, C. Counsel, and Julia Lawall. Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study. In *International Conference on Computer Languages*, 1998.

[32] NSF TeraGrid Project. http://www.teragrid.org/.

[33] POSIX the Portable Operating System Interface. http://www.knosof.co.uk/posix.html.

[34] Predictive Patterns Software. Manhattan Distance Function Metric. http://www.predictivepatterns.com/docs/WebSiteDocs/Clustering/Clustering_Pa%rameters/Manhattan_Distance_Metric.htm.

[35] C. Pu, T. Autrey, A. Black, C. Counsel, C. Cowan, J. Inouya, L. Kethana, J. Wapole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating System Principles*, December 1995.

[36] C. Pu, H. Massalin, and J. Ioannidis. The Synthetics Kernel. *Computing Systems*, 3(1), 1990.

[37] Cray Inc., Red Storm Cluster. http://www.cray.com/media/2003/october/rsproduct.html.

[38] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guilemont, F. Herrman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[39] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages*, October 2002.

[40] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[41] C. Small and M. Seltzer. VINO: An integrated platform for operating system and database research. Technical Report Technical Report TR-30-94, Harvard Univ. Cambridge, MA, 1994.

[42] C. Small and M. Seltzer. Self-monitoring and Self-adapting Operating Systems. In *Workshop on Hot Topics in Operating Systems*, May 1997.

[43] S. Soman, C. Krintz, and D. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, October 2004.

[44] SpecJVM'98 Benchmarks. http://www.spec.org/osg/jvm98.

[45] S. Sucu and C. Krintz. ACE: A Resource-Aware Adaptive Compression Environment. In *International Conference on Information Technology: Coding and Computing (ITCC03)*, April 2003.

[46] A. Tamches and B. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. *International Journal of High-Performance and Applications*, 13(3), 1999.

[47] NCSA Tungsten Linux Cluster. http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/XeonCluster/.

[48] D. Turner and X. Chen. Protocol-Dependent Message-Passing Performance on Linux Clusters. In *IEEE Cluster Computing*, September 2002.

[49] User Mode Linux Home Page. http://user-mode-linux.sourceforge.net/.

[50] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, 1998.