K42: An Infrastructure for Operating System Research

Dilma Da Silva,
Orran Krieger,
Robert W. Wisniewski,
Amos Waterland,
IBM TJ Watson Research Center
Yorktown Heights, New York
{dilmasilva,okrieg,bobww,
apw}@us.ibm.com

David Tam
University of Toronto
Toronto, Canada
tamda@eecg.toronto.edu

Andrew Baumann
University of South Wales &
National ICT Australia
Sydney, Australia
andrewb@cse.unsw.edu.au

ABSTRACT

K42 is an open-source scalable research operating system well suited to support systems research. The primary goals of K42's design that support such research include flexibility to allow a multitude of policies and implementations to be supported simultaneously, extensibility to allow new policies and implementations to be readily added, and scalability to enable good performance for both small and large applications on both small and large multiprocessor systems. The goals are accomplished via key features including an object-oriented structure that allows specialized resource management implementations and policies on a per-resource, per-application basis, implementation in user-level servers of much of the system functionality, and a sophisticated set of underlying services that provides a programming model for developing system software in a scalable and modular fashion.

These characteristics make K42 an attractive framework for prototyping new operating system ideas. In addition, K42 has a sophisticated performance monitoring infrastructure allowing a thorough understanding of new ideas to be gained. The above framework combined with a consistent emphasis on scalability makes K42 well suited for high-end computing initiatives. In this paper, we describe the structure of K42 which contributes to the advantageous prototyping environment, and demonstrate how to utilize it by describing ongoing research efforts.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

1. INTRODUCTION

There is a mismatch between resource management on commodity operating systems and the requirements of large scientific applications [2] defined by the High End Computing (HEC) community. As in other areas of computing, researchers in HEC recognize the value of specializing the operating system (OS) to address their unique requirements such as scalability, fault containment, and memory hierarchy management. Current efforts in supporting new programming models for HEC systems (e.g., X10 [16], Chapel [15], and Fortress [3]) can benefit from operating systems support of new customized interfaces, mechanisms, and policies. Long-lived applications common in HEC environments benefit from resource management policies that can change as the application goes through different phases.

The benefits obtained by customizing operating system services are attractive. However, OS specialization is hindered by the fact that operating systems are complex and their development and maintenance is expensive. Sustainable specialization will be more readily achievable given a model that enables customization in a first-class manner.

Current commodity operating systems were not designed to scale or to accommodate specialization; most of their components utilize global data structures and global policies. Some of their sub-systems, e.g. the Virtual File System (VFS) layer [26], are based on a more modular design and have supported a higher degree of customizability. In general though, existing systems have been shown to be difficult to configure or adapt beyond the basic workloads that were targetted. For example, the work on scaling commodity operating systems for a large number of processors has shown that scheduling and memory management algorithms that work well on small machines may need significant changes to perform reasonably on larger configurations [14]. This form of specialization requires much effort and is unlikely to be re-incorporated into the main code-base because it does not address mainstream workload scenarios. In the open-source community, major variants to a mainline Linux kernel are implemented as *patches*. The difficulty in specializing Linux is illustrated by the nature of these patches, where changes are scattered throughout the kernel sources. For example, the resource management changes done in VServer [1] to isolate users from each other on time-shared servers modifies 211 files and introduces 40 new ones, the device driver recovery provided in the Nooks [37] project includes 108 modified kernel files and 68 new files [20].

Research on customizable and extensible operating systems such as Exokernel [19], Spin [11], Vino [35], and K42 [8, 36] has demonstrated that performance gains can be achieved if the operating system is customized to the needs of specific applications. In contrast to K42, the approach taken by the other projects did not allow for an environment where existing mainstream applications could run. Thus, K42 provides a good opportunity for users with significant existing appli-

cations to explore the impact of potential OS customizations on their applications.

Because we have taken the strategy of providing Linux ABI/-API functionality in K42, a large number of Linux binaries already execute on K42. We have run applications and benchmarks such as Apache, dbench, most of SPEC SDET, SPECfp, SPECint, UMT2K, an ASCII nuclear transport simulation, test suites such as LTP (Linux Test Project), the J9 Java virtual machine, SPECjbb, SPECjvm, and the DB2 database management system. Support for unhandled system calls may be added on a case-by-case basis to get a particular application running.

K42 contributes to exploring HEC specialization in the following ways:

1. as a framework where specialized resource management approaches can be easily prototyped and evaluated through the execution of relevant workloads;
2. as an environment where the experiences and performance data gained with the above approaches provide a stronger case for incorporating that approach into a mainstream operating system;
3. as a design supporting customization of services thereby allowing for a better understanding of the trade-offs in design flexibility. This knowledge can be useful for determining ways of incrementally changing existing commodity operating systems to allow for more specialization.

This paper discusses K42's support for implementing customized services and describes ongoing projects using and supporting those services. Section 2 presents the aspects in K42's design that enable flexibility and simplify the experimentation work needed to evaluate new ideas. Section 3 illustrates how the infrastructure is used to explore dynamic adaptation of services. Section 4 describes examples of ongoing operating system research projects by collaborators using K42. Section 5 discusses advantages and disadvantages of K42's infrastructure, and Section 6 concludes.

2. K42'S STRUCTURE

K42's goal was to start with a "clean slate" and examine the system structure needed to achieve excellent performance in a scalable, maintainable, and extensible system. We aimed at fully supporting existing applications, therefore we did not introduce a new OS personality, but instead made K42 fully Linux API- and ABI-compatible [5].

K42 has explored new approaches in memory management, scheduling, inter-process and intra-process communication, event management, file systems, performance monitoring, scalable data structures, and dynamic adaptation [27]. In this paper we focus on the design characteristics that directly impact the customization of services.

The entire system has been designed with an object-oriented approach. Each virtual resource (e.g., virtual memory region, network connection, file, process) and physical resource (e.g., memory bank, network card, processor, disk) is managed by a different set of object instances. Each object encapsulates the meta-data necessary to manage the resource as well as the locks necessary to manipulate the meta-data. We avoided global locks, data structures, and policies. We adhered to a very modular design in order to simplify the addition and evaluation of alternative resource management strategies. By pursuing the separation of concerns, we enable developers to easily add new implementations without requiring an understanding of the entire system and to confine the impact of these changes to the specific set of modified components.

K42 provides an enhanced object-oriented model, called *clustered objects* [6, 21], that can improve access locality by enabling selective partitioning, replication, and distribution of object implementations. The systematic integration of support for flexible data distribution on a per-object basis has yielded a simpler, incremental approach to scalable system design and implementation. The clustered object infrastructure facilitates the addition of new scalable services by allowing the developer to focus initially on functional aspects through a non-distributed version, and then extend the implementation incrementally (for example, on a per-method basis) to a distributed version.

K42's object model introduces technology to simplify the synchronization of object access and destruction. Traditionally, the error of using a stale pointer to deleted storage is avoided by existence locks or use counts to protect pointers. Full-scale garbage collection can also solve this problem, but is not appropriate for low-level operating system code. K42 uses an independently-developed mechanism similar to read copy update (RCU) [29], in which deletion of K42 objects is deferred until all currently running threads have finished. This mechanism results in a programming style where an object releases its locks before calling a method of another object, thus improving base system performance, increasing scalability, and eliminating the need for complex lock hierarchies and the resulting complex deadlock avoidance algorithms.

K42 is structured around a client-server model. Much of the functionality traditionally implemented in the kernel or servers is moved to libraries in the application's address space. For example, all thread scheduling is done by a user-level scheduler library that is linked into each process. This design supports flexibility on a per-application basis. The specialization of services for a class of applications (e.g., games, scientific applications, databases, JVMs) is achieved by choosing the objects that are appropriate for the requirements and packaging them into a library. Security is not affected because only information that would have been accessible to an application is stored in the library. Overhead is reduced in many cases because crossing address space boundaries to invoke system services can be avoided. Also, space and time is consumed in the application rather than in the kernel or servers. For example, an application can have a large number of threads or file descriptors without consuming any additional kernel memory.

As multi-core chips become more prevalent, the scalability of the operating system becomes an important issue. K42 has

been designed to achieve good multiprocessor performance by maintaining the following characteristics. (1) Independent requests to different resources proceed independently because there are no shared data structures to be traversed and no shared locks to be accessed, (2) locality is maintained for resources accessed by a small number of processors, and (3) the use of clustered-object technology allows widely-accessed objects be implemented in a distributed fashion.

Because we designed the system from scratch, we were able to integrate tracing and performance monitoring from the earliest stages, allowing an efficient, unified, and scalable facility [39]. This infrastructure encompass all aspects of the software stack, providing for correctness debugging, performance debugging, and performance monitoring of the system. The infrastructure allows for cheap and parallel logging of events by all levels of the system including applications, libraries, servers, and the kernel. This event log may be examined, written to disk, or streamed over the network, all while the system is running. Information in the trace events can be used to determine system behavior characteristics that are useful for driving further specialization of services.

3. K42'S SUPPORT FOR DYNAMIC ADAPTATION

The elements of K42's design described in Section 2 are instrumental in providing a framework for implementing specialized services, experimenting with them, and enabling applications to deploy the best-suited implementations. Although this support for static configuration of system services is useful, a more dynamic mechanism is sometimes needed. There are many applications that can benefit from further specialization of services when decisions are based on information available exclusively during execution.

K42 addresses the need for dynamic adaptation by providing mechanisms to replace object implementations on-the-fly. K42's dynamic customization mechanisms are *hot-swapping* [36] and *dynamic update* [8, 9]. *Hot-swapping* allows individual object instances, that are used to implement a service, to be tuned to varying service demands. Hot-swapping replaces an active component instance with a new component instance that provides the same interface. To maintain availability and correctness of the service provided, the new component resumes from where the old one had paused. Internal state from the old component is transfered to the new one, and external references will resolve to the new object. Thus, hot-swapping allows component replacement without disrupting the entire system and does not place additional requirements on the clients of the component. *Dynamic upgrade* uses hot-swapping to replace all objects in the system providing a given service. If an upgrade was made to the object representing a process, for example, dynamic upgrade would replace all objects in the system that represented each process. Thus, it can be viewed as the hot-swapping of classes rather than objects.

In the High End Computing arena, hot-swapping is useful to address application behavior characteristics that depend on input data or vary as the application progresses through its processing phases. Dynamic upgrade is valuable for applying changes to the overall system (affecting all applications using the code being updated) without interrupting long-running applications to reboot the system.

The following adaptation scenarios illustrate how the support for flexibility in K42 can be leveraged to handle dynamic application requirements, implementing specialized resource management policies not available in commodity operating systems today.

1. **Performance optimization for common cases:** For many OS resources the common access pattern is simple and can be implemented efficiently. However, the implementation becomes expensive when all complex corner cases must be supported. Dynamic replacement allows efficient implementations of common paths to be used when applicable and less efficient uncommon implementations to be switched in when necessary.

   An example of this scenario is the support for file sharing. While most applications have exclusive access to their files, occasionally files are shared among a set of applications. In K42, when a file is accessed exclusively by one application, an object in the application's address space handles the file control structures, allowing it to take advantage of mapped file I/O. When the file becomes shared, a new object dynamically replaces the old, specialized object. This new object communicates with the file system to maintain the control information, handling the sharing requirements. This customization results in a 34% performance improvement [36] for Postmark, a benchmark designed to model a combination of electronic mail, netnews, and web-based commerce transactions [24].

   Other examples where similar optimizations for the most-common case are possible are a pipe with a single producer and consumer (in which case the implementation of the pipe can use shared memory between the producer and consumer) and network connections that have a single client on the system (in which case zero-copy communication can proceed between the server and the client.)

2. **Optimizing for file attributes:** Several specialized file system structures have been proposed to optimize file layout and caching of files with different attributes [28, 34]. Dynamic replacement can take advantage of these different structures by implementing each and switching between them when appropriate.

   For example, while the vast majority of files accessed are small ($<$ 4KB), OSes must support both large files and files that grow. Dynamic replacement can take advantage of file size characteristics to optimize application performance. In K42, in the case of a small unshared file, an object in the application's address space services requests to that file thus reducing the number of interactions between the client and file system. Once a file grows to a larger size, the implementation is dynamically switched to another object that communicates with the file system to satisfy requests. Running the Postmark benchmark with the small-file cache optimization resulted in a 40% improvement on top of the exclusive-use optimization improvement of 34% [36].

3. **Access patterns:** There is a plethora of literature focused on optimizing the caching and prefetching of file blocks and memory pages from disk based on application access patterns [23, 25]. Researchers have shown up to 30% fewer cache misses by using the appropriate policy. Hot-swapping can exploit these policies by interposing monitoring code to track access patterns, and then switching between policies based on the current access pattern.

4. **Exploiting architecture features:** Many features of modern processors are under-utilized in today's multi-platform OSes. To ensure portable code, without making global code paths unduly complex, these features are generally either crippled or ignored entirely because implementers need to provide a single implementation to be used across all platforms. For example, there is only limited support today for large pages even though a large number of processors support them. Hot-swapping makes it easier to take advantage of architectural features because special-purpose objects can be introduced and used without requiring that all corner case functionality be implemented in every object.

5. **Multiprocessor optimizations:** In large multiprocessor systems, parallel applications can benefit from processor locality. To exploit this locality, some OSes implement services in a partitioned fashion. However, these partitioned implementations consume more memory and incur larger overheads on some operations, e.g., file destruction and process destruction. Conversely, shared implementations can minimize space and time overheads for sequential applications. With hot-swapping, service implementations can be dynamically selected to better match the state of the system and the dynamic behavior of the applications.

6. **Enabling client-specific customization:** Extensible OSes offer new interfaces that enable clients to customize OS components. By using components optimized for a particular application, it is possible to achieve significant performance improvements in a variety of system services [11,19,32,35]. For example, the Exokernel's Cheetah web server demonstrated factor of two-to-four increases in throughput from network stack and file cache specializations [22]. Hot-swapping enables extensibility by allowing applications to replace OS components. Hot-swapping improves upon most existing extensible systems by allowing on-the-fly switching as well as replacement of generic system components. On the other hand, previous extensible systems enabled safe extension of the OS, whereas hot-swapping requires trusting the loaded code.

7. **Exporting system structure information:** Technologies such as compiler-directed I/O prefetching [13] and storage latency estimation descriptors [30] have shown over 100% performance increases for applications, but require detailed knowledge about the state of system structures. Inserting the necessary profiling information can affect the performance of applications that do not require this information. Hot-swapping allows applications to gather more information about the state of system structures by interposing monitoring objects into the kernel. By inserting these monitors only when applications will benefit, overall system performance will not degrade. Without hot-swapping, the additional cost of monitoring and increased system complexity hampers a researcher's ability to consider algorithms designed for rare conditions that may be important for certain applications.

8. **Supporting specialized workloads:** With existing monolithic systems, optimizations introduced for one workload often negatively impact the performance of other workloads. One strategy being considered by some groups is to ship multiple versions of an OS, where each version is tuned for a particular critical workload. Another approach is through incremental specialization, where specific portions of the kernel are recompiled to optimize them for a particular workload. Using incremental specialization, Pu et al. reported performance improvements as high as 70% for small amounts of data read from files [33]. Using hot-swapping, the system can dynamically select or replace these optimized components to handle each specific workload.

4. ONGOING PROJECTS

This section highlights some of the projects investigating new operating system ideas using the K42 platform.

CMP/SMT

K42 is currently being used as a research platform for exploring the impact of chip multiprocessor (CMP) and simultaneous multithreading (SMT) processor architectures on operating system design. The goal of this research is to obtain higher performance by exploiting the new hardware properties of these processors from an operating systems perspective. As the number of processor cores on an integrated chip increases, performance scalability issues will also arise. Established scalability techniques on traditional shared-memory multiprocessors may be applicable in many cases. In other cases, new techniques will need to be developed because the new hardware properties of these processors may invalidate past assumptions and design decisions. Currently, modifications to the K42 scheduler are being explored in an attempt to exploit the significantly lower costs of execution and data migration among processor cores within a single integrated chip. The two-level scheduler design of K42, inspired by Anderson et al. [4], separates user-level thread scheduling within an application from kernel-level process scheduling across different applications. The advantage of this design is that it suggests a logical strategy of investigating migration possibilities among threads within a process separately from migration possibilities among processes. Migration at each level has different challenges, costs, and benefits. After exploring each level, a structured integration and interaction of these two dimensions can be examined. In summary, K42 facilitates a divide-and-conquer strategy in this study.

Resource Containers

Resource containers, originally proposed by Banga et al. [7], were explored on the K42 research platform by a team at the University of Toronto. The resource container mechanism was implemented to track processor and memory usage on K42. The microkernel design of K42, where the operating system consisted of (1) a small exception-handling component, (2) a fast inter-process communication (IPC) mechanism, and (3) user-level servers to provide operating system services, led to two advantages in choosing K42 as the research platform in this study. First, the interaction model of the microkernel, servers, and applications, all via IPC, naturally suggested a simple model for resource container transfer and tracking; resource containers are remapped upon IPC events. Second, the microkernel design of K42 minimized the amount of time that would have otherwise been spent in the kernel of a monolithic operating system. This property helped to reduce resource container design complexity and potential resource accounting leakage. During execution within a monolithic kernel, tracking resource usage is a more complex task due to the global pooling of resources and task multiplexing. A major disadvantage of K42 in this study was the two-level scheduler design, which closely follows the ideas of Anderson et al. [4]. Although a two-level scheduler offers flexibility, it was a source of complexity for resource container integration. In a multithreaded server application, such as a file-system server, where each thread belongs to a different resource container, thread switching at the user-level scheduler requires notifying the kernel of a corresponding resource container switch in order to maintain accurate resource accounting. This requirement would eliminate the fast, light-weight thread switching advantages of a user-level scheduler. In a traditional one-level scheduler, resource containers can be switched and resource limits can be enforced while executing in kernel space, since thread switches are done at the kernel level.

Dynamic Update of Operating Systems

At the University of New South Wales, K42 is being used for experimenting with dynamic update for operating systems [9]. The ultimate goal of this work is to be able to take any update to the OS and apply it to a running system without loss of service. The choice of K42 as an experimental platform for this work was motivated primarily by its support for hot-swapping, but additional unanticipated advantages were found in the process.

In order to be able to update a system, and to contain and reason about the effects of the update, it is necessary to be able to define an updatable unit with well-defined interfaces and boundaries. K42's object-oriented structure was found to be extremely useful here; objects are the natural choice for the updatable unit, and the strong data encapsulation enforced by the object-oriented design allows us to update data structures and data formats with ease. In contrast, in considering the application of the same features to Linux, the research team found that Linux's lack of clearly enforced module boundaries or data encapsulation, while surmountable in many cases, made dynamic update more complex to implement [10].

File Systems

K42 was used at the University of Toronto as a research platform for examining file-system scalability on large-scale shared-memory multiprocessors [38]. In order to focus solely on the file system and to push this component to its scalability limits, a provenly scalable operating system, such as K42, was needed. Using K42 meant that the underlying software infrastructure, which is used by file system, would not be a bottleneck point. Any scalability problems would be exclusively due to the file system component. Useful scalable infrastructure provided by K42 included: (1) the locality-sensitive memory allocation facilities, and (2) locking facilities optimized for scalability. Experience in this study showed that K42 (1) provided a scalable operating system that never became the bottleneck point and (2) no fundamental design-incompatibility issues were encountered between K42 and the file system. However, because K42 was in its early development stages during the study, well-known standard benchmarks and applications could not be executed on this platform. Since then, K42 has matured to be capable of running the desired workloads.

A project at the University of São Paulo advanced the customization support in K42's File System (KFS). KFS is a fine-grained adaptable file system that is customizable at the granularity of files and directories, allowing K42 to satisfy the requirements and usage access patterns of various workloads [17]. In KFS, each file or directory may have a custom-tailored service implementation, which can also be replaced on-the-fly. By doing so, KFS addresses the difficulties found in traditional file systems, which are typically designed to handle a specific set of requirements and assumptions about file characteristics, expected workload, and usage and failure patterns. This project extended KFS to include meta-data snapshotting, allowing it to have the properties of a journalled file system but with significantly lower performance overhead. KFS was shown to induce less overhead than a write-ahead journalling file system and to scale better as the number of clients and file system operations increases [17]. This effort also resulted in KFS becoming available as a file system for Linux 2.4 with reasonable performance.

Memory Fragmentation

A team at the IBM Linux Technology Centre in Australia is currently investigating a new memory management technique to reduce memory fragmentation and better support, for example, large-pages. Memory is allocated to a consumer (e.g., a process or group of files) in larger (variable-sized) chunks and then internally sub-allocated by the consumer as necessary. All memory given to a single consumer has the same expected lifetime, e.g., the computation memory handed to a process has the expected lifetime of that process. Hence, allocating contiguous memory to a consumer naturally avoids fragmentation. Preliminary results are very promising. We also expect, once we have tuned the implementation, that allocating and managing memory in larger units will result in performance improvements due to reduced synchronization and more cache-friendly data structures.

Although ultimately aimed at being implemented in Linux, K42 was chosen as the platform to prototype this idea because its object-oriented design was naturally suitable to

this effort. In K42 there are already objects that represent the key consumers, and requests to allocate memory already go through these consumers. Hence, caching chunks in these consumers and causing them to allocate memory in chunks was a very natural effort. If we determine that the benefits are large enough, we will do the much larger work to transfer this technique to Linux.

HEC Support

The SmartApps [18] project offers a framework to achieve application-centric optimization through dynamic algorithmic adaptation, reconfiguration of system parameters, and tuning of operating system services. SmartApps is integrating K42's support for specialization of services into a middleware stack designed to monitor application behavior and determine beneficial application restructurings.

At the Lawrence Berkeley National Laboratory, researchers are leading an effort to investigate the operational architecture of large-scale operating systems by extending the K42 research to clusters.

Researchers at the University of New Mexico are leveraging K42 to support specialized workloads and client customizations in the HEC environment. HEC applications are resource constrained and therefore their performance is determined by the policies and mechanisms under which these resources are managed.

This research leverages client customization support in K42 to provide HEC applications with: (1) finer-grained control of resources, and (2) the ability to reconfigure and relocate OS services. Memory is an example of a resource in which finer-grained control is beneficial in HEC applications. Inefficient memory management policies can have a negative performance impact on HEC applications. As an example, MPI applications communicate with other nodes through internal local buffers of the application. The MPI specification does not dictate the location of communication buffers that must reside in the application address space. Many HEC network interface cards (NICs) are capable of directly writing and reading data to or from an application's address space. However, these communication buffers must be pinned, i.e., page mappings must be fixed and present in memory, before the NIC can access them.

Traditional MPI implementations require kernel interaction for the pin operation and therefore incur significant overhead. This overhead includes time to trap into the kernel, walk and modify page-table entries, and populate pages in memory (possibly involving swap operations). As communication buffers change throughout the course of the application these operations will adversely affect performance. The kernel cannot infer *a priori* the application's communication patterns, and therefore cannot optimize memory management usage. In contrast, an application knows its communication patterns and can achieve better performance using this knowledge. For example, it can maintain an appropriately sized pool of permanently pinned memory which it can manage for communication operations and thus better utilize memory resources. Furthermore, an application can take advantage of memory optimizations such as large page support, specialized page replacement and pinning policies, etc., based upon its communication patterns.

Additionally, HEC applications can benefit from the ability to reconfigure and relocate OS services based on usage patterns and availability. One such example is cache injection, which allows a device to put data directly into a processor's cache, bypassing an expensive fetch from memory. This research focuses on the infrastructure involved to support a NIC that is capable of cache injection. This infrastructure involves careful interaction with other OS services such as scheduling, processor affinity and migration, and memory management.

5. PROS AND CONS

Operating systems researchers need a platform to implement and evaluate their ideas. The environment may constrain how throughly the idea can be explored. As in other areas of computer science (e.g., programming languages), there is a choice between pursuing innovation through incremental evolution of existing frameworks or through brand new initiatives.

OS researchers can choose between the following options for working on their goals: (1) working on a commodity operating system, (2) starting the research environment from scratch, and (3) working on an existing extensible framework.

The advantages of the first approach are that existing operating systems come with a rich set of workloads and the perception that they are an effective medium to take ideas from research labs into the real world. Also, the learning curve for working with these systems is usually not a challenge, because most of their structuring follows traditional principles and accepted practices described in textbooks. The disadvantage of commodity operating systems as a platform for experimental work is that they incorporate a large number of design and implementation decisions that can not be easily altered. Some new ideas may be infeasible to implement due to the mechanisms and policies in place. In particular, theses systems were not designed to accommodate specialization of services.

Starting from scratch eliminates the constraints from legacy approaches, but the required investment is daunting, and in many cases impossible given the resources available. It deviates the effort from the main research inquiry into infrastructure work. By working on a radically novel environment, it may be more difficult to prove the applicability of the new ideas on mainstream systems.

By using an existing extensible operating system as a basis, it is possible to leverage previous work done to develop the infrastructure. If the system is able to handle the type of workload necessary to carry out the research, then the problem of application availability is also addressed. If the OS is also flexible enough to accommodate a reasonably efficient implementation of the research idea, then it constitutes a promising platform for the investigation. Possible disadvantages of this approach are the lack of stability in the code, insufficient or uncertain support from the framework's original developers, lack of functionality, and framework complexity.

K42 has been designed to offer a higher degree of static and dynamic customization and it is able to execute a broad range of workloads. The experience with K42 so far has highlighted the following advantages and disadvantages:

Pros

- K42 can provide applications or sub-systems with resource management tailored to their needs, because per-resource object instances allow multiple policies and implementations to be supported simultaneously by the system. Because each resource instance is implemented by an independent object, resource management policies and implementations can be controlled on a per-virtual-resource basis, thus different applications can use different resource management policies. Even within a given application, different policies may be supplied for different instances of a given resource. For example, every open file may have a different prefetching policy, and different page caches may have different replacement policies.
- The modular nature of the system makes it more maintainable by providing a clean model for supporting new applications and new hardware. For each new platform or application that K42 supports, additional objects may be created. Thus, each object remains simple and easy to program for the given platform or application. As code does not affect all users in the system, developers working on features that apply to a narrow audience may still contribute to K42 because only those applications for which features are advantageous need to use them. Further, because modifying a given object does not involve code from many aspects of the system, non-kernel developers may more easily develop a new resource management policy for their application.
- For many subsystems (e.g., memory management) it is possible to pursue sophisticated new approaches without knowledge of the rest of the system.
- The code structure has allowed new implementations of objects to support old and new policies with minimal added complexity to the source code.
- The performance monitoring infrastructure is invaluable in profiling functionality and identifying major performance bottlenecks quickly.
- Some of the characteristics of K42 are definite advantages in an experimentation environment: K42 runs on the Mambo full system simulator [12], it supports the GDB interfaces necessary for remotely debugging the running system, and it offers a *fast-reboot* feature for rebooting a new K42 kernel image without having to go through the firmware and bootloader stages. This K42 feature pre-dates Linux's similar *kexec* feature.

Cons

- The effort required in setting up, building, and booting K42 is non-trivial.
- The limitation on hardware alternatives to run K42 has been an issue. However, the situation has improved recently with the availability of Apple Xserve machines.
- K42's learning curve is fairly steep, but once the developer becomes familiar with the object model and basic design principles, the difficulties lessen considerably. A side-effect of the object-oriented approach is the difficulty in gaining an understanding of the system by code inspection, because statically the code path may be non-obvious.
- Parts of the system are very suitable for customizability (e.g., memory management, scheduling, filesystems), but the parts that consist of Linux components (e.g., drivers, network stack) are not.
- For *fork*-intensive workloads, K42 performs much worse than Linux, due to a performance problem in K42's support for *fork* [5] that has yet to be fixed.
- Because K42 is a research environment that differs significantly from commodity operating systems, researchers must be able to argue that the results obtained through K42 experiments are representative of trends on a real system, and that the ideas and results are applicable on commodity systems.
- As more and more implementations are developed to represent alternative service specializations, interface changes and bug fixes may result in maintenance complexity. K42's regression test suite covers only a few basic configurations of the system, therefore it may take several months before we learn that a change in a base common class has caused one of the available specializations to fail. Jeff Mogul argues that the type of flexibility offered by extensible operating systems seems to create more problems than solutions [31]. K42's experience indicates that although there are many challenges in maintaining and testing a flexible system, the benefits may be worth it.

6. CONCLUSIONS

K42's design is an attractive platform for conducting operating systems research because of its built-in support for customizability of services, its sophisticated tracing and performance monitoring infrastructure, and its ability to execute a rich, general set of workloads. The experience from several ongoing projects using K42's infrastructure so far indicates that K42's flexibility can be successfully deployed to explore and evaluate new approaches to resource management.

K42 is available as open source under an LGPL license. Please see our home page `www.research.ibm.com/K42` for additional papers on K42 or to participate in this research project.

Acknowledgments

A kernel development project is a massive undertaking, and without the efforts of many people, K42 would not be in the state it is today. In addition to the authors, the following people have contributed much appreciated work to K42: Marc Auslander, Bryan Rosenburg, Jonathan Appavoo, Jimi Xenidis, Michal Ostrowski, Maria Butrico, Mark Mergen, Chris Yeoh, Jeremy Kerr, Reza Azimi, Michael Britvan, Chris Colohan, Phillipe DeBacker, Alexandre Depoutovitch, Khaled Elmeleegy, David Edelsohn, Raymond Fingas, Hubertus Franke, Ben Gamsa, Garth Goodson, Darcie Gurley,

Kevin Hui, Edgar Leon, Craig MacDonald, Iulian Neamtiu, Michael Peter, Jim Peterson, Eduardo Pinheiro, Bala Seshasayee, Rick Simpson, Livio Soares, Craig Soules, Michael Stumm, Adrian Tam, Manu Thambi, Nathan Thomas, Gerard Tse, and Timothy Vail.

This work was partially supported by DARPA under contract NBCH30390004. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs. The work from the University of Toronto is funded by NSERC Canada (Natural Sciences and Engineering Research Council of Canada) and the *Ontario Graduate Scholarship* program.

7. REFERENCES

[1] Linux VServer Project. http://linux-vserver.org/.

[2] Report from the 4th FAST-OS (forum to address scalable technologies for runtime and operating systems) workshop. http://www.cs.unm.edu/ fastos/03workshop/report.pdf, July 2003.

[3] Eric Allen, David Chase, Victor Luschangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification (version 0.785). Sun Microsystems, Inc., November 2005.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb 1992.

[5] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Freenix track, USENIX Technical Conference*, pages 323–336, San Antonio, TX, June 9-14 2003.

[6] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger, and Craig Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, pages 3–8, 2002.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI'99: Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[8] Andrew Baumann, Jonathan Appavoo, Dilma da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *Workshop of Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*, pages 21–27, Boston Massachusetts, October 9 2004.

[9] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, and Robert W. Wisniewski. Providing dynamic update in an operating system. In *USENIX Technical Conference*, pages 279–291, Anaheim, CA, April 2005.

[10] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proc. of 6th Linux.conf.au (LCA)*, Canberra, April 2005.

[11] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyn, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating System Principles*, 3–6 December 1995.

[12] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[13] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001.

[14] Ray Bryant, John Hawkes, and Jack Steiner. Scaling Linux to the extreme: from 64 to 512 processors. In *Ottawa Linux Symposium*. Linux Symposium, 2004.

[15] Brad Chamberlain. An introduction to Chapel: Cray Cascade's High Productivity Language. In *AHPCRC DARPA Parallel Global Address Space (PGAS) Programming Models Conference*, Minneapolis, MN, September 2005.

[16] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[17] Dilma da Silva, Livio Soares, and Orran Krieger. KFS: Exploring flexilibity in file system design. In *Proc. of the Brazilian Workshop in Operating Systems*, Salvador, Brazil, August 2004.

[18] Francis Dang, Maria Jesus Garzaran, Milos Prvulovic, Ye Zhang, Alin Jula, Hao Yu, Nancy Amato, Lawrence Rauchwerger, and Josep Torrellas. Smartapps: An application centric approach to high performance computing: Compiler-assisted software and hardware support for reduction operations. In *Proc. Next Generation Software Workshop*, pages 10–19, Fort Lauderdale, Florida, April 2002.

[19] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, pages 251–266, 3–6 December 1995.

[20] Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. Patch (1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, June 2005. USENIX.

[21] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, February 22-25 1999.

[22] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceno, Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[23] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 115–126. ACM Press, 1997.

[24] Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance.

[25] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Symposium on Operating Systems Design and Implementation*, pages 119–134. USENIX Association, 23–25 October 2000.

[26] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of 1986 Summer USENIX Annual Technical Conference*, pages 238–247, Atlanta, GA, USA, 1986.

[27] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a real operating system. In *Proceedings of EuroSys'2006*, pages 133–145. ACM SIGOPS, April 2006.

[28] Orran Krieger and Michael Stumm. HFS: a performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, 1997.

[29] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 338–367, 26–29 June 2002.

[30] Rodney Van Meter and Minxi Gao. Latency management in storage systems. In *Symposium on Operating Systems Design and Implementation*, pages 103–117. USENIX Association, 23–25 October 2000.

[31] Jeff Mogul. Operating systems should support business change. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Fe, NM, June 2005. USENIX.

[32] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Symposium on Operating Systems Design and Implementation*, volume 30, pages 153–167. ACM, 1996.

[33] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *ACM Symposium on Operating System Principles*, volume 29, 3–6 December 1995.

[34] ReiserFS home page, April 2002. http://www.namesys.com.

[35] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. An introduction to the architecture of the VINO kernel. Technical report, Harvard University, 1994.

[36] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX Technical Conference*, pages 141–154, San Antonio, TX, June 9-14 2003.

[37] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Procedings of the 6th ACM/USENIX ymposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Francisco, CA, December 2004. USENIX.

[38] David Tam. Performance analysis and optimization of the Hurricane File System. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada, January 2003.

[39] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, Phoenix Arizona, November 17-21 2003.