A Case for Secure and Scalable Hypervisor using Safe Language*

Haibo Chen
Institute of Parallel and Distributed Systems
School of Software
Shanghai Jiaotong University
haibochen@sjtu.edu.cn

Binyu Zang
Parallel Processing Institute
Fudan University
byzang@fudan.edu.cn

ABSTRACT

System virtualization has been a new foundation for system software, which is evidenced in many systems and innovations, as well as numerous commercial successes in desktop, datacenter and cloud. However, with more and more functionality being built into the virtualization layer, the trustworthiness of the hypervisor layer has been a severe issue and should no longer be an "elephant in the room". Further, the advent and popularity of multi-core and many-core platforms, the scalability of the virtualization layer would also be a serious challenge to the scalability of the whole software stack.

In this position paper, we argue that it is the time to rethink the design and implementation of the virtualization layer using recent advances in language, compilers and system designs. We point out that the use of safe languages with scalable system design could address the trustworthiness and scalability issues with virtualization. We also argue that applying language innovations to the hypervisor layer avoids the need of an evolutionary path, as it is relatively small in scale and has little backward compatibility issue.

Categories and Subject Descriptors

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

General Terms

Design, Languages

Keywords

Hypervisor, Verification, Scalability, Trustworthiness, Object-oriented Design, Aspect-oriented design

1. INTRODUCTION

System virtualization has recently gained a resurgent interest and made system research more relevant. Evidences include many systems and innovations [40, 8], as well as commercial successes [27, 38]. It was even claimed that hypervisors have become "the new foundation for system software" [20] [1]. Much system-related research has been conducted in the hypervisor layer due to its global view on system resources. Meanwhile, commercial pressures also push the improvement of hypervisor to support new business models (e.g., virtual appliances [30] and cloud computing) and new applications.

However, modern hypervisors such as Xen and VMWare are partly derived from contemporary operating systems, e.g., reusing their code and/or sharing their design decisions. Unfortunately, many of the design decisions "have remained unchanged, even as hardware and software have evolved" [24], thus are somewhat inadequate in the context of advances in modern architecture and the context of evolving usage scenarios. For example, the monolithic design using unsafe procedure-based languages (i.e., C) results in poor maintainability and robustness, while the use of global policies (including global data structures and locks) limits its scalability for large-scale processors [36, 34]. Further, current hypervisors are hard to extend their functionalities. An adoption of functionalities usually involves changes spreading across multiple files in a hypervisor, which are hard to maintain and error prone.

Meanwhile, there have been a number of advances in both processor architectures and languages. On the hardware side, recent processors have stepped into the multicore era, where it is no surprise to see tens of CPU cores in a single commodity server. As a new form of the Moore's law, multicore or many-core processors are still evolving fast in terms of both scale and heterogeneity. On the language side, advances in safe language design as well as the corresponding code and model verification techniques have made it practical to apply safe languages and verification techniques to even an operation system [24, 25]. Further, efforts in over twenty years of improvements in operating system scalability have also accumulated a number of systems and innovative ideas [29, 14, 11, 12, 9, 39, 33].

In this position paper, we rethink the design and implementation of the virtualization layer in the context of numerous advances in processors, language and system design techniques. We argue that it is time to consider the reconstruction of the hypervisor layer using safe languages as well as new scalable system design techniques, with the goal of yielding a secure, scalable, verifiable and extensible hypervisor. We argue that the hypervisor is a better place to integrate innovations in language, compiler, verification and sys-

*This work was funded by China National Natural Science Foundation under grant numbered 61003002 and 90818015, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100 and Shanghai Leading Academic Discipline Project (Project Number: B114), and Fundamental Research Funds for the Central Universities in China.

[1]Here, we only focus on hypervisors which directly execute on bare hardware. Hypervisors running on a host operating system (e.g. VMWare workstation and KVM) are similar.

tem designs, due to the fact that a hypervisor sits in a much lower layer in the software stack. For this reason, reconstructing a hypervisor using safe languages and system design techniques will suffer very little compatibility issues given that most hypervisors provide a similar hardware interface in the hope of transparently supporting legacy and new operating systems originally running on bare metals.

Specifically, we believe the following language techniques could be helpful to remedy the situation. First, a type-safe language with verifiable extension can improve the code quality of hypervisors by enabling formal verification [37] and soundness analysis [4]. Second, object-oriented and aspect-oriented design should be helpful to improve the customizability, extensible and maintainability. Third, deliberate object management (e.g., clustered objects [29] and cache-aware data distribution and scalable locks [12]) should be promising to improve the locality and scalability of hypervisors. Finally, proper object replication and partition [14] should be helpful to improve the fault containment of hypervisors.

The following sections are organized as follows. The next section describes issues with current hypervisor designs and discusses the opportunities in the face of innovations in languages and system designs. Then, we argue why a hypervisor is a good candidate to apply various novel language innovations, using the criteria of practicality and cost-effectiveness. Then, we describe the design consideration and possible design of a system (called SafeHype in this paper). Finally, we survey literatures on various related research and then conclude.

2. PROBLEMS AND OPPORTUNITIES

This section briefly describes the problems with existing virtualization and discusses related advances in programming languages, formal verification, and system designs, which open the opportunities to address issues raised in this paper.

2.1 Issues with Current Hypervisor Design

Ideally, we believe the following possibly absent but demanding features are necessary for modern hypervisors to match their roles in software stack and satisfy changing business needs.

2.1.1 Trustworthiness

As hypervisors be gradually used in mission-critical applications, their trustworthiness becomes vitally important. However, there have been various issues with current hypervisor layers. First, there are very limited security guarantees in current hypervisor layer, which is, however, an "elephant in the room" and largely ignored in many security systems [18]. For example, recent research [41, 18] and evidences from security vulnerability database [1] show that current hypervisor layer is huge and complex, and thus can be easily tampered with by either external attackers or malicious operators. Hence, many current cloud platforms only provide very limited security guarantees to users' data [28, 3], which does not match the increasing trust being put on the virtualized cloud platforms.

	VMM	*Dom0 Kernel*	*Tools*	TCB
Xen 2.0	45K	4,136K	26K	4,207K
Xen 3.0	121K	4,807K	143K	5,071K
Xen 4.0	270K	7,560K	647K	8,477K

Table 1: Trusted computing base of the Xen virtualization layer (by Lines of Code) [41].

Second, with more and more virtual machines being deployed in a single platform managed by a single hypervisor, the robustness and fault containment have also been critical issues for current hypervisors. Unfortunately, current virtualization layers are pretty large in terms of code size and complex in functionalities that it implements. Thus, it may contain a number of bugs. One reason is that most hypervisors are implemented using unsafe programming languages, risking of breaking type or memory safety. One may argue that a hypervisor is relatively small thus easy to ensure it is bug-free via testing and code review. However, the hypervisor layer, including both the hypervisor, management VM as well as tools, are pretty large in code size and the code size is still steadily increasing, as shown in Table 1. Worse even, many virtualization developers are less sophisticated hackers than the sole hypervisor designers, integration of their code will likely degrade hypervisor trustworthiness. Further, the flexibility of C programming language makes it difficult to ascertain the correctness of programs [13].

Worse even, the dramatic advances of hardware technology will also come with significant reliability problem [2]: high-density cores per-chip also increases the probability of both transient and permanent failures. As the resource manager for the system platform, the hypervisor layer should be designed with awareness of fault containment, to survive it from both transient and permanent hardware failures.

2.1.2 Scalability and Extensibility

Scalability: Recent technology advances have accelerated the prevalence of multi-core or many-core systems. Eight to twelve cores in a chip have already been commercially prevalent with low cost now. Viewpoints from Intel show that "even hundreds of cores" in a chip will be available in the next decade [10]. The advances in processor architectures demand the hypervisor, a new foundation for system software, be highly scalable. However, recent measurements show that commodity hypervisors (e.g., Xen) experience with low scalability due to the use of system-wide and domain-wide locks and data structures [36, 34].

Customizability: With the prevalence of system virtualization in both academic research and industry, hypervisors have been used in various usage scenarios. Meanwhile, as the global manager for computing resources, a hypervisor is likely to host many significantly different VMs (operating systems) and applications. Thus, it is desirable to adjust the hypervisor resource management policies and even modules to suit diverse usage scenarios. Global policies or a unified view are not always suitable to satisfy various applications (e.g., computation-intensive vs. I/O-intensive and latency-intensive vs. through-intensive). In contrast, local policies should be more suitable for existing applications. Moreover, the ability of online-adjustment of policies is desirable to avoid loss of availability. Also, the number of local policies (per-module or per-object) may be much larger than global policies. They should be able to be described using a simple policy language.

Extensibility and Maintainability: Ever since the emergence of virtualization, a lot of research work as well as industry efforts have been conducted to extend existing hypervisors to gain better performance and security and support more functionalities. However, only a very few have been integrated into the mainstream hypervisors. It should not merely be attributed to laziness. We believe two main reasons cause the situation. First, mainstream hypervisors inherit the monolithic and structural design from contemporary operating systems, preventing easy extensions of the hypervisors. Our experiences in several projects [17, 16, 15, 23, 33] in extending hypervisors show that a single extension usually span across a number of files and could incur likely conflicting changes. Second, traditional *diff* and *patch* approach shows poor maintainability to system software [21]. To support easy integration of extensions and shorten the time-to-market of new features, it is demanding that the hypervisor be highly extensible and maintainable. Here, both static and online extensibility are useful to increase the flexibility and availability.

2.2 Opportunities

Here, we claim that we are not that pessimistic as would be interpreted from the above argument. Instead, we actually are encouraged by recent advances in the programming languages, formal verification and system designs, which we believe can be put together to solve parts of the problems described above.

In the programming language community, years of research has yielded a number of safe language and its runtime support. Researchers have also successfully demonstrated the practicality of applying it to even both new [24] and commodity [19] operating systems. Further, formal verification techniques have also shown its capability of verifying a small operating system kernel [25]. Finally, language design such as Aspect-oriented programming may improve the code maintainability of code. There have been several dialects of common languages supporting Aspect-oriented programming, including Aspect Java and Aspect C#.

For system design techniques, previous work on scalable operating systems [11, 22, 5] has shown various approaches that could yield good scalability on large-scale shared memory multiprocessors. For example, the Corey operating system [11] advocates controlling sharing by applications among kernel objects and has provided several kernel abstractions to improve operating system scalability. The Hive operating system [14] and the multikernel design [9] advocate treating multicore platform as a distributed system and leverage multiple cooperative kernels to work together to manage a large-scale shared memory machine. The sloppy counter [12] approach provides scalable counting mechanism on multicore system. The Read-Copy Update [26] mechanism leverages a lazy memory management scheme to allow concurrent read and write operations. All these techniques have great potential to be incorporated into the hypervisor layer to provide scalable performance. Actually, we do notice that the Read-Copy Update mechanism have already been used in the Xen hypervisor.

3. WHY A VMM IS A GOOD CANDIDATE?

None of the above mentioned language techniques and system designs are new in essence. Some of the literatures have already been used in the context of operating systems. For example, the Singularity operating system [24] also employs type-safe language (i.e., Sing#) and code verification techniques to improve the dependability and trustworthiness of operating systems. Tornado [22] and K42 [5] have demonstrated the power of object-oriented design to improve the customizability and scalability of operating systems. Yet, using the criteria of practicality and cost-effectiveness, we believe building a hypervisor using modern language innovations could be more applicable than in operating system level.

Brewer et al. [13] point out that a major concern in replacing C in system software is the lack of an “evolutionary path”, which can avoid efforts of porting or rewriting existing software. This is the case for an operating system due to its large code size (millions to tens of millions of code) and possibly loss of backward compatibility (e.g., Singularity [24]) to millions of legacy applications. However, we argue that for a hypervisor the impact of losing an evolutionary path can be mitigated at a minimal level and thus practical enough to apply innovations in language and system design.

First, a hypervisor itself is much less complex than an operating system measured in code size or implementation complexity. A hypervisor is a software layer managing the underlying machine resources and exposing them to operating systems in the form of virtual machines (VMs). The major roles of a hypervisor include abstraction and management of resources, isolation and sharing of resources among VMs and handling inter-VM control and data communication. Although these resemble the roles of operating systems at first glance, the level of hypervisor abstraction and management are at a much lower level. Thus, a hypervisor incurs a significant less complexity in code size than an operating system. Moreover, modern hypervisors (e.g. Xen) further move the I/O device virtualization out of the hypervisor, making them even smaller. Although the code size of a hypervisor will likely expand in light of numerous innovations and extensions, the porting effort can be still much less and new extensions can be developed under the new framework. Though the management VM would be a problem due to its large code size, there is actually very little reason to leverage a commodity operating system to host the management tools. We believe it is practical to either replace the operating system for a management VM with an existing operating system written in safe language (e.g., Singularity) or even write a new one from scratch with only necessary functionalities needed to support the management tools.

Second, there are little or no backward compatibility issues for a hypervisor. As a hypervisor multiplexes resources in VM-level, it generally does not directly interact with applications. Thus, dramatic changes in a hypervisor usually do not require any change in applications. For operating system compatibility, full-system virtualization obviously does not have compatibility issues. For para-virtualized virtual machines (i.e., operating systems), if the hypervisor can retain the existing VM interfaces, it can also be backward compatible with existing para-virtualized operating systems. As a VM interface is generally much smaller than operating system interface (i.e., system calls) with less complex semantics (such as ioctl), it is much easier to comply with.

Finally, innovations in hypervisors have a little performance implication. Using a safe language may incur additional performance overhead due to the added call indirections and changed code layout. However, as the proportion execution time in a hypervisor is much less than an operating system for a well-formed (e.g., non-blocking) hypervisor, the possibly added execution time should contribute little to the overall performance of applications. Further, using a scalable design and language can likely improve the performance for medium- or large-scale processors due to the possibly improved cache locality and scalability.

4. THE CASE OF SAFEHYPE

In this section, we present the design decisions of a possible hypervisor called SafeHype, which uses various programming language innovations and new system designs to improve its trustworthiness, scalability, customizability, extensibility and maintainability. The major design decisions include using scalable system design and type-safe language with verifiable extensions. We first present specifically the design decisions in the SafeHype system, and then provide the overall possible design of the SafeHype.

4.1 Design Decisions

4.1.1 Programming Language

“Good languages lead to good systems” [13]. To build a robust hypervisor, language is thus a critical issue. Any replacement of C programming language in system software should not sacrifice its expressiveness and efficiency [13, 7], which are critical for efficiently managing low-level objects (e.g., page tables and segments). Also, it should retain programmers' custom and be easy to ascertain the correctness. Here, we choose to use a type-safe C++ subset with verifiable extension.

We choose C++ because of its object-oriented nature and comparable expressiveness with C. We introduce a garbage collector to the SafeHype to manage object allocation and deallocation. To ensure safety, we intend to discard the unsafe features in C++ (such as pointer-arithmetic and implicit type casting). Also, we intend to provide a static verification and soundness analysis tool to detect and reject suspicious errors early in compilation time, to reduce the overhead of runtime checking.

4.1.2 *Trustworthiness*

Using a type-safe language and software verification tool can significantly improve the security and robustness of the hypervisor. We plan to incorporate existing formal verification techniques in Singularity [24] and sel4 [25] to verify the correctness and security properties of SafeHype. As the case for fault containment, SafeHype provides two levels of containment of failures: core-level and node level. SafeHype is designed with SMP and NUMA awareness. Processors with uniform memory access are considered in the same node. As in Hive [14], SafeHype replicates the code and data in each node in case of node failure. The wide use of object-oriented design and clustered objects makes it easy for hypervisor replications. Within the same node, each processor monitors the liveness of other processors and communicates with each other to assure the liveness in a fixed time interval. Upon a failure, they coordinate the failure recovery by attempting to recover the failed processes and detach the failed processors.

4.1.3 *Scalability*

Two key elements in hypervisor scalability are good cache locality and less lock contention. To improve cache locality, a hypervisor should be designed to avoid possible false sharing. To reduce lock contention, a hypervisor should avoid the use of global locks and data structures. As a result, SafeHype plans to adopt the following novel design and structures: clustered objects [29], sloppy counters [12], and Read-Copy Update [26]. In clustered objects, objects accessed by different processors are mapped to different physical addresses, yet in a uniform object-oriented interface. The adoption of clustered object can efficiently support object replication, migration and distribution, to minimize cache eviction and lock contention. The sloppy counter design can make the pervasively used counting mechanism in a hypervisor being scalable. The Read-Copy Update allows concurrent accesses to many data structures. Moreover, SafeHype avoids the use of global data structures and mostly relies on per-object locks. Large system objects are partitioned or replicated among processors to avoid access contention. Finally, SafeHype tries to minimize sharing by explicitly allowing sharing only when necessary and provide mechanism from Corey [11] to support efficient sharing.

4.1.4 *Customizability*

The adoption of object-oriented design naturally fits the requirement for customizability. SafeHype relies on inheritance and polymorphism to implement customizability. SafeHype provides a modular implementation for each system resource and control. One can provide various implementations of policies to manage system resources, under a uniform interface. To support runtime transformation among different policies, each implementation should provide a state transforming function [6] to transfer the resident state to a new instance. The means to customize the system thus mainly relies on changing different policies of each system resource. To efficiently resolve conflicts among various policies, we intend to provide a simple policy language as well as a tool to check the validity of an overall policy description.

4.1.5 *Extensibility and Maintainability*

As with customizability, SafeHype also relies on inheritance and polymorphism to implement extensibility. Extending a hypervisor can be done via inheriting and extending existing objects. To assist dynamic extensions, SafeHype also supports dynamically downloading code into the hypervisor. The downloaded code is first verified by the hypervisor and then is relinked and relocated to the hypervisor.

To assist the static extension or adjustment of crossing-cutting code (such as logging and debugging) code, SafeHype also supports the use of aspect-oriented programming (Aspect C++ [35]). However, the use of Aspect C++ also complicates the soundness analysis. To overcome this, the code of SafeHype will first be transformed by Aspect C++ compiler (a source-to-source compiler) to normal C++ source code. Then, the transformed code will be applied with soundness analysis and software verification process.

4.2 The Case of SafeHype

We plan to implement SafeHype as a research hypervisor that investigates novel language innovations to address various issues with commodity hypervisors. It is still in a very early stage. We have made an initial design and the design is still under intensive discussion. SafeHype is designed with awareness of modern architectural advances and supports and optimizes for large-scale shared memory multiprocessor (NUMA-aware). To minimize the device porting efforts and make SafeHype measurable, SafeHype is designed to be compatible with Xen, a popular open-source hypervisor. SafeHype intends to support both para-virtualization and hardware-assisted full-virtualization (e.g., support Intel VT and AMD-V).

Figure 1 shows the general architecture of the SafeHype systems. SafeHype is implemented in a modular manner to enable reuses of each module. The detailed objects are not listed on due to space constraints. The base system contains several modules managing and virtualizing underlying processors and memory. Extensible modules such as inter-VM communication, security manager and fault-containment are built upon these basic modules. The Garbage Collector manages object allocation and deallocation as well as the RCU objects in SafeHype. The object adaptor controls the policies of each object. The object loader handles runtime extension of the hypervisor, which resembles module loader in Linux. All communications between the virtual machines and SafeHype are handled by the virtual machine interface. The virtual environment manager resides in a control virtual machine and provides an interface to customize and extend the hypervisor.

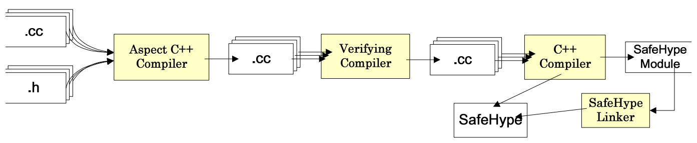

Figure 2: The compilation process of SafeHype and its modules

Figure 2 depicts the compilation process of SafeHype and its modules. Code must first be preprocessed by Aspect C++ compiler to transfer it to normal C++ files, which are then applied with soundness analysis and verification by the verifying compiler. Then

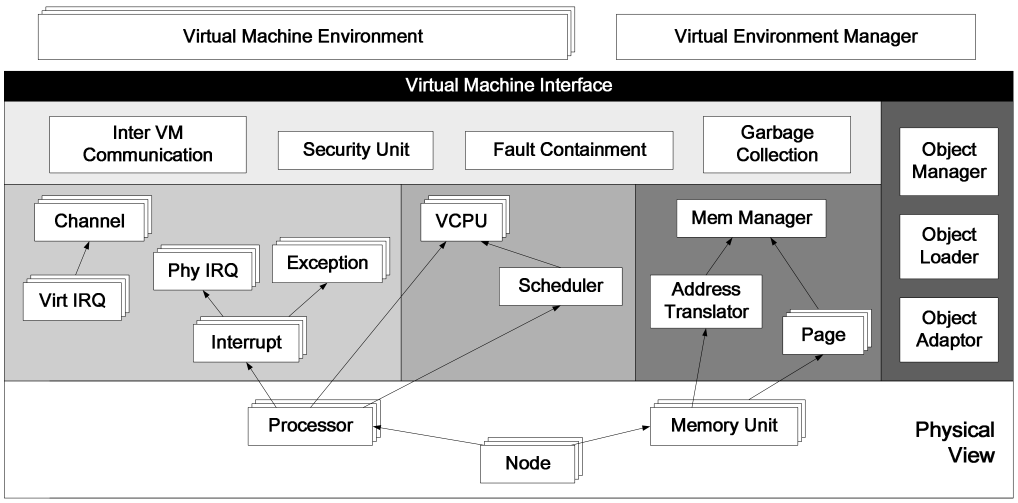

Figure 1: The general architecture of SafeHype

the code is compiled using C++ compiler into SafeHype binary or linkable SafeHype modules. The modules can be dynamically loaded into a running SafeHype to take effect. The SafeHype linker validates the module by verifying the signature embedded in the module, which was signed by the verifying compiler.

5. RELATED WORK

Other than the system designs and language advances mentioned before, a number of work has been conducted in language support for system software. Previous work can be classified into two categories: providing good substitutes for the C programming language, and exploring language techniques to operating systems. To our knowledge, we are the first to explore language innovation to increase the scalability, customizability, extensibility, maintainability and reliability of a hypervisor.

In the effort of substituting C programming language, the major concerns include type-safety and expressiveness. Ivy [13] and BitC [32] are both C-compatible extensions with type-safe and verifiable features. SysObjC [7] provides inheritance support with efficient low-level object layout to standard C. Sing# [24] is a C# extension with verifiable extension and have been used to construct the Singularity operating system. All these languages are good candidates to be used to construct our SafeHype system. However, for object-orientation features and availability reason, we adopt a type-safe C++ subset with verifiable extension to construct our system.

The benefits of object-orientation in operating systems have been heavily studied. Tornado and K42 have used various object-oriented design techniques to improve the scalability and customizability of operating systems. Singularity [24] and Coyotos [31] are both new operating systems built using type-safe languages. However, as we argued before, it is far more heavyweight to apply language innovations to operating systems than to hypervisors measured by implementation complexity and manual-effort.

6. CONCLUSION

Advances in languages, compilers and hardware make it possible and demanding to improve system software. In this paper, we have argued that object-oriented design, aspect-oriented design, type-safe and verifiable language extension could be helpful to address various issues of modern hypervisors, to suit their new role in the software stack. We have also argued that applying software innovations to hypervisors is both practical and cost-effective, due to its medium code size and little backward compatibility issues. To demonstrate the effect and applicability, we have discussed how specifically language innovations are useful in improve the scalability, customizability, extensibility, maintainability and reliability of hypervisors. As an initial effort to applying language innovations to hypervisors, we have presented the initial design of the SafeHype, our intended research hypervisor, which aimed at combining innovations in languages, verification, compilers and systems.

7. REFERENCES

[1] Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[2] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 470–481, New York, NY, USA, 2007. ACM Press.

[3] Amazon Inc. Amazon web service customer agreement. `http://aws.amazon.com/agreement/`, 2011.

[4] Z. Anderson, E. Brewer, J. Condit, R. Ennals, D. Gay, M. Harren, G. Necula, and F. Zhou. Beyond Bug-Finding: Sound Program Analysis for Linux. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[5] J. Appavoo, M. Auslander, M. Butrico, D. da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. Wisniewski, and J. Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[6] J. Appavoo, K. Hui, C. Soules, R. Wisniewski, D. Da Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. Ganger, et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[7] Á. Balogh and Z. Csörnyei. Sysobjc: C extension for development of object-oriented operating systems. In *Proceedings of the 3rd workshop on Programming languages and operating systems*, New York, NY, USA, 2006. ACM Press.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. SOSP*, 2009.

[10] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, and S. Pawlowski. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. *Technology@ Intel Magazine*, March 2005.

[11] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. OSDI*, 2008.

[12] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proc. OSDI*, 2010.

[13] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. Thirty Years is Long Enough: Getting Beyond C. In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[14] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 12–25. ACM Press New York, NY, USA, 1995.

[15] H. Chen, J. Chen, W. Mao, , and F. Yan. Daonity-grid security from two levels of virtualization. *Elsevier Information Security Technical Report*, 12(3):123–138, 2007.

[16] H. Chen, R. Chen, F. Zhang, B. Zang, and P. chung Yew. Mercury: Combining Performance with Dependability Using Self-virtualization. In *Proceedings of 36th International Conference on Parallel Processing (ICPP 2007)*, 2007.

[17] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew. Live updating operating systems using virtualization. In *Proc. VEE*, pages 35–44. ACM Press New York, NY, USA, 2006.

[18] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proc. SOSP*, pages 189–202, 2011.

[19] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proc. SOSP*, number 6, pages 351–366. ACM, 2007.

[20] S. Crosby and D. Brown. The virtualization reality. *Queue*, 4:34–41, 2006.

[21] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Patch (1) considered harmful. In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*. Usenix, 2005.

[22] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. OSDI*, pages 87–100, 1999.

[23] Y. Huang, H. Chen, and B. Zang. Optimizing crash dump in virtualized environments. In *Proc. VEE*, number 7, pages 25–36, 2010.

[24] G. Hunt and J. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proc. SOSP*, pages 207–220. ACM, 2009.

[26] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of Linux Symposium*, pages 338–367, 2002.

[27] Microsoft Corporation. Microsoft virtual server. *See www.microsoft.com/windowsserversystem/virtualserver*, 2005.

[28] Microsoft Inc. Microsoft online services privacy statement. http://www.microsoft.com/online/legal/?langid=en-us&docid=7, March
2011.

[29] E. Parsons. (de-) clustering objects for multiprocessor system software. In *IWOOOS '95: Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, Washington, DC, USA, 1995. IEEE Computer Society.

[30] C. Sapuntzakis and M. S. Lam. Virtual appliances in the collective: a road to hassle-free computing. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. USENIX Association, 2003.

[31] J. Shapiro, M. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proc. NICTA Formal Methods Workshop on Operating Systems Verification, Sydney, Australia*, 2004.

[32] J. Shapiro, S. Sridhar, and S. Doerrie. *BitC Language Specification*. http://www.bitc-lang.org/docs/bitc/spec.html, 2006.

[33] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with os clustering. In *Proc. EuroSys*, pages 61–76. ACM, 2011.

[34] X. Song, H. Chen, and B. Zang. Characterizing the performance and scalability of many-core applications on virtualized platforms. *Parallel Processing Institute, Fudan University*, 2010.

[35] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. *Proceedings of the Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications-Volume 10*, pages 53–60, 2002.

[36] A. Theurer, K. Rister, O. Krieger, R. Harper, and S. Dobbelstein. Virtual Scalability: Charting the Performance of Linux in a Virtual World. In *Proc. of Linux Symposium*, 2006.

[37] H. Tuch, G. Klein, and G. Heiser. OS Verification - Now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 7–12, 2005.

[38] VMware. The VMWare software package. *See http://www.vmware.com*, 2006.

[39] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *Operating System Review*, 2008.

[40] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proc. of USENIX'02*, pages 195–209, 2002.

[41] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor : Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. SOSP*, pages 203–216, 2011.