
When OO Meets System Software: Rethinking the Design of VMMs

Haibo Chen, Pengcheng Liu, Rong Chen, Binyu Zang
{hbchen,pcliu,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Parallel Processing Institute Technical Report
Number: FDUPPITR-2007-08003
August 2007

Parallel Processing Institute, Fudan University
Software Building, 825 Zhangheng RD.
PHN: (86-21) 51355363-18
FAX: (86-21) 51355358
URL: <http://ppi.fudan.edu.cn>

NOTES: This report has been submitted for early dissemination of its contents. It will thus be subjective to changes without prior notice. It will also be probably copyrighted if accepted for publication in a referred conference or journal. Parallel Processing Institute makes no guarantee on the consequences of using the viewpoints and results in the technical report. It requires prior specific permission to republish, to redistribute and to copy the report elsewhere.

When OO Meets System Software: Rethinking the Design of VMMs

Haibo Chen, Pengcheng Liu, Rong Chen, Binyu Zang
{hbchen,pcliu,chenrong,byzang}@fudan.edu.cn
Parallel Processing Institute, Fudan University

Fudan University, Parallel Processing Institute, PPITR-2007-08003

August 2007

Abstract

System virtualization has increasingly become a disruptive force in system research and an enabler for many commercial uses. While current VMMs partly evolve from traditional operating systems, they also inherit some problems associated with them. First, many of them are designed and optimized for small systems, yielding low scalability for large systems (many-core or large scale SMP systems) due to global policies, global data structures and large locks. Second, they are built in a monolithic and structural manner, resulting in wretched customizability and extensibility. Finally, almost all of them are built using C programming language and the code size is still growing, making it difficult to maintain the code and verify their correctness.

In this paper, we argue that advances in languages and compilers, as well as object-oriented and aspect-oriented design, could be helpful to address the scalability, customizability, extensibility, maintainability and reliability issues of modern virtual machine monitors (VMMs). The marriage of language innovations and VMMs results in an object-oriented VMM (using Sing#, an extension of C#), namely OVM, our ongoing research VMM, built with the above mentioned features in mind. We argue that applying language innovations to a VMM is more practical and cost-effective than to an operating system, due to the concern on code size and backward compatibility. We present the design decisions and initial design of OVM to show how modern programming languages and design are helpful.

1 Introduction

1.1 Motivations

System virtualization has recently gained a resurgent interest and made system research more relevant. Evidences include many systems and innovations [28, 6], as well as commercial successes [19, 27]. It is claimed that VMMs have become "the new foundation for system software" [15]¹. Many system-related researches have been conducted in VMM layer due to its global view on system resources. Meanwhile, commercial pressures also push the improvement of VMM to support new business models (e.g. virtual appliances [22]) and new applications.

Modern VMMs are partly derived from contemporary operating systems, e.g., reusing their code, and sharing their design decisions. Unfortunately, many of the design decisions "have remained unchanged, even as hardware and software have evolved" [18], thus are somewhat inadequate in the context of advances in modern architecture and programming languages. 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 [25].

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

Scalability: Recent technology advances have accelerated the prevalence of multi-core or many-core systems. Two to eight cores in a chip have already been commercially available now. Viewpoints from

¹Here, we only focus on VMMs which directly execute on bare hardware. VMMs running on a host operating system (e.g. VMWare workstation) can be similarly discussed.

Intel show that "dozens, and even hundreds of cores" will be available in the next decade[9]. The advances in processor architectures demand the VMM, the new foundation for system software, be highly scalable. However, recent measurements show that commodity VMMs (e.g. Xen) experience with low scalability due to the use of system-wide and domain-wide locks and data structures [25]. Fortunately, previous work on scalable operating systems [17, 3] has shown that object-oriented design could yield good scalability for large-scale shared memory multi-processors. It is thus promising to apply object-oriented design to improve the scalability of VMMs.

Customizability: With the prevalence of system virtualization in both academic research and industry, VMMs have been used in various usage scenarios. Meanwhile, as the global manager for computing resources, a VMM is likely to host many significantly different VMs (operating systems) and applications. Thus, it is desirable to adjust the VMM resource management policies to suit diverse applications. Global policies are not always suitable to satisfy various applications (e.g. computation 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 research work as well as industry efforts have been conducted to extend existing VMMs to gain better performance and investigate novel applications. However, only a few have been integrated into the mainstream VMMs. It should not merely be attributed to laziness. We believe two reasons cause the situation. First, mainstream VMMs inherit the monolithic and structural design from contemporary operating systems, preventing easy extensions of the VMMs. Our experiences in several projects [14, 13, 12] in extending VMMs show that a single extension could span across a couple of files and incur likely conflicting changes. Second, traditional *diff* and *patch* approach shows poor maintainability to system software [16]. To support easy integration of extensions and shorten the time-to-market of new features, it is demanding that the VMM be highly extensible and maintainable. Here, both static and online extensibility are useful to increase the flexibility and availability.

Reliability: As VMMs be gradually used in mission-critical applications, their reliability becomes vital important. Here, we focus on robustness and fault containment issues of a VMM. First, most VMMs are implemented using unsafe programming languages, risking of broking type or memory safety. One may argue that a VMM is relatively small thus easy to ensure it is bug-free via testing and code review. However, as more and more functionalities are integrated into a VMM, its code size will inevitably grow. Worse even, most VMM-extendors are less sophisticated hackers than VMM designers, integration of their code will likely degrade VMM trustworthiness. Further, the flexibility of C programming language makes it difficult to ascertain the correctness of programs [10]. To address this issue, we believe a type-safe language with verifiable extension and soundness analysis tools [2] should be helpful and applicable to increase the robustness of VMMs.

Second, the dramatic advances of hardware technology will also come with significant reliability problem [1]: high-density cores-per-chip also increase the probability of both transient and permanent failures. As the resource manager for the system platform, a VMM should be designed with awareness of fault containment, to survive it from both transient and permanent hardware failure.

1.2 Rethinking the Design of VMMs

To meet the above requirements, we believe that a reconstruction of modern VMM is necessary to suit the evolution of modern hardware and software techniques. 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 VMMs by enabling formal verification [26] and soundness analysis [2]. Second, object-oriented and aspect-oriented design should be helpful to improve the customizability, extensibility and maintainability. Third, deliberate object management (e.g. clustered objects [20], cache-aware data distribution and local locks) should be promising to improve the locality and scalability of VMMs. Finally, proper object replication and partition [11] should be helpful to improve the fault containment of VMMs. The combination of language advances and VMMs results in our ongoing research VMM, namely OVM, aiming at satisfying the described features above.

The following sections are organized as follows. The next section argues why a VMM is a good candidate to apply various novel language innovations, using the criteria of practicality and cost-effectiveness. Then, we provide the design consideration and the overall design of the OVM system. Finally, we survey literatures on various related research and then conclude.

2 Why a VMM is a Good Candidate?

Nothing of the above mentioned language techniques are new in essence. Some of the literatures have already been used in operating systems. For example, Singularity operating system [18] also employs type-safe language (i.e. Sing#) and software verification techniques to improve the dependability and trustworthiness of operating systems. Tornado [17] and K42 [3] 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 VMM using modern language innovations could be more applicable than in operating system level.

Brewer et al [10] points out that a major concern in replacing C in system software is the lack of an "evolutionary path", which can avoid efforts to 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 [18]). However, we argue for a VMM the impact of losing an evolutionary path can be mitigated at a minimal level and thus practical enough to apply language innovations.

First, a VMM is much less complex than an operating system measured in code size or implementation complexity. A VMM 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 VMM 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 VMM abstraction and management are at a much lower level. Thus, a VMM incurs a significant less complexity in code size than an operating system. Moreover, modern VMMs (e.g. Xen) further move the I/O device virtualization out of the VMM, making them even smaller. Although the code size of a VMM 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.

Second, there are little or no backward compatibility issues for a VMM. As a VMM multiplexes resources in VM-level, it generally does not directly interact with applications. Thus, dramatic changes in a VMM usually do not require any change in applications. For operating system compatibility, full-system virtualization obviously does not have compatibility issues. For para-virtualized operating systems, if the VMM 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), it is much easier to comply with.

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

3 The OVM: When OO Meets VMMs

In this section, we present the design decisions of our ongoing OVM, which uses various programming language innovations to improve its scalability, customizability, extensibility, maintainability, robustness and fault-containment. The major design decisions include using object-oriented design and type-safe language with verifiable extensions. We first present specifically the design decisions in the OVM system, then provide the overall design of the OVM.

3.1 Design Decisions

3.1.1 Programming Language

”Good languages lead to good systems” [10]. To build a robust VMM, language is thus a critical issue. Any replacement of C programming language in system software should not sacrifice its expressiveness and efficiency [10, 5], 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 Sing# [18] which is an extension to Spec# [7]. Spec# itself is an extension to Microsoft’s C#.

We choose Sing# because of its object-oriented and type-safe nature. As C#, Sing# is a garbage collected language. The difference is it can support specification constructs like pre- and postconditions, non-null types, and some facilities for higher-level data abstractions. To ensure safety, the Sing# compiler checks type safety during compilation. After that a verifying compiler is used to detect and reject suspicious errors early in the development cycle, which reduces the overhead of runtime-check.

3.1.2 Performance Scalability

Two key elements in VMM scalability are good cache locality and less lock contention. To improve cache locality, a VMM should be designed to avoid possible false sharing. To reduce lock contention, a VMM should avoid the use of global locks and data structures. As a result, OVM widely adopts clustered objects [20], by which 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. Moreover, OVM 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.

3.1.3 Customizability

The adoption of object-oriented design naturally fits the requirement for customizability. OVM relies on inheritance and polymorphism to implement customizability. OVM 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 [4] 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.

3.1.4 Extensibility and Maintainability

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

To assist the static extension or adjustment of crossing-cutting code (such as logging and debugging), OVM also supports the use of aspect-oriented programming. While Sing# don’t have existing support for aspects, aspects can be supported through language extensions[21] or using features of the existing language to specify aspect behavior and cross cutting[8].

3.1.5 Reliability

Using a type-safe language and software verification tool can significantly improve the robustness of the VMM. As the case for fault containment, OVM provides two levels of containment of failures: core-level and node level. OVM is designed with SMP and NUMA awareness. Processors with uniform memory access are considered in the same node. As in Hive [11], OVM 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 VMM replications. Within the same node, each processor monitors the liveness of other processors and communicates with each

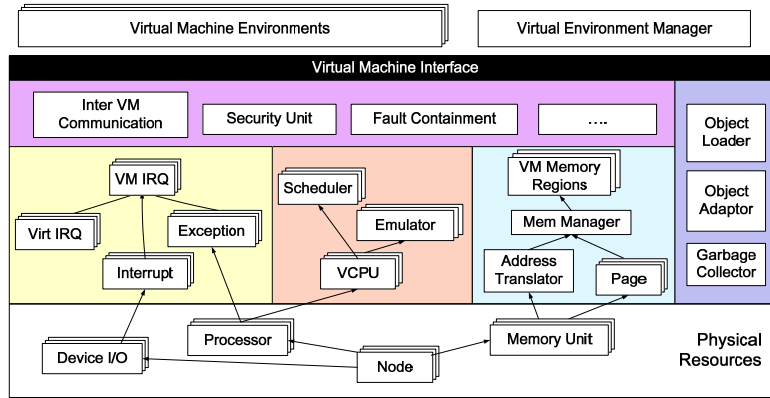


Figure 1: The general architecture of OVM

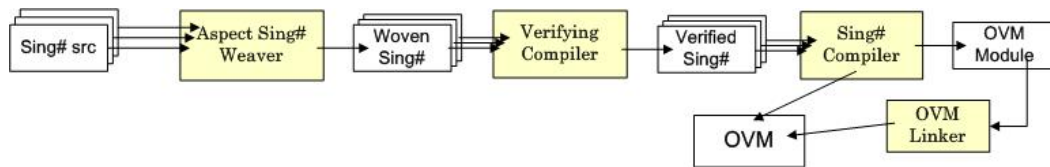


Figure 2: The compilation process of OVM and its modules

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 discard the failed processors.

3.2 The Design of OVM

OVM is our ongoing research VMM that investigates novel language innovations to address various issues with commodity VMMs. It is still in an early stage. We have made an initial design and the coding is in progress. OVM 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 OVM measurable, OVM is designed to be compatible with Xen, a popular open-source VMM. OVM 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 OVM systems. OVM is implemented in a modular manner to enable reuses of each modules. 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 in OVM. The object adaptor controls the policies of each object. The object loader handles runtime extension of the VMM, which resembles module loader in Linux. All communications between the virtual machines and OVM 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 VMM.

Figure 2 depicts the compilation process of OVM and its modules. Code must first be preprocessed by Sing# weaver to transfer it to normal Sing# files, which are then applied with soundness analysis and verification by the verifying compiler. Then the code is compiled using Sing# compiler into OVM binary or linkable OVM modules. The modules can be dynamically loaded into a running OVM to take effect. The OVM linker validates the module by verifying the signature embedded in the module, which was signed by the verifying compiler.

4 Related Work

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 VMM.

In the effort of substituting C programming language, the major concerns include type-safety and expressiveness. Ivy [10] and BitC [24] are both C-compatible extensions with type-safe and verifiable features. SysObjC [5] provides inheritance support with efficient low-level object layout to standard C. Sing# [18] 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 OVM system. However, for object-orientation features and availability reason, we adopt Sing# 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 [18] and Coyotos [23] are both new operating systems built using type-safe languages. However, as we argued before, it is far more heavy-weight to apply language innovations to operating systems than to VMMs measured by implementation complexity and manual-effort.

5 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 VMMs, to suit their new role in the software stack. We have also argued that applying software innovations to VMMs 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 VMMs. As an initial effort to applying language innovations to VMMs, we have presented the initial design of the OVM, our ongoing research OVM aiming at combining innovations in languages, compilers and systems.

References

- [1] 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.
- [2] 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.
- [3] J. Appavoo, M. Auslander, M. Butrico, D. da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenberg, 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.
- [4] 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.
- [5] 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.
- [6] 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.

- [7] B. Barnett, M. Leino and A. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*. Springer Verlag, Marseille, France, 2004.
- [8] M. A. Blackstock. Aspect Weaving with C# and .NET. See <http://www.cs.ubc.ca/michael/publications/AOPNET5.pdf>, 2004.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] H. Chen, J. Chen, W. Mao, , and F. Yan. Daonity-grid security from two levels of virtualization. *Elsevier Information Security Technical Report (To appear)*, 2007.
- [13] 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) (To appear)*, 2007.
- [14] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44. ACM Press New York, NY, USA, 2006.
- [15] S. Crosby and D. Brown. The virtualization reality. *Queue*, 4(10):34–41, 2006.
- [16] 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.
- [17] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Usenix Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.
- [18] G. Hunt and J. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.
- [19] Microsoft Corporation. Microsoft virtual server. See www.microsoft.com/windowsserversystem/virtualserver, 2005.
- [20] E. Parsons. (de-) clustering objects for multiprocessor system software. In *IWOOS '95: Proceedings of the 4th International Workshop on Object-Oriented in Operating Systems*, Washington, DC, USA, 1995. IEEE Computer Society.
- [21] M. Prasad and B. Chaudhary. AOP Support for C#. In *AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software*, pages 49–53, March 2003.
- [22] 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*, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.
- [23] 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.
- [24] J. Shapiro, S. Sridhar, and S. Doerrie. *BitC Language Specification*. <http://www.bitc-lang.org/docs/bitc/spec.html>, 2006.

- [25] A. Theurer, K. Rister, O. Krieger, R. Harper, and S. Dobbstein. Virtual Scalability: Charting the Performance of Linux in a Virtual World. In *Proc. of Linux Symposium*, 2006.
- [26] 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.
- [27] VMware. The VMWare software package. See <http://www.vmware.com>, 2006.
- [28] 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.