Virtualized Recomposition: Cloudy or Clear?

Chris Matthews, Yvonne Coady
University of Victoria
{cmatthew,ycoady}@cs.uvic.ca

Abstract

Virtualization provides a coarse-grained isolation mechanism that results in large systems, with full operating systems and a complete software stack as their foundation. Though much of this foundation is not strictly necessary, the programmatic burden of building systems at a finer-granularity, on a smaller foundation, has previously been shown to be prohibitive. The aim of this work is to revisit this tension, and present an alternative, lightweight and composable approach to virtualization that we call MacroComponents—software components that run in isolation from the rest of the system, but without the full foundations of their more traditionally virtualized counterparts. We argue that this approach will provide a more scalable and sustainable approach for composing robust services in cloud environments, both in terms of dynamic system properties and software engineering qualities.

1. Introduction

One of the challenges of cloud computing is the complexity of composing services in ways that will ultimately be not only be efficient, secure, and robust, but also scalable and sustainable in the face of system evolution. Virtualization has provided critical leverage within this domain. However, current coarse grained virtualization is arguably a mismatch in terms of cloud service compositions. Intuitively, system virtualization allows multiple virtual machines (VMs) to be run on one physical host—providing an efficient isolated duplicate of the real machine [1]. Vendors like Citrix [2], Microsoft [3], Sun [4], and VMware [5] all have successful offerings in the virtualization space, and each offer the same granularity for system decomposition, consisting of a full operating system and a complete software stack.

As an alternative, we present *MacroComponents*, designed specifically to combine the power of the isolation currently provided by virtualization with software engineering qualities borrowed from component systems for scalability and sustainability in cloud environments. In systems composed of MacroComponents, components run in isolation from the rest of the system, but without the full foundations of their more traditionally virtualized counterparts.

This paper starts with an overview of some background on the well-known advantages of virtualization (Section 2), then considers current liabilities in the context of service composition (Section 3). The requirements of an approach to address these liabilities, MacroComponents, is presented (Section 4) and some implementation considerations are discussed (Section 5).

2. Background and Related Work

Some of the well-known advantages of virtualization that make it an alluring platform for cloud computing include improved resource utilization, security, robustness and decomposition.

2.1 Utilization

With respect to the efficient use of system resources, virtualization can effectively address system bottlenecks that may arise in cloud environments. For example, given a physical machine with four processors, some services may not naturally absorb these resources. Take the case of a database running on a VM which only makes use of two processors, a second VM with a second instance of the database can then be configured to use up the remaining two processors. Applying this further throughout a cloud environment, VMs can naturally solve the underutilization of hardware resources, as their isolation means that if one VM does not fully load system, more VMs can be added to that system until it is fully utilized.

2.2 Security

The isolation provided by virtualization has security benefits. Assuming a working and trusted virtualization platform, there should be no way for resources to be unintentionally shared between VMs. In terms of a malicious attack on the system, virtualization narrows the attack area to explicitly

exposed and shared resources. In terms of privacy, there should be no way for one VM to see the contents of another. This strong isolation has been used successfully to create honeypot systems, with less risk of compromising other VMs on the same physical system [6]. Simple invariants can also be enforced on a VM. Treating the virtualized system as a 'black box', assertions can be made about the system's interactions with the rest of the world. For example, one assertion could be that a system running a web server would only accept traffic on a particular network port. Since the system's traffic can be made to pass through the virtualization platform, it can be easily checked for this property. More complex assertions and monitoring have also been shown to be possible [8].

2.3 Robustness

By spreading a software system across more than one VM, the system as a whole can be made more robust. The first way isolation makes the system more robust is by confining software failures. Ideally, a software failure in one VM should not be able to affect another VM. Although most modern virtualization platforms cannot produce perfect isolation, they do a good enough job to contain many software errors. Furthermore, if an attacker gains access to a particular VM they will still be unable to access the other VMs in that system. Isolation can also help with confinement of hardware failures [7].

2.4 Decomposition

The isolation imposed by virtualization provides a mechanism by which a system's engineer can decompose the system. For example, consider a web server and e-mail server on the same physical machine. In a virtualized environment, they may reside in different VMs on the same physical machine. This potentially separates the stakeholders of the system, and/or isolates proprietary modules from the rest of the system. Additional motivation to decompose a system along these lines can stem from issues of compatibility in legacy systems. In brief, some of these issues of compatibility include:

- legal: the license that software is written under,
- technical: two software products do not interact well when installed on the same system,
- security: not trusting other software on the system,
- quality of service: can only guarantee software will work well without other software installed.

Another important contribution of this separation is that it potentially better matches the conceptual understanding of the system. Since each server provides a separate service, having them running on separate platforms allows a coarse-granularity of reasoning along the lines of separation of concerns. When the maintainers of a system are reasoning about the services it provides, their reasoning need not include details of where the services instantiated. For example, if the e-mail server was moved behind a firewall, but the web server was not, virtualization would support this conceptual model; however, this is not the case if the e-mail server and the web server running on the same operating system.

Similarly, developers targeting cloud computing need the same ability to reason at this high level. For example, in the Amazon elastic compute cloud [10], VMs are the unit that can be purchased for computing. Although users do not know the actual structure of the system (and may never be told), they are able to purchase an amount of computing power in the form of a VM that has some physical parameters such as the number of processors in the amount of memory. Purchasing time in these logical units does not have any correspondence to the physical resources that Amazon is actually offering, and this transparency is an important feature of the model provided.

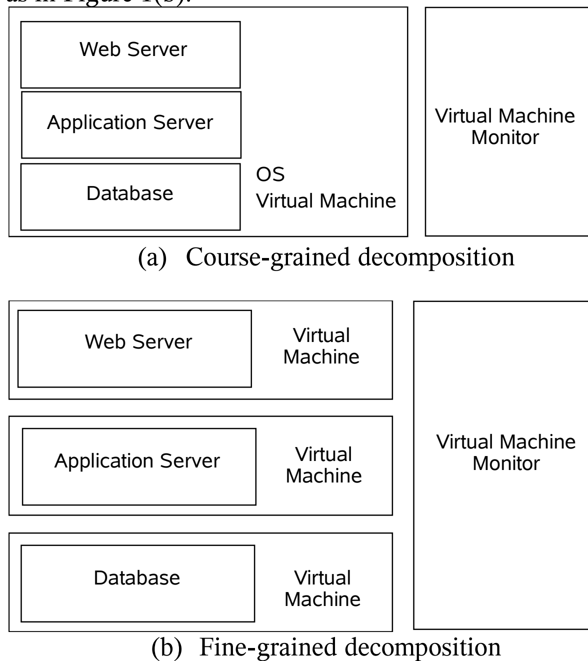
Given all these motivating factors, it is not surprising that the notion of a *virtual appliance* has been gaining in popularity in recent years. A virtual appliance is a VM with software prepackaged and preconfigured. All the user of the virtual appliance has to do is download the virtual appliance and start it, adopting all of the advantages of virtualization implicitly.

There are many more advantages to leveraging virtualization for decomposition. In some systems, VMs are able to dynamically migrate between hosts. Additionally, *snapshotting* enables the system to make a copy of the live VM state so that it can be restored later or archived. Flash cloning is an extension of snapshotting that allows the system to instantiate snapshots very quickly. In Potemkin [6], flash cloning was used to instantiate a new VM for every socket connection a machine made. Commercial cluster control platforms are also available, in which virtualization creates a number cluster nodes per physical machine [11].

2.5 An Example: Web Services

In order to further consolidate the argument for decomposition, consider, for example, a web services stack. One possible configuration of modern web services stack could be a web server coupled with an application server and database. All three applications can be in one VM on a physical machine, as in

Figure 1(a), or each element could be in a separate VM as in Figure 1(b).



(a) Course-grained decomposition

(b) Fine-grained decomposition

Figure 1: Decomposition leveraging virtualization

The system's stakeholder will reap the expected benefits from decomposing the system:

- Isolation: it would be harder for the three elements of the system to unintentionally affect each other. There are very few shared resources between the elements, and those that are shared (memory, CPU, disk) are all tightly controlled by the VM monitor to ensure fairness.
- Security: all of the communication with the database's VM would take place through the standard TCP database communication port. All other communication with the databases domain could be disallowed with a firewall.
- Privacy: controls can be placed on the database interface to assure that private information is not leaked. This should be the only way the information could leave the system.
- Migration and snapshotting, as previously described.

3. Composition: Virtualization's Liability?

Assuming these benefits could hold true for even finer-grained elements of the system, further decomposition could be desirable. For example, the web sever could be further decomposed to remove its request handlers and authentication mechanisms so that the web sever could benefit from their isolation.

We believe the costs associated with decomposition based on virtualization break down into two categories: (1) scalability in terms of dynamic system properties, and (2) sustainability in terms of software engineering challenges of constructing service composition.

3.1 Scalability: Dynamic Compromise

One observation that arises from Figure 1 is that the overall time to satisfy a request coming into the system may increase. This is because the communication between VMs is not as fast as the communication within a VM. Similarly, there are other costs associated with the differences between a virtualized duplicate and the physical machine. Timing of events may differ, and resources presented to any single VM may be reduced as a result of sharing and contention. Thus, although VMs can be functionally equivalent to their physical counterparts, they will display some different behavioral characteristics, particularly at a large-scale.

Additionally, in practice, isolation provided by VMs is incomplete. Modern VM monitors compromise between efficiency and complete isolation. Though the degree to which a VM monitor isolates VMs varies amongst types of virtualization platforms and implementations, each approach incorporates its own built-in set of assumptions that come into play when VMs are used as finer-grained building blocks within a larger system composition. The impact of these characteristics are also compounded at scale. An example of initial experiments to better quantify these costs can be found in [12]. These are just some of the dynamic costs that the stake holder would have to take into consideration when deciding to leverage virtualization as a more fine-grained approach to system composition.

3.2 Sustainability: The Burden of Composition

Though decomposition is effective in terms of separation of concerns, virtualization actually does not support a similarly nice model for composition. This is at least in part due to the coarse granularity of the approach. For example, given the example in Figure 1(b), instead of installing one operating system, the stakeholder would have to install and prepare an operating system for each element, tripling that work in this example, and possibly tripling licensing fees as well. Each element would have to be individually secured and set-up to perform the task required. The configuration of each element's VM may have to be different as well. Each VM would have to be setup to communicate, and be given a mechanism to do so

with. Enforcement of system policy would have to be done with virtualization unaware OS mechanisms.

At scale, these costs are potentially prohibitive, and instead we suggest a lightweight model based on first principles for componentization.

4. MacroComponents: Requirements

Component systems also allow developers to build programs from scratch out of compositions of components which they further customize. Component systems allow replacement of existing parts without requiring major changes to the rest of the program. Characteristics that a component model needs to function effectively include componentization, composition, communication and control. Here we discuss the requirements of a virtualized component model in these terms.

4.1 Componentization

One important issue with component design is finding the appropriate component size. This could also be referred to as the granularity of the system. In comparison to a VM running a full OS, we want to encourage a relatively small component size. Small component size would allow for the minimal foundation needed to run a component instead of a full OS. There are at least two factors that could affect our component size: first, the computational overhead of components, second, the development burden of the components. There are also more pragmatic concerns in the design of a component system; for example, how the components are packaged and how resources and information shared between components. These are things that still need to be addressed.

4.2 Composition

The second characteristic of a component system is how components are combined to form a working system. In most component systems there is a simple programmatic way in which you can reference and then invoke a component. Some systems have dynamic models in which components can be found at runtime then invoked. Ideally our system would have programmatic dynamic composition of components. Some of the component model design points in this area include:

- the means by which components are referenced,
- the control of the life-cycle of the components and
- their customization within anticipated parameters.

Conventional component models provide insights into how these design points could be satisfied. However, one design point that conflicts with modern component systems is the question of where the composition actually happens. In modern virtual architectures the control and creation of VMs happens outside of the normal system in a special 'privileged' area. This mismatch in the location of control has to be address in a component system design.

4.3 Communication

Most modern component systems are designed to run in a single address space, so communication is as simple as a function call. Inter-process communication (IPC) and middleware are more representative of the types of communication a virtual component model would have. They are more representative because they send communication through some sort of shared medium like shared memory or a network instead of being able to directly invoke a function or access data.
Some of the design points that have to be address with respect to communication include:

- the medium used communicate between VMs,
- the way entities in the system named and registered,
- the interface if typed communication is required,
- communication semantics and
- the identification of essential services.

4.4 Control

The mechanisms for controlling the communication and behavior of components should follow the key design principle aligned with the separation of policy and mechanism. A good system design should allow the application designer to specify component's policy, and have the system follow though that policy with mechanism. As a result, some of the design points that have to be addressed with respect to control within a system that uses VMs for fine grained decomposition are:

- definition of the mechanisms necessary to correctly control a component,
- how and where are these mechanisms controlled from, in terms of system decomposition and
- where the policies of components is implemented.

4.5 Component Models

Although we know of no systems that target composition of virtual machines, there are several interesting solutions to composition in general. The OSGi Framework [15] is a component model that is

primarily targeted at Java. Besides the requirements mentioned above, the MacroComponents framework could be designed around the OSGi Framework and provide a similar model for composition of VMs. OSGi could likely lend solutions to several of the problems outlined in this section.

Similarly, systems like WSO_2's Carbon platform [16] might be able to be used to create a minimal operating environment for individual MacroComponents.

5. Implementation Considerations

There are a number of implementation challenges that would have to be addressed before using virtualization as a composition mechanism would be possible. A few key challenges include:

- Changing the centralized control mechanisms employed in current virtualization platforms.
- The scale of the system in terms of the number of VMs and the amount of inter-VM communication may need to be addressed.
- The speed with which VMs can be created, and the latency of inter-VM communication needs to be addressed.

The following details how we have begun addressing the VM creation problem.

5.1 Virtual Machine Creation

The process of creating a new VM on a system is a relatively heavy weight operation. The virtual machine monitor (VMM) has to allocate the necessary memory for a VM, as well as prepare any resources that it will share with the rest of the system. In operating systems, one commonly used optimization to increase the speed of process creation and reduce overall memory consumption of the system is copy-on-write [13]. Copy-on-write is a technique in which two parties who have similar memory to each other transparently share common memory pages instead of duplicating them. Only when a shared copy is changed, is it no longer common to both parties. Instead, the divergent local copies are maintained independently.

Copy-on-write is one feature needed in a system to have VM creation via a fork/clone like model, mimicking process creation in operating systems. A checkpointing mechanism is used to provide a VM level fork. As with processes, this allows for fast creation of VMs. Closely related would be creating a VM clone, a VM created from a pre-described image. VM cloning has been used successfully in Potemkin [6] to quickly provide a VM to service a single request in a system. In theory, a system like this benefits from the isolation properties of VMs on a very fine grained level.

Ultimately, a copy-on-write facility would decrease the memory usage of a system running similar code bases in separate VMs, and decrease the time it takes to create those VMs. In a system creating and running lots of MacroComponents with similar base code, a fork/clone facility is a valuable optimization while still maintaining strong isolation between VMs.

5.2 A Copy-on-write Subsystem

Copy-on-write is implemented by having two separate address ranges in virtual memory point to one common set of physical pages. In the common case, when a process reads from those virtual addresses, it is able to access the shared pages as if they are its own. However, the pages are marked as read-only by the system, so if a process writes to them, the memory management unit (MMU) triggers a page fault, and the operating system's virtual memory subsystem is able to deal with the resulting trap. In the fault handling code, the page is duplicated, and one copy is assigned to each process. The procedure is transparent to both processes using the data.

Unfortunately, preserving copy-on-write semantics across VMs is more complex than it is in operating systems, and consequently more difficult to implement. There are several domains of control, and each has to participate to successfully implement a copy-on-write within a target VM. Some of the points we considered in this implementation are outlined as questions below.

- Where does the spare memory come from to duplicate the written pages? And, which VM of the forked VMs gets the copy (and experience memory fragmentation)?
- How will this work with different memory models? For example, Xen [14] has several different memory models that all work slightly differently.
- How is a consistent image of memory possible when it is possibly changing as you read it?

5.3 Design Decisions

We have started to develop a copy-on-write subsystem for Xen. The subsystem is centralized around a user space library that sits in Domain 0 (the Xen control VM). From there, the library is able to interface with the Domain 0 kernel to allocate memory, map memory from the target domain and interface with the hypervisor. This last step involving the hypervisor is

necessary to get copied pages and perform administration activities like starting and stopping a particular copy-on-write. To address some of the questions listed above we have made several design decisions, and assumptions.

- The copy-on-write subsystem is designed to run only on 64-bit HVM VMs. This allows us to focus on one memory model.
- When duplicating the faulting page, the snapshot gets the duplicate page. This keeps the faulting VM's memory contiguous.
- To provide a consistent image, the user-level library will not give its clients direct access to the VM's memory, but rather access through an interface. Behind the interface, the copy-on-write subsystem will synchronize the requests for memory with the live memory state of the VM. If high performance is needed, the clients can circumvent the library's synchronization, but will risk seeing inconsistent memory images.

6. Conclusions

Though virtualization improves system utilization, security and robustness, possible liabilities that surface as we attempt to leverage virtualization for software recomposition include: (1) scalability in terms of dynamic system properties, and (2) sustainability in terms of software engineering challenges of constructing service composition.

MacroComponents offer a lightweight container for software components that run in isolation from the rest of the system, but without the full foundations of their more traditionally virtualized counterparts. By reducing the foundation upon which virtualization is built, and by incorporating first principles of component based software development, this approach can provide a more scalable and sustainable approach for composing robust services in cloud environments.

Acknowledgments

We would like to thank Stephen Neville, Andrew Warfield, and Jonathan Appavoo for their invaluable feedback, and many helpful comments and contributions to this paper.

7. References

[1] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[2] Citrix, “Citrix XenServer 5: Virtualization for every server in the enterprise,” Website, 2008, http://www.citrix.com/English/ps2/products/product.asp?contentID=683148.

[3] Microsoft, “Microsoft Virtualization: Home,” Website, 2008, http://www.microsoft.com/virtualization/.

[4] Sun, “Sun Virtualization Solutions,” Website, 2008, http://www.sun.com/solutions/virtualization/.

[5] VMware, “VMware: Virtualization via Hypervisor, Virtual Machine & Server Consolidation,” Website, 2008, http://www.vmware.com/.

[6] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M.Voelker, and S. Savage, “Scalability, fidelity, and containment in the potemkin virtual honeyfarm,” in *SOSP ’05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2005,pp. 148–162.

[7] T. C. Bressoud and F. B. Schneider, “Hypervisor-based fault tolerance,” in SOSP ’95: Proceedings of the fifteenth ACM symposium on Operating systems principles . New York, NY, USA: ACM, 1995, pp.1–11.

[8] Dunlap, G. W.; King, S. T.; Cinar, S.; Basrai, M. A. & Chen, P. M. “ReVirt: enabling intrusion analysis through virtual-machine logging and replay” *SIGOPS Oper. Syst. Rev., ACM,* 2002*, 36*, 211-224

[9] J. Chapin, “Hive: Operating system fault containment for shared-memory multiprocessors,” Stanford University, Stanford, CA, USA, Tech. Rep., 1997.

[10] Amazon inc, “Amazon Elastic Compute Cloud”, Website, 2008, http://aws.amazon.com/ec2/.

[11] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. “Large-scale Virtualization in the Emulab Network Testbed.” *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008.

[12] C. Matthews, Y. Coady. S. Neville, “Quantifying Artifacts of Virtualization: A Framework for Mirco-Benchmarks” in QuEST '09: *The 2009 IEEE International Workshop on Quantitative Evaluation of large-scale Systems and Technologies*, *IEEE*, 2009

[13] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP’03*, 2003, pp. 1–14.

[15] OSGi Alliance, “OSGi - The Dynamic Module System for Java”, Website, 2008, http://www.osgi.org.

[16] WSO2 Inc “WSO2 Carbon”, Website, 2008, http://wso2.org/projects/carbon.