Scalable Elastic Systems Architecture

Technical Report BUCS-TR-2012-008

Jonathan Appavoo and Dan Schatzberg, Boston University
{jappavoo,dschatz}@bu.edu

This material is based upon work supported in part by the Department of Energy Office of Science under its agreement number DE-SC0005365 and upon work supported in part by National Science Foundation award #1012798.

Scalable Elastic Systems Architecture

Jonathan Appavoo & Dan Schatzberg, Boston University
{jappavoo,dschatz}@bu.edu

Abstract

Cloud computing has spurred the exploration and exploitation of elastic access to large scales of computing. To date the predominate building blocks by which elasticity has been exploited are applications and operating systems that are built around traditional computing infrastructure and programming models that are in-elastic or at best coarsely elastic. What would happen if application themselves could express and exploit elasticity in a fine grain fashion and this elasticity could be efficiently mapped to the scale and elasticity offered by modern cloud hardware systems? Would economic and market models that exploit elasticity pervade even the lowest levels? And would this enable greater efficiency both globally and individually? Would novel approaches to traditional problems such as quality of service arise? Would new applications be enabled both technically and economically?

How to construct scalable and elastic software is an open challenge. Our work explores a systematic method for constructing and deploying such software. Building on several years of prior research, we will develop and evaluate a new cloud computing systems software architecture that addresses both scalability and elasticity. We explore a combination of a novel programming model and alternative operating systems structure. The goal of the architecture is to enable applications that inherently can scale up or down to react to changes in demand. We hypothesize that enabling such fine-grain elastic applications will open up new avenues for exploring both supply and demand elasticity across a broad range of research areas such as economic models, optimization, mechanism design, software engineering, networking and others.

Contents

1 Cloud Computing Systems Software Gap

Current systems software, burdened by legacy compatibility, is complex and does not have the right abstractions to enable the pervasive parallelism or elasticity offered by the modern datacenter-scale systems that form cloud platforms. Additionally, systems software is not designed for service oriented cloud applications that can dramatically expand and contract resource consumption across hundreds, if not thousands, of computers in minutes or even seconds. Collectively, we refer to the lack of support for cloud hardware and usage as the cloud computing systems software gap. This gap and the benefits of addressing it have driven researchers at Berkeley to also observe that "The Datacenter needs an Operating System"[53]. This paper discusses our method for filling the gap.

Hardware is Different Cloud computing hardware is evolving into consolidated extreme-scale systems. These systems are hybrids of chip-level multiprocessor nodes[1] and massive clusters composed of tens of thousands of such nodes. The nodes themselves will soon contain hundreds to thousands of cores. The cluster networks will be supercomputer like interconnects capable of micro-second latencies, with tens to hundreds of gigabits of bandwidth per node, and integrate advanced features like Remote Direct Memory Access (RDMA). Additionally, cloud hardware is elastic, permitting software to efficiently change the fraction of system resources it executes on in seconds. The platforms on which today's system software developed are much smaller in scale, typically solely shared memory or message passing in nature and not elastic.

Usage is Different Cloud Computing applications are service oriented in nature, focusing on scale and elasticity. Applications are characterized by transaction processing that is triggered by dynamic requests to an application service interface. Maximizing the scale and efficiency that an application can execute at ensures that it can satisfy the largest possible number of concurrent requests for service. Given the elastic nature of the hardware and usage based billing, applications are driven to exploit elasticity in requests to minimize cost. Applications that today's system software were originally designed to support rarely focused on extreme scales and did not have the ability to dynamically change the scale and composition of the hardware.

This research explores the cloud computing systems software gap through a unique hybrid system software model that acknowledges and reflects the structure of cloud computing hardware, usage and realities of application development. The intent is not to produce a single new operating system, but rather to explore an architecture for developing and deploying scalable and elastic applications without a homogenous OS.

[1] A node refers to a unit of hardware that contains one or more micro-processors, local memory and communication hardware – examples include a single computer of a cluster, a single blade in a blade center or a compute card in a supercomputer. In the systems I consider, a node is the maximum boundary for hardware supported shared memory. In a virtualized environment, a node refers to a Virtual Machine (VM) instance.

Approach

Our approach, derived from several years of our research (described in the next section), is a new scalable elastic systems architecture (SESA). SESA combines in a novel way three old ideas, object-oriented programming, event-driven programming, and library operating systems, to explore the cloud computing system software gap. More specifically, the SESA work explores these three ideas:

- The use of a distributed object abstraction, rather than shared memory or message passing, as the programming model. The goal is to address the challenges and tensions associated with developing scalable and elastic software on evolving data-center scale hardware while providing a degree of isolation from hardware complexity and details.
- The use of an event model across all layers to expose and enable elastic resource usage and management.
- A new distributed library OS model that factors out support for scalable and elastic applications from a traditional OS, alleviating the burden for the construction of a complete scalable and elastic OS.

Object Orientation Our prior work uniquely established that object-orientation, typically used for software engineering benefits, can be exploited for scalable and elastic performance. Objects can provide an effective programming abstraction, over shared memory and message passing, to optimize the parallel runtime structure induced by software (layout on physical resources and induced communication patterns). Objects, when combined with an appropriate event model, can lead programmers to develop software that 1. exploits partitioning, replication and distribution in implementation, 2. defines a model for scaling resource consumption based on demand, 3. moves communication from hot-paths, and 4. adapts to workload changes.

Event Model Our prior work established that an event-driven programming model combined with object-orientation can be used to develop software that scales in response to changes in workload and system size. Specifically, methods of an object are event handlers that respond to accesses on particular processors. The processor associates the event with a particular set of physical resources on a specific node. In this way, all software becomes elastic with respect to system size. As an object is accessed in a different location, it must define how to consume that location's resources. A runtime for this type of event-driven object model, through appropriate resource allocators and communication primitives, can map the objects and their operation efficiently onto the physical structure and communication features of the hardware.

Library Operating Systems Library Operating Systems (LibOSs) were proposed as a way to isolate an application and its dependence on a particular legacy OS interface into a stand-alone portable package. This is achieved through a library that provides the OS interface and is linked into the application binary. Under the covers, the library can provide an alternative implementation of the legacy OS calls. This includes translating them into calls to a new underlying OS that exports a different interface. For example, a LibOS might provide an application with legacy `read` and `write` calls while implementing the functionality in the library with calls to a new underlying OS's memory mapped IO calls [32]. At IBM, our research group exploited a variant of this technique to support an alternative OS interface across multiple nodes of a data-center scale system. SESA leverages

this design point to enable new SESA applications to be written without the need for an entire new OS to be developed. Furthermore, the technique allows applications to be developed, launched, managed and even use features from an existing legacy OS.

Our goal is to construct a SESA prototype and establish, through experimental evaluation, its ability to enable a set of applications to efficiently exploit the parallelism and elasticity of a large scale cloud computing platform. In the next section we discuss in more detail the research background for SESA and its evolution.

2 Background

The SESA research agenda has developed in a rich systems research context. In this section we briefly describe some of the relevant system software research, including our own, that strongly influences the design of SESA.

Shared Memory MultiProcessor OS Services: Around 2005, despite continuing growth in transistor count, the industry reached a plateau in the speed at which a serial sequence of instructions is executed [13]. New processors have and continue to exploit the increasing number of transistors by increasing the number of cores, and future improvements in application performance depends on exploiting multi-core parallelism. This ensures that cloud platforms will have rising core counts. This change has resulted in a re-vitalization in OS research focused on multiprocessor scalability [12, 14, 28, 48]. A common trend is to achieve scalability by building an OS that relies on message passing, rather than shared memory, to communicate system state between cores.

While the ubiquity of multi-core processors has resulted in this recent interest, there is a rich history of research in multi-processor OS scalability [49, 38, 42, 44, 45, 47, 18, 16]. We were heavily involved in projects that pre-dated and influenced much of the current work in this area [8, 10, 31, 3, 5, 7, 26].

We found that when constructing the Tornado and K42 operating systems for scalable Shared Memory MultiProcessor hardware, performance depends on both matching the demands of the applications and the communication characteristics of the hardware. Through experience, we found it necessary to exploit the semantics of individual software components, the load they were under, and the hardware attributes to achieve scalable performance. We concluded that there was no single solution for all software components or hardware structures but, rather, achieving scalable performance was a runtime mapping problem. To this end, we adopted a *partitioned object* programming abstraction rather than pure shared memory or message passing.

Partitioned Object Models In the *partitioned object* model [50], an externally visible object is internally composed of a set of distributed *Representative* objects. Each Representative object locally services requests, possibly collaborating with one or more other Representatives[2]. Cooperatively, all the Representatives implement the complete functionality of the object. To the clients, the partitioned object appears and behaves like a traditional object. The distributed nature make them ideally suited for the design of scalable software, which often requires a high degree of modularity and yet benefits from the sharing, replicating and partitioning of data on a per-resource

[2]Using the system provided communication models and facilities such as shared memory and message passing

(object) basis. Several examples of partitioned object models have been developed [34, 15, 46, 29]. We developed Clustered Objects [5] as a partitioned object model to simplify the task of designing and developing high-performance software for scalable multiprocessors.

SESA will explore a new partitioned object model called Elastic Building Blocks (EBBs) that targets cloud platforms which combine shared memory and message based high-performance interconnects.

Library OSs: Library OSs factor out the majority of an OS's application interface and function into a library that executes directly in the address space of an application. OS research in the 1990's explored the library OS model [2, 19, 24, 33] as a way of introducing application specific customization and performance optimizations. In this model, an application only interacts with the operating system function via the library's provided implementation. Thus each application could have a library implementation that was tuned to its needs. The libraries themselves, when necessary, interact with an underlying operating system via more primitive and restricted interfaces. Drawbridge, a Window's 7 library OS by Microsoft Research, is a more recent example of how a library OS approach can support complex commercial legacy applications and rapid system evolution [40].

While at IBM, we were part of the team that built the Libra Library OS [1]. Libra was unique in the use of the library operating system model. Libra was used in a multi-node environment to optimize a distributed application. Rather than supporting a standard OS application programming interface (API), our library supported an interface that integrated with the internals of IBM's production Java Virtual Machine (JVM). Libra booted a node directly into the application, dramatically decreasing the time needed to add or reduce the nodes used by the distributed application.

The execution of the `java` command on a front-end node transparently caused a new instance of the Libra JVM to be launched on other nodes. On the front-end, the Libra JVM instance appeared in the process list and could be killed like any other local process. Additionally, the Libra Library OS exploited the front-end to offload legacy function such as TCP/IP connectivity to exterior networks and file system access. This proved to be a very powerful way of integrating a new advanced distributed application with a legacy OS. The Libra work influences the Library OS model we are pursuing with SESA. It provides a natural path for introducing SESA based applications that aggressively exploit scale and elasticity while usefully leveraging existing software.

Supercomputers To date, cloud computing has focused on commercial models of computing based on general-purpose workloads and networking. It has achieved standardization and high levels of elasticity through virtualization but allows little or no direct hardware access, hinders specializations, and limits predictability and performance. In contrast, scientific computing has exploited specialized infrastructure, including non-standard communication networks, to achieve predictable high performance through direct hardware access and physical dedication but with little or no elasticity.

Amazon's EC2 and Microsoft's Azure are examples of cloud computing offerings available today. Although little is published about the internal structure of the platforms, we do see a trend in the products targeting cloud computing, such as SGI's CloudRack^{TM}[20], which exploits consolidation to integrate up to 1824 cores per rack. Such racks form the basic unit by which cloud datacenter systems are constructed. While such systems still rely on standard external Ethernet switch infrastructure for connectivity, a trend of offering advanced supercomputing like interconnects such

as InfiniBand™[30], Fibre Channel[25], and NUMAlink™[35] can be observed.

While at IBM Research, we founded and developed Project Kittyhawk[9, 4, 11]. The key idea behind Project Kittyhawk was to exploit data-center scales and hardware consolidation to enable support for fine grain physical resource allocation and deallocation that was not tied to shared memory but could still exploit high performance communication interconnects. We exploited features found on super-computer class machines to provide a global-scale Hardware as a Service (HaaS)[51] model on which a user could acquire and release physical resources in sub-second time scales. Given an associated ability to meter the resources, this system could enable a cloud computing system on which users could construct services that would be able to scale resource consumption on time scales associated with interactive use.

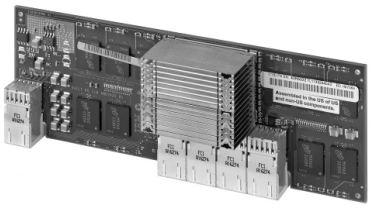

Figure 1: BG/P Node

Using the Blue Gene/P hardware we constructed a prototype system that was composed of control systems software and associated prototype OS and application software. Our model proved to be very useful for several projects internal to IBM. Since joining Boston University, we have worked with IBM to open source the project and install it on a hardware system at Argonne National Labs. The system is now being used by several research groups and continues to be an important experimental testbed for our work. The following highlights the consolidation, scale and capacity provided by the Blue Gene/P platform. The basic building block is a node (Figure 1) composed of a quad-core processor with five integrated network interfaces and RAM. The networks include partition-able tree, 3D torus and dedicated management networks. A Blue Gene/P rack provides a total of 1024 nodes, up to a total of 4 TB of RAM and 640 Gbit/s of external IO bandwidth. The system supports up to 256 racks totaling over 1 million cores, 1 Petabyte of RAM and 10.4 Petabit/s of aggregate external bandwidth. All of the internal networks seamlessly extend across all racks of an installation.

Cloud Computing Infrastructure: In 2006, when studying data-center scale systems for cloud computing[9, 4, 11, 51], we observed four trends: 1) many new applications are constructed around a service model that is very similar to our prior model of Shared Memory MultiProcessor OS services, 2) new applications are typically being written against user-level runtimes such as Java, Ruby, Python, and others that naturally separate them from the underlying OS, 3) predominately, a single application process is executed on a node, and 4) data-center scale systems can exploit large scale consolidation to offer advanced hardware features for communication and elasticity making them a hybrid between a shared memory multiprocessor and a cluster. The Kittyhawk experience exposed an opportunity for a new OS runtime that could enable service oriented applications to be better mapped to the scale and elasticity of cloud data-center systems. If the fine grain elasticity of the hardware could be exploited by the software, then resources could migrate between services without the need for complex multiplexing in software. This would alleviate the need for services to hoard resources for the sake of over-provisioning.

Since 2006, several models for elastic applications have developed: Google AppEngine[27], Ruby On Rails[41] and other web frameworks, DryadLINQ[52], Sawzall[39], Pig[36], MapReduce[21], Dynamo[22], BigTable[17]. However, all of these rely on a coarse grain model of multiple nodes executing a complete legacy OS instance. They are structured as a collection of TCP/IP based servers that form a traditional middleware distributed system. This means that they, along with any applications, can at best exploit the elasticity and communications afforded by the OS they

run on. SESA attempts to integrate support for elasticity into the system layers thus enabling applications to exploit the underlying elasticity of the hardware platforms.

Summary

In this section we have presented the major work that influenced how we came to identify the cloud computing software gap and the SESA approach to exploring a solution. Both have come from several years of our prior work in Shared Memory Multiprocessor OSs, Partitioned Objects, Library OSs, Supercomputers, and Cloud Computing Infrastructure. The timeliness and need for this work has recently been articulated by Zaharia et al. [53]:

> Datacenters have become a major computing platform, powering not only popular Internet services but also a growing number of scientific and enterprise applications. We argued that, as the use of this new platform grows, datacenters increasingly need an operating system-like software stack for the same reasons that single computers did ... This kind of software stack is already emerging in an ad-hoc manner, but now is the right time for researchers to take a long-term approach to these problems and have a lasting impact on the software infrastructure for this new computing platform.

The SESA approach is presented next.

3 Scalable Elastic Systems Architecture

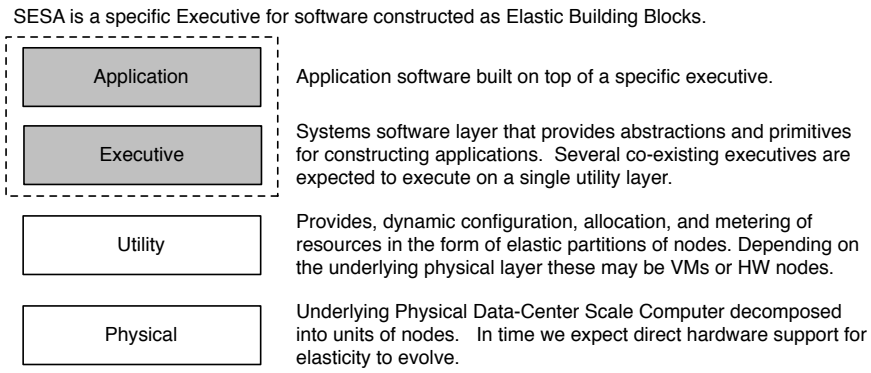

Figure 2: Generic Context for SESA

The Scalable Elastic Systems Architecture (SESA) is designed within the context of a generic four layer model of cloud computing, illustrated in Figure 2. SESA investigates the top two gray shaded layers of the figure, namely a specific architecture for an executive layer and its associated application model.

In practice, the SESA work will result in a library OS that forms a distributed executive. The set of nodes obtained from the utility layer on which the executive runs can be expanded and contracted by the application. The executive provides a programming model that combines a distributed component and an event model for the construction of scalable and elastic software. Software is formed out of dynamically allocated instances of components called Elastic Building Blocks (EBBs) that respond to events.

The EBBs themselves are distributed entities. That is to say that an EBB is accessible and available on any node that currently forms the executive. However, internally, the EBB may be composed of one or more representatives that are located on specific nodes. When an EBB is first accessed on a new node, a new access event is generated and the EBB must be programmed

to respond appropriately based on internal knowledge of its semantics. The executive, exploiting hardware features, provides primitives for the internal representatives of an EBB to efficiently locate each other and communicate. A prebuilt library of EBBs will provide the primary interface to all systems features. Additional libraries of EBBs will be used to create application runtimes and applications themselves.

The SESA research will be focused around the construction of three example applications:

1. An elastic resilient data-store (based on our prior work on scalable hash tables).
2. A Java Virtual Machine (port of openJDK[37]) along with a commercial Java application.
3. A Matlab like numerical computing environment (port of Octave[23]) along with a Matlab based neuro-medical imaging application.

The goal is to evaluate SESA with respect to the scale and elasticity the applications can ultimately achieve but also with respect to SESA's ability to enable the development of these real-world applications.

While SESA is not fixed to a specific utility or physical layer, the work will target advanced scale and features based on system trends. Using our prior work on Project Kittyhawk, we will be able to develop and test SESA on two IBM Blue Gene/P Systems, one composed of 4096 cores and 4 terabytes of ram and the other 163840 cores and 175 terabytes of ram both located at Argonne National Labs. To ensure generality and ease development we will, with our collaborators, maintain versions of SESA on standard and commercially available virtualization based cloud platforms.

Given Project Kittyhawk and Blue Gene systems, this work can legitimately target and evaluate very large scale future cloud platforms. We consider a physical layer that:

- contains in excess of 40×10^3 nodes and 160×10^3 cores,
- exploits consolidation and scale in manufacturing to optimize total cost of ownership, and
- integrates advanced features with respect to power, cooling, density, communications, configuration and metering.

With respect to the utility layer, Kittyhawk lets us explore a utility system capable of the following:

- Sub-second to second latencies for scalable node allocation. That is to say the utility layer, exploiting features of the physical layer, can expand an allocation of nodes by one to several thousands of nodes within a second.
- The utility layer can provide precise control over how a node is configured with respect to the other nodes it is permitted to communicate with.
- Finally, the utility layer permits access to advanced communication features of the physical system such as hardware supported remote direct memory access, multi-cast, and precise local and remote interrupt steering.

In the next section, we discuss various aspects of SESA in more detail.

4 Discussion

In this section we discuss in more detail how SESA combines object orientation, an event model and a library operating system approach.

4.1 Object Orientation

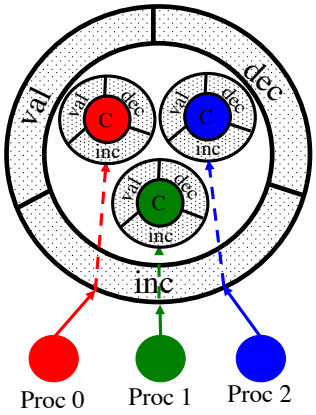

Figure 3: Abstract Simple EBB Distributed Counter

As discussed, one of the approaches that SESA uses to address the cloud computing system software gap is to introduce a programming abstraction above shared memory and message passing that enables programmers to develop scalable and elastic software. This abstraction comes in the form of Elastic Building Blocks whose internal structure can be physically distributed in the systems resources and whose structure is lazily constructed in response to events.

Figure 3 abstractly illustrates an EBB of a simple distributed integer counter. A single instance of the counter, represented by the outer ring labeled with the counter's interface (`inc`, `val` and `dec`), is accessed by code executing on the processors at the bottom of the diagram. All processors invoke the `inc` method of the instance. Transparent to the invoking code, the invocations are directed to internal per-processor Representatives, illustrated by the three inner rings in the diagram. Each Representative supports the same interface but encapsulates its own data members. This ensures that the invocation of `inc` on each processor results in the update of an independent per-processor counter, avoiding sharing and ensuring good increment performance. In this way, EBBs permit distributed data structures and algorithms to be used to minimize communication overheads on hot-paths, thereby optimizing parallel performance.

4.2 Event Model

The EBB model incorporates features to enable elasticity via integration with an event driven model. Execution of EBB software is done via events. An event occurs at a specific location. The location associates the event with a particular set of physical resources on a specific node. EBBs must define an 'access' handler that is invoked when an EBB is accessed at an event location. The EBB runtime implements a lazy access protocol for all EBBs. When an EBB is accessed by an event the runtime will create an 'access miss' and invoke the access handler associated with the EBB. The handler can then, at that location, implement various policies – create a local representative and map it via the runtime to avoid future access misses, map an existing representative, install a redirection proxy, etc.

In our simple example, each arrow represents an event on a specific processor. In many ways EBB access events and method invocations can be considered synonymous. However, events introduce the hook for elasticity. To understand this, consider the life cycle of the distributed counter EBB. Let's assume that our application is initially composed of a single physical node. An initial application setup event creates an instance of the counter. Initially, the counter does not have any internal representatives but is bound to an EBB identifier that can be used to invoke methods of its interface on any processor the application is running on. At some point in the future, a processor on a node in the system invokes the `inc` method of the instance. This generates an access event and handling is directed by the runtime to the instance's access miss handler (not illustrated in the diagram).

The miss-handler, in the case of the distributed counter, allocates resources for a new representative via a system provided allocator EBB[3]. The handler, using runtime primitives, can ensure that future invocations of the EBB's interface on that processor will no longer suffer a miss but

[3]Which itself can suffer a miss and setup its local representative that manages resources for that processor.

will directly invoke the associated function[4]. In this way, we see that, either by writing or choosing from a template of implementations, the programmer achieved the distributed internal structure via definition of the access miss handling behavior. Having achieved this distributed structure, the programmer of the distributed counter implementation will define the `val` method to use runtime provided primitives for aggregating the sum of the currently existing representatives[5].

Several advantages are achieved through this approach:

- Every EBB must define its elastic behavior by specifying what to do when it is accessed at locations it has never been accessed before.
- The resources consumed by an EBB are dynamically allocated in a lazy fashion, avoiding non-scalable initialization.
- Locality optimizations can naturally be implemented at the time of an access miss by ensuring resources for an EBB are allocated locally.
- EBBs can have arbitrarily complex internal structures – all accesses can be forwarded to a single representative, every processor can have its own representative, all processors of a node can share a representative, all cores of a multi-core can share a representative, etc.
- The programmer must implement an EBB to work in the face of elastic changes in its internal composition.

While some of these advantages may seem like burdens on the programmer, our prior experience with Clustered Objects establishes that libraries of template EBBs and access handlers can mitigate this cost.

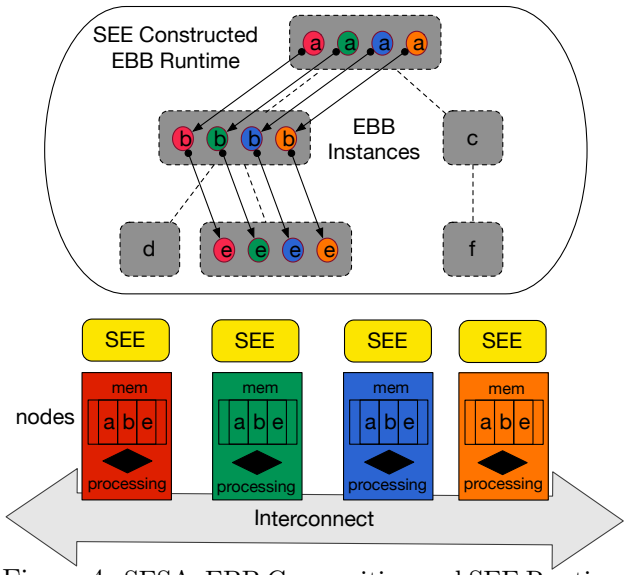

Figure 4: SESA: EBB Composition and SEE Runtime

Given the basic primitive of an EBB, event and event locations, SESA software is constructed as compositions of EBBs where the processing of an event flows through a graph of EBBs. Figure 4 illustrates the EBB runtime model. Each participating node executes a local instance of the Scalable Elastic Executive (SEE). The SEE instances cooperatively form the EBB runtime. The SEE instances locally provide the necessary supporting infrastructure. At the top is a composition of EBBs, labeled a-f, with dashed lines showing how they are connected to each other via references. The solid arrows illustrate an event path that transited the EBBs along a path through a, b and e. The interior of these instances show the constituent representatives that were created based on access. In this fashion, we see that those EBBs are elastically expanded to consume resources on the physical nodes. All system paths in the Tornado

[4]The EBB runtime is constructed to interact with various language calling conventions.

[5]While potentially loosing portability, the programmer can also define their own aggregation primitives that directly use hardware features

and K42 operating systems were constructed in a similar fashion using Clustered Objects, albeit within a single shared memory domain.

The figure illustrates how the EBB approach enables scalable and elastic applications to evolve both with respect to runtime and development. As execution flows along a path of EBBs, the EBB implementations, if all programmed for locality, will transparently and elastically realize a purely local handling of that path. From a development perspective, however, this need not be done by constructing a complete set of advanced distributed implementations. A developer can start with simple centralized implementations and incrementally introduce more advanced distributed implementations on an EBB by EBB basis. This can be done iteratively based on observed performance. This approach proved to be invaluable when implementing the K42 virtual memory management services [8].

4.3 Library Operating System

The Tornado and K42 work acknowledge that achieving practical scalable performance required considering how to integrate advanced models for scalable software given the large investments in current non-scalable software. To this end the projects had the following goals:

1. Provide a structure which allows good performance and scalability to be achieved with standard tools, programming models and workloads without impacting the user or programmer. Therefore, the OS must support standards while efficiently mapping any available concurrency and independence to the hardware without impacting the user level view of the system.

2. Enable high performance applications to side step standards to utilize advanced models and facilities in order to reap maximum benefits without being encumbered by traditional interfaces or policies which do not scale.

These goals are equally important with respect to addressing the cloud computing systems software gap. The prior projects attempted to satisfy these goals by constructing an advanced OS runtime that supported both legacy functions as well as native scalable functions. This proved to be very difficult and challenging with respect to maintenance and software complexity[6].

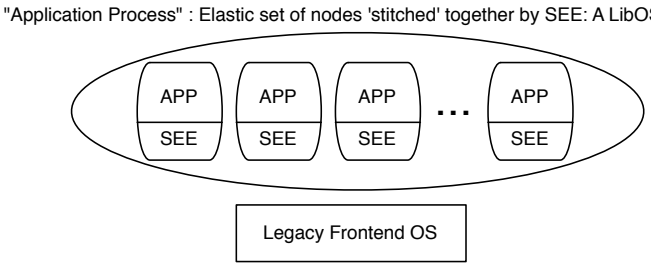

Figure 5: SESA: Example LibOS Deployment

Based on our experience with Libra, SESA attempts to satisfy these goals by adopting a LibOS approach. Rather than trying to construct a complete and legacy compatible OS, the SESA work factors out the support for EBB applications into a standalone LibOS that can be integrated into existing legacy OS environments. Figure 5 abstractly illustrates the approach.

Applications are designed and written as EBBs to the SEE runtime. The runtime is provided as a library that is linked with the application. The library includes a low-level runtime that permits it to be directly executed on a node of the utility provider. An integrated environment is constructed as a collection of nodes running the SESA Application and a front-end node running a legacy OS. The SEE-App nodes can offload legacy function as needed to the frontend via a transfer protocol

and user-level server running on the frontend. Reflexively, the frontend is provided with user-level software that integrates SESA Apps into its application model so that it appears like a native application.

The LibOS approach will allow SESA to avoid the complexities of a complete operating system while enabling a viable incremental development and deployment path. One can optimize a single critical application and within that application focus only on the core function that requires scalable performance.

4.4 Summary

To summarize, consider the SESA approach to developing a new cloud data store service. Given the unification provided by the SESA model, developing an application is more akin to writing software for a single computer. The developer begins by decomposing the application into a set of modules and defining their interfaces. In the case of a data-store, a core component is likely to be a hash-table. To implement a hash-table EBB, the developer would need to consider the following questions:

- What state should be placed in the EBB representatives and hence be local?
- What state should be centralized?
- How should the representatives be created and initialized? At what point should new representatives participate in the global operations of the object?
- How should local operations and global operations synchronize?
- How should the EBB representatives be organized and accessed?

The developer may turn to existing templates and libraries to address these questions. In the case of a hash-table EBB, the developer is likely to consider various organizations ranging from a simple centralized implementation to a partitioned implementation to a fully distributed implementation.

The centralized implementation could be quickly and simply implemented by a single representative located on the instantiating node. This representative would contain the underlying data structures used to store the data items. Access on other processors, both on the same node and other nodes, would be directed to the single representative. To provide access on remote nodes, the programmer could utilize EBB primitives for function and or data shipping. While functional and easy to get started with, this approach has the disadvantage of poor scalability and lack of elasticity.

From here the developer can progressively evolve the implementation. At first, a developer might simply design the EBB to provide additional representatives on new nodes that cache the data of the first representative. While the capacity of the table is still limited to a single node, the performance for read mostly data would dramatically improve. This could further be evolved to use a replica directory, maintained on the first representative, to track and manage replicas of a data item. In this fashion, the developer can continue to evolve more complex distributed implementations that are tuned for the usage patterns of the data being stored. This might also involve adding support for a Chord-like[43] key space partition.

The purpose of system infrastructure is the enablement of computer system use in efficient and novel ways. In this section we have briefly described how SESA uniquely combines object-orientation, event-driven programming and a LibOS model to enable the development and deployment of scalable elastic applications.

References

[1] Glenn Ammons, Robert W. Wisniewski, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, and Eric Van Hensbergen. Libra. In *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, 2007.

[2] T.E. Anderson. The case for application-specific operating systems. In *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*, 1992.

[3] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with k42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Syst. J.*, 2005.

[4] J. Appavoo, V. Uhlig, A. Waterland, B. Rosenburg, D. Da Silva, and J. E. Moreira. Kittyhawk: enabling cooperation and competition in a global, shared computational system. *IBM J. Res. Dev.*, 2009.

[5] Jonathan Appavoo. *Clustered objects*. PhD thesis, University of Toronto, 2005.

[6] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a linux API on the scalable k42 kernel. In *Freenix*, San Antonio, TX, June 2003.

[7] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the first ACM SIGSOFT workshop on Self-Healing Systems (WOSS'02)*, pages 3–8. ACM Press, 2002.

[8] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25, August 2007.

[9] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. *Project Kittyhawk: building a global-scale computer*. ACM Press, 2008.

[10] Jonathan Appavoo, Volkmar Ulig, and Dilma Da Silva. Scalability: The software problem. In *Proceedings of the 2nd Workship on Software Tools for Multi-core Systems*, 2007.

[11] Jonathan Appavoo, Amos Waterland, Dilma Da Silva, Volkmar Uhlig, Bryan Rosenburg, Eric Van Hensbergen, Jan Stoess, Robert Wisniewski, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[13] Shekhar Y. Borkar, Hans Mulder, Pradeep Dubey, Stephen S. Pawlowski, Kevin C. Kahn, Justin R. Rattner, and David J. Kuck. *Platform 2015: Intel Processor and Platform Evolution for the Next Decade*. Intel, 2005.

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th symposium on Operating systems design and implementation*, 2008.

[15] Georges Brun-Cottan and Mesaac Makpangou. Adaptable replicated objects in distributed environments. Technical Report BROADCAST TR No.100, ESPRIT Basic Research Project BROADCAST, June 1995.

[16] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-16)*, pages 143–156. ACM Press, October 1997.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[18] John Chapin. *HIVE: Operating System Fault Containment for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1997.

[19] David R. Cheriton and Kenneth J. Duda. A Caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*, pages 179–193, 1994.

[20] SGI CloudRack™. http://www.sgi.com/products/servers/cloudrack/.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[23] John W. Eaton. *GNU Octave Manual*. Network Theory Limited, 2002.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995.

[25] Fibre Channel. http://www.fibrechannel.org/.

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

[27] Google. Google App Engine. http://code.google.com/appengine/.

[28] David A. Holland and Margo I. Seltzer. Multicore oses: looking forward from 1991, er, 2011. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[29] Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995. http://www.cs.vu.nl/~steen/globe/publications.html.

[30] InfiniBand™. http://www.infinibandta.org/.

[31] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[32] Orran Krieger, Michael Stumm, , and Ron Unrau. The alloc stream facility: A redesign of application-level stream i/o. *IEEE Computer*, 27:75–82, 1992.

[33] Ian M Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 1996.

[34] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In Thoman L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, Los Alamitos, California, 1994.

[35] NUMAlink™. http://www.sgi.com/products/servers/altix/numalink.html.

[36] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[37] Open JDK. http://openjdk.java.net/.

[38] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–104, February 1980.

[39] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13:277–298, October 2005.

[40] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, 2011.

[41] Ruby on Rails. http://rubyonrails.org/.

[42] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proc. ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, page 70, Seattle, WA, March 1990. In ACM SIGPLAN Notices 25:3.

[43] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 2003.

[44] Josep Torrellas, Anoop Gupta, and John L. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174. Boston, Massachusetts, 1992.

[45] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1–2):105–134, 1995.

[46] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed sytem. Technical Report IR-442, Vrige Universiteit, De Boelelann 1105, 1081 HV Amsterdam, The Netherlands, March 1997.

[47] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosemblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Cambridge, Massachusetts, October 1996. ACM Press.

[48] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, 2010.

[49] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Frederick Pollack. HYDRA: The kernel of a multiprocessor operating system. *CACM*, 17(6):337–345, June 1974.

[50] Guray Yilmaz and Nadia Erdogan. Partitioned Object Models for Distributed Abstractions. In *Proc. of 14th International Symp. on Computer and Information Sciences (ISCIS XIV)*, pages 1072–1074, Kusadasi, Turkey, 1999. IOS Press.

[51] L. Youseff and D. Da Silva. Towards a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop 2008*, 2008.

[52] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[53] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. The datacenter needs an operating system. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.