A Unified Execution Model for Cloud Computing

Eric Van Hensbergen
IBM Austin Research Lab
bergevan@us.ibm.com

Noah Paul Evans
Alcatel-Lucent Bell-Labs
npe@plan9.bell-labs.com

Phillip Stanley-Marbell
IBM Zürich Research Lab
pst@zurich.ibm.com

ABSTRACT

This article presents the design goals and architecture for a *unified execution model (UEM)* for cloud computing and clusters. The UEM combines interfaces for logical provisioning and distributed command execution with integrated mechanisms for establishing and maintaining communication, synchronization, and control. In this paper, the UEM architecture is described, and an existing application which could benefit from its facilities is used to illustrate its value.

1. MOTIVATION

The term *cloud computing* is often used to describe cluster computing configurations which permit fluid allocation, provisioning, and configuration of services. In such systems, end-users can easily request and provision computing resources (typically a complete server instance with a chosen operating system installation) as needed, typically through well-defined application programming interfaces (APIs). These computing resources may also be dynamically connected to virtualized storage or virtualized networks, which may also be requested and provisioned dynamically.

Services such as Amazon's EC2 allow end-users to dynamically acquire and provision new resources programmatically, with the new systems brought online in the time span of minutes versus the hours (or days) it would typically take to order, build and configure a physical server. Services may be released at a similar pace, allowing users to scale back the expense of hosting a service when demand is low. Within such a fluid environment, with resources commonly appearing and disappearing at nondeterministic intervals, a static configuration is no longer viable, and mechanisms for dynamic organization of available compute, communication and storage resources into a single logical view are desirable.

Flexibility from an administrative standpoint is only one part of the story. With many different and dynamically varying resources spread out across clusters, users and applications require a new set of systems software interfaces to take maximum advantage of the added dynamism. It is our belief that the first step towards these new interfaces is to *unify the logical node and resource provisioning interfaces with a system-provided remote application execution mechanism.* This new unified interface should be directly accessible by the applications in an operating system-, programming language-, and runtime-neutral fashion.

In this article, we present an approach to providing such a *unified execution model.* The next section details related efforts to provide such interfaces in the high-performance computing community as well as other cloud-based solutions. Section 3 details the key design elements of our approach. In Section 4, we walk through an example of using these interfaces to implement a cloud-based distributed full-system simulator and we conclude with a discussion in Section 5.

2. RELATED WORK

There have been numerous research efforts in the area of distributed execution models and computing systems, such as the Cambridge Distributed Computing System [11], Amoeba [10], V [3], and Eden operating systems [2]. Among the prevalent contemporary approaches employed in high performance computing (HPC) and commercial datacenter/cloud applications, the two most prominent paradigms are MapReduce [4] and MPI [5]—both of which were designed with a particular application structure in mind. We seek a more general-purpose execution model based on system software primitives versus those provided by a language-specific library or runtime system.

The Plan 9 distributed operating system [12] established a cohesive model for accessing resources across a cluster, but it employed only a rudimentary methodology for initiating and controlling remote execution. Its *cpu* facility provided a mechanism to initiate remote execution while providing seamless access to certain aspects of the initiating terminal's resources, using Plan 9's dynamic private (per-process) namespace facilities [13]. While the *cpu* facility provided an elegant mechanism for remote execution, it was limited to a single remote node, which was explicitly selected either by the user or via DNS configuration. This worked well enough for small clusters of terminals with a few large servers whose performance is upgraded over time (i.e., *scale-up*), but is less appropriate for today's clouds, where increased computing resources are added not to a single server, but rather achieved by adding more servers to a cluster (i.e., *scale-out*).

The XCPU runtime infrastructure [7] was an attempt at bringing the power of Plan 9's *cpu* facility to HPC systems running mainstream operating systems such as Linux. It improves upon the basic *cpu* application design by incorporating facilities for starting large numbers of threads on a large number of nodes. It optimizes binary deployment using a *treespawn* mechanism which allows it to task clients as servers to aggregate the deployment of the application executable, required libraries, and associated data and configuration files. The XCPU client also includes provisions for supporting multi-architecture hosts. A significant limitation to deploying XCPU in a cloud context is that it relies on static configuration of member nodes and utilizes an

ad-hoc authentication mechanism which isn't persistent and ends up being rather difficult to use. It also doesn't incorporate workload scheduling or resource balancing itself, relying instead on external tools. XCPU also makes no provisions for interconnecting the processes it starts, relying instead on MPI or other infrastructure to facilitate communication.

The Snowflock [9] logical partition fork mechanism provides an interesting approach to unifying applications and virtual machines via an extension of traditional UNIX fork semantics to entire virtual machine instances. This creates a very powerful and natural method for spawning new virtual machines to complete tasks. Our proposed UEM model could be used as an underlying implementation for this fork model, but also provides broader applicability by being able to utilize heterogeneous resources as well as deploy fan-out style distributed computations.

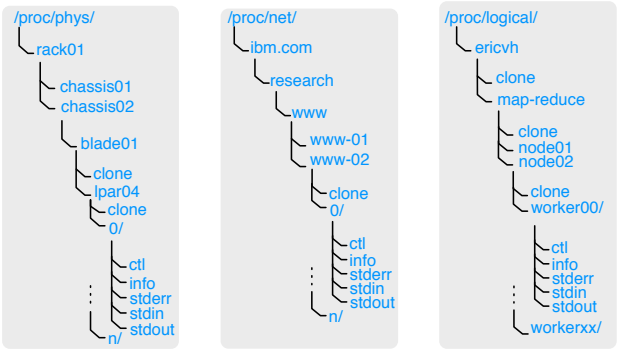
3. APPROACH

The interface to the proposed *unified execution model (UEM)* is structured as a synthetic filesystem similar to the *proc* filesystem pioneered by UNIX and later extended by Plan 9 and adopted by Linux. Within these systems, every process on the system is represented by a *synthetic file* (in the case of historical UNIX), or a directory (in the case of Plan 9 and Linux); synthetic files and directories are entries visible in the filesystem with no corresponding data store on disk. When processes are represented by synthetic directories, there are a number of synthetic files within each process' directory serving as interfaces to process information, events and enabling process control. The XCPU system built upon this simple abstraction in two ways: it allowed nodes to mount each other's *proc* filesystem interfaces and provided the ability to instantiate new processes via a filesystem interface. UEM takes the control of processes via a synthetic filesystem one step further, enabling process creation, control, and inter-application pipelining (in the spirit of Unix pipes), across multiple compute system instances. Importantly, this interface is *distributed*, eliding the need for a central control or coordination point, and facilitating scalability.

Physical resources (and logical multiplexors such as hypervisors) publish their services using the Zeroconf protocol. Distributed registries at key aggregation points routinely scan for changes in service availability and publish consolidated lists to registries higher in the hierarchy. Interactions with the UEM happen with local interfaces, and propagate to different levels of the hierarchy as user specified requests and resource constraints dictate. While our current implementations mostly deal with local hierarchies of clustered resources, nothing implicit in the mechanisms prevent us from cross-connecting multiple hierarchies representing physically or logically distant resources. In other words, nothing prevents a hierarchy from one data center from leveraging resources from a distinct hierarchy at a data center across the globe—the root registries merely need to be aware of one another.

3.1 Namespace organization

The cloud computing systems at which UEM is targeted contain on the order of tens of thousands of computing nodes. The size of these systems necessitates careful consideration of scalability with regards to a synthetic filesystem interface; a single flat organizational structure will simply not scale. A viable approach is the use of a hierarchical structure matching the physical organization of the nodes (Figure 1(a)). A related model would be to use network topology in order to address resources (Figure 1(b)). The nature of either of these hierarchical organizations provides natural points for aggregation, allowing deployment of UEM "concentrators" which offset scalability issues in monitoring and provisioning infrastructures [17]. Yet another model would be to use a logical topology based on characteristics such as the account using the resources, task names, etc. (Figure 1(c)).



(a) Match physical organization. (b) Mimic network topology. (c) Based on resource user.

Figure 1: Three examples of organizational views for UEM synthetic filesystem interfaces.

In practice, any chosen organizational structure will not be optimal for every type of access. To support dynamic and user-defined organizations, the UEM architecture is based on a multi-dimensional semantic filesystem hierarchical structure. Instead of a single hierarchy, UEM provides access to as many different hierarchical organizations as make sense for the end user, supporting physical, logical, network, or other user-defined views. This multiple-view facility is enabled through a key/value tagging mechanism for individual leaf nodes. In the synthetic subdirectory corresponding to each process managed by the UEM are five entries—`ctl`, `info`, `stdin`, `stdout` and `stderr`. Tags are applied or modified via messages written to a node's `ctl` file and are reported as part of the `info` file. These generic tags can then be used by organization and policy modules to provide structured views into the resources based on relevant attributes. This solution works with both physical resources as well as user-defined tasks and logical resources.

3.2 Dynamic namespace views

Synthetic filesystems may also enable more dynamic uses of a path to access resources. For example, the act of accessing a path might be used to initiate (and determine the search terms of) a search, in a manner similar to a RESTful web queries. A top level query-view may thus be provided by a module, allowing end-users and applications to embed attribute regular expressions into the filesystem paths. This would permit searching for resources with certain capabilities, possibly with partial matches. It would also permit searches based on the current state of resources, as captured, e.g., in the `info` entries of processes managed via UEM.

If the path in a query corresponds to a directory, a read will return a directory listing composed of leaf nodes match-

```
% ls /proc/query/x86/gpus=1/mem=4G
# will return a list of physical systems matching
0/
1/
2/
% echo kill > /proc/query/user=ericvh/ctl
# will terminal all LPARs and/or threads belonging
# to user ericvh
% echo cat /proc/query/os=linux/status
# returns status of all Linux logical partitions
node01 up 10:35, 4 users, load average: 1.08
node02 up 05:23, 1 users, load average: 0.50
node03 down
...
```

Figure 2: Semantic query hierarchy example.

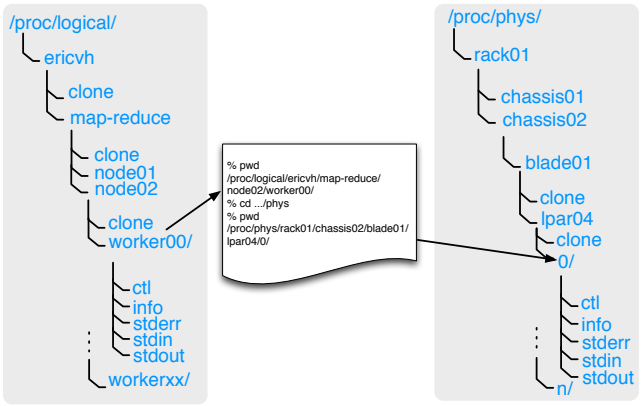


Figure 3: Using dot-dot-dot to shortcut from one hierarchy to another.

ing the query. When a query-path no longer matches any nodes, a file-not-found-error will be returned. An alternate top-level hierarchy will be provided for users who wish to block on traversing a query-path until a resource becomes available. Examples can be seen in Figure 2.

In addition to switching between views of a hierarchy (as seen from the hierarchy's root), it is also useful to be able to change views while at a navigation point *within* the hierarchy, without losing the implicit state associated with the current position in a synthetic filesystem hierarchy.

One possible approach is to define a semantic filesystem shortcut, henceforth referred to as *dot-dot-dot* (Figure 3), which allows users to switch semantic views while maintaining context of their existing location. In Figure 3, the dot-dot-dot shortcut is used to switch between the logical view and the physical view while maintaining the context of the current process. This mechanism could also be used to allocate a new thread (or even a logical partition) on the same physical machine of an existing thread.

Given the presence of many different views of the organizational hierarchy of a system, it will be necessary to have a single canonical view of the hierarchy in which all nodes see the same leaf nodes at the same location. This is necessary both for use by administrative tools which might require a more static view of the resources as well as to be able to communicate path-based references to particular resources. This will be particularly important in order to establish an abstract addressing model for I/O and communication. Devising a meaningful canonical view that meets these criteria is one of our ongoing efforts.

3.3 Policy

The allocation of resources is controlled by policy modules. Physical allocation policy modules are by far the most simple, providing reservation based access to physical resources. The logical policy modules are built on top of the physical policy modules, allocating virtual machines and/or tasks on top of pre-allocated physical or logical resources. The default policy modules provide a space-based partitioning model which allows for simple logical node allocation and task deployment with each task getting its own logical node.

While we provide default allocation policy modules, the UEM is not limited to a single policy. Users and/or administrators may define their own policy modules which can make provisioning decisions based on machine constraints, resource load, quality of service contracts, or external quantitative influences such as who is willing to pay more for the service. These user-defined policy modules can govern any subset of resources and may be deployed at different levels of the organizational hierarchy.

Beyond allocation based policies, users or administrators may deploy monitoring agents and runtime policy modules throughout the infrastructure using the same execution methodology they would use to deploy tasks. Policy agents may also operate directly on the UEM synthetic file hierarchy, gathering and aggregating monitoring information and (infrastructure permitting) reorganize resources and task execution location to optimize for power efficiency, bandwidth, cost or other factors.

3.4 Execution

The mechanism behind initiating execution is based on XCPU's example of using a synthetic file (conventionally named `clone`,) with special semantics, to allocate new resources. Clone files are used in the Plan 9 operating system's synthetic file servers to atomically allocate and access an underlying resource.

An `open()` system call on a `clone` file doesn't return a file descriptor to the (synthetic) `clone` file itself. Instead, a new synthetic subdirectory is created to represent the resource containing control/status files for that instantiated resource. The file descriptor returned from opening the `clone` file points to a control file (conventionally named `ctl`,) within the newly allocated subdirectory so that the application/user can directly interact with the resource they just allocated. All the resources allocated by the initial opening of the `clone` file are released and garbage-collected when the user/application closes the `ctl` file.

For example, a new task can be initiated by opening the `clone` file in the UEM `/proc` filesystem, using a path, as described in Section 3.1, to describe the required resources. Such an operation will initiate interaction with relevant policy modules to reserve the required resources, if available, and return a handle to a `ctl` file. In a fashion similar to Plan 9's *cpu* command, it will also export local resources such as the filesystem, standard input, output, and the user's current shell environment, to the newly allocated resources. Following the convention of Inferno's devcmd [1] and XCPU,

we initiate execution by writing a command to the open `ctl` file handle detailing the (local) path to the executable and command line arguments. Other configurations, such as alternate namespace configuration, environment variables, etc., can be specified either through direct interaction with the control file or through other filesystem interfaces. The remote node will then setup the namespace and environment and initiate execution of the application, redirecting standard I/O to the originating user context (unless otherwise specified as mentioned later in Section 3.6). Since this same interface is available on every node of the system, subsequent task executions can be triggered from within the cluster providing a much more dynamic environment than is available in many of today's cluster workload management systems.

3.5 Configuration

The same approach as taken for initiating execution can be used to provision a logical node or other resource within a cluster. Instead of an application binary, a disk image or an XML virtual machine specification may be passed in the commands being sent to a `ctl` interface in the UEM namespace. In the case of logical nodes, standard I/O handles in the file system are hooked to the console of the virtualized system. Simply allocating a logical node on a particular piece of physical hardware is somewhat less compelling. Instead we take advantage of the dynamic aspects of the query hierarchy to help allocate machines with specific attributes (i.e., `/cloud/x86/gpus=1/mem=4G/clone`) without having to specify a physical node. The same technique could be a general mechanism to find hardware capable of supporting the objtype for a given application (or allocating new logical nodes on demand as necessary if none are currently available).

A variation of this attribute specification can be used to allocate a cluster of nodes (i.e., `/cloud/x86/gpus=1/mem=4G/num=16/clone`). In the case that insufficient physical resources are available to satisfy a logical (or physical) node request a file not found error message will be provided back to the provisioning user or application. As mentioned earlier, doing directory listings at any level of the attribute query semantic hierarchy will detail available nodes matching the classification and the user can use the blocking query hierarchy to wait for resources to become available.

In the case of a group allocation, opening the `clone` file allocates a new node subdirectory which actually represents the set of nodes allocated. In addition to control and status files which provide aggregated access to the individual logical nodes, it will provide subdirectories for each of the logical nodes allocated providing individual access and control. Commands sent to the top level control file will be broadcast to the individual nodes (allowing, for instance, them to all boot the same disk image). We are refining the syntax of the control commands on these meta files to allow for certain keywords which enable necessary differentiation (specifying for instance individual copy-on-write disk images and network MAC addresses). This same approach can be used to launch a set of processes on many remote nodes to perform a scale-out operation such as MapReduce workloads or Monte Carlo simulations.

3.6 Communication

```
# proc102 | proc56 | proc256
# usage: splice <proc_path> <src_fd> <local_fd>
% echo splice /proc/net/remote.com/102 1 0 \
   > /proc/net/mybox.com/56/ctl
% echo splice /proc/net/mybox.com/56 1 0 \
   > /proc/net/farther.com/256/ctl
```

Figure 4: Creating a three-way distributed pipe using UEM.

```
push -c '{
  ORS=./blm.dis
  du -an files |< xargs os \\
   chasen | awk '{print \$1}' | sort | uniq -c \\
   >| sort -rn
```

Figure 5: Examples of *fan-out* and *fan-in* with the PUSH shell.

The UEM takes the UNIX idea of linking processes together with local pipelines and generalizes it to distributed systems by allowing file descriptors to be managed from within the synthetic filesystem control interfaces. This allows the composition of distributed workflows out of simple sets of tools designed to do one thing well. Establishing a UNIX-style pipe between a local and remote task is straightforward, since the local standard I/O context (the `stdout` synthetic file) is exported to the remote task which accesses it directly. Attempting to establish a deep multi-stage pipeline is however more complex, as it is undesirable for all I/O between stages to flow through the initiating node. To avoid this, file descriptors in UEM can be redirected *directly* between nodes executing the stages of the pipeline, with the redirection initiated through the UEM control interface (`ctl` files). This is particularly important for long pipelines and for fan-out workflows.

This approach is best illustrated by an example. Assume that we want to run a three-stage pipeline with each process (process IDs 102, 56 and 256) residing on a separate system (`remote.com`, `mybox.com` and `farther.com`). We assume that the processes have already been instantiated via the UEM interface. The operation of splicing the I/O can be seen in Figure 4. The splice command takes three arguments: the canonical path to the source process, the file descriptor number within the source process, and the target file descriptor within the local process we wish to stream the source into.

Tying all this together is a new shell, named PUSH [6], which uses the facilities of the UEM to allow pipelining of distributed computation. The central goal of PUSH is to provide shell abstractions for the capabilities of UEM that provide a language-independent, terse, and simple way of instantiating large distributed jobs that are traditionally performed by middleware in modern *data intensive supercomputing (DISC)* systems. PUSH provides *fan-out*, *fan-in*, and *hash-pipe* operators providing map, reduce, and all-to-all communication mechanisms between pipeline stages. User-defined modules can be specified to decompose and reconstitute byte streams into records and back into byte streams again, allowing output from one process to go to many and vice versa.

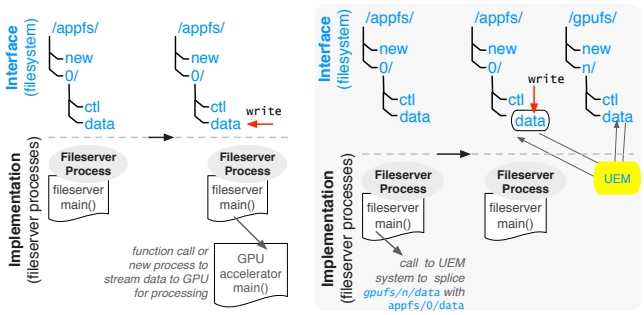


Figure 6: Illustration of time-evolving dynamic synthesized filesystems.

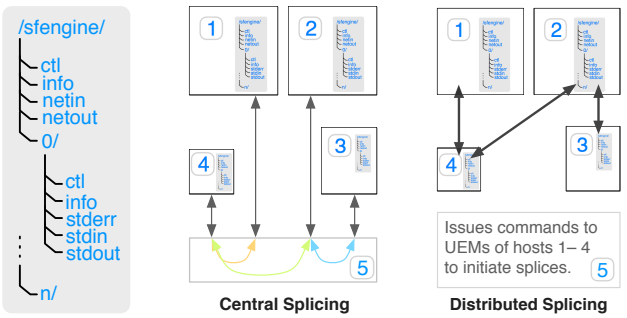


Figure 7: Illustration of potential for removal of the central inter-interface splicing facilitated by the unified execution model's in-network streaming. The filesystem interface at each of the five hosts (four simulation hosts and one control interface), is shown in (a).

4. CASE STUDY

One example application of the UEM is in systems that themselves use synthetic filesystems for representing interfaces to compute or communication resources. If, in such systems, new entries in the served namespace may be dynamically created, or if new interactions between existing entries may occur (Figure 6), static approaches to interfacing and interconnection will be insufficient.

The Sunflower full-system simulator for networked embedded systems [15] provides one illustration of such a system. Sunflower is a microarchitectural simulator intended for use in modeling large networks of embedded systems. Due to the tension between the computation requirements of the instruction-level hardware emulation it performs, and the desire to emulate hardware systems comprising thousands of complete embedded systems, it implements facilities for distributing simulation across multiple *simulation hosts*. To facilitate distributed simulation, Sunflower includes built-in support for launching simulation engines on Amazon's EC2. Each of these individual simulation engine instances executes as a filesystem server [14], serving a small hierarchy of synthetic interface files for interacting with the simulation engine instance and the resources simulated within the engine instance.

4.1 Distributed simulation in Sunflower

Each simulation host taking part in a distributed simulation in Sunflower exposes its modeled resources and control interfaces as a dynamically synthesized filesystem (Figure 7(a)). Through this interface, it is possible to access all the state (processor state, network packets, modeled analog signals, etc.) within components of the system modeled at a given host. When executing a distributed simulation, a central *interface host* connects to each host filesystem, and launches multiple concurrent threads to cross-connect the exposed interfaces to achieve a single system (Figure 7(b)). For example, by cross-connecting (by continually reading from one and writing to the other) the `netin` and `netout` of multiple simulation engines (on possibly-different simulation hosts, e.g., hosts 1–4 in Figure 7(b)), the simulated interconnects in the systems are unified. The central host also ensures the coherent evolution of time across these different simulation hosts, by implementing algorithms from the domain of parallel discrete-event simulation.

4.2 Distributed splicing with UEM

In-network splicing in UEM, as described in Section 3.6, provides a way for cross-connection of file descriptors within a hierarchy of files, with the cross-connection (splicing) occurring on one of the nodes involved in the splice, and with the *splicing facilitated transparently by the system*. Thus, for example, rather than having to execute processes to stream data between the file interfaces to the modeled interconnect between the node pairs (1, 4), (2, 4) and (2, 3) (by cross-connecting the `netin` and `netout` interfaces with processes that continually stream data), the central control interface only needs to *initiate* the cross connection between these pairs (Figure 7(c)). This can be done using the mechanism described in Section 3.6, Figure 4.

4.3 Dynamic call chains with UEM

The interaction between multiple hosts' simulation engines may go beyond static interconnections such as those described thus far. Per-host statistics tracers may, for example, be triggered to commence detailed logging of network traces or architectural state based on time or machine-state information. This could lead to *fileservers within a UEM filesystem triggering the instantiation of new fileservers*. Thus, although a simulation of a particular system will typically involve the creation of an initial static distributed filesystem of simulation state, further work may be triggered by this filesystem at runtime.

5. DISCUSSION

This article presented the architecture for a *unified execution model (UEM)* for structuring the provisioning, and runtime usage of large dynamically changing collections of computing systems, as is typical of so-called cloud computing systems.

By leveraging user- and system-provided policy, monitoring and organizational modules that interface to the UEM, the UEM is extensible, allowing exploration of alternate organizational models, scheduling and resource allocation policies. This will hopefully facilitate new ways of interacting with the dynamic distributed resources which today's cloud computing infrastructures provide.

In addition to the topics covered in this article, we are also actively investigating the application of the UEM to hy-

brid computing environments with GPU and Cell Processor based accelerators [8]. We are also investigating its application on extreme scale high performance computing systems such as BlueGene and Roadrunner [16]. At extreme scale, we are exploring methods of aggregating monitoring, command, and control of the resources providing logical grouping models for system services and automated workload rebalancing and optimization. Another area of active investigation is determining mechanisms for providing fault-tolerance both within our infrastructure and for application tasks which are deployed by it.

The true potential of cloud computing systems will only be realized when cloud interfaces and management mechanisms are integrated into end-user environments as well as the applications themselves. The importance of the consolidated interface that the UEM embodies is that it permits the organization and orchestration of computing resources in a logical manner, maintaining a holistic view of the distributed system. The use of synthetic semantic filesystems enables the creation of a co-operating environment which isn't tightly bound to a particular operating system, programming language, or runtime system; this facilitates its use on legacy and heterogeneous systems, such as those containing computing accelerators (e.g., GPUs or dedicated hardware such as cryptographic encoder/decoders). Providing interfaces for allocating new resources as well as integrated mechanisms to deploy and connect tasks on those resources is the first step towards cloud enabling applications and workflows providing new degrees of flexibility, efficiency, reliability, and productivity.

6. ACKNOWLEDGEMENTS

This work has been supported in part by the Department of Energy Office of Science Operating and Runtime Systems for Extreme Scale Scientific Computation project under contract #DE-FG02-08ER25851.

7. REFERENCES

[1] Inferno Man Pages. *Inferno 3rd Edition Programmers Manual*, 2.

[2] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The eden system: a technical review. *IEEE Transactions on Software Engineering*, 11(1):43–59, 1985.

[3] D. R. Cheriton. The V distributed system. *Communications of the Association of Computing Machinery*, 31(3):314–333, Mar. 1988.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for mpp and workstations. *Commun. ACM*, 39(7):84–90, 1996.

[6] N. P. Evans and E. Van Hensbergen. Push, a disc shell. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 306–307, New York, NY, USA, 2009. ACM.

[7] L. Ionkov, R. Minnich, and A. Mirtchovski. The xcpu cluster management framework. In *First International Workshop on Plan9*, 2006.

[8] D. Jamsek and E. Van Hensbergen. Experiences with Cluster GPU Acceleration for Circuit Design. *Proc. of Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

[9] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[10] S. J. Mullender, C. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Stavern. Amoeba: a distributed operating system for the 1990s. 23(5):44–53, May 1990.

[11] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley Publishers Limited, London, 1982.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[13] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[14] P. Stanley-Marbell. Implementation of a distributed full-system simulation framework as a filesystem server. In *Proceedings of the First International Workshop on Plan 9*, 2006.

[15] P. Stanley-Marbell and D. Marculescu. Sunflower: Full-System, Embedded Microarchitecture Evaluation. *2nd European conference on High Performance Embedded Architectures and Computers (HiPEAC 2007) / Lecture Notes on Computer Science*, 4367:168–182, 2007.

[16] E. Van Hensbergen, C. Forsyth, J. McKie, and R. Minnich. Holistic aggregate resource environment. *SIGOPS Oper. Syst. Rev.*, 42(1):85–91, 2008.

[17] E. Van Hensbergen and R. Minnich. System Support for Many Task Computing. *Proc. of Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.