Towards Elastic Operating Systems

Amit Gupta, Ehab Ababneh, Richard Han, Eric Keller
University of Colorado, Boulder

Abstract

Realizing elasticity in cloud applications today is often a cumbersome process, requiring applications to integrate with services like elastic load balancers and/or be rewritten to accommodate distributed frameworks like map/reduce or cluster-based operating systems. In this paper, we introduce the concept of ElasticOS, which enables a process (or even a single thread) to stretch its associated resource boundaries across multiple machines automatically, expanding and contracting on demand without requiring the application to be re-designed or configured with a complex combination of additional tools and frameworks. Our initial implementation within Linux 3.2 and a study of a MySQL execution trace provide hope that the ElasticOS vision is achievable.

1 Introduction

A key property of cloud-based based systems is elasticity, namely the ability to dynamically provision resources to applications on demand in order to scale up/down to accommodate changing requirements for CPU, memory, storage, and network bandwidth. Achieving elasticity in the cloud today remains non-trivial as it relies on a management soup of various frameworks, independently configured services, and frequently requires applications to be (re)written.

In particular, there are four general steps that a cloud based application must go through to achieve elasticity. First, the application must be partitioned into independent units as the current granularity of adaptation is either on the order of an entire virtual machine or a process that does not overload a single machine. Second, the developer must use a monitoring system and write extra logic to implement their own heuristic to trigger expansion or contraction of number of VMs. Third, the developer must implement or configure an existing system to distribute the workload among the partitioned application instances. Fourth, the developer must incorporate mechanisms to overcome partitions that have some shared state. These patterns are exemplified in the deployment procedures of cloud applications like web servers (used with load balancers like HAProxy [8]), compute intensive jobs (whose execution is often managed by frameworks like HTCondor [12]), or databases modified and specially configured for high scale operations (like MySQL's cluster software [19]). Commercial cloud offerings have attempted to address this complexity through custom services with custom APIs such as load balancing [2], auto scaling [1], or a job processing framework [3]. While useful, these are simply extra services to configure by custom scripting.

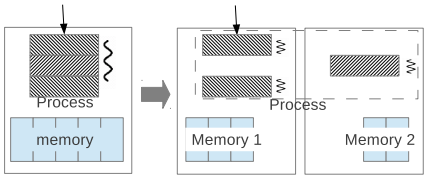


Figure 1: Example of elasticity of memory.

We can see that the current limitations imposed by the cloud today could lead to missed opportunities for elasticity. By limiting elasticity to fixed application-specific units based on what can run on a single machine, we crucially miss the opportunity to elastically scale to take advantage of the collective distributed resources provided by many cloud machines. Recent approaches in improving SMP scalability [26] and elasticity [24] tend to favor/target specific application types. Instead of requiring a great deal of additional applications, services, or frameworks, or targeting a specific class of applications, we should provide to applications a generic OS service that allows them to automatically elasticize processing, memory or I/O, without code modification or additional frameworks/scripting.

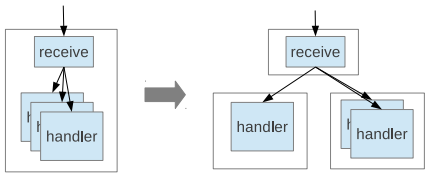


Figure 2: Example of elasticity of processing or I/O .

An ElasticOS with Elastic Page Tables

Following on many of the innovations of modern operating systems, which handle the complex demands of todays software "under the hood", we propose that elasticity be a first order design concern of modern cloud operating systems, and as such be integrated seamlessly into an OS such as Linux to allow large scale applications like those in data centers to automatically expand/contract/scale to accommodate changing load, without being rewritten. For example, in the case of a large in-memory database, the system would automatically scale to incorporate newly available memory on other cloud machines (Figure 1). For Web servers, the system would automatically load balance, moving the code for handlers to other machines while retaining the code which receives requests and divvies them out to worker threads (*e.g.,* a load balancer) on the entry node (Figure 2). Importantly, in contrast to proposed distributed operating systems like fos [28] and Barrelfish [4], we believe this can and should be achieved without requiring a brand new OS and forcing application developers to adopt new programming models.

In order to achieve this, we propose to implement an ability to *stretch* processes elastically on demand as memory, CPU, network and storage demands increase. In particular, we propose to implement memory elasticity via the concept of *elastic page tables*. Conventional page tables map a virtual page number within an address space to a physical page number in memory under the standard assumption that all of the physical memory is local. However, if we relax this assumption and permit different code and data pages from the same process to be placed or spread out across many machines, then this enables us to realize elasticity by adaptively changing the placement of pages across machines in response to the execution flow and data page access patterns.

One basic use case would involve for example a node that has run out of local memory and begun thrashing. In this case, a new cloud machine with available memory could be found, and some of the data pages in the current process could be moved to the newly available memory, thus stretching the process across two or more machines simultaneously. In this way, a process such as a large in-memory database could expand to operate over the collective memory of a large number of rack-mounted machines. Conversely, as demand drops, the process would contract to occupy less physical memory over fewer machines. While this may raise concerns about performance, we are inspired by prior studies that have shown that accessing memory on another machine across the network is faster than accessing disk, for example for swap space [17]. Even more, Section 2 discusses how locality in access patterns of code and data pages for a common application like a MySQL database allows placement of clustered groups of pages together on the same physical node to minimize network traffic from distributed stretched execution.

Revisiting / Extending Distributed Shared Memory

A key mechanism in our proposed approach for ElasticOS is to move thread execution context towards its frequently accessed data pages (Figure 3) to amortize access cost. This is in contrast with Distributed Shared Memory (DSM) systems [5, 6, 10, 23, 25, 27], where support for shared memory, used for IPC by parallel applications, was achieved by replicating data and maintaining consistency through the use of coherency protocols (Figure 4). By choosing not to use data page replicas for the purpose of page access optimization, we will avoid the network cost of coherency messages [1]. Our interest is in investigating a model in which there will be one "active" copy of data pages – *i.e.,* the copy that threads read/write to. The read-only nature of code pages, however, allows them to be replicated on every node without coherency costs. The "active" data page set spread over several nodes is therefore like the distributed equivalent of a working set of data pages of a typical process. In the earlier DSM research mentioned above, execution context was fixed on a node. More recently, movement of the execution stream was explored in DSM/Single System Image (SSI) related research [11, 22, 13, 14, 16], but only under the umbrella of full process migration used for balancing CPU load across a cluster. These (and even newer projects [7]) retain the paradigm of data being moved (and cached) closer to execution streams on-demand and being kept current by the use of costly coherency mechanisms.

We think exploring our design choice to dynamically move thread execution context to follow data could potentially reveal several advantages for ElasticOS, some of which are: the system will be able to reactively co-locate fragments of code and data that are strongly coupled and in doing so, naturally align available resources with any inherent thread level parallelism in the application; the

[1] Of course, data page replicas will exist for fault tolerance purposes, but outside the execution path with looser consistency requirements that do not block the application.

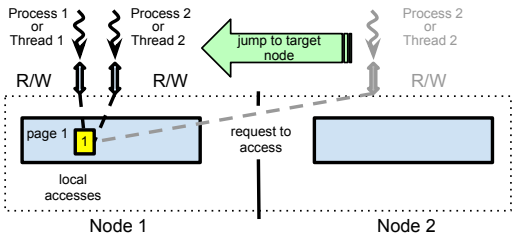


Figure 3: Accesses in ElasticOS

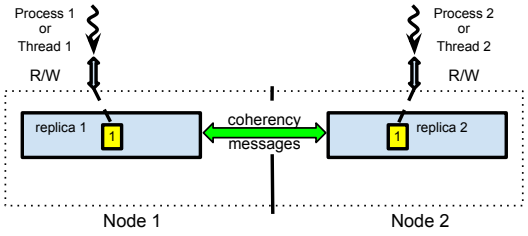


Figure 4: Accesses in Traditional DSM Systems

system can introduce new optimizations that are possible because of the ability to move thread execution context – *e.g.,* using lock wait time to co-locate a waiting thread to a machine where the lock owner is executing. Such co-location could have two fold benefits of **(a)** reducing data access traffic (a promising scaling property) and **(b)** being able to leverage existing, well engineered mechanisms in modern operating systems that handle concurrent access of data between threads/processes.

We should point out that our goal extends beyond just a global virtual address space as we envision elasticity of resources such as network I/O as well. While in some sense we are proposing revisiting the problem domain of DSM[2], with advances in technology such as faster networks, new applications beyond high-performance computing, and with a new goal of elasticity in cloud computing, we are advocating a concept that extends beyond DSM. Of course, this elastic approach introduces a host of interesting new research issues and challenges, which are described in the following sections.

2 Is Elasticity Achievable ?

To gauge the feasibility of this vision, we conducted initial experiments based on 2 fundamental questions.

Can locality be detected in common applications at run-time? : It is our goal that an in-memory application that is elastic would eventually form natural groupings of data pages that are frequently accessed together on each node. As access patterns of the application temporally change, these groupings can change. In order to detect these changes "on the fly" an ElasticOS would have to employ a **page placement algorithm**. As an initial algorithm, we designed the *Multinode Adaptive LRU algorithm* (illustrated in Figure 5) that works as follows: when an execution stream references a page on a remote node, we initially choose to move the page to the location of the execution stream – that is, pull the remote page to the current node. Eviction of a page in order to accommodate a pulled page is done on a least recently used (LRU) basis *i.e.,* the working set of a process on each node over time becomes an LRU based grouping from which pages are accessed together on that node. Upon repeated data page "pulls", beyond a temporal threshold (for our experiment we used a % of pages pulled from a remote node in the last 10 pages accessed), we move the execution context to the target node. In doing this we are likely to exploit locality in the subsequent page accesses.

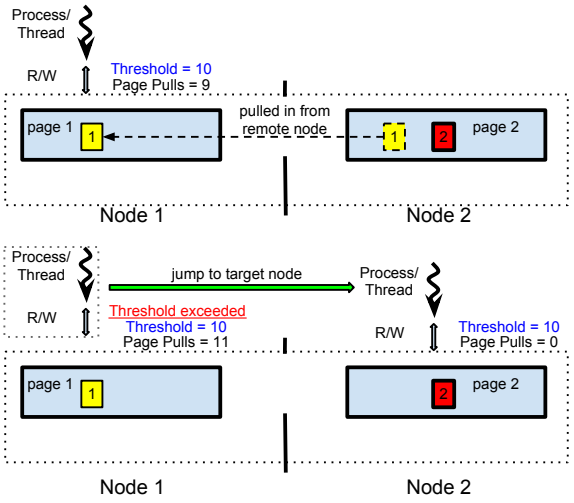


Figure 5: Multinode adaptive LRU algorithm

We conducted a 2-node experiment based on this algorithm to study locality within a common, memory-intensive real world application, MySQL. We instrumented a MySQL daemon using Intel's PIN [9, 15] framework to retrieve a trace of its accessed pages during execution of a customized MySQL-Bench [20] workload. Using this trace we ran a simulation of our algorithm and measured how many execution context jumps and data page pulls took place. The simulation was repeated for various threshold values. Lower threshold values make the algorithm more responsive in moving execution context, which can hurt performance by moving away from locality prematurely. Higher thresholds make the algorithm more sluggish in moving execution context, and can cause an application to be late in exploiting locality at a remote node.

As seen in Figure 6, our simulation results show that

[2]Much as virtualization technology from the 70s was revisited in the late 90s and led to a great wave of innovation in cloud computing.

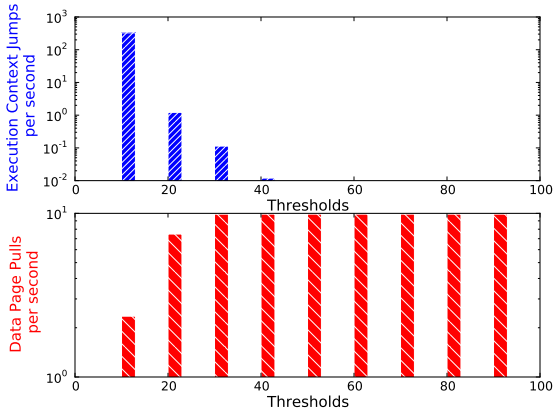
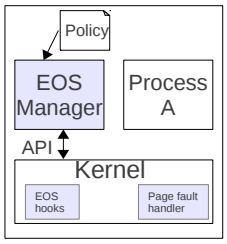


Figure 6: Simulation of the multinode adaptive LRU algorithm for a 334 second MySQL execution trace.

as we vary the threshold, we can tradeoff between execution context jumps and data page pulls. Finding the right *"sweet spot"* (found in this experiment at Threshold = 20%) between this tradeoff in real time, is one of the problems we plan to investigate in our future research.

What is the latency cost of execution context jump/data page pull on today's hardware? : As others have shown, it is extremely fast to pull a data page from across the network [21]. In order to verify that performing an execution context jump to a remote machine is likewise fast, we set out to measure this within the Linux 3.2 kernel by implementing an initial version of a stretched process. Our test process consisted of a simple program that loops and increments a counter stored on a data page and a second counter stored in a register. We wrote a system call for this test process to invoke the movement of execution context and a data page to a location where we had pre-distributed its code pages. The actual state transfer mechanism was implemented within our initial version of the ElasticOS Manager component (described futher in Section 3), and was done over a persistent TCP connection. This test verified the feasibility of execution state transfer – *e.g.,* the counters would count to 10 on one machine, then execution state would be transferred, and the counters would resume counting up 11-20 on the other machine, followed by another state transfer and a count up 21-30 on the original machine, and so forth. The average measured latency of a state transfer was approximately **0.4 ms** over a gigabit Ethernet link. This 0.4ms represents a pause in the execution of a single thread (parallel execution within the entire application ensures that this is not simply idle time). As such, this results in a slowdown of the application as compared to the case where a single machine has infinite memory (and therefore never swaps to disk). For



Physical addr	Valid	node
0x1234...1234	1	local
0x2341...2341	0	local
0x3412...3412	0	10.1.1.1
0x4123...4123	1	local

Figure 7: ElasticOS Architecture.

the threshold of 20%, the combined transitions (jumps + pulls) of our MySQL trace was approximately 9 per second. Using 0.4ms for each, implies a delay of 36ms for every 1 second of actual execution, or a slowdown of only 3.6% (we do not expect slower execution, but instead improvements over operations such as swapping to disk).

Based on these results, our algorithm was effective for this particular workload in being able to maintain a reasonable page grouping that dynamically captures some application locality. We intend to explore this design space further in our search for an algorithm that adapts well to various applications and realistic workloads.

3 Elastic OS Architecture

We identified the following major new components to support an ElasticOS architecture, as shown in Figure 7.

Elastic Page Table for Tracking Remote Pages
We extend the traditional page table to include an extra field to note the machine that is holding the page – nominally shown as an IP address for illustration as any addressing scheme can be used. Note that the machine holding the page may be the local machine, which indicates the page is on the local disk (*i.e.,* valid bit set means in memory, not valid and machine ID is local means on disk, not valid and machine ID is remote means the page is on a remote machine). Of course, the processor's memory management unit (MMU) will not (currently) recognize the ElasticOS modified page table entry, so it is implemented as a separate data structure, but used internally by the operating system as a single table. Maintaining the page to machine mapping falls to the ElasticOS Manager discussed below.

Modified Page Fault Handler to Transition Execution
The traditional page fault handler must be modified to handle the new case where a page fault does not necessarily mean fetch from disk. If the machine ID in the elastic page table indicates the page is local, then the page fault handler will proceed as normal and swap from disk. If the page is remote, there are two possible actions that can occur: (i) pull the remote page across the net-

work to the local memory (as with network-based memory swapping [17, 21]), (ii) transition execution to the other machine. In either case, with ElasticOS we will put the process in the suspended state and notify the ElasticOS manager (EOM), which will perform the action.

ElasticOS Manager for Intelligent Orchestration

The EOM is a continually running process that monitors the system and provides intelligence through two basic functions: (i) make page placement decisions across the machines, and (ii) oversee execution transition.

The first is to group related memory pages across the machines in such a way that minimizes the need for transitioning execution between machines. To achieve this, the EOM monitors the entire execution of the process and analyzes the relationships between memory pages. Upon an event in the system (*e.g.,* thrashing) that indicates the need to expand to another machine, the EOM may move pages between machines (possibly speculatively). The decision to transition execution to another machine is based on the overall page placement policy – *e.g.,* based on analysis of whether this will lead to execution that accesses more remote pages or whether this is a single page but future execution will mostly be on the current machine. Our initial multinode adaptive LRU algorithm was a policy which pulled first, then jumped after some number of pulls.

EOM's second function is to perform execution transition. If the decision was made to move execution to the node hosting the page, the page fault handler would have put the process in a suspended state. The EOM must then copy to the remote machine the necessary register state and any memory pages needed for execution to continue in the exact same state. Note that this transition of execution is simply stretching the process across multiple machines which is distinctly different from entire process migration [13].

Kernel Hooks to Enable User-space Management

Putting complexity in user-space not only increases security and stability of the system but also provides greater flexibility when adding functionality. As such, we choose to place the ElasticOS Manager in user-space. The ElasticOS kernel must provide hooks to enable this user-space management. While the full extent of the API can not be listed here for space reasons, we highlight its major functionalities. First, the ability to get and set pages is needed to perform the page movement. This is needed to enable smart page clustering, fault tolerance, and execution transition. Second, access to register state of a process is needed during the transition of execution. Finally, counters and various other measurements need to be exposed to enable intelligent page placement decisions.

4 Open Research

There are a host of additional exciting open research questions that need to be solved in order to fully realize the promise of ElasticOS.

Page placement distributed decision making: Since ElasticOS spans multiple machines, then the EOM and page placement policies are distributed across many computers. An open question we expect to explore is to what extent should this elastic decision-making be centralized in say a master-slave(s) arrangement, or be fully decentralized.

Elastic I/O: Stretching virtual memory across a network provides only one dimension of stretching. We also envision stretching of other computer resources such as CPU and I/O (namely network I/O). Network I/O presents an interesting challenge due to addressing issues. In particular, a packet directed to the process has a destination IP address that, due to the current technology, will be directed to a single server. Outgoing packets can come from any of the stretched machines. Stretching the I/O without custom virtual appliances is a significant research direction to explore. We intend to examine leveraging advances in software-defined networking (such as with OpenFlow [18]) as an interface to achieve this. Encouraged by contemporary research [24] in this direction that targets network middleboxes, we would like to extend network elasticity more generally to a large range of applications.

Fault Tolerance: Stretching a process over multiple machines has the potential to increase the application's susceptibility to failure. To combat this, we intend to investigate snapshot techniques, rollbacks, distributed check-pointing, and replication.. In addition, we will strive to place fault-tolerance functions off the critical path of normal execution to minimize their impact on performance.

5 Conclusion

This paper introduced the concept of elasticity in an operating system by stretching the execution of a single process or thread over many machines without requiring new programming models or cumbersome new scripting logic. Memory elasticity is achieved via elastic page tables, and an initial study of a multi-node adaptive LRU page placement algorithm explored feasibility. The ElasticOS architecture was outlined, and a host of exciting new research issues in elasticity were highlighted for OS researchers to solve.

Acknowledgements

We would like to thank Blake Caldwell and Ryan Tidwell for their early help on this research project.

References

[1] Amazon. Auto scaling. `http://aws.amazon.com/autoscaling/`.

[2] Amazon. Elastic load balancing. `http://aws.amazon.com/elasticloadbalancing/`.

[3] Amazon. Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proc. ACM Symposium on Operating systems principles (SOSP)*, pages 29–44, 2009.

[5] J. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: where we are and where we should be headed. In *Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on*, pages 119–122, 1995.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, Aug. 1995.

[7] M. Chapman and G. Heiser. vnuma: A virtual shared-memory multiprocessor. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 2–2. USENIX Association, 2009.

[8] HAProxy. The reliable, high performance tcp/http load balancer. `http://haproxy.1wt.eu/`.

[9] INTEL. Intel pin binary instrumentation tool, rev 55942. `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, Dec. 2012.

[10] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. 1988 Int. Conf. Parallel Processing*, volume 2, pages 94–101, 1988.

[11] LinuxPMI. Linux-pmi project webpage. `http://linuxpmi.org/trac/`, Dec. 2012.

[12] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[13] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. Openmosix, openssi and kerrighed: a comparative study. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 1016–1023 Vol. 2, 2005.

[14] R. Lottiaux, L. Rilling, and M. Fertré. Architecture of the kerrighed hotplug mechanism. 2010.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[16] A. Lbre, R. Lottiaux, E. Focht, and C. Morin. Reducing kernel development complexity in distributed environments. In E. Luque, T. Margalef, and D. Bentez, editors, *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 576–586. Springer Berlin Heidelberg, 2008.

[17] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *Proc. USENIX Annual Technical Conference*, 1996.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communications Review*, 38(2), 2008.

[19] MySQL. MySQL 5.5 Reference Manual, 17.1 MySQL Cluster Overview. `http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-overview.html`.

[20] MySQL. Sql-bench. `http://dev.mysql.com/doc/refman/5.0/en/mysql-benchmarks.html`, Dec. 2012.

[21] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters. In H. Kosch, L. Bszrmnyi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 1160–1169. Springer Berlin Heidelberg, 2003.

[22] OpenSSI. Openssi project webpage. `http://openssi.org/cgi-bin/view?page=openssi.html`, Dec. 2012.

[23] A. L. C. P. Keleher, S. Dwarkadas and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[24] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. NSDI, 2013.

[25] D. J. Scales and K. Gharachorloo. Design and performance of the shasta distributed shared memory protocol. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 245–252, New York, NY, USA, 1997. ACM.

[26] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with os clustering. In *Proceedings of the sixth conference on Computer systems*, pages 61–76. ACM, 2011.

[27] M.-C. Tam, J. M. Smith, and D. J. Farber. A taxonomy-based comparison of several distributed shared memory systems. *SIGOPS Oper. Syst. Rev.*, 24(3):40–67, July 1990.

[28] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.