Tiago de Almeida LUC - COMP 410 - Fall 2014

Graduate Student Research Paper

Paper #1: Decoupling Cores, Kernels, and Operating Systems (Zellweger, 2014)

The paper presents an extension to the Barrelfish OS, named Barrelfish/DC, which decouples physical cores from the OS, as well as the kernel itself from the rest of the OS. This extension is capable of quickly replacing native kernel code on any core, moving kernel state between cores and adding and removing cores from the system in a way that is transparent to applications and the OS, which do not interrupt their execution.

In order to provide such capabilities, Barrelfish/DC presents two novel ideas. The first is the abstraction of a CPU core as a special case of a peripheral device, through the use of boot drivers, which can start, stop and restart a core while running elsewhere. The second is the use of a partitioned capability system for memory management, which externalizes all OS state for a core. As a result, a kernel can be essentially stateless and easily replaced while Barrelfish/DC is running.

Designing a processor is a dynamic task, driven by ever-changing demands. It is commonplace to remark that core counts are increasing, both on a single chip and in a complete system. Cores themselves are becoming non-uniform and future computers will contain large numbers of heterogeneous cores, which will be powered on and off individually in response to workload changes. A key motivation for this is power reduction for embedded systems: under high CPU load, complex, high-performance cores can complete tasks more quickly, resulting in power reduction in other areas of the system. Under light CPU load, however, it is more efficient to run tasks on simple, low power cores. Besides that, specialized cores will be responsible for specific kinds of computation. Just like graphic processor are common today, other specific processing units will become common for tasks like natural language processing and a contextual computing processing.

Current OS designs like Linux and Windows assume a static number of homogeneous cores, with recent extensions to allow core hotplugging. There is increasing interest in modifying, upgrading, patching or replacing OS kernels at runtime. According to the authors, the key challenges in updating an OS online are to maintain critical invariants across the update and to do so with minimal interruption of service. This is particularly hard in a multiprocessor kernel with shared state. The implications for a future OS, the authors say, are that it must manage a dynamic set of physical cores, and be able to adjust to changes in the number, configuration, and microarchitecture of cores available at runtime, while maintaining a stable execution environment for applications.

Barrelfish/DC is an OS design based on the principle that all cores are fully dynamic. It exploits the “multikernel” architecture to separate the OS state for each core. A key challenge with dynamic cores is safely disposing of per-core OS state when removing a core from the system: this process takes time and can dominate the hardware latency of powering the core down, reducing any benefit in energy consumption. Barrelfish/DC addresses this challenge by externalizing all the per-core OS and application state of a system into objects called OSnodes, which can be executed lazily on another core.

Barrelfish/DC can completely replace the OS kernel code running on any single core or subset of cores in the system at runtime, without disruption to any other OS or application code, including that running on the core. Kernels can be upgraded or bugs fixed without downtime, or replaced temporarily, for example, to enable detailed instrumentation, to change a scheduling algorithm, or to provide a different kind of service such as performance-isolated, hard real-time processing for a bounded period. Furthermore, per-core OS state can be moved between slow, low-power cores and fast, energy-hungry cores. Multiple cores' state can be temporarily aggregated onto a single core to further trade-off performance and power, or to dedicate an entire package to running a single job for a limited period. Parts of Barrelfish/DC can be moved onto and off cores optimized for particular workloads.

This work may have a huge impact in operating systems design. The diversity of processing units coupled together in a single chip or system is becoming common. The use of different specialized cores in the same processor has revealed itself as an interesting approach for managing the workload in a more intelligent way. This may be related to performance, but it is increasingly also related to energy efficiency and power consumption. Operating systems need to follow this approach by managing cores accordingly. This paper shows that current operating systems do not have the ability to manage dynamic cores. Barrelfish/DC presents a radically different vision of how cores are exploited by an OS and the applications running above it. As hardware becomes more dynamic, system software will have to change its assumptions about the underlying platform, and adapt to a new world with constantly shifting hardware.

As for future directions, the authors plan to investigate the power management opportunities afforded by the ability to replace cores and migrate the running OS around the hardware. They also intend to use core replacement as a means to improve OS instrumentation and tracing facilities, by dynamically instrumenting kernels running on particular cores at runtime, removing all instrumentation overhead in the common case. The ultimate goal is to minimize whole-system reboots as much as possible by replacing single kernels on the fly.

Paper #2: f4: Facebook's Warm BLOB Storage System (Muralidhar, 2014)

Facebook has grown to be a huge company. Since its experience is based on data sharing, its storage capacity to store that data needs to be big and efficient. An important class of data stored by Facebook is Binary Large OBjects (BLOBs), which are immutable binary data. This kind of data is created once, read many times, never modified and sometimes deleted. Obvious types of BLOBs on Facebook include photos and videos. Documents, traces, heap dumps and source code also frequently fall into this category.

Facebook's original BLOB storage, called Haystack, is designed for IO-bound workloads. It replicates data for fault tolerance and to support a high request rate. As Facebook has grown, characteristics of BLOBs have changed. The diversity in size and create, read, and delete rates has increased. There is now a large and increasing number of BLOBs with low request rates. For these BLOBs, triple replication results in over provisioning from a throughput perspective, even though it provides important fault tolerance guarantees.

This paper describes a specialized BLOB storage system called f4. It provides the same fault tolerance guarantees as Haystack but at a lower effective-replication-factor. f4 is a storage system specially designed for warm BLOBs. Warm BLOBs are objects with lower request rates than those stored on Haystack, thus not "hot". The design of f4 is simple, modular, scalable and fault tolerant. It handles the request rate of warm BLOBs and responds to requests with sufficiently low latency. It is tolerant to disk, host, rack and datacenter failures, and it provides all of this at a low effective-replication-factor.

It is really important to notice how well crafted a solution may be when designed for the right set of requirements. Generic solutions usually need to provide guarantees for several environments and situations. It makes them useful, but usually not optimized. Once you take away some of those requirements, you are able to design an optimized solution for a narrower set of situations. You are therefore able to take advantage of the fewer requirements in order to gain efficiency. In this specific scenario, the lower request rate of warm BLOBs enables them to provision a lower maximum throughput for f4 than Haystack, and the low delete rate for warm BLOBs enables them to simplify f4 by not needing to physically reclaim space quickly after deletes.

The authors also present a few lessons they learned along the course of designing, building, deploying and refining f4. The importance of simplicity in the design of a system for keeping its deployment stable, according to them, crops up in many systems within Facebook and was reinforced by their experience with f4. The importance of measuring underlying software for your use case's efficiency was highlighted by the authors. Measuring and understanding the underlying software that f4 was built on top of helped improve the efficiency of the system. Finally, the authors mention how they learned about the importance of heterogeneity in the underlying hardware for f4. According to them, there was an episode when a crop of disks started failing at a higher rate than normal. In addition, one of their regions experienced higher than average temperatures that exacerbated the failure rate of the bad disks. This combination of bad disks and high temperatures resulted in an increase from the normal ~1% AFR (Annualized Failure Rate) to an AFR over 60% for a period of weeks. As a lesson learned, they plan on using hardware heterogeneity to decrease the likelihood of such correlated failures in the future.

Companies like Facebook currently face unprecedented challenges when it comes to storage. Data is being stored like never before, and the amount of data continues to increase.

Coming up with solutions for specific scenarios becomes a necessity. In this case, it is really interesting to see how Facebook organizes the storage of data according not only to its type, but also to its temperature, a measure that defines how frequently an object is target of reading, modifying and deleting operations. In Facebook's scenario, data's age is strongly correlated to its temperature. For instance, newly created BLOBs are requested in an order of magnitude higher than week-old BLOBs. Therefore, BLOBs are initially stored in the hot storage system Haystack and later moved to f4. As a result, f4 saves a significant amount of storage while still providing enough guarantees to keep the service running and the user experience acceptable.

Their lesson on the importance of heterogeneity in the underlying hardware is also really interesting. Fault tolerance techniques apply similar reasoning on software, using different implementations for the same job and comparing the results. Using equipment from different models and manufacturers makes sense, since an unexpected fault may be related to that specific model or manufacturer.

Paper #3: Torturing Databases for Fun and Profit (Zheng, 2014)

When a process crashes in your system, you're frequently able to re-open it and re-compute the data you were working on. When a failure happens in a storage system, however, you may not be able to restore the data. Data losses are extremely damaging and practically unacceptable for services that store user's data in the cloud.

Among storage systems, databases provide the strongest reliability guarantees. The atomicity, consistency, isolation, and durability (ACID) properties databases provide make it easy for application developers to create highly reliable applications. However, these properties are far from trivial to provide, particularly when high performance must be achieved. This leads to complex and error-prone code. Even at a low defect rate of one bug per thousand lines, the millions of lines of code in a commercial OLTP database can harbor thousands of bugs. Checking for the ACID properties under failure is notoriously hard since a failure scenario may not be conveniently reproducible.

The paper proposes a method to expose and diagnose violations of the ACID properties, focusing on an ostensibly easy case: power faults. Unexpected loss of power is a particularly interesting fault, since it happens in daily life and is a threat even for sophisticated data centers and well-prepared important events. Clean loss of power causes the termination of the I/O operation stream, which is the most idealized failure scenario and is expected to be tolerated by well-written storage software.

Despite extensive test suites, existing databases still contain bugs. Verifying if a database run is correct after a fault injection is a non-trivial task. The authors present four workloads developed for evaluating databases under the power fault model. They stress all four ACID properties and allow the easy identification of incorrect database states. Given the workloads, a framework is built to efficiently test the behavior of databases under fault. A high-fidelity block I/O trace is recorded using a modified iSCSI driver. Then, each time a fault is to be simulated during that run, the fault model is applied to the collected trace, generating a new synthetic trace. A new disk image representative of what the disk state may be after a real power fault is then created by replaying the synthetic trace against the original image. After restarting the database on the new image, the consistency checker is ran for the workload and database under test. This record-and-replay feature allows the systematical injection of faults at every possible point during a workload

Something interesting on this work is that the authors understand that simply triggering errors is not enough. Given the huge code base and the complexity of databases, diagnosing the root cause of an error could be even more challenging. Instead, they go further and collect detailed traces during the working and recording phases, including function calls, systems calls, SCSI commands, accessed files and accesses blocks, in order to help in diagnosing and fixing the discovered bugs. These multi-layer traces, which span everything from the block-level accesses to the workloads' high-level behavior, facilitate much better diagnosis of root causes.

8 common databases are evaluated using the implemented framework, ranging from simple open-source key-value stores such as Tokyo Cabinet and SQLite up to commercial OLTP databases. Because the file system may be a factor in the failure behavior, the databases are tested on multiple file systems (ext3, XFS and NTFS) as applicable. As a surprising result, the authors show all 8 databases exhibit erroneous behavior, with 7 of the 8 clearly violating the ACID properties. The errors range from only a few corrupted records to entire tables lost. By

using the detailed multi-layer traces mentioned before, the authors are able to indicate the root causes of the errors for open-source systems. According to them, architects of the commercial systems could quickly correct their defects given similar information.

The method for exposing reliability issues in storage systems under power fault presented on this paper is considered cross-platform. By intercepting in the sSCSI layer, the framework can test databases on different operating systems. By recording SCSI commands (which are what disks actually see), faults can be injected with high fidelity. Further, SCSI tracing allows systemic fault injection and ease of repeating any error that is found.

An additional interesting contribution of this work is the identification of 5 low-level patterns that indicate the most vulnerable points in database operations from a power-fault perspective. Further analysis of the root causes verifies that these patterns are closely related to incorrect assumptions on the part of database implementers. User-space applications including databases often have assumptions about what the OS, file system, and block device can do, and violations of these assumptions typically induce incorrect behavior.

The authors close the paper with the conclusion that even ostensibly well-tested databases can fail and consequently lose data. According to them, undirected testing is not enough: Thorough testing requires purpose-built workloads designed to highlight failures, as well as fault injection targeted at those situations in which designers are likely to make mistakes.

This work may have a considerably big impact in the designing and implementation of storage solutions. The identification of clear violations of the ACID properties even in well-known and trusted databases shows how difficult it is to develop huge systems like these. The development of testing frameworks and workloads like the ones presented on this paper is of fundamental importance for the development process of database systems. The platform agnostic approach of the presented solution is also important: it allows different platforms to use the framework for extensible testing, and also gives an opportunity to compare different storage systems regarding reliability levels.

References:

Zellweger, G. et al (2014). Decoupling Cores, Kernels, and Operating Systems. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

Muralidhar, S. et al (2014). F4: Facebook's Warm BLOB Storage System. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

Zheng, M. et al (2014). Torturing Databases for Fun and Profit. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.