Outfoxing the Mammoths

Marek Olszewski   Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Some of last year's FIT talks motivated and provided uses for highly sought-after features for tomorrow's programming languages. The topics ranged from compilation of deterministic parallel programs [7] to ultra fast always-on precise race detectors [4]. Many programmers dream about having deterministic execution of parallel programs to help with their debugging, and fast race detectors could dramatically simplify programming language specifications by eliminating the need to define the semantics of memory races. The faster we can create such language features and semantics, the faster we can dig ourselves out of the multicore crisis.

We argue that if we want to solve these daunting problems we need additional ammunition from the architecture and operating system communities. However, operating systems and computer architectures are changing too slowly for us to rely on. These *mammoth* systems are extremely large and complex and as a result are extremely slow to move. Thus, rather than waiting and hoping to receive the ammunition, we need to outfox the mammoths so that we can obtain the tools we need today.

To do so, we need to learn from the greatest and most innovative invention since the transistor: The TiVo. The TiVo revolutionized the way we experience TV programming, by allowing users to not only pause and rewind live television, but also to record not one, but *two* shows at the same time for later viewing without advertisements. This ability to "time-shift" television programming dramatically improved the experience for people willing to invest in a simple add-on box that plugged in between their televisions and cable boxes. Though the lesson behind TiVo is not in the features that it provided, but in the way that it was able to provide them. TiVo introduced these new features transparently, despite the fact that it was an outsider to the television industry, through the use of a simple add on box.

We believe that just like the TiVo revolutionized the television watching experience, we need to revolutionize the system stack by creating our own TiVos to help us with our research. Thus, in a similar way that TiVo nudged cable companies to start integrating DVR (Digital Video Recording) features directly into their cable boxes, such an action could provoke the operating systems community to start offering us the new features we need to achieve our research goals. Most importantly, we must realize that in order to bring about these changes we cannot succeed by implementing them directly into the operating system. Operating systems are often too large, too complicated, or too closed source for us to easily modify to fit our needs. Moreover, there are too many operating systems out there to change and it takes too long for such changes to propagate into the hands of researches. Instead, we must build software equivalents of TiVo boxes that can simply plug into the software stack, much like a TiVo plugs into the home entertainment system. To do so, we must build *OS-Top Boxes* – the software stack equivalent of a television set-top box.

Such OS-top boxes already exist today in the form of hypervisors. Hypervisors fit neatly underneath the operating system and can augment computer systems to include features such as machine virtualizing and record-replay. Moreover, implementing new features in a hypervisor is generally easier than in an operating system because hypervisors are simpler and contain fewer lines of code. Additionally, hypervisors provide portability across a range of operating systems and can be released to users faster than new operating systems. Thus, when it comes to nudging the operating system mammoths, hypervisors provide a natural avenue to do so.

It is important to note however that over time hypervisors will grow in complexity as new features are added and as the source code becomes more mature. As a result, we must be careful to avoid constructing our own mammoths, and so we should not shy away from using multiple hypervisors, with one on top of another (i.e.: nested virtualization). After all, we have no problems with placing an Apple TV on top of a

TiVo and using them both together in the same system.

1. Example OS-Top Box

Following this approach, we are in the process of developing Aikido: a hypervisor that we hope will provide a useful service to programming language researchers. In this section we describe Aikido and postulate some possible uses. We hope that this example can spark the imagination of the reader into thinking about other hypervisors that might also be possible and useful.

The Aikido hypervisor enables user-space threads running within a shared address space to page-protect memory on a per-thread basis. Thus, Aikido can be used to cheaply detect sharing habits at a page-level granularity for today's multithreaded applications, which almost always use shared address spaces. Aikido defines a new API for user-space applications and tools that uses *hyper-calls* to enable direct calls into the hypervisor, which bypass the operating system. Through careful manipulation of shadow page tables, Aikido is able to construct per-thread virtual memory protections for the same virtual memory pointing to the same physical memory. The system is portable to many operating systems which are unaware of the changes being made to the system's page tables.

By allowing programs to dynamically and cheaply discover which pages of memory are shared between threads, Aikido allows dynamic instrumentation systems to avoid instrumenting instructions that only access pages containing just private memory. Thus, for applications that are not dominated with accesses to shared data, Aikido can enable a number of tools, systems, or language features that could operate with low overheads. We envision the following potential uses:

Language Support and Enforcement of Data Sharing Information provided by a type system that describes ownership and sharing of data, could be enforced using Aikido.

Sharing Detector By instrumenting only instructions that access shared pages, a data sharing detector tool could run significantly faster than if it were to instrument all memory accesses. Such a tool could be used to profile sharing patterns of parallel applications.

Software Transactional Memory A software transactional memory system might benefit significantly if it could rely on page protection, rather than expensive fine grained instrumentation, to detect conflicting memory accesses in concurrent transactions. The technique is especially appealing for *strong atomicity* systems where all program code (whether inside or outside of a transaction) must detect conflicts and where greater amounts of transactional data is private.

Thread Level Speculation In a similar way, Aikido could be used to help detect data dependency violations across speculative threads used in thread level speculation systems and in systems such as Grace [2].

Race Detector A race detector that only instruments instructions that access shared pages might run significantly faster than today's race detectors. If such a race detector can be made precise, it may be feasible to use it for enforcing language specifications that disallow race conditions.

Deterministic Multithreading Finally, by instrumenting only instructions that access shared pages, it may be possible to enforce a deterministic interleaving of instructions that access shared data with little overhead. Such a system could provide *strong determinism* guarantees [6], with lower overheads than today's strong determinism systems [1, 3, 5].

References

[1] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS '10*, New York, NY, USA, 2010. ACM.

[2] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09*, pages 81–96, New York, NY, USA, 2009. ACM.

[3] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA '09*, New York, NY, USA, 2009. ACM.

[4] Hans Boehm. Simple thread semantics require race detection. *PLDI FIT Talk*, 2009.

[5] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09*, New York, NY, USA, 2009. ACM.

[6] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09*, New York, NY, USA, 2009. ACM.

[7] Nalini Vasudevan and Stephen Edwards. A determinizing compiler. *PLDI FIT Talk*, 2009.