MAKING LOW LATENCY STORES PRACTICAL AT CLOUD SCALE

by

Chinmay Satish Kulkarni

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Chinmay Satish Kulkarni**
has been approved by the following supervisory committee members:

Ryan Stutsman ,	Chair(s)	__ Date Approved
Robert Ricci ,	Member	__ Date Approved
John Regehr ,	Member	__ Date Approved
Kobus Van Der Merwe ,	Member	__ Date Approved
Badrish Chandramouli ,	Member	__ Date Approved

by **Mary Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B Kieda** , Dean of The Graduate School.

ABSTRACT

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

CHAPTER 1

EXTENSIBILITY AND MULTI-TENANCY

Since the end of Dennard scaling, disaggregation has become the norm in the datacenter. Applications are typically broken into a compute and storage tier separated by a high speed network, allowing each tier to be provisioned, managed, and scaled independently. However, this approach is beginning to reach its limits. Applications have evolved to become more data intensive than ever. In addition to good performance, they often require rich and complex data models such as social graphs, decision trees, vectors [6, 7] etc. Storage systems, on the other hand, have become faster with the help of kernel-bypass [3, 8], but at the cost of their interface – typically simple point lookups and updates. As a result of using these simple interfaces to implement their data model, applications end up stalling on network round-trips to the storage tier. Since the actual lookup or update takes only a few microseconds at the storage server, these round-trips create a major bottleneck, hurting performance and utilization. Therefore, to fully leverage these fast storage systems, applications will have to reduce round-trips by pushing compute to them.

Pushing compute to these fast storage systems is not straightforward. To maximize utilization, these systems need to be shared by multiple tenants, but the cost for isolating tenants using conventional techniques is too high. Hardware isolation requires a context switch that takes approximately 1.5 microseconds on a modern processor [5]. This is roughly equal to the amount of time it takes to fully process an RPC at the storage server, meaning that conventional isolation can hurt throughput by a factor of 2 (Fig 1.1). Splinter relies on a type- and memory-safe language for isolation. Tenants push extensions – a tree traversal for example – written in the Rust programming language [9] to the system at runtime. Splinter installs these extensions. Once installed, an extension can be remotely

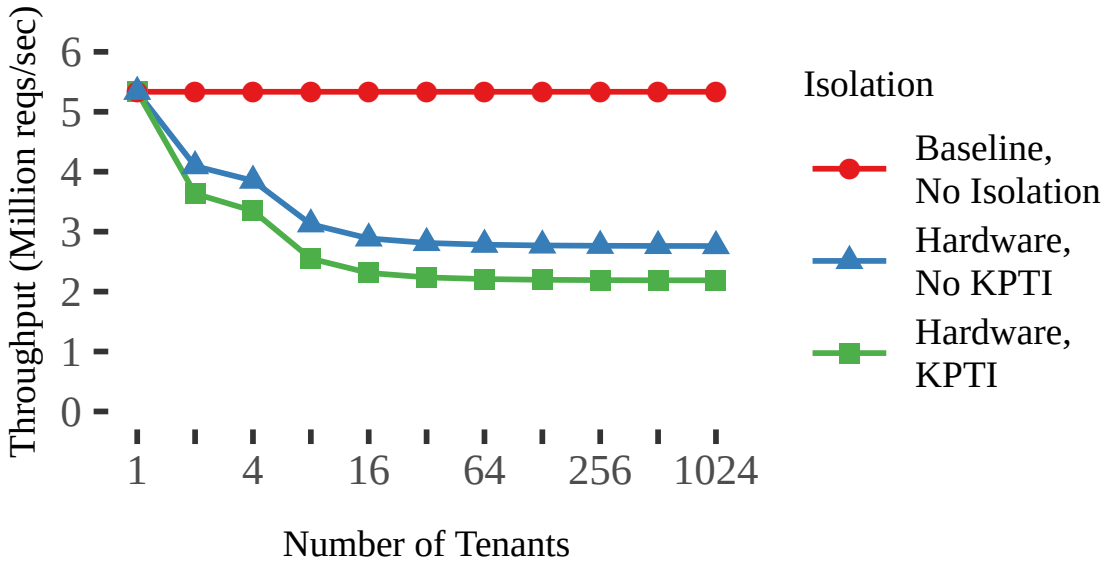

Figure 1.1: Simulated throughput of a system that isolates pushed code using hardware. The baseline represents the upper bound when there is no isolation. At high tenant density, isolation hurts throughput by nearly 2x.

invoked (executed) by the tenant in a single round-trip. For applications such as tree traversals which would ordinarily require round-trips logarithmic in the size of the tree, splinter can significantly improve both throughput and latency.

In addition to lightweight isolation, splinter consists of multiple mechanisms to make pushing compute feasible. Cross-core synchronization is minimized by maintaining *tenant locality*; tenant requests are routed to preferred cores at the NIC [4] itself, and cores steal work from their neighbour to combat any resulting load imbalances. Pushed code (an extension) is scheduled *cooperatively*; extensions are expected to yield down to the storage layer frequently ensuring that long running extensions do not starve out short running ones. This approach is preferred over conventional multitasking using kthreads because preempting a kthread requires a context switch, making it too expensive for microsecond timescales. Uncooperative extensions are identified and dealt with by a dedicated watchdog core. Data copies are minimized by passing immutable references to extensions; the rust compiler statically verifies the lifetime and safety of these references. With the help of these mechanisms, Splinter can isolate 100's of granular tenant extensions per core while serving millions of operations per second with microsecond latencies.

Overall, Splinter adds extensibility to fast kernel-bypass storage systems, making it easier for applications to use them. An 800 line Splinter extension implementing

Facebook's TAO graph model [1] can serve 2.8 million ops/s on 8 threads with an average latency of 30 microseconds. A significant fraction of TAO operations involve only a single round-trip. Implementing these on the client using normal lookups and implementing the remaining operations using the extension helps improve performance to 3.2 million ops/s at the same latency. This means that an approach that combines normal lookups/updates with Splinter's extensions is the best for performance; the normal lookups do not incur isolation overhead (no matter how low), and the extensions reduce the number of round-trips. In comparison, FaRM's [2] implementation of TAO performs 6.3 million ops/s on 32 threads with an average latency of 41 microseconds. This makes Splinter's approach, which performs 0.4 million ops/s per thread, competitive with FaRM's RDMA based approach, which performs 0.2 million ops/s per thread.

REFERENCES

[1] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, *TAO: Facebook's Distributed Data Store for the Social Graph*, in Proceedings of the 2013 USENIX Annual Technical Conference, USENIX ATC '13, San Jose, CA, 2013, USENIX Association, pp. 49–60.

[2] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, *FaRM: Fast Remote Memory*, in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI '14, Berkeley, CA, 2014, USENIX Association, pp. 401–414.

[3] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, *No compromises: Distributed transactions with consistency, availability, and performance*, in Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, New York, NY, USA, 2015, Association for Computing Machinery, p. 54–70.

[4] Intel Corporation., *Flow Director*. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf`. Accessed: 2018-09-27.

[5] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, *Splinter: Bare-metal extensions for multi-tenant low-latency storage*, in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, Oct. 2018, USENIX Association, pp. 627–643.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, *Scaling distributed machine learning with the parameter server*, in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, Oct. 2014, USENIX Association, pp. 583–598.

[7] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, *Scaling memcache at facebook*, in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, Apr. 2013, USENIX Association, pp. 385–398.

[8] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, and et al., *The ramcloud storage system*, ACM Trans. Comput. Syst., 33 (2015).

[9] *The Rust Programming Language*. `http://www.rust-lang.org/en-US/`. Accessed: 2018-09-27.