Exploiting MISD Performance Opportunities in Multi-core Systems

Patrick G. Bridges, Donour Sizemore, and Scott Levy*
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Abstract

A number of important system services, particularly network system services, need strong scaling on multi-core systems, where a fixed workload is executed more quickly with increased core counts. Unfortunately, modern multiple-instruction/multiple-data (MIMD) approaches to multi-core OS design cannot exploit the fine-grained parallelism needed to provide such scaling. In this paper, we propose a replicated work approach to parallelizing network system services for multi-core systems based on a multiple-instruction/single-data (MISD) execution model. In particular, we discuss the advantages of this approach for scalable network system services and we compare past methods for addressing the challenges this approach presents. We also present preliminary results that examine the viability of the basic approach and the software abstractions needed to support it.

1 Introduction

A number of important system services need *strong scaling*, where a fixed workload is executed more quickly with increased core counts. This is particularly true for data-intensive, network-oriented system services because network interface card (NIC) line rates continue to increase but the individual cores that service them are not getting any faster. In cases like this, multiple cores must closely coordinate activities and multi-core system designs which rely on only intermittent synchronization will bottleneck on inter-core synchronization. As a consequence, the resulting system performance can be disappointing.

*This work was supported in part by gifts from Sun and Intel Corporations, a faculty sabbatical appointment at Sandia National Laboratories, and a grant from DOE Office of Science, Advanced Scientific Computing research, under award number DE-SC0005050, program manager Sonia Sachs.

Recent research projects on multi-core system software designs propose a distributed systems-oriented approach, where data structures are replicated or segregated between processors [1, 2]. These designs use a multiple-instruction/multiple-data (MIMD) parallel execution model, and generally focus on services for which coarse-grained synchronization is sufficient. Because of this, such systems generally provide only *weak scaling*, where more cores can be used to handle a larger workload (e.g. more TCP/IP connections) but cannot make an existing workload (e.g. a single TCP/IP connection) run faster.

In this paper, we propose a replicated work approach based on a multiple-instruction/single-data (MISD) execution model to take advantage of unexploited parallelization opportunities in multi-core systems. This approach is particularly important in the network stack, as it is needed to enable strongly scalable network services. We begin by discussing example network applications and related system services that require strong scaling. We then discuss a replicated work execution model for building strongly-scalable network system services based on MISD parallelism, along with specific challenges that this execution mode presents, including consistency, scheduling, and programmability. Finally, we discuss initial experiments we have conducted on this approach and the infrastructure to support it.

2 Motivating Applications and Services

The combination of increasing network card line rates and plateauing single-core CPU speeds is driving the demand for scalable network services. The most obvious motivating example of this is high-speed network connectivity over TCP-based network connections. Traditional operating systems have difficulty saturating a 10Gbps Ethernet NIC using a single

TCP connection [14] without complex hardware assistance such as TCP offload engines. However, such hardware solutions present well-known challenges to system software design (e.g. packet filtering, security, portability).

Past work has shown that providing scalable network connections in software is quite challenging [11, 14]. Perhaps as a result, recent work on scaling TCP/IP network stacks has focused primarily on scaling the number of connections or flows that can be supported rather than on the data rate of individual connections. Corey, for example, uses a private TCP network stack on each core [3], and RouteBricks segregates each IP flow to its own core [6]. These approaches allow systems to scale to handle a large number of connections or flows, but limit the data rate achievable on each flow to well below the throughput of modern network cards.

One frequently attempted workaround to these problems is to use multiple parallel connections for these applications, turning a strong scaling problem into a weak scaling one. Unfortunately, this approach has numerous drawbacks. Most importantly, such approaches can break existing network resource allocation systems [13]. It also places significant burden on application and library programmers for managing data ordering and reliability.

Without a scalable network service, a wide range of applications such as high-bandwidth file systems, databases, and content distribution systems, become difficult to deploy. For example, the performance of middleware systems that buffer, aggregate, and analyze data as it is streamed between data warehouse systems (e.g. Netezza, LexisNexis) and high-end supercomputing systems (e.g. ADIOS [9]) depends upon the availability of scalable network services.

3 Building Strongly Scalable System Services

The primary challenge in building strongly scalable networked system services is handling shared state. Current MIMD-based approaches to parallelizing system services for multi-core systems rely on relatively expensive synchronization mechanisms (e.g. messages, locks) that prevent the exploitation of fine-grained parallelism. TCP/IP, for example, includes important shared state (e.g. sliding window state) that must be updated on each packet sent or received. TCP also includes elements that can potentially be paralellized, for example data delivery, acknowledgment generation, and timer management. Unfortunately, explicit or implicit inter-core communication costs are too large compared to packet processing times to be successfully amortized away.

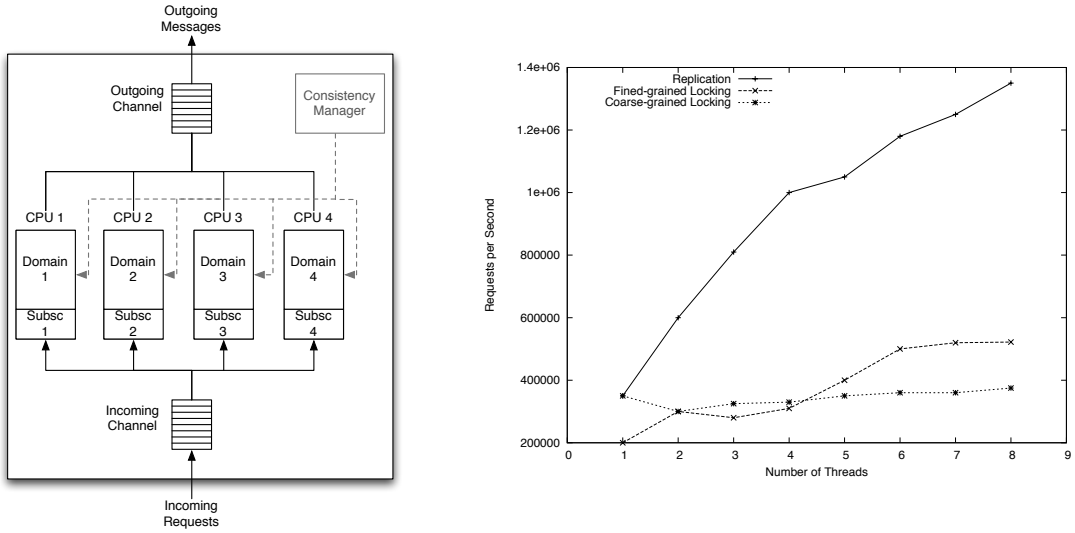
3.1 MISD-Based Parallelism

We propose using a multiple instruction/single data (MISD) execution model (i.e. replicated work) to provide fine-grained data consistency between cores and to eliminate expensive inter-core interactions. In this approach, shown in Figure 1(a), state is replicated into *domains* on separate cores, and requests that modify shared state are broadcast to *every* domain using a ringbuffer-based *channel* abstraction. The first replica that dequeues a request becomes the primary replica for that request and is responsible for fully processing it, including any updates with globally visible side effects (e.g. data delivery, packet reconstruction, acknowledgment generation). Other replicas that process the request will, on the other hand, only partially process each request to maintain state consistency.

This approach is particularly appropriate for many network services because it parallelizes expensive per-byte processing (e.g. data copies, encryption/decryption, and error correction) and replicates per-header processing costs that are known to be small [7]. It also retains the reduced locking and caching costs of systems like Barrelfish [1] while adding the ability to perform fine-grained updates to logically shared state. Finally, we note that this approach is similar to the execution model used in many group RPC systems [5]

To examine the potential for this approach compared to lock-based and atomic instruction-based MIMD approaches, we constructed a simple synthetic test. In this test, processing each request requires some amount of work that can be done by one core without synchronization or replication (i.e. parallelizable work), and some work that must be synchronized or replicated. Specifically, the parallelizable work is a set of memory updates done on per-core memory, while the synchronized work is the set of memory updates that must be performed: (a) on every replica; (b) while holding a shared lock; or (c) by using atomic memory update instructions.

Figure 1(b) shows the potential performance advantages of this approach using a 10:1 ratio of parallelizable to replicated work; we chose this ratio based



(a) Replicated-work parallelization architecture

(b) Comparison of lock, atomic instruction, and replicated-work parallellization approaches

Figure 1: Architecture and basic performance of a replicated-work parallelization approach

on our our studies of production TCP/IP stacks. As can be seen, the lock-based model scales remarkably poorly and the atomic instruction approach is only slightly better. The replicated work approach, in contrast, scales well to 8 cores, the limit of the 2x4-core Intel Xeon machine on which we tested.

3.2 Consistency Management

While this approach avoids the inter-core synchronization and communication costs of other approaches, it does introduce consistency management problems. Specifically, domains that subscribe to multiple broadcast channels, for example separate channels for incoming and outgoing packets are by default PRAM consistent—they see requests on the same channel in the same order, but may see different interleavings of the requests from the different channels.

If these consistency problems are not handled properly, inconsistency between replicas could result in both poor performance and incorrect behavior. In the networking context, for example, a replica could dequeue an incoming acknowledgement for a packet sent by another replica *before* it sees dequeues the request that generated the packet being acknowledged from the outgoing request channel.

There are a variety of approaches to handling these consistency management problems. The simplest solution is for each domain to route all requests through a single broadcast channel that supports multiple producers, resulting in sequential consistency between replicas—all replicas see the same requests in the same order—at the cost of extra synchronization when enqueuing requests on channels. A wide range of other well-known distributed consistency and message ordering techniques are also potentially applicable.

In addition, addressing consistency problems is in many ways *simpler* for network services than for other services. In particular, inconsistency between replicas in network services results in anomalous protocol behavior, and many network services must be able to tolerate unexpected protocol behaviors due to unusual network conditions. As a result, consistency management between replicas for these services need only focus on inconsistency that results in non-recoverable behavior (e.g. spurious connection resets) or that significantly impacts performance and scalability.

We are taking an approach where each service manages its own consistency demands in three distinct ways. Most coarsely, each service implementation will determine how to use channels to route requests to domains, allowing it to determine which requests are sequentially consistent and which are PRAM consistent. At a finer granularity, each domain implements a channel scheduler which chooses from which chan-

nel to dequeue the next request, based on examination of the head of each channel; this allows it to implement service-specific scheduling rules, such as: "process outgoing requests required to fill the congestion control window prior to processing incoming packets".

3.3 Scheduling

Using a bounded broadcast channel to provide MISD parallelism, as we propose, also raises a variety of inter-domain scheduling issues. In particular, domains that lag behind in processing requests can cause the channel to fill up, preventing new requests from being enqueued and slowing service. However, well-known gang scheduling and load balancing techniques can be used to address such issues. In addition, because every replica processes the same set of requests, it should be possible for a lagging replica to roll forward to the state of another replica that periodically saves a snapshot of its state in shared memory.

3.4 Programmability

The new replicated work execution model described above also presents programmability challenges, an important factor to consider in new system software designs. This execution model, like a purely distributed one, avoids the need for complex locking protocols and read-copy-update mechanisms. It does, however, require replicas to segregate program execution into work that must be executed on every request (to maintain consistency) and work that must be executed when the replica is the primary on a request.

We believe that an event-based programming model [12] is ideal for programming such services. Event-based programming provides a simple mechanism for directing execution based on whether or not the replica is the primary for a request (e.g. by raising separate events), and for reusing shared code between primary and non-primary processing paths. In addition, MISD-style execution preserves the event atomicity expected in event-based systems, and event-based systems have been successfully used in a number of related distributed systems projects [8].

4 Implementation

We have begun implementing our proposed approach in a framework called Dominoes. As a first step, we have implemented a replicated ring buffer channel to broadcast requests to multiple domains. We have also begun porting network stacks to this framework.

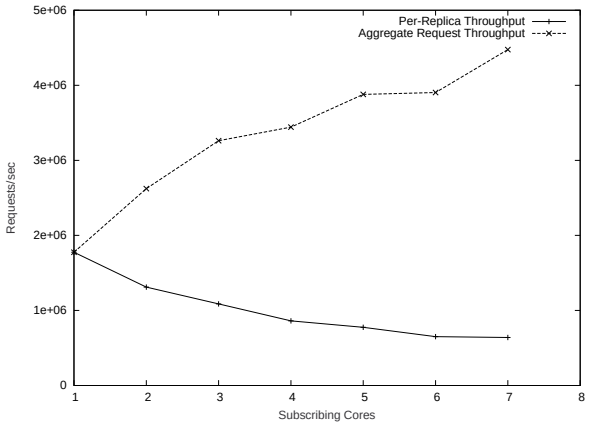


Figure 2: Replicated ring buffer channel performance on an 2x4-core 2.4GHz AMD Shanghai system

The basic replicated ring buffer channel is a traditional single producer/single consumer ring buffer augmented with per-replica head pointers and a global per-entry reference count updated using an atomic decrement instruction. Upon dequeuing an entry, a replica advances its local head pointer and atomically decrements the entry's reference count. If the updated reference count is one less than the total number of replicas, the replica knows that it is the first to dequeue the request and is therefore responsible for fully processing it. On the other hand, if the updated reference count is zero the replica knows that it is the last to dequeue and it can advance the global head pointer, thereby freeing space in the ring buffer for new entries. We have also implemented a version that allows for multiple producers as opposed to a single producer.

Figure 2 shows the per-core and aggregate dequeue rates from a channel based on this data structure on a 2x4-core 2.4GHz AMD Shanghai system, where one core is dedicated to enqueueing elements and some number of the remaining cores dequeue elements. Even with 1500 byte packets, this service rate is sufficient at 4 replicas to handle data rates well in excess of 10 Gbps. In addition, more sophisticated reference counting techniques than the one used by this implementation (e.g. sloppy counters [3]), could also potentially improve the scalability of this implementation.

In addition to this basic framework construction,

we have also begun porting the Scout [10] and the CTP [4] networking frameworks to Dominoes. We have completed initial ports of the (essentially stateless) UDP and IP protocols, where preliminary scaling results are encouraging, and are currently working on a TCP protocol implementation. We have also begun adapting the Cactus framework [8] used to implement CTP to provide an event-based programming model in Dominoes, and porting basic components of CTP to the resulting system.

References

[1] 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 *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.

[2] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 2010 USENIX Symposium on Operating System Design and Implementation*, 2010.

[4] P. G. Bridges, M. A. Hiltunen, R. D. Schlichting, G. T. Wong, and M. Barrick. A configurable and extensible transport protocol. *ACM/IEEE Transactions on Networking*, 15(6):1254–1265, 2007.

[5] S. T. Chanson, D. W. Neufeld, and L. Liang. A bibliography on multicast and group communications. *ACM Operating Systems Review*, 23(4):20–25, 1989.

[6] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[7] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[8] M. A. Hiltunen, R. D. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, 1999.

[9] J. Lofstead, F. Zhang, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, Washington, DC, USA, 2009. IEEE Computer Society.

[10] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–168, 1996.

[11] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[12] J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *1996 USENIX Technical Conference*, 1996. Invited Talk.

[13] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367 – 378. ACM Press, New York, 2004.

[14] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–96, June 2006.