Improving Network Connection Locality on Multicore Systems

Aleksey Pesterev* Jacob Strauss† Nickolai Zeldovich* Robert T. Morris*
*MIT CSAIL †Quanta Research Cambridge
{alekseyp, nickolai, rtm}@csail.mit.edu jacob.strauss@qrclab.com

Abstract

Incoming and outgoing processing for a given TCP connection often execute on different cores: an incoming packet is typically processed on the core that receives the interrupt, while outgoing data processing occurs on the core running the relevant user code. As a result, accesses to read/write connection state (such as TCP control blocks) often involve cache invalidations and data movement between cores' caches. These can take hundreds of processor cycles, enough to significantly reduce performance.

We present a new design, called Affinity-Accept, that causes all processing for a given TCP connection to occur on the same core. Affinity-Accept arranges for the network interface to determine the core on which application processing for each new connection occurs, in a lightweight way; it adjusts the card's choices only in response to imbalances in CPU scheduling. Measurements show that for the Apache web server serving static files on a 48-core AMD system, Affinity-Accept reduces time spent in the TCP stack by 30% and improves overall throughput by 24%.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Measurement, Performance

Keywords Multi-core, Packet Processing, Cache Misses

1. Introduction

It is well known that a good policy for processing TCP connections on a multiprocessor is to divide the connections among the cores, and to ensure that each connection is handled entirely on one core [18]. This policy eliminates contention for the locks that guard each connection's state in the kernel, and eliminates cache coherence traffic that would be caused if a connection's state were used on multiple cores. The policy is an instance of a more general rule for parallelism: activities on different cores should interact as little as possible. Many server programs are designed to follow this rule. Web servers, for example, typically process a stream of requests from many independent clients. They process the requests concurrently on many cores, and the requests' independence allows them to be processed with little serialization due to shared state, and thus allows for good parallel speedup.

In practice, however, even with independent requests it is difficult to avoid all interactions between activities on different cores. One problem is that the kernel typically serializes processing of new connections on a given TCP port (e.g., the single UNIX `accept()` queue). A second problem is that there may be no way to cause all of the activities related to a given connection to happen on the same core: packet delivery, kernel-level TCP processing, execution of the user process, packet transmission, memory allocation, etc. A third problem is that the connection-on-one-core goal may conflict with other scheduling policies, most notably load balance. Finally, the application's design may not be fully compatible with independent processing of connections.

This paper describes Affinity-Accept, a design that achieves the goal of executing all activity related to a given connection on a single core, and describes an implementation of that design for Linux. The starting point for Affinity-Accept is an Ethernet controller that distributes incoming packets over a set of receive DMA rings, one per core, based on a hash of connection identification fields. New connection requests (SYN packets) are added to per-core queues, protected by per-core locks, so that connection setup can proceed in parallel. Each server process that is waiting for a connection accepts a new connection from its own core's queue, and we assume that the application processes the connection on the same core where it was accepted. Since the Ethernet controller will place subsequent incoming packets for this connection into the DMA ring for the same core, the end effect is that all packet and application processing for this connection will take place on the same core. This minimizes contention for shared cache lines and locks. Affinity-Accept also balances load at multiple time-scales to handle connections with differing processing times as well as persistent imbalances due to unrelated computing activity.

An alternate solution is to use a "smart" Ethernet card with a configurable flow steering table that routes packets to specific cores. The kernel could configure the card to route packets of each new connection to the core running the user

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

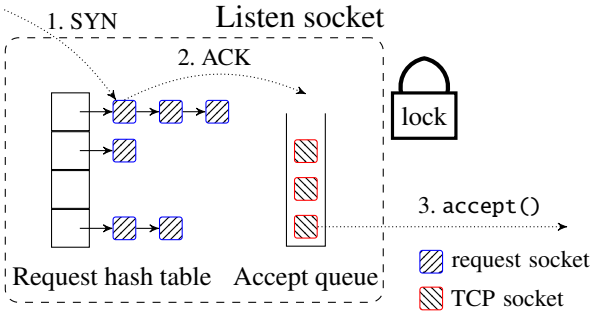

Figure 1: A TCP *listen socket* in Linux is composed of two data structures: the *request hash table*, which holds request sockets, and the *accept queue*, which holds established TCP sockets. The listen socket performs three duties: (1) tracking connection initiation requests on SYN packet reception, (2) storing TCP connections that finished the three-way handshake, and (3) supplying TCP connections to applications on calls to `accept()`.

space application. Unfortunately, this does not work due to limited steering table sizes and the cost of maintaining this table; §7 gives more details.

We present an evaluation on a 48-core Linux server running Apache. First, we show that, independent of Affinity-Accept, splitting up the new connection queue into multiple queues with finer-grained locks improves throughput by 2.8×. Next, we evaluate Affinity-Accept and show that processing each connection on a single core further reduces time spent in the TCP stack by 30% and improves overall throughput by 24%. The main reason for this additional improvement is a reduction in cache misses on connection state written by other cores.

The rest of the paper is organized as follows. §2 describes the problems with connection management in more detail, using Linux as an example. §3 presents the kernel components of Affinity-Accept's design, with application-level considerations in §4, and implementation details in §5. §6 evaluates Affinity-Accept. §7 presents related work and §8 concludes.

2. Connection Processing in Linux

This section provides a brief overview of Linux connection initiation and processing, and describes two problems with parallel connection processing: a single lock per socket and costly accesses to shared cache lines.

2.1 TCP Listen Socket Lock

The Linux kernel represents a TCP port that is waiting for incoming connections with a *TCP listen socket*, shown in Figure 1. To establish a connection on this port the client starts a TCP three-way connection setup handshake by sending a SYN packet to the server. The server responds with a SYN-ACK packet. The client finishes the handshake by returning an ACK packet. A connection that has finished the handshake is called an established connection. A listen socket has two parts to track this handshake protocol: a hash table recording arrived SYNs for which no ACK has arrived, and an "accept queue" of connections for which the SYN/SYN-ACK/ACK handshake has completed. An arriving SYN creates an entry in the hash table and triggers a SYN-ACK response. An arriving ACK moves the connection from the hash table to the accept queue. An application pulls connections from the accept queue using the `accept()` system call.

Incoming connection processing scales poorly on multicore machines because each TCP port's listen socket has a single hash table and a single accept queue, protected by a single lock. Only one core at a time can process incoming SYN packets, ACK packets (in response to SYN-ACKs), or `accept()` system calls.

Once Linux has set up a TCP connection and an application process has accepted it, further processing scales much better. Per-connection locks allow only one core at a time to manipulate a connection, but in an application with many active connections, this allows for sufficient parallelism. The relatively coarse-grained per-connection locks have low overhead and are easy to reason about [9, 18].

Our approach to increasing parallelism during TCP connection setup is to partition the state of the listen socket in order to allow finer grained locks. §3 describes this design. Locks, however, are not the whole story: there is still the problem of multiple cores sharing an established connection's state, which we discuss next.

2.2 Shared Cache Lines in Packet Processing

The Linux kernel associates a large amount of state with each established TCP connection, and manipulates that state whenever a packet arrives or departs on that connection. The core which manipulates a connection's state in response to incoming packets is determined by where the Ethernet interface (NIC) delivers packets. A different core may execute the application process that reads and writes data on the connection's socket. Thus, for a single connection, different cores often handle incoming packets, outgoing packets, and copy connection data to and from the application.

The result is that cache lines holding a connection's state may frequently move between cores. Each time one core modifies a cache line, the cache coherency hardware must invalidate any other copies of this data. Subsequent reads on other cores pull data out of a remote core's cache. This sharing of cache lines is expensive on a multicore machine because remote accesses are much slower than local accesses.

Processing the same connection on multiple cores also creates memory allocation performance problems. An example is managing buffers for incoming packets. The kernel allocates buffers to hold packets out of a per-core pool. The kernel allocates a buffer on the core that initially receives the packet from the RX DMA ring, and deallocates a buffer on the core that calls `recvmsg()`. With a single core processing a connection, both allocation and deallocation are fast because they access the same local pool. With multiple cores performance suffers because remote deallocation is slower.

3. An Affinity-aware Listen Socket

To solve the connection affinity problem described in the previous section, we propose a new design for a listen socket, called Affinity-Accept. Affinity-Accept's design consists of three parts. As a first step, Affinity-Accept uses the NIC to spread incoming packets among many RX DMA rings, in a way that ensures packets from a single flow always map to the same core (§3.1).

One naïve approach to ensure local packet processing would be to migrate the application-level thread handling the connection to the core where the connection's packets are delivered by the NIC. While this would ensure local processing, thread migration is time-consuming, both because migration requires acquiring scheduler locks, and because the thread will incur cache misses once it starts running on a different core. For short-lived connections, thread migration is not worthwhile. Another approach would be to tell the NIC to redirect the packets for a given flow to a different core. However, a single server may have hundreds of thousands of established TCP connections at a given point in time, likely exceeding the capacity of the NIC's flow steering table. Moreover, the time taken to re-configure the NIC for each flow is also prohibitively high for short-lived connections.

Thus, the second part of Affinity-Accept's design is that instead of forcing specific connections or threads to a particular core, Affinity-Accept arranges for the `accept()` system call to preferentially return local connections to threads running on the local core (§3.2). As long as the NIC's flow steering does not change, and application threads do not migrate to another core, all connection processing will be done locally, with no forcible migration of flows or threads.

Since Affinity-Accept does not use forcible migration to achieve connection affinity, it assumes that the NIC will spread load evenly across cores. However, there are many reasons why some cores may receive more or less load at any given time. Thus, the third part of Affinity-Accept's design is a mechanism to dynamically balance the load offered by the NIC's RX DMA rings to each core, to counteract both short- and long-term variations (§3.3).

3.1 Connection Routing in the NIC

Affinity-Accept assumes that a multi-core machine includes a NIC, such as Intel's 82599 10Gbit Ethernet (IXGBE) card, that exposes multiple hardware DMA rings. Assigning each core one RX and one TX DMA ring spreads the load of processing incoming and outgoing packets among many cores. Additionally, a performance benefit of multiple hardware DMA rings is that cores do not need to synchronize when accessing their own DMA rings.

To solve the connection affinity problem, Affinity-Accept must configure the NIC to route incoming packets from the same connection to the same core. To do this, Affinity-Accept leverages packet routing support in the NIC. The NIC hardware typically hashes each packet's flow identifier five-tuple (the protocol number, source and destination IP addresses, and source and destination port numbers) and uses the resulting hash value to look up the RX DMA ring where the packet will be placed. Since all packets from a single connection will have the same flow hash values, the NIC will deliver all packets from a single connection into a single DMA ring.

The IXGBE card supports two mechanisms to map flow hash values to RX DMA rings; here, we omit some details for simplicity. The first is called Receive-Side Scaling, or RSS [5]. RSS uses the flow hash value to index a 128-entry table. Each entry in the table is a 4-bit identifier for an RX DMA ring. A limitation of this mechanism is that a 4-bit identifier allows for routing packets to only 16 distinct DMA rings. This is a limitation particular to the IXGBE card; other NICs can route packets to all DMA rings.

The second mechanism supported by IXGBE uses a flow steering table and is called Flow Direction, or FDir [14]. FDir can route packets to 64 distinct DMA rings, and Affinity-Accept uses FDir. FDir works by looking up the flow hash value in a hash table. This hash table resides in the NIC, and the kernel can modify it by issuing special requests to the card. Each entry in the hash table maps a flow's hash value to a 6-bit RX DMA ring identifier. The table is bounded by the size of the NIC's memory, which is also used to hold the NIC's FIFOs; in practice, this means the table can hold anywhere from 8K to 32K flow steering entries.

Affinity-Accept requires the NIC to route every incoming packet to one of the available DMA rings. As described above, FDir can only map specific flows to specific DMA rings, and the FDir table does not have sufficient space to map the hash value for every possible flow's five-tuple. To avoid this problem, we change the NIC's flow hash function to use only a subset of the packet's five-tuple. In particular, we instruct the NIC to hash the low 12 bits of the source port number, resulting in at most 4,096 distinct hash values. Each hash value now represents an entire family of flows, which we call a *flow group*. We then insert FDir hash table entries for each one of these 4,096 flow groups, and map them to RX DMA rings to distribute load between cores. This method frees the kernel from communicating with the NIC on every new connection, and avoids the need for an entry per active connection in the hardware table. As we will describe later, achieving good load balance requires having many more flow groups than cores.

3.2 Accepting Local Connections

In order to efficiently accept local connections, Affinity-Accept must first eliminate the single listen socket lock, shown in Figure 1. The listen socket lock protects two data structures: the request hash table, and the accept queue. To remove the single listen socket lock, we use the well-known technique of splitting a single lock protecting a data structure into many finer-grained locks each protecting a part of the data structure. Affinity-Accept removes the lock

by partitioning the accept queue into per-core accept queues, each protected by its own lock, and by using a separate lock to protect each bucket in the request hash table. These changes avoid lock contention on the listen socket. We describe these design decisions in more detail in §5.

To achieve connection affinity, Affinity-Accept modifies the behavior of the `accept()` system call. When an application calls `accept()`, Affinity-Accept returns a connection from the local core's accept queue for the corresponding listen socket, if available. If no local connections are available, `accept()` goes to sleep (as will be described in §3.3, the core first checks other core's queues before going to sleep). When new connections arrive, the network stack wakes up any threads waiting on the local core's accept queue. This allows all connection processing to occur locally.

3.3 Connection Load Balancer

Always accepting connections from the local accept queue, as described in §3.2, addresses the connection affinity problem, but introduces potential load imbalance problems. If one core cannot keep up with incoming connections in its local accept queue, the accept queue will overflow, and the kernel will drop connection requests, adversely affecting the client. However, even when one core is too busy to accept connections, other cores may be idle. An ideal system would offload connections from the local accept queue to other idle cores.

There are two cases for why some cores may be able to process more connections than others. The first is a short-term load spike on one core, perhaps because that core is handling a CPU-intensive request, or an unrelated CPU-intensive process runs on that core. To deal with short-term imbalance, Affinity-Accept performs *connection stealing*, whereby an application thread running on one core accepts incoming connections from another core. Since connection stealing transfers one connection at a time between cores, updating the NIC's flow routing table for each stolen connection would not be worthwhile.

The second case is a longer-term load imbalance, perhaps due to an uneven distribution of flow groups in the NIC, due to unrelated long-running CPU-intensive processes, or due to differences in CPU performance (e.g., some CPUs may be further away from DRAM). In this case, Affinity-Accept's goal is to preserve efficient local processing of connections. Thus, Affinity-Accept must match the load offered to each core (by packets from the NIC's RX DMA rings) to the application's throughput on that core. To do this, Affinity-Accept implements *flow group migration*, in which it changes the assignment of flow groups in the NIC's FDir table (§3.1).

In the rest of this subsection, we first describe connection stealing in §3.3.1, followed by flow group migration in §3.3.2.

3.3.1 Connection Stealing

Affinity-Accept's connection stealing mechanism consists of two parts: the first is the mechanism for stealing a connection from another core, and the second is the logic for determining when stealing should be done, and determining the core from which the connection should be stolen.

Stealing mechanism. When a *stealer* core decides to steal a connection from a *victim* core, it acquires the lock on the victim's local accept queue, and dequeues a connection from it. Once a connection has been stolen, the stealer core executes application code to process the connection, but the victim core still performs processing of incoming packets from the NIC's RX DMA ring. This is because the FDir table in the NIC cannot be updated on a per-flow basis. As a result, the victim core is still responsible for performing some amount of processing on behalf of the stolen connection. Thus, short-term connection stealing temporarily violates Affinity-Accept's goal of connection affinity, in hope of resolving a load imbalance.

Stealing policy. To determine when one core should steal incoming connections from another core, Affinity-Accept designates each core to be either *busy* or *non-busy*. Each core determines its own busy status depending on the length of its local accept queue over time; we will describe this algorithm shortly. Non-busy cores try to steal connections from busy cores, in order to even out the load in the short term. Busy cores never steal connections from other cores.

When an application calls `accept()` on a non-busy core, Affinity-Accept can either choose to dequeue a connection from its local accept queue, or from the accept queue of a busy remote core. When both types of incoming connections are available, Affinity-Accept must maintain efficient local processing of incoming connections, while also handling some connections from remote cores. To do this, Affinity-Accept implements proportional-share scheduling. We find that a ratio of 5 : 1 between local and remote connections accepted appears to work well for a range of workloads. The overall performance is not significantly affected by the choice of this ratio. Ratios that are too low start to prefer remote connections in favor of local ones, and ratios that are too high do not steal enough connections to resolve a load imbalance.

Each non-busy core uses a simple heuristic to choose from which remote core to steal. Cores are deterministically ordered. Each core keeps a count of the last remote core it stole from, and starts searching for the next busy core one past the last core. Thus, non-busy cores steal in a round-robin fashion from all busy remote cores, achieving fairness and avoiding contention. Unfortunately, round robin does not give preference to any particular remote queue, even if some cores are more busy than others. Investigating this trade-off is left to future work.

Tracking busy cores. An application specifies the maximum accept queue length in the `listen()` system call. Affinity-Accept splits the maximum length evenly across all cores; this length is called the *maximum local accept queue*

length. Each core tracks the instantaneous length of its local queue which we simply call the local accept queue length.

Each core determines its busy status based on its local accept queue length. Using the max local accept queue length, Affinity-Accept sets high and low watermark values for the queue length. These values determine when a core gets marked as busy, and when a core's busy status is cleared. Once a core's local accept queue length exceeds the high watermark, Affinity-Accept marks the core as busy. Since many applications accept connections in bursts, the length of the accept queue can have significant oscillations. As a result, Affinity-Accept is more cautious about marking cores non-busy: instead of using the instantaneous queue length, it uses a running average. This is done by updating an Exponentially Weighted Moving Average (EWMA) each time a connection is added to the local queue. EWMA's alpha parameter is set to one over twice the max local accept queue length (for example, if an application's max local accept queue length is 64, alpha is set to $1/128$). A small alpha ensures that the EWMA tracks the long term queue length, because the instantaneous queue length oscillates around the average. The work stealer marks a core non-busy when this average drops below the low watermark.

We have experimentally determined that setting the high and low watermarks to be 75% and 10% of the max local accept queue length works well with our hardware; the numbers may need to be adjusted for other hardware. Developers of Apache and lighttpd recommend configuring the accept queue length to 512 or 1024 for machines with a low core count. The length must be adjusted for larger machines. We found a queue length of 64 to 256 per core works well for our benchmarks.

To ensure that finding busy cores does not become a performance bottleneck in itself, Affinity-Accept must allow non-busy cores to efficiently determine which other cores, if any, are currently busy (in order to find potential victims for connection stealing). To achieve this, Affinity-Accept maintains a bit vector for each listen socket, containing one bit per core, which reflects the busy status of that core. With a single read, a core can determine which other cores are busy. If all cores are non-busy, the cache line containing the bit vector will be present in all of the cores' caches. If the server is overloaded and all cores are perpetually busy, this cache line is not read or updated.

Polling. When an application thread calls `accept()`, `poll()`, or `select()` to wait for an incoming connection, Affinity-Accept first scans the local accept queue. If no connections are available, Affinity-Accept looks for available connections in remote busy cores, followed by remote non-busy cores. If no connections are available in any accept queues, the thread goes to sleep.

When new connections are added to an accept queue, Affinity-Accept first tries to wake up local threads waiting for incoming connections. If no threads are waiting locally, the local core checks for waiting threads on any non-busy remote cores, and wakes them up.

To avoid lost wakeups, waiting threads should first add themselves to the local accept queue's wait queue, then check all other cores' accept queues, and then go to sleep. We have not implemented this scheme yet, and rely on timeouts to catch lost wakeups instead.

3.3.2 Flow Group Migration

Connection stealing reduces the user-space load on a busy core, but it neither reduces the load of processing incoming packets nor allows local connection processing of stolen connections. Thus, to address long-term load imbalance, Affinity-Accept migrates flow groups (§3.1) between cores.

To determine when flow group migration should occur, Affinity-Accept uses a simple heuristic based on how often one core has stolen connections from other cores. Every 100*ms*, each non-busy core finds the victim core from which it has stolen the largest number of connections, and migrates one flow group from that core to itself (by reprogramming the NIC's FDir table). In our configuration (4,096 flow groups and 48 cores), stealing one flow group every 100*ms* was sufficient. Busy cores do not migrate additional flow groups to themselves.

4. Connection Affinity in Applications

Applications must adhere to a few practices to realize the full performance gains of Affinity-Accept. This section describes those practices in the context of web servers, but the general principles apply to all types of networking applications.

4.1 Thundering Herd

Some applications serialize calls to `accept()` and `poll()` to avoid the *thundering herd* problem, where a kernel wakes up many threads in response to a single incoming connection. Although all of the newly woken threads would begin running, only one of them would accept a connection. The kernel would then put the remainder back to sleep, wasting cycles in these extra transitions. Historically, applications used to serialize their calls to `accept()` to avoid this problem. However, Linux kernel developers have since changed the `accept()` system call to only wake up one thread. Applications that still serialize calls to `accept()` do so only for legacy reasons.

For Affinity-Accept to work, multiple threads must call `accept()` and `poll()` concurrently on multiple cores. If not, the same scalability problems that manifested due to a single socket lock will appear in the form of user-space serialization. Many web servers use the `poll()` system call to wait for events on the listen socket. The Linux kernel, unfortunately, still wakes up multiple threads that use `poll()` to wait for the same event. Affinity-Accept significantly reduces the thundering herd problem for `poll()` by reducing the number of threads that are awoken. It uses multiple accept queues and only wakes up threads waiting in `poll()` on the local core.

4.2 Application Structure

To get optimal performance with Affinity-Accept, calls to `accept()`, `recvmsg()`, and `sendmsg()` on the same connection must take place on the same core. The architecture of the web server determines whether this happens or not.

An event-driven web server like lighttpd [4] adheres to this guideline. Event-driven servers typically run multiple processes, each running an event loop in a single thread. On calls to `accept()` the process gets back a connection with an affinity for the local core. Subsequent calls to `recvmsg()` and `sendmsg()` therefore also deal with connections that have an affinity for the local core. The designers of such web servers recommend spawning at least two processes per available core [6] to deal with file system I/O blocking a processes and all of its pending requests. If one process blocks, another non-blocked process may be available to run. The Linux process load balancer distributes the multiple processes among the available cores. One potential concern with the process load balancer is that it migrates processes between cores when it detects a load imbalance. All connections the process accepts after the migration would have an affinity to the new core, but existing connections would have affinity for the original core. §6 shows that this is not a problem because the Linux load balancer rarely migrates processes, as long as the load is close to even across all cores.

The Apache [2] web server has more modes of operation than lighttpd, but none of Apache's modes are ideal for Affinity-Accept without additional changes. In "worker" mode, Apache forks multiple processes; each accepts connections in one main thread and spawns multiple threads to process those connections. The problem with using this design with Affinity-Accept is that the scheduler disperses the threads across cores, causing the accept and worker threads to run on different cores. As a result, once the accept thread accepts a new connection, it hands it off to a worker thread that is executing on another core, violating connection affinity.

Apache's "prefork" mode is similar to lighttpd in that it forks multiple processes, each of which accepts and processes a single connection to completion. Prefork does not perform well with Affinity-Accept for two reasons. First, prefork uses many more processes than worker mode, and thus spends more time context-switching between processes. Second, each process allocates memory from the DRAM controller closest to the core on which it was forked, and in prefork mode, Apache initially forks all processes on a single core. Once the processes are moved to another core by the Linux process load balancer, memory operations become more expensive since they require remote DRAM accesses.

The approach we take to evaluate Apache's performance is to use the "worker" mode, but to pin Apache's accept and worker threads to specific cores, which avoids these problems entirely. However, this does require additional setup configuration at startup time to identify the correct number of cores to use, and reduces the number of threads which the Linux process load balancer can move between cores to address a load imbalance. A better solution would be to add a new kernel interface for specifying groups of threads which the kernel should schedule together on the same core. Designing such an interface is left to future work.

5. Implementation

Affinity-Accept builds upon the Linux 2.6.35 kernel, patched with changes described by Boyd-Wickizer et al [9], and includes the TCP listen socket changes described in §3. Affinity-Accept does not create any new system calls. We added 1,200 lines of code to the base kernel, along with a new kernel module to implement the connection load balancer in about 800 lines of code.

We used the 2.0.84.9 IXGBE driver. We did not use a newer driver (version 3.3.9) because we encountered a 20% performance regression. We modified the driver to add a mode that configures the FDir hardware so that the connection load balancer could migrate connections between cores. We also added an interface to migrate flow groups from one core to another. The changes required about 700 lines of code.

We modified the Apache HTTP Server version 2.2.14 to disable a mutex (described in §4) used to serialize multiple processes on calls to `accept()` and `poll()`.

5.1 Per-core Accept Queues

One of the challenging aspects of implementing Affinity-Accept was to break up the single lock and accept queue in each listen socket into per-core locks and accept queues. This turned out to be challenging because of the large amount of Linux code that deals with the data structures in question. In particular, Linux uses a single data structure, called a `sock`, to represent sockets for any protocol (TCP, UDP, etc). Each protocol specializes the `sock` structure to hold its own additional information for that socket (such as the request hash table and accept queue, for a TCP listen socket). Some functions, especially those in early packet processing stages, manipulate the `sock` part of the data structure. Other operations are protocol-specific, and use a table of function pointers to invoke protocol-specific handlers. Importantly, socket locking happens on the `sock` data structure directly, and does not invoke a protocol-specific function pointer; for example, the networking code often grabs the `sock` lock, calls a protocol-specific function pointer, and then releases the lock. Thus, changing the locking policy on `sock` objects would require changing the locking policy throughout the entire Linux network stack.

To change the listen socket locking policy without changing the shared networking code that deals with processing `sock` objects, we *clone* the listen socket. This way, there is a per-core copy of the original listen socket, each protected by its own socket lock. This ensures cores can manipulate their per-core clones in parallel. Most of the existing code does not need to change because it deals with exactly the same

type of object as before, and the `sock` locking policy remains the same. Code specific to the listen socket implementation does manipulate state shared by all of the clones (e.g., code in `accept` that performs connection stealing), but such changes are localized to the protocol-specific implementation of the listen socket. Additionally, we modified generic socket code that expects only one socket to be aware of cloned sockets. For example, we aggregate data from all cloned sockets when reporting statistics.

5.2 Fine-grained Request Hash Table Locking

The other data structure protected by the single listen socket lock is the request hash table (Figure 1). One approach to avoid a single lock on the request hash table is to break it up into per-core request hash tables, as with the accept queue above. However, this leads to a subtle problem when flow groups are migrated between cores. Due to flow group migration, a connection's SYN packet could arrive on one core (creating a request socket in its per-core request hash table), and the corresponding ACK packet could arrive on another core. The code to process the ACK would not find the request socket in its local request hash table. The same problem does not occur with established TCP sockets because the kernel maintains a global hash table for established connections, and uses fine-grained locking to avoid contention.

At this point, we are left with two less-than-ideal options: the core that receives an ACK packet could drop it on the floor, breaking the client's connection, or it could scan all other cores' request hash tables looking for the corresponding request socket, which would be time-consuming and interfere with all other cores. Since neither option is appealing, we instead maintain a single request hash table, shared by all clones of a listen socket, and use per-hash-table-bucket locks to avoid lock contention. We have experimentally verified using the benchmark described in §6 that this design incurs at most a 2% performance reduction compared to the per-core request hash table design.

6. Evaluation

In this section we describe the experimental setup (§6.1) and workload (§6.2). Then we show that the listen socket lock in stock Linux is a major scalability bottleneck, and splitting the single lock into many fine grained locks alleviates the problem (§6.3). After removing the lock bottleneck, data sharing is the predominant problem and Affinity-Accept further improves throughput by reducing sharing (§6.4). Clients can experience poor service using Affinity-Accept without a load balancer, and our proposed load balancer addresses this imbalance (§6.5). We end with a characterization of the type of workloads Affinity-Accept helps (§6.6).

6.1 Hardware Setup

We run experiments on two machines. The first is a 48-core machine, with a Tyan Thunder S4985 board and an M4985 quad CPU daughterboard. The machine has a total of eight 2.4 GHz 6-core AMD Opteron 8431 chips. Each core has private 64 Kbyte instruction and data caches, and a 512 Kbyte private L2 cache. The cores on each chip share a 6 Mbyte L3 cache, 1 Mbyte of which is used for the HT Assist probe filter [8]. Each chip has 8 Gbytes of local off-chip DRAM. Table 1 shows the access times to different memory levels. The machine has a dual-port Intel 82599 10Gbit Ethernet card, though we use only one port for all experiments. That port connects to an Ethernet switch with a set of load-generating client machines.

	Local Latency (cycles)				Remote Latency (cycles)	
	L1	**L2**	**L3**	**RAM**	**L3**	**RAM**
AMD	3	14	28	120	460	500
Intel	4	12	24	90	200	280

Table 1: Access times to different levels of the memory hierarchy. Remote accesses are between two chips farthest on the interconnect.

The second machine is composed of eight 2.4 GHz 10-core Intel Xeon E7 8870 chips. Each core has a private 32-Kbyte data and instruction cache, and a private 256 Kbyte L2 cache. All 10 cores on one chip share a 30 Mbyte L3 cache. Each chip has 32 Gbytes of local off-chip DRAM. Table 1 shows the memory access times for the Intel machine. The machine is provisioned with two dual-port Intel 82599 10Gbit Ethernet cards. Each port exposes up to 64 hardware DMA rings, which is less than the machine's core count. For experiments that use 64 or fewer cores we use a single port; for experiments with more than 64 cores we add another port from the second card. A second port adds up to 64 more DMA rings and each core can have a private DMA ring.

6.2 Workload

To evaluate the performance of network-heavy applications with and without Affinity-Accept, we measure the rate at which a machine can serve static web content, in terms of HTTP requests per second. This is distinct from connections per second because a single HTTP connection can issue multiple requests.

We profile two web servers, lighttpd [4] and Apache [2], to show that two different architectures work with Affinity-Accept. We use 25 client machines with a total of 54 cores, running the httperf [3] HTTP request generator. A single httperf instance issues many requests in parallel, up to the maximum number of open file descriptors (1024). We run 4 httperf instances per client machine core so that httperf is not limited by file descriptors. Httperf works by generating a target request rate. In all experiments we first search for a request rate that saturates the server and then run the experiment with the discovered rate.

The content is a mix of files inspired by the static parts of the SpecWeb [7] benchmark suite. We do not use SpecWeb

directly because it does not sufficiently stress the network stack: some requests involve performing SQL queries or running PHP code, which stresses the disk and CPU more than the network stack. Applications that put less stress on the network stack will see less pronounced improvements with Affinity-Accept. The files served range from 30 bytes to 5,670 bytes. The web server serves 30,000 distinct files, and a client chooses a file to request uniformly over all files.

Unless otherwise stated, in all experiments a client requests a total of 6 files per connection with requests spaced out by think time. First, a client requests one file and waits for 100*ms*. The client then requests two more files, waits 100*ms*, requests three more files, and finally closes the connection. §6.6 shows that the results are independent of the think time.

We configure lighttpd with 10 processes per core for a total of 480 processes on the AMD machine. Each process is limited to a maximum of 200 connections. Having several processes handling connections on each core limits the number of broken connection affinities if the Linux scheduler migrates one of the processes to another core, and reduces the number of file descriptors each process must pass to the kernel via `poll()`.

We run Apache in worker mode and spawn one process per core. Each process consists of one thread that only accepts connections and multiple worker threads that process accepted connections. We modify the worker model to pin each process to a separate core. All threads in a process inherit the core affinity of the process, and thus the accept thread and worker threads always run on the same core. A single thread processes one connection at a time from start to finish. We configure Apache with 1,024 worker threads per process, which is enough to keep up with the load and think time.

We use a few different implementations of the listen socket to evaluate our design. We first compare Affinity-Accept to a stock Linux listen socket that we call "Stock-Accept" and then a second intermediate listen socket implementation that we refer to as "Fine-Accept". Fine-Accept is similar to Affinity-Accept, but does not maintain connection affinity to cores. On calls to `accept()`, Fine-Accept dequeues connections out of cloned accept queues in a round-robin fashion. This scheme performs better than Stock-Accept's single accept queue, because with multiple accept queues, each queue is protected by a distinct lock, and multiple connections can be accepted in parallel. The Fine-Accept listen socket does not need a load balancer because accepting connection round-robin is intrinsically load balanced: all queues are serviced equally. In all configurations we use the NIC's FDir hardware to distribute incoming packets among all hardware DMA rings (as described in §3.1) and we configure interrupts so that each core processes its own DMA ring.

6.3 Socket Lock

First, we measure the throughput achieved with the stock Linux listen socket, which uses a single socket lock. The Stock-Accept line in Figure 2 shows the scalability of Apache on the AMD machine. The number of requests each core can process decreases drastically as the number of cores increases (in fact, the total number of requests handled per second stays about the same, despite the added cores). There is an increasing amount of idle time past 12 cores because the socket lock works in two modes: spinlock mode where the kernel busy loops and mutex mode where the kernel puts the thread to sleep.

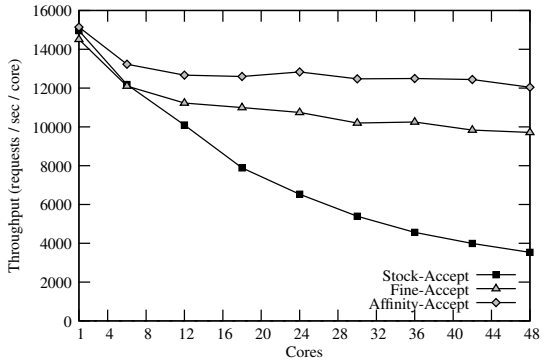

Figure 2: Apache performance with different listen socket implementations on the AMD machine.

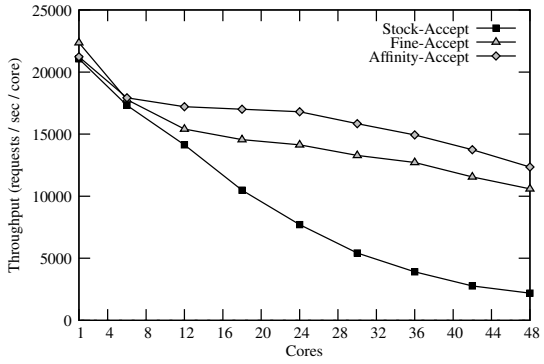

Figure 3: Lighttpd performance with different listen socket implementations on the AMD machine.

To understand the source of this bottleneck, the Stock-Accept row in Table 2 shows the cost of acquiring the socket lock when running Apache on a stock Linux kernel on the AMD machine. The numbers are collected using `lock_stat`, a Linux kernel lock profiler that reports, for all kernel locks, how long each lock is held and the wait time to acquire the lock. Using `lock_stat` incurs substantial overhead due to accounting on each lock operation, and `lock_stat` does not track the wait time to acquire the socket lock in mutex mode; however, the results do give a picture of which locks are contended. Using Stock-Accept, the machine can process a request in 590 μs, 82 μs of which it waits to acquire the listen socket lock in spin mode and at most 320 μs in mutex mode. Close to 70% of the time is spent waiting for another core. Thus, the decline observed for Stock-Accept in Figure 2 is due to contention on the listen socket lock.

Listen Socket	Throughput (requests / sec / core)	Total Time	Idle Time	Non-Idle Time		
				Socket Lock Wait Time	Socket Lock Hold Time	Other Time
Stock-Accept	1,700	590 μs	320 μs	82 μs	25 μs	163 μs
Fine-Accept	5,700	178 μs	8 μs	0 μs	30 μs	140 μs
Affinity-Accept	7,000	144 μs	4 μs	0 μs	17 μs	123 μs

Table 2: The composition of time to process a single request with Apache running on the AMD machine with all 48 cores enabled. These numbers are for a `lock_stat` enabled kernel; as a consequence the throughput numbers, shown in the first column, are lower than in other experiments. The total time to process a request, shown in the second column, is composed of both idle and non-idle time. The idle time is shown in the third column; included in the idle time is the wait time to acquire the socket lock in mutex mode. The last three columns show the composition of active request processing time. The fourth column shows the time the kernel waits to acquire the socket lock in spinlock mode and the fifth column shows the time the socket lock is held once it is acquired. The last column shows the time spent outside the socket lock.

To verify that Apache's thread pinning is not responsible for Affinity-Accept's performance advantage, Figure 3 presents results from the same experiment with lighttpd, which does not pin threads. Affinity-Accept again consistently achieves higher throughput. The downward slope of Affinity-Accept is due to lighttpd's higher performance that saturates the NIC: the NIC hardware is unable to process any additional packets. Additionally the higher request processing rate triggers a scalability limitation in how the kernel tracks reference counts to file objects; we have not yet explored workarounds or solutions for this problem.

The Affinity-Accept line in Figure 2 shows that the scalability of Apache improves when we use the Affinity-Accept listen socket. Part of the performance improvement comes from the reduced socket lock wait time, as shown in Affinity-Accept row of Table 2. Part of the improvement also comes from improved locality, as we evaluate next.

6.4 Cache Line Sharing

To isolate the performance gain of using fine grained locking from gains due to local connection processing, we analyze the performance of Fine-Accept. The Fine-Accept row in Table 2 confirms that Fine-Accept also avoids bottlenecks on the listen socket lock. However, Figures 2 and 3 show that Affinity-Accept consistently outperforms Fine-Accept. This means that local connection processing is important to achieving high throughput, even with fine-grained locking. In case of Apache, Affinity-Accept outperforms Fine-Accept by 24% at 48 cores and in the case of lighttpd by 17%.

Tracking misses. In order to find out why Affinity-Accept outperforms Fine-Accept, we instrumented the kernel to record a number of performance counter events during each type of system call and interrupt. Table 3 shows results of three performance counters (clock cycles, instruction count, and L2 misses) tracking only kernel execution. The table also shows the difference between Fine-Accept and Affinity-Accept. The `softirq_net_rx` kernel entry processes incoming packets. These results show that Fine-Accept uses 40% more clock cycles than Affinity-Accept to do the same amount of work in `softirq_net_rx`. Summing the cycles column over network stack related system calls and interrupts, the improvement from Fine-Accept to Affinity-Accept is 30%. The application level improvement due to Affinity-Accept, however, is not as high at 24%. The is because the machine is doing much more than just processing packets when it runs Apache and lighttpd. Both implementations execute approximately the same number of instructions; thus, the increase is not due to executing more code. The number of L2 misses, however, doubles when using Fine-Accept. These L2 misses indicate that cores need to load more cache lines from either the shared L3, remote caches, or DRAM.

Kernel Entry	Cycles		Instructions		L2 Misses	
	Total	Δ	Total	Δ	Total	Δ
`softirq_net_rx`	97k / 69k	28k	33k / 34k	-788	352 / 178	174
`sys_read`	17k / 10k	7k	4k / 4k	260	60 / 31	29
`schedule`	23k / 17k	6k	9k / 8k	450	79 / 38	41
`sys_accept4`	12k / 7k	5k	3k / 2k	666	38 / 19	19
`sys_writev`	15k / 12k	3k	5k / 4k	120	53 / 33	20
`sys_poll`	12k / 9k	3k	4k / 4k	94	39 / 17	22
`sys_shutdown`	8k / 6k	3k	3k / 3k	55	28 / 7	21
`sys_futex`	18k / 16k	3k	8k / 8k	357	56 / 45	11
`sys_close`	5k / 4k	707	2k / 2k	29	12 / 10	2
`softirq_rcu`	714 / 603	111	212 / 204	8	4 / 3	1
`sys_fcntl`	375 / 385	-10	275 / 276	-1	0 / 0	0
`sys_getsockname`	706 / 719	-13	277 / 275	2	1 / 1	0
`sys_epoll_wait`	2k / 2k	-29	568 / 601	-33	3 / 2	1

Table 3: Performance counter results categorized by kernel entry point. System call kernel entry points begin with "sys", and timer and interrupt kernel entry points begin with "softirq". Numbers before and after the slash correspond to Fine-Accept and Affinity-Accept, respectively. Δ reports the difference between Fine-Accept and Affinity-Accept. The kernel processes incoming connection in `softirq_net_rx`.

To understand the increase in L2 misses, and in particular, what data structures are contributing to the L2 misses, we use DProf [19]. DProf is a kernel profiling tool that, for a particular workload, profiles the most commonly used data structures and the access patterns to these data structures. Table 4 shows that the most-shared objects are those associated with connection and packet processing. For example the `tcp_sock` data type represents a TCP established socket. Cores share 30% of the bytes that make up this structure.

Data Type	Size of Object (bytes)	% of Object's Cache Lines Shared	% of Object's Bytes Shared	% of Object's Bytes Shared RW	Cycles Accessing Bytes Shared in Fine-Accept, per HTTP Req
tcp_sock	1664	85 / 12	30 / 2	22 / 2	54974 / 30584
sk_buff	512	75 / 25	20 / 2	17 / 2	17586 / 9882
tcp_request_sock	128	100 / 0	22 / 0	12 / 0	5174 / 3278
slab:size-16384	16384	5 / 1	1 / 0	1 / 0	1531 / 1123
slab:size-128	128	100 / 0	9 / 0	9 / 0	1117 / 51
slab:size-1024	1024	38 / 0	4 / 0	4 / 0	882 / 24
slab:size-4096	4096	19 / 5	1 / 0	1 / 0	417 / 136
socket_fd	640	10 / 10	2 / 2	2 / 2	348 / 23
slab:size-192	192	100 / 33	17 / 2	17 / 2	- / -
task_struct	5184	10 / 0	2 / 0	2 / 0	- / -
file	192	100 / 100	8 / 8	8 / 8	- / -

Table 4: DProf results for Fine-Accept, shown before the slash, and Affinity-Accept, shown after the slash. Results for Fine-Accept show that the data types shared across multiple cores are those used by the network stack. The `tcp_sock` and `tcp_request_sock` are established and request TCP socket objects respectively. A `sk_buff` holds packet data and a `socket_fd` is used to represent an established or listen socket as a file descriptor. Data types that start with "slab" are generic kernel buffers (including packet buffers). Entries with "-" indicate that we did not collect latency data for the data structure.

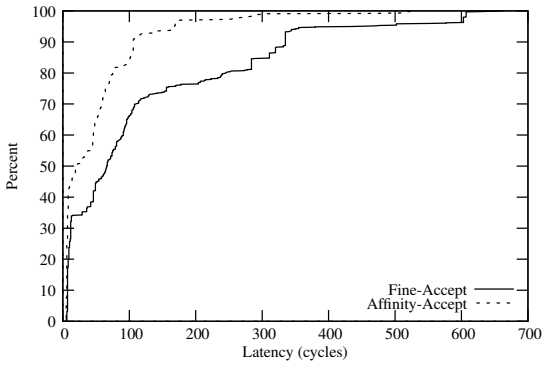

Figure 4: CDF of memory access latencies to shared memory locations reported in the right column of Table 4.

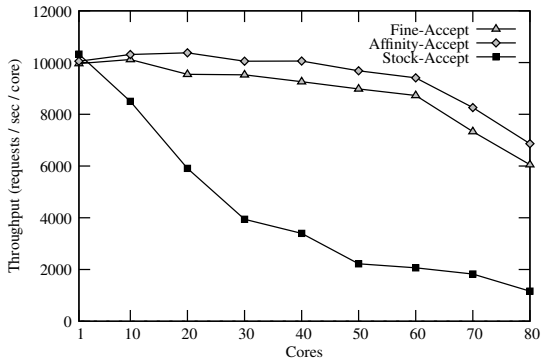

Figure 5: Apache performance with different listen socket implementations on the Intel machine.

Worse yet, these shared bytes are not packed into a few cache lines but spread across the data structure, increasing the number of cache lines that cores need to fetch. For an established socket, cores share 85% of the cache lines.

To get an understanding of how long it takes a core to access shared cache lines, we measure access times for individual load instructions directly. For the top shared data structures, DProf reports all load instructions that access a shared cache line. The set of instructions is bigger for Fine-Accept than for Affinity-Accept because Fine-Accept shares more cache lines. When collecting absolute memory access latencies for Affinity-Accept, we instrument the set of instructions collected from running DProf on Fine-Accept; this ensures that we capture the time to access data that is no longer shared. We measure the access latency by wrapping the load with `rdtsc` and `cpuid` instructions. The `rdtsc` instruction reads a time stamp which the hardware increments on each cycle. We use the `cpuid` instruction to force in-order execution of the `rdtsc` and load instructions. The difference between the before and after time stamps is the absolute access latency. The absolute access latency is distinct from the duration a load instruction stalls a core. The core may not stall at all if it can execute other instructions while waiting for the load to finish. Nevertheless, the measurement is a good indication of from how far away the data is fetched.

The last column in Table 4 shows the total per request absolute latency to access the shared cache lines of each data type. Affinity-Accept reduces the number of cycles needed to access shared cache lines of `tcp_sock` by more than 50%. We also plot the results as a CDF in Figure 4. Accesses with long latencies are either directed to memory or to remote caches, and the CDF shows that Affinity-Accept considerably reduces long latency memory accesses over Fine-Accept.

The results in Table 4 show that Affinity-Accept removes most of the sharing. The sharing that is left is due to accesses to global data structures. For example, the kernel adds `tcp_sock` objects to global lists. Multiple cores manipulate these lists and thus modify the cache lines that make up the linked list pointers. The minimal amount of sharing remaining with Affinity-Accept suggests that it will continue to scale better than Fine-Accept as more cores are added.

Intel machine. We also evaluate the performance of Affinity-Accept on a machine with a different architecture. Figure 5 shows the performance of Apache and Figure 6

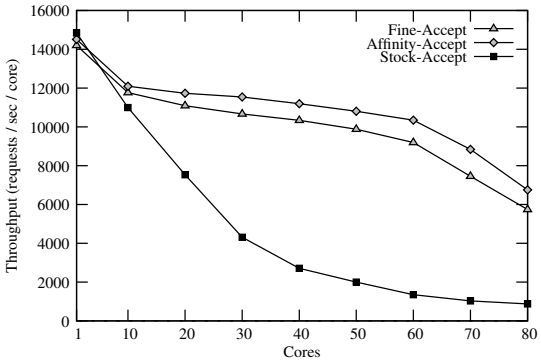

Figure 6: Lighttpd performance with different listen socket implementations on the Intel machine.

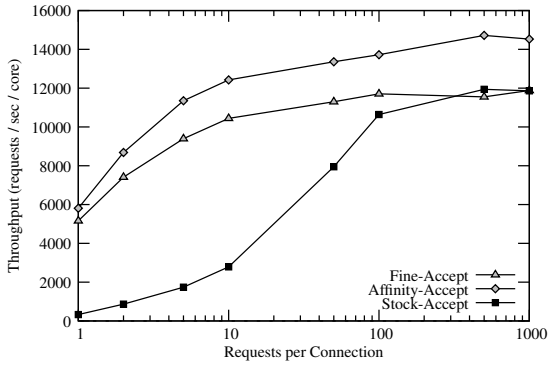

Figure 7: The effect of TCP connection reuse on Apache's throughput running on the AMD machine. The accept rate decreases as clients send more HTTP requests per TCP connection.

shows the performance of lighttpd on our Intel system. Affinity-Accept outperforms Fine-Accept by a smaller margin on this system than on the AMD machine. We suspect that this difference is due to faster memory accesses and a faster interconnect.

6.5 Load Balancer

To evaluate the load balancer we want to show two things. First, that without connection stealing, accept queues overflow and affect the performance perceived by clients. Second, that flow group migration reduces the incoming packet load on cores not processing network packets, and speeds up other applications running on these cores.

The first experiment illustrates that the load balancer can deal with cores that suddenly cannot keep up with the incoming load. We test this by running the web server benchmark on all cores but adjusting the load so the server uses only 50% of the CPU time. For each connection, the client terminates the connection after 10 seconds if it gets no response from the server. To reduce the processing capacity of the test machine, we start a build of the Linux kernel using parallel `make` on half of the cores (using `sched_setaffinity()` to limit the cores on which `make` can run). Each client records the time to service each connection, and we compute the median latency.

Running just the web server benchmark yields a median and 90th percentile latency of 200*ms* with and without the load balancer. This includes both the time to processes the 6 requests, as well as the two 100*ms* client think times, indicating that request processing takes under 1*ms*. When `make` is running on half of the cores, the median and 90th percentile latencies jump to 10 seconds in the absence of a load balancer, because the majority of connections time out at the client before they are serviced by the web server. The timeouts are due to accept queue overflows on cores running `make`. Enabling load balancing reduces the median latency to 230*ms*, and the 90th percentile latency to 480*ms*. The extra 30*ms* in median latency is due to the 100% utilization of cores still exclusively running lighttpd, and is a consequence of taking a workload that uses 50% of CPU time on 48 cores and squeezing it onto 24 cores. If the initial web server utilization is less than 50%, the median latency falls back to 200*ms*.

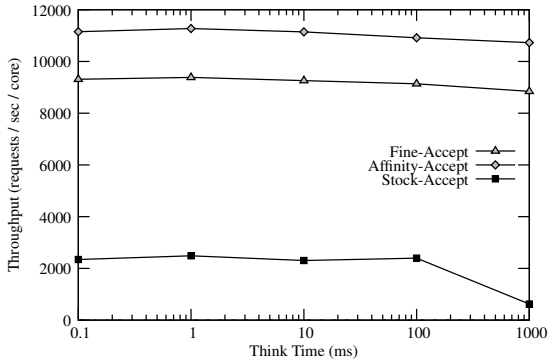

Figure 8: The effect of increasing client think time on Apache's throughput running on the AMD machine.

The second experiment shows that flow group migration improves non-web server application performance. We run the same experiment as above, with connection stealing enabled, but this time measure the runtime of the kernel compile. As a base line, the compilation takes 125*s* on 24 cores without the web server running. Adding the web server workload with flow group migration disabled increases the time to 168*s*. Enabling flow group migration reduces the time to 130*s*. The extra 5*s* is due to the time it takes flow group migration to move all flow groups away from the cores running `make`. This migration actually happens twice, because the kernel `make` process has two parallel phases separated by a multi-second serial process. During the break between the two phases, Affinity-Accept migrates flow groups back to the cores that were running `make`.

6.6 Variations to the Workload

The evaluation thus far concentrated on short-lived connections. This section examines the effect of three parameters of the workload on Affinity-Accept performance: accept rate,

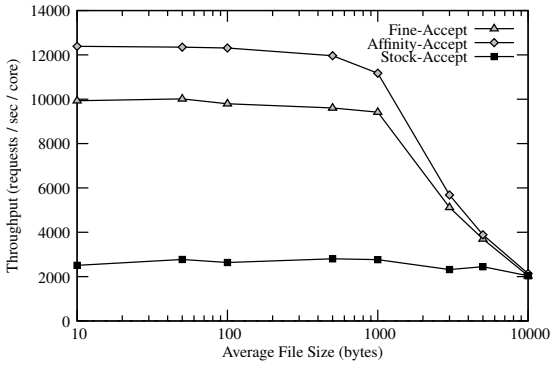

Figure 9: The effect of different average file sizes on Apache's throughput running on the AMD machine. The average file size is of all serviced files.

client think time, and average served file size. All of the experiments in this section were run with all CPUs enabled.

Accept Rate. The first workload variation we consider is HTTP connection reuse. A client can send multiple requests through a single HTTP connection, which reduces the fraction of accepts in the server's total network traffic. In previous experiments the request per connection ratio is fixed to 6; Figure 7 shows the performance of Apache as the number of requests per connection varies. In this example, Apache is configured to permit an unbounded number of requests per connection instead of the default configuration which limits connection reuse. When the number of requests per connection is small, Affinity-Accept and Fine-Accept outperform Stock-Accept as described in the earlier experiments. As connection reuse increases, total throughput increases, as there is less overhead to initiate and tear down connections. Affinity-Accept outperforms Fine-Accept at all measured points. At very high rates of connection reuse (above 5,000 requests per connection), lock contention for the listen socket is no longer a bottleneck for Stock-Accept, and its performance matches that of Fine-Accept.

Figure 7 also shows that Affinity-Accept provides a benefit over Fine-Accept even when accepts are less frequent, since Affinity-Accept reduces data sharing costs after the kernel accepts connections.

Think Time. Figure 8 shows the effect of increasing the lifetime of a connection by adding think time at the client between requests sent over the same TCP connection. This experiment holds connection reuse constant at 6 requests per connection, so it does not vary the fraction of network traffic devoted to connection initiation. It does add a variable amount of client-side think time between subsequent requests on the same connection to increase the total number of active connections that the server must track. The range of think times in the plot covers the range of delays a server might experience in data center or wide area environments, although the pattern of packet arrival would be somewhat different if the delays were due to propagation delay instead of think time. Beyond the rightmost edge of the plot ($1s$), the server would need more than half a million threads, which our kernel cannot support. Stock-Accept does not perform well in any of these cases due to socket lock contention. Affinity-Accept outperforms Fine-Accept and the two sustain a constant request throughput across a wide range of think times.

This graph also points out the problem with NIC assisted flow redirection. In this experiment at $100ms$ of think time there are more than 50,000 concurrently active connections and at $1s$ of think time more than 300,000 connections. Such a large number of active connections would likely not fit into a current NIC's flow steering table.

Average File Size. Figure 9 shows how file size affects Affinity-Accept. The average file size for previous experiments is around 700 bytes and translates to 4.5 Gbps of traffic at 12,000 requests/second/core. Here we change all files proportionally to increase or decrease the average file size. The performance of Stock-Accept is once again low due to lock contention. At an average file size larger than 1 Kbyte, the NIC's bandwidth saturates for both Fine-Accept and Affinity-Accept; as a consequence, the request rate decreases and server cores experience idle time. The Stock-Accept configuration does not serve enough requests to saturate the NIC, until the average file size reaches about 10 Kbytes.

7. Related Work

There has been previous research that shows processing packets that are part of the same connection on a single core is critical to good networking performance. Nahum et al. [18], Yates et al. [24], Boyd-Wickizer et al. [9], and Willmann et al. [23] all demonstrate that a network stack will scale with the number of cores as long as there are many connections that different cores can processes in parallel. They also show that it is best to process packets of the same connection on the same core to avoid performance issues due to locks and out of order packet processing. We present in detail a method for processing a single connection on the same core. Boyd-Wickizer et al. [9] used an earlier version of this work to get good scalability results from the Linux network stack.

RouteBricks [10] evaluates packet processing and routing schemes on a multicore machine equipped with multiple IXGBE cards. They show that processing a packet exclusively on one core substantially improves performance because it reduces inter-core cache misses and DMA ring locking costs.

Serverswitch [17] applies recent improvements in the programmability of network components to data center networks. Using similar features within future NIC designs could enable a better match between hardware and the needs of systems such as Affinity-Accept.

In addition to network stack organization, there have been attempts to address the connection affinity problem. We describe them in the next two sections.

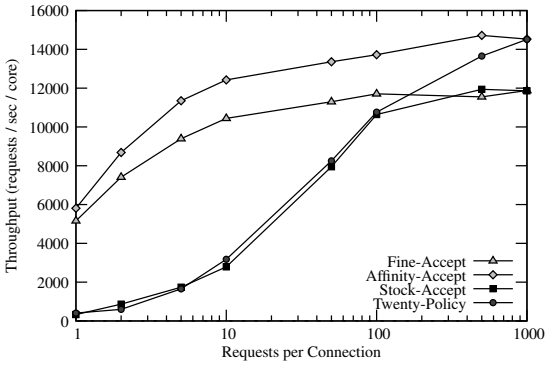

Figure 10: The effect of TCP connection length on Apache's throughput running on the AMD machine. This is a duplicate of Figure 7 but includes "Twenty-Policy": stock Linux with flow steering in hardware.

7.1 Dealing with Connection Affinity in Hardware

The IXGBE driver authors have tried to use FDir to route incoming packets to the core processing outgoing packets. They do so by updating the FDir hash table on every 20th transmitted packet to route incoming packets to the core calling `sendmsg()`. We call this scheme "Twenty-Policy" and Figure 10 shows its performance. At 1,000 requests per connection the NIC does a good job of routing packets to the correct core and the performance matches Affinity-Accept. At 500 requests per connection, however, maintaining the hardware table limits performance. Socket lock contention limits performance below 100 requests per connection; the table maintenance problems would still limit performance even if the socket lock were not a bottleneck.

There are a few problems with Twenty-Policy. First, it is expensive to talk to the network card. It takes 10,000 cycles to add an entry into the FDir hash table. The bulk of this cost comes from calculating the hash value, and the table insert takes 600 cycles.

Second, managing the hash table is difficult. The driver cannot remove individual entries from the hash table, because it does not know when connections are no longer active. The driver, instead, flushes the table when it overflows. It takes up to 80,000 cycles ($\sim 40\mu s$) to schedule the kernel to run the flush operation, and 70,000 cycles ($\sim 35\mu s$) to flush the table. The driver halts packet transmissions for the duration of the flush. A rate of 50,000 connections/second and a hash table with 32K entries requires a flush every 0.6 seconds. We have also confirmed that the NIC misses many incoming packets when running in this mode. Although we do not have a concrete reason, we suspect it is because the NIC cannot keep up with the incoming rate while the flush is in progress. The stopped transmission and missed packets cause TCP timeouts and delays, and the end result is poor performance.

Tighter integration with the network stack can reduce many of these costs. This is exactly the approach Accelerated Receive Flow Steering (aRFS) [13] takes. Instead of on every 20th transmitted packet, an aRFS enabled kernel adds a routing entry to the NIC on calls to `sendmsg()`. To avoid the kernel calculating the connection hash value on a hash table update the NIC reports, in the RX packet descriptor, the hash value of the flow and the network stack stores this value. Unfortunately, the network stack does not notify the driver when a connection is no longer in use so the driver can selectively shoot down connections. Instead, the driver needs to periodically walk the hardware table and query the network stack asking if a connection is still in use. Just as in Twenty-Policy, we see the need for the driver to search for dead connections as a point of inefficiency.

NIC	HW DMA Rings	RSS DMA Rings	Flow Steering Table (# connections)
Intel [14]	64	16	32K
Chelsio [1]	32 or 64	32 or 64	"tens of thousands"
Solarflare [16]	32	32	8K
Myricom [15]	32	32	-

Table 5: Comparison of features available on modern NICs. Entries with "-" indicate that we could not find information.

Even with aRFS, flow steering in hardware is still impractical because the third problem is the hard limit on the size of the NIC's table. Table 5 lists the table sizes for different modern 10Gbit NICs. FreeBSD developers, who are also implementing aRFS-like functionality, have raised similar concerns over hardware table sizes [20].

Additionally, currently available 10Gbit NICs provide limited hardware functionality in one way or another. Table 5 summarizes key NIC features. Each card offers either a small number of DMA rings, RSS supported DMA rings, or flow steering entries. For example, using the IXGBE NIC there is no way to spread incoming load among all cores if FDir is also used to route individual connections to particular cores. In this case, we would have to use RSS for load balancing new connections, which only supports 16 DMA rings. It is imperative for NIC manufacturers to grow the number of DMA rings with increasing core counts and provide functionality to all DMA rings.

7.2 Dealing with Connection Affinity in Software

Routing in software is more flexible than routing in hardware. Google's Receive Flow Steering patch [11, 12] for Linux implements flow routing in software. Instead of having the hash table reside in the NIC's memory, the table is in main memory. On each call to `sendmsg()` the kernel updates the hash table entry with the core number on which `sendmsg()` executed. The NIC is configured to distribute load equally among a set of cores (the routing cores). Each routing core does the minimum work to extract the information needed to do a lookup in the hash table to find the destination core. The routing core then appends the packet to a destination core's queue (this queue acts like a virtual DMA ring). The

destination core processes the packet as if it came directly from the NIC. Unfortunately, routing in software does not perform as well as in hardware: achieving a 40% increase in throughput requires doubling CPU utilization [11]. Our analysis of RFS shows similar results, and points to remote memory deallocation of packet buffers as part of the problem.

The Windows [5], FreeBSD [20], and Solaris [22] kernels have multi-core aware network stacks and support RSS hardware. Both FreeBSD and Solaris have had a form of RFS before Linux. These kernels have the same connection affinity problem as stock Linux: user space and interrupt processing can happen on different cores and lead to cache line sharing. These kernels would benefit from this work.

FlexSC [21] introduces an asynchronous system call interface to the Linux kernel that allows a core other than the local core to execute a system call. With the ability to route system calls to different cores the kernel can execute system calls that touch connection state on the core processing incoming packets. The drawbacks are similar to RFS: cores need to communicate with each other to exchange request and response messages.

8. Conclusion

This paper introduced Affinity-Accept, a new design for operating systems to align all phases of packet processing for an individual network connection onto the same core of a multi-core machine, and implemented this design in Linux. This approach ensures that cores will not suffer from lock contention between cores to modify connection state, as well as long delays to transfer cache lines between cores.

An evaluation of Affinity-Accept shows that on workloads that establish new connections at high rates, such as web servers, these modifications significantly improve application performance on large multi-core machines.

Acknowledgments

This research was supported by NSF awards CNS-0834415 and CNS-0915164, and by Quanta. We thank our shepherd Jeff Mogul and the anonymous reviewers for making suggestions that improved this paper.

References

[1] Chelsio Terminator 4 ASIC. White paper, Chelsio Communications, January 2010. http://chelsio.com/assetlibrary/whitepapers/Chelsio T4 Architecture White Paper.pdf.

[2] Apache HTTP Server, October 2011. http://httpd.apache.org/.

[3] Httperf, October 2011. http://www.hpl.hp.com/research/linux/httperf/.

[4] Lighttpd Server, October 2011. http://www.lighttpd.net/.

[5] Receive Side Scaling, October 2011. http://technet.microsoft.com/en-us/network/dd277646.

[6] SMP and Lighttpd, October 2011. http://redmine.lighttpd.net/wiki/1/Docs:MultiProcessor.

[7] SpecWeb2009, October 2011. http://www.spec.org/web2009/.

[8] AMD, Inc. Six-core AMD opteron processor features. http://www.amd.com/us/products/server/processors/six-core-opteron/Pages/six-core-opteron-key-architectural-features.aspx.

[9] 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 *Proc. OSDI*, 2010.

[10] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proc. SOSP*, 2009.

[11] T. Herbert. RFS: Receive Flow Steering, October 2011. http://lwn.net/Articles/381955/.

[12] T. Herbert. RPS: Receive Packet Steering, October 2011. http://lwn.net/Articles/361440/.

[13] T. Herbert. aRFS: Accelerated Receive Flow Steering, January 2012. http://lwn.net/Articles/406489/.

[14] Intel. 82599 10 GbE Controller Datasheet, October 2011. http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[15] Linux 3.2.2 Myricom driver source code, January 2012. drivers/net/ethernet/myricom/myri10ge/myri10ge.c.

[16] Linux 3.2.2 Solarflare driver source code, January 2012. drivers/net/ethernet/sfc/regs.h.

[17] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. NSDI*, 2011.

[18] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proc. OSDI*, 1994.

[19] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proc. EuroSys*, 2010.

[20] Robert Watson. Packet Steering in FreeBSD, January 2012. http://freebsd.1045724.n5.nabble.com/Packet-steering-SMP-td4250398.html.

[21] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proc. OSDI*, 2010.

[22] Sunay Tripathi. FireEngine: A new networking architecture for the Solaris operating system. White paper, Sun Microsystems, June 2004.

[23] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proc. USENIX ATC*, June 2006.

[24] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers. In *Proc. SIGMETRICS*, 1996.