A PROTOTYPE OF AN EFFICIENT MULTITHREADED PACKET FILTER

Matthew B. Smith, Brown University

1.0 Introduction

Efficient multithreaded applications are becoming increasingly necessary as single-processor machines are replaced with machines that have two or more processors. Multithreaded applications involve significant challenges that are not present when writing single-threaded applications. In this project I designed a software prototype of a multi-threaded packet-filter, and I assessed its performance.

The prototype handles both control packets and data packets. A control packet sets permissions regarding which data packets should be kept or discarded, based on the address of the sender and recipient. A data packet contains information which requires significant work to be done by the software.

To assess performance of the prototype I used eight sets of mixtures of packets, each with different characteristics. Differing characteristics included the number of addresses available, the percentage of control packets that disabled or enabled address pairs, and the number of addresses used at one time. The objective was to analyze how well the software performed and traits which allowed the software to perform better or worse.

2.0 Approach

2.1 Design of Software Prototype

The software prototype was created using Java, and packets are processed in a linearizable manner.

A single distribution thread is responsible for receiving and distributing packets to worker threads. Packets are distributed to queues which threads process in first-in-first-out order. The distribution thread sends packets to worker threads based on their address.

The software contains a shared data structure with striped locking for storing and retrieving information about which sending addresses are permitted to send packets to particular receiving addresses. When a thread accesses the shared data structure, it caches the value stored or retrieved into a hash map, so that future packets with the same sending and receiving addresses can be processed more quickly. In this way, significant lock contention can be avoided.

When a control packet changes access permissions for a particular sending and receiving address pair, some cached entries may become incorrect. To deal with this, when a thread processes a control packet, it marks the affected addresses as obsolete for each thread. When a thread has a value cached for a sending and receiving address pair, but one of the addresses has been marked as obsolete, the thread overwrites its cache with whatever data is found in the shared cache structure.

The caches are limited in size. When a cache expands beyond its allocated size, the least recently used cached address pair is removed.

2.2 Testing Approach

Eight mixtures of packets with different characteristics were tested. Initially I chose two optimization parameters of the prototype with which to experiment. I varied the number of worker threads from one to three. I did not run tests with more than three threads because the machine I was using had only four CPUs available (when there are three worker threads there is also

one distribution thread, making four threads total). I also altered the size of the cache, measured by the number of entries it is permitted to contain.

The Linux machine used gave the following information with a call to lscpu:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	1
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	AuthenticAMD
CPU family:	16
Model:	4
Stepping:	2
CPU MHz:	800.000
BogoMIPS:	6429.10
Virtualization:	AMD-V
L1d cache:	64K
L1i cache:	64K
L2 cache:	512K
L3 cache:	6144K
NUMA node0 CPU(s):	0-3

All data points tested were run 10 times for 10 seconds, and the number of packets processed were counted, so that the average number of packets per second could be computed.

As a result of my initial testing, I identified interesting trends that required additional testing to better understand the results.

3.0 Results

In Figure 1, using a constant cache size (16×1024), I altered the number of threads and plotted how this affected throughput for the eight parameter mixes. Plotting packet per second values makes it difficult to see important relationships. Instead, I plotted for each mix the ratio between the amount of work done with different numbers of threads to the amount of work done with one thread. This allows us to see how well performance scales when more threads are added.

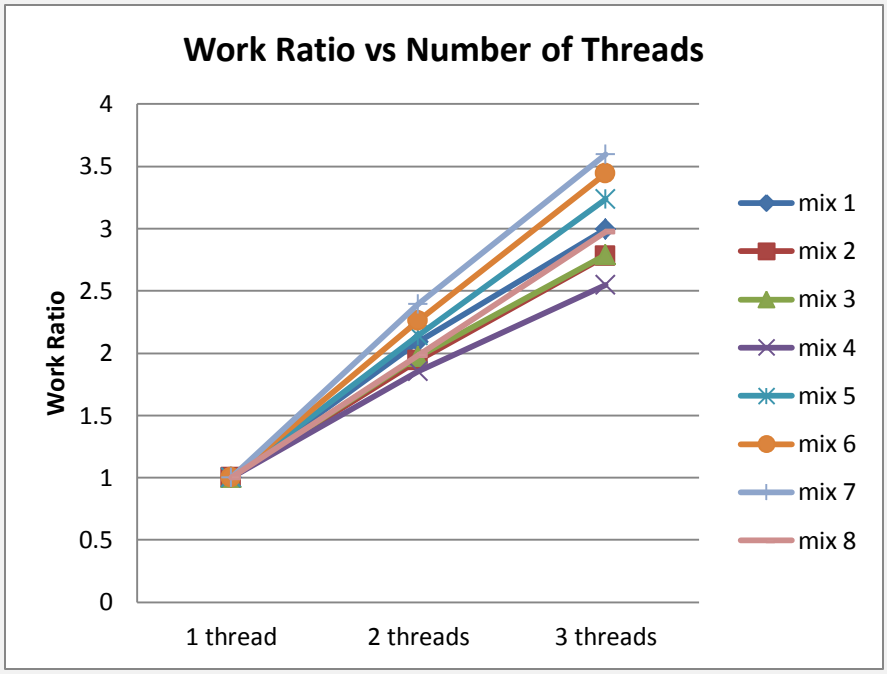

Figure 1

This figure shows how the work ratio changes as more threads are used. There is a line for each of the eight different packet mixtures. For all data points, the cache size was limited to 16×1024 entries. Each data point is averaged over 10 runs, 10 seconds each.

At first glance, the results in Figure 1 seem impossible. Three threads should not be able to do more than triple the work of one thread. Mix 7 has the highest work ratio, achieving 3.59 times more work with three threads than with one.

I considered the possibility that this result could be caused by a larger effective cache size with three threads than with one. Although each test in Figure 1 was run with a cache size of 16×1024, this is the size of the cache of each thread, rather than the cache size of the system as a whole.

This possibility can be quickly eliminated by the results in Figure 2. Figure 2 shows the packets per second over the maximum allowed size of the cache for the different mixes. In Figure 2, it is easily seen that, for the range of cache sizes tested, a smaller cache size actually results in

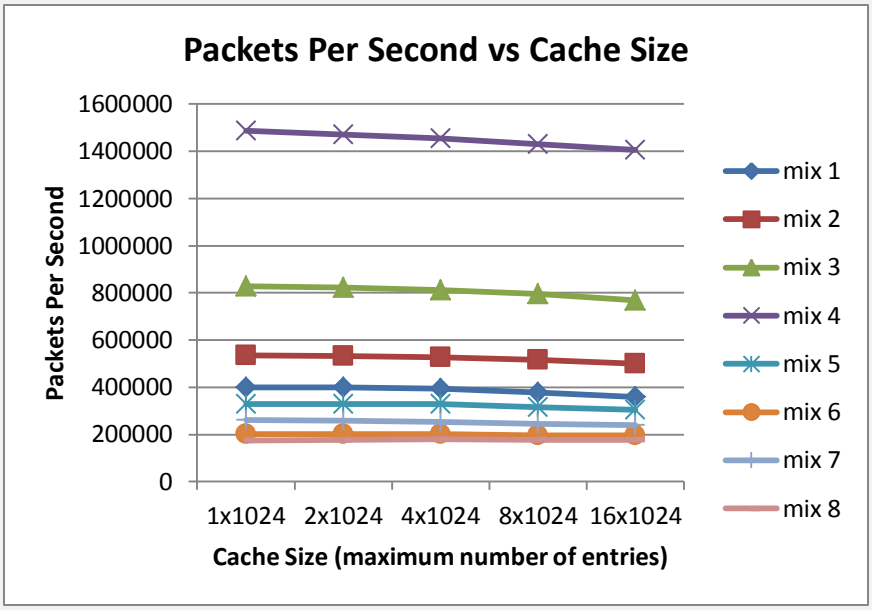

Figure 2

This figure shows how the packets per second rate achieved by the prototype changes when the maximum cache size changes. There is a line for each of the eight different packet mixtures. For all data points, 3 threads were used. Each data point is averaged over 10 runs, 10 seconds each.

better performance. This is most likely a result of limited memory available in RAM and in hardware caches.

Next I considered the possibility that the aberrant results charted in Figure 1 are caused by the manner by which the packets are distributed to worker threads. Packets are distributed to worker threads based on their address modulo the number of threads. This means that a control packet with a particular address, and a data packet with a particular destination address, will always be put into the same worker thread's queue. This may improve the effectiveness of the cache for each thread, as the threads will have a smaller number of addresses to contend with when more threads are used. To test this theory, I ran the software prototype with a modification in which the packets are randomly distributed to worker threads. The results of this test are shown in Figure 3. As in Figure 1, Figure 3 was created with a cache size of 16×1024.

It is very clear in Figure 3 that random assignment of packets to threads eliminates the unusual multithreaded performance shown in Figure 1: each thread performs less than three times better when three threads are used. Also notice that even parameter mixtures in Figure 1 which had ordinary results performed worse under the conditions used to generate Figure 3. It is easy to conclude that dividing the packets up by address when distributing them to worker threads causes the work for each packet to be significantly easier.

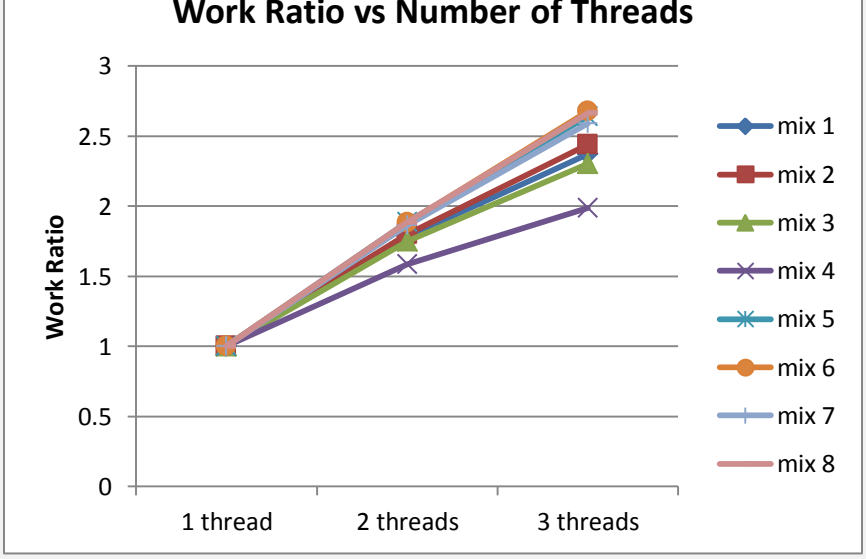

Figure 3

This figure shows how the work ratio changes as more threads are used. There is a line for each of the eight different packet mixtures. For all data points, the cache size was limited to 16×1024 entries. Each data point is averaged over 10 runs, 10 seconds each. The data for this chart was not created with the beneficial packet distribution system (which was used to create Figure 1), instead, packets were distributed randomly.

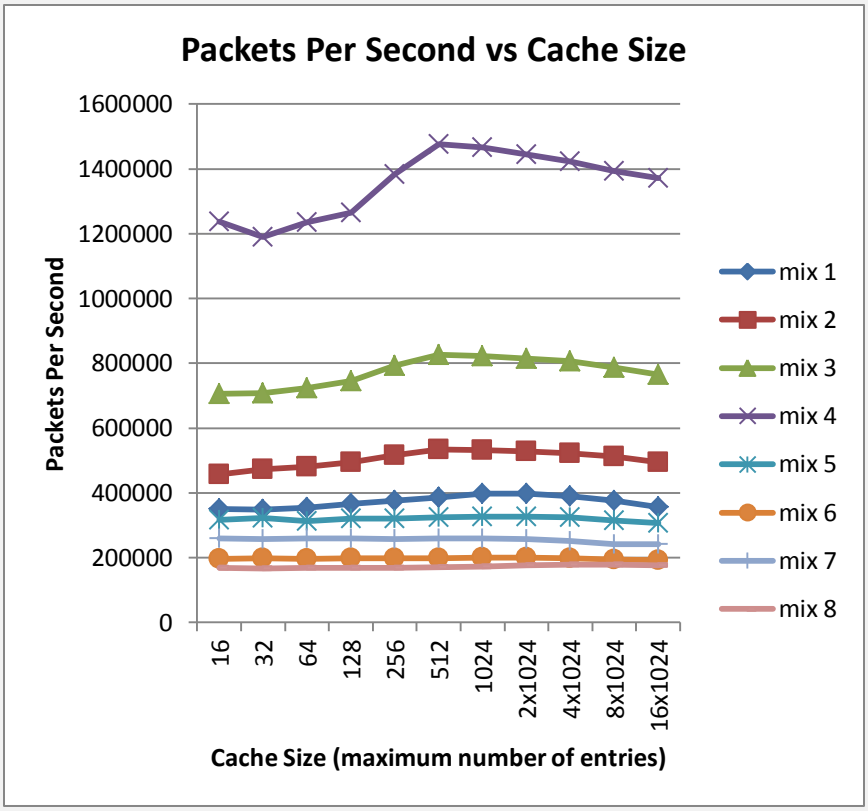

Figure 4

This figure shows how the packets per second rate achieved by the prototype changes when the maximum cache size changes. This figure is similar to figure 2, but has additional data points for smaller cache sizes. There is a line for each of the eight different packet mixtures. For all data points, 3 threads were used. Each data point is averaged over 10 runs, 10 seconds each.

After the data used in Figure 2 showed me that the cache size I had selected was too large, I decided to try to find the optimal cache size. I ran tests to estimate the number of packets per second using three threads with maximum cache sizes as low as 16. The results are shown in the plot in Figure 4. For each packet mixture, the maximum packet rate occurs between a maximum cache size of 512 and one of 2×1024, the only exception being mix 8, which achieves its maximum packet rate with a cache size of 4×1024. Regardless, each mixture is close to optimal when the cache size is limited to 1024 entries.

It is important to note that the results in Figure 4 are only relevant for the machine this was tested on. Other machines with more or less space in RAM and on hardware caches may experience very different performance.

4.0 Conclusion

Multithreaded applications often scale poorly to large numbers of processors. I have created a prototype of a multithreaded packet-filter with exceptional multithreaded performance. By distributing different packets with the same address to the same worker thread every time, my software was able to achieve more than n times the packet rate with n threads than with one, which is highly unusual.

The exceptional results make it difficult to extrapolate performance for higher numbers of threads, because it seems likely that the advantages gained by my superior packet distribution strategy would decay when large numbers of threads are used. Tests of this software with more threads would help to better understand how performance changes as the number of threads increases.