Effective Search Strategies for Application-Independent Speedup in UDP Demultiplexing

Joseph T. Dixon and Kenneth L. Calvert
{jdixon,calvert}@cc.gatech.edu

GIT-CC-97-02

Networking and Telecommunications Group
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

We present UDP datagram demultiplexing techniques that can yield potentially substantial application-independent performance gains over BSD-derived UDP implementations. Our demultiplexing strategies exploit local host and UDP implementation features -- (1) how UDP processes connection-less datagrams, (2) local host application as client or server, and (3) local host application "density" -- resulting in straight-forward hash-based search strategies that caused demultiplexing speedups as high as 24-to-1 over BSD's one-behind cache. Furthermore, while past researchers have shown that cache-based schemes yield little performance benefit for UDP, we show that cache-based implementations can actually degrade demultiplexing performance. Finally, we recommend simple, non-protocol altering local host modifications for existing and future UDP implementations. We used four server traffic traces and eight algorithms in our trace-driven simulations, and executed more than 60 simulations to obtain our results.

1. Introduction

1.1 Background

The recent Internet explosion has placed greater demand on TCP/IP's performance. As a result, several works have presented implementations that can yield significant TCP/IP performance gains over BSD-derived implementations. [Clark89, Part&Pink93, Dix&Cal96] A common strategy is to speed up specific processing steps that are potential performance bottlenecks. In this paper, we show how unicast datagram demultiplexing -- the successful delivery of a datagram to its intended communication end-point/process -- can be improved so that overall packet processing improves. In particular, we analyze UDP's[1] unicast datagram demultiplexing, which can bottleneck packet processing when a large end-point store is searched to find the destination end-point and when datagram/end-point matching rules allow exhaustive store searches. UDP performance gains are of particular interest because UDP's transport services are utilized by some of the Internet's most heavily used applications.[Com&Stev91, Wri&Stev95]

We describe UDP's unicast datagram demultiplexing using the destination host model shown in figure 1.

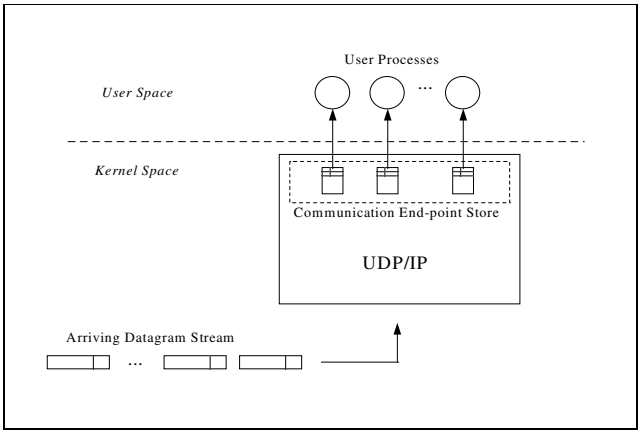

Figure 1. Destination host model for UDP datagram demultiplexing.

Our model assumes the following:

- UDP/IP performs UDP datagram demultiplexing and is implemented in the operating system kernel.
- Each active network application in user space is represented by a user process and has an associated kernel level "end-point" that UDP/IP uses to maintain the process' network related kernel resources. We make this end-point/process association one-to-one; therefore, the end-point is UDP/IP's identifier for the user process.
- Each arriving datagram contains a "tag" that UDP/IP uses to correctly identify a datagram's correct destination end-point.
- Each arriving datagram is destined for a single user process.
- UDP/IP keeps end-points in a store that must be searched in order to find a datagram's correct destination end-point.

Thus, UDP/IP receives an arriving datagram, determines its tag, searches the store for the end-point identified by the tag, and delivers the datagram payload to the end-

[1] UDP, or User Datagram Protocol, is the Internet suite's unreliable, datagram-oriented transport protocol.

point associated user process. (We assume that all other typical UDP/IP datagram processing also takes place.)

1.2 BSD's PCB and UDP Implementations

BSD-derived UDP implementations use protocol control blocks (pcbs) as communication end-points and manage the pcbs using a circular linked list store combined with a *one-behind* cached pcb pointer.[2] [Wri&Stev95] Traditionally, `in_pcblookup` is the kernel function that performs the pcb list search. This function matches the source (or *foreign*) and destination (or *local*) IP addresses and port numbers of a just-arrived packet with the foreign/local sockets of a pcb (foreign and local sockets make up the tag); the latter may include wildcard addresses, indicating that the pcb can accept packets with any value in that field. The only mandatory value specified in a wildcarded pcb is the local port number; any combination of foreign IP address, foreign port and local IP address may be unspecified. When an inbound packet is received, the `in_pcblookup` function searches through a circular linked list of pcbs (a pointer to which is passed as an argument) and identifies the pcb with the fewest wildcard matches. A NULL value is returned when no pcb matching the packet can be found.

The one-behind cached pointer always points to the pcb that was referenced by the last arriving UDP packet. When reference locality is high, i. e., when a single or small set of pcbs is preferentially referenced, one would expect the cached pointer to avoid potentially many otherwise costly linear list searches to find matching pcbs. (TCP benefits from this search strategy.) However, well documented investigations show that this cached pointer yields practically no UDP demultiplexing performance benefits.[Part&Pink93] This one-behind cache is not effective as a performance enhancer, even when pcb reference locality is substantial, primarily because fully specified UDP pcbs (i. e., ones with specific values for foreign and local address and port) seldom exist: the one-behind cache saves a linear list search only for cached fully specified pcbs. This condition results from several UDP implementation characteristics:

- A list search's goal is to produce a "best" match, which may result in an exact match with a fully specified or a partial match with a wildcard pcb, whichever pcb yields the most matching fields and least wildcarded ones. A partially specified pcb has wildcarded (i. e., don't care) local address or foreign address or port values.
- Wildcard pcbs are typical for UDP because UDP is primarily used as a connection-less transport -- pcbs that always have specific IP addresses and service ports are generally associated with connection-oriented communication. In practice, UDP pcbs are fully-specified only long enough to successfully send an outgoing datagram.[3]
- The objective of caching is to find only an exact match. Caching a wildcard pcb for the last arriving packet is useless because a better match for the current packet may exist further in the list, even when the cached wildcard pcb and current packet partially match.

1.3 Solution Requirements

We propose a pcb search strategy that significantly speeds up UDP packet demultiplexing over the implementation described in section 1.2. Our solution is a BSD Net/3 UDP re-implementation. Since our goal to make modifications that have a high benefit-to-effort ratio, our solution conforms to specific requirements:

- UDP is not altered from its specification;
- Changes are isolated to individual hosts;
- Impact on system resources is not altered significantly;
- Code changes are straight-forward and sparse.

In an earlier paper, we asserted that TCP and UDP packet demultiplexing may warrant separate solutions. [Dix&Cal96] In this paper, we investigate UDP to determine more precise features of effective pcb search strategies. While arriving UDP packets can be unicast, broadcast, or multicast, we focus on unicast packet demultiplexing because broadcast and multicast UDP packets always require an exhaustive list search: a packet's payload must be delivered to all matching pcbs.

1.4 Previous Work

Several researchers have investigated TCP demultiplexing efficiency. [Clark89, McK&Dove92, Dix&Cal96, GIT-CC-96-08] Fewer, however, have addressed UDP demultiplexing:

- Mogul, who investigated persistence and temporal locality at the process level, showed traffic traces in which half of all datagrams received are replies to the last datagram that was sent. He suggested that a *last-sent* pcb cache might result in UDP demultiplexing speedup.[Mogul92]

[2] This investigation focuses entirely on existing BSD-derived UDP implementations (such as BSD Net/3 and SunOS 4.1.X). Other UDP implementations may deviate from the packet processing behavior we describe herein (e. g., Solaris 2.X).

[3] The Sockets API `bind` function sets local IP address and port, while `connect` or `sendto` set the foreign IP address and port. In practice, UDP clients that utilize an actual connection (via `connect`) are optional; such UDP server implementations are rare. [Wri&Stev95]

- Partridge and Pink observed that the one-behind cache yielded little performance benefit for UDP. They also tested a last-received and last-sent pcb cache UDP implementation on a general purpose system and showed a cache hit rates of up to 57% and 30%, respectively.[Part&Pink93]

Our UDP research differs from past investigations in several important ways:

1. We investigate local host and UDP implementation characteristics as search algorithm design factors rather than pcb reference locality;
2. We quantify execution cost at a lower granularity -- instruction counts rather than cache hit rate.
3. We offer specific implementation recommendations.

Subsequent sections present our work in detail: experiments, simulation results, implementation recommendations, and conclusions.

2. Experiment Overview

2.1 Network Server Traces

Four network server traces drive our analysis. They were collected from 4:35PM to 6:19PM, April 7, 1995 (during peak usage) using the UNIX `tcpdump(1)` command. The servers were directly attached to the Georgia Institute of Technology College of Computing's 100Mbps FDDI backbone and provided a variety of TCP and UDP services to both on and off-campus hosts. Table 1 shows total incoming packets, number of UDP lookups performed, and percent of all UDP lookups in which a server application is bound to the destination pcb.

Server Name	Primary UDP Services	Total Incoming Packets	Total UDP Lookups	% UDP Server Lookups
cleon	xterm, other	183462	85975	85%
gaia	xterm, other	190564	106969	99%
lennon	nfs, smtp, other	174949	61437	96%
siwenna	nfs, nntp	1222643	143830	99%

Table 1. Summary of the four network server traces, number of UDP lookups, and the percentage of UDP lookups to server applications.

2.2 Demultiplexing Algorithms

We targeted local host and UDP characteristics rather than pcb reference locality to devise search algorithms that avoid unnecessary packet-to-pcb comparisons. Each characteristic we considered has search strategy implications:

1. How UDP processes connection-less datagrams - We mentioned earlier how UDP pcbs are generally wildcarded and how delivering a packet payload to such pcbs is different than fully-specified pcbs. Given these observations, we separate fully specified pcbs from wildcarded pcbs for separate processing.
2. Local host application as client or server - UDP server applications (and their pcbs) tend to remain in the pcb list longer than client applications. How long a pcb remains in the pcb list directly affects search cost because longer lists make exhaustive pcb list searches more expensive – a critical issue for UDP server applications. We partition the pcb search space into many shorter lists rather than a single long one to reduce the search cost of long-lived wildcard pcbs.
3. Local host application "density" - Several of our servers support a diverse set of UDP server applications, which implies potentially many long-lived wildcard pcbs. Intuitively, the larger the pcb search space, the greater the speedup potential when the search space is partitioned. We partition the pcb search space by service type (i. e., port number). Service-based partitions turn out to be an effective strategy for servers that support a variety of applications.

2.2.1 Linear Search - `in_pcblookup`

We measured `in_pcblookup` performance to serve as a benchmark. Recall that `in_pcblookup` performs a linear search of a circular linked list. All other algorithms implemented for this study eventually invoke `in_pcblookup`.

2.2.2 One- and Two-entry Pcb Cache

The BSD 4.3-Reno UNIX release was the first to include a single cached pointer to the last pcb referenced (i. e., *one-behind* cache). If the next packet that arrives is destined for the pcb that the cache identifies, then a list search is avoided; otherwise, `in_pcblookup` is invoked to find the best match pcb. We simulated this algorithm and measured its performance to "sanity check" our experiment methodology.

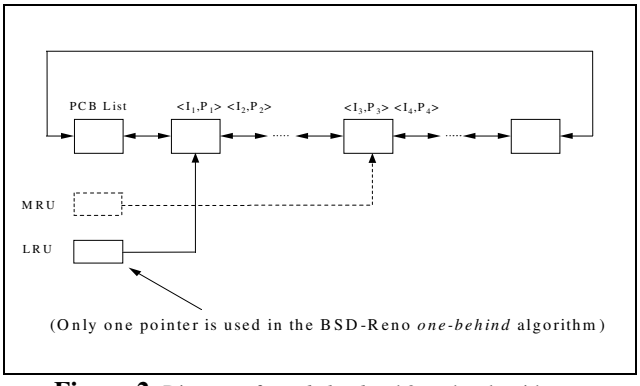

Figure 2. Diagram of *one-behind* and 2-cache algorithms.

We also devised a novel two-cache algorithm that attempts to exploit pcb reference locality. Though conceptually simple, multi-entry cache algorithms have an inherent complexity: more than one element in the cache requires policies governing access order and replacement. Our two-cache implementation enforces *most-recently-used* (MRU) access and *least-recently-used* (LRU) replacement policies. Figure 2 shows the structures used in the 2-cache algorithm after packets destined for pcbs $<I_1,P_1><I_2,P_2>$ and $<I_3,P_3><I_4,P_4>$ are demultiplexed. The 2-cache algorithm executes the following steps when a received packet is demultiplexed.

```
Check the MRU cache;
If !(MRU cache hit) {
       Check the LRU cache;
       If !(LRU cache hit)
              LRU cache =
              in_pcblookup(PCB_list, ...);
       swap(MRU cache, LRU cache);
       }
Return MRU cache as the destination PCB;
```

This algorithm accesses the MRU cache first, but replaces it last; it accesses the LRU cache last, but replaces it first. We examined cache-based demultiplexing algorithms mainly to corroborate past researchers' findings.

2.2.3 Hashing to Multiple Pcb Lists

McKenney and Dove introduced a demultiplexing algorithm that combines caching with multiple hash chains.[McK&Dove92] The algorithm maintains a linear list of pcbs for each of several hash chains. Each hash chain has an associated cache that points to the last pcb found on that chain. When a packet arrives, it is routed to a hash chain via a hash function. The packet is then assigned to its pcb via a cached pcb comparison or linear hash chain search if the cache comparison fails. Figure 3 shows the structures used in this algorithm after packets destined for pcbs tagged with $<I_1,P_1><I_2,P_2>$ and $<I_3,P_3><I_4,P_4>$ are demultiplexed.

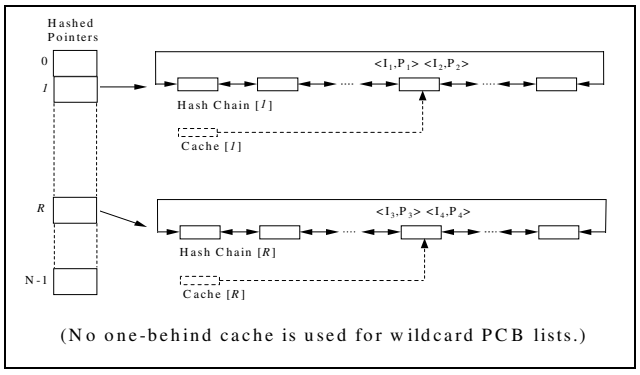

Figure 3. Diagram of an *N* hash chain algorithm with 1 cache per chain.

The 1-cache/multiple hash chain algorithm we used executes the following steps when it demultiplexes a packet identified by $<I_1,P_1><I_2,P_2>$.

```
R = hash(<I1,P1><I2,P2>), R ∈ [0,(N-1)];
Check cache[R];
If !(cache hit)
       cache[R] =
       in_pcblookup(hash_chain[R], ...);
Return cache[R] as the destination PCB;
```

In our TCP work, we showed that this algorithm was extremely effective on pcb lists that had a large fraction of fully-specified pcbs. For UDP, however, we know the one-behind cache is ineffective because of UDP's large fraction of wildcard pcbs; thus, we used a multiple hash chain without the single entry cache per chain on wildcard pcbs in our final UDP algorithm.

2.2.4 Hashing to Separate Wildcard and Fully-specified Pcb Lists

We designed a two-stage multiple hash chain algorithm to capitalize on pcb reference locality for local hosts with two features: (i) the host's supported applications consist of a large fraction bound to fully specified pcbs and (ii) a sufficient fraction of its arriving packets is destined for those pcbs. We have not been able to identify such applications but, since UDP client applications may be implemented optionally with `sendto` or `connect`, we felt compelled to address this case. (We show later that this algorithm can perform well, even in the absence of fully-specified pcbs.)

The two-stage UDP demultiplexing algorithm has several notable features:

1. Wildcard pcbs and fully-specified pcbs are managed in separate stores, since searching for each is inherently different.
2. The fully-specified pcb store is searched before the wildcard pcb store so an exact match (if one exists) can be found early in the search.
3. Fully connected pcb searches (possibly for UDP client applications) use the McKinney and Dove type multiple hash chain algorithm we described earlier.
4. Wildcard pcb searches (primarily from connection-less server applications) use a multiple hash chain algorithm without the one-behind cache per chain, since we know the cache is not effective on them.
5. We chose a simple, "cheap" hash function that partitioned each search space so that no single hash chain per store was inordinately longer than another.

The two-stage multiple hash chain algorithm executes the following steps:

```
Let the arriving packet be identified by
the foreign and local socket pair
(<I1,P1><I2,P2>).

R = hash(<I1,P1><I2,P2>), R ∈ [0,(N-1)];
Check f_cache[R];
If !(cache hit){
        inp = in_pcblookup(f_chain[R], ...);
        If (inp == NULL)
                inp = in_pcblookup(w_chain[R],
        ...);}
else
        f_cache[R] = inp;
Return inp as the destination PCB;
```

Our multiple hash chain algorithm is consistent with our TCP recommendations[GIT-CC-96-08];

- We only use 64 hash chains for each store. (We showed that additional performance speedup may not justify system resource costs when more than 64 hash chains are used.)
- We use a straight-forward and efficient hash chain function. `Hash` is merely the destination/local host's *service port modulo-64*. It requires only four to six assembly language instructions (depending on code optimization) and, as will be shown later, yields impressive results.

Note that pcbs must be hashed to their respective chains as they are bound. Note also that `w_hash` segregates wildcard pcbs by port number; this is intuitively appealing, since it reduces the chances of any single hash chain on a UDP hosts that serves many applications of becoming inordinately long.

2.3 Metrics

We measure performance in terms of assembly language instructions required per lookup. We used the following procedure to determine the number of instructions executed by each algorithm:

1. Implement a C-language version of the algorithm.
2. Generate an optimized assembly language version of the C program using the *gcc* compiler.
3. Determine the correct mapping between the assembly language instructions and the C program instructions. (This step determines the exact cost of each logical processing path.)
4. Imbed the code to compute instruction counts for each lookup (from step 3) in the C program.

We also report UDP demultiplexing speedup for each algorithm relative to the original `in_pcblookup` using the simple ratio [Hen&Pat96]:

$$Speedup = \frac{Avg(InstructionCost_{LinearSearch})}{Avg(InstructionCost_{NewAlgorithm})}$$

2.4 Assumptions

We made several assumptions so our simulation results would be as realistic as possible. (Some were also necessary to overcome packet traces limitations.)

- All arriving packets will be delivered to a valid pcb.
- The local host is a server for application *L*, where *L* is a well-known port, if *L* is specified as the local port in an arriving trace packet.
- The local host is a client for application *F*, a well-known port, if *F* is specified as the foreign port in an arriving trace packet and *L*, not well-known, is specified as the local port.
- The local host's function, as client or server for an application, is undetermined for arriving packets that have non-well-known local and foreign ports.
- Each local host application binds to an existing wildcard pcb when it sends a datagram.
- Undetermined applications are assumed to be clients and are bound to fully-specified pcbs. (Still, less than one percent of all arriving packets in our traces are delivered to fully specified pcbs.)
- All server application processes are invoked at system startup. (i. e., the server's wildcard pcb is in UDP's pcb store at simulation startup and exists for the entire trace.)
- Once a single datagram is delivered to a client application, its pcb is removed from the pcb list.
- Client applications bound to fully-specified pcbs exist from first to last trace packet destined to that pcb.

These assumptions follow directly from various well-documented discussions concerning BSD-derived TCP/IP implementations. [Wri&Stev95,Stevens90,Com&Stev91]

3. Results

3.1 Caching can add cost to demultiplexing.

We expected the cached pcb pointer to be ineffective because the vast majority of arriving UDP packets are delivered to wildcard pcbs. Our simulations show, however, that caching can actually add cost to demultiplexing because we consider assembly language instructions executed rather than cache hit/miss rates. While both the BSD 4.3-Reno one-behind cache and our MRU/LRU 2-cache algorithm sped up TCP demultiplexing, both performed worse than `in_pcblookup`'s simple linear search for UDP. Table 2 shows the mean number of instructions executed per UDP pcb lookup for the linear search, BSD 4.3-Reno

algorithm, and the 2-cache algorithms. Table 2 also shows the one-behind and 2-cache algorithms' speedup over a linear search.

Server	linear search	1-behind	1-behind Speedup	2-cache	2-cache Speedup
cleon	886.1	905.7	0.98	913.3	0.97
gaia	839.2	858.2	0.98	865.2	0.97
lennon	1254.2	1273.3	0.98	1280.4	0.98
siwenna	91.4	110.5	0.83	117.5	0.78

Table 2. Mean instructions executed per UDP pcb lookup and speedup for in_pcblookup, BSD 4.3-Reno, and 2-cache algorithm.

3.2 A multiple hash chain algorithm provides substantial speedup.

We expected a multiple hash chain algorithm to speed up UDP demultiplexing primarily because (1) the multiple hash chain algorithm breaks a single pcb list into multiple smaller lists (as long as the algorithm's hash function reasonably distributes pcbs) and (2) UDP demultiplexing usually exhaustively searches its pcb list. We simulated a modified version of the multiple hash chain algorithm we described in a previous section - we removed the one-behind cache for each hash chain - and realized substantial UDP demultiplexing speedup. Table 3 shows various the algorithm's instruction execution cost.

Server	Non-caching Multi-Hash Chain (Avg.)	90-percentile	Speedup over linear search
cleon	45.0	46	19.6
gaia	45.1	46	18.6
lennon	50.8	52	24.7
siwenna	33.4	34	2.7

Table 3. Mean instructions per UDP pcb lookup, 90-percentile, and speedup for the modified multiple hash chain algorithm.

The multiple hash chain algorithm, in its original form (i. e., with a one-behind cache per hash chain), performed substantially better than in_pcblookup, but not as well as the non-caching hash chain algorithm.

3.3 If the search space is partitioned by *service port*, then pcbs are effectively distributed across multiple hash chains.

The speedups in Table 3 result mainly from our simple, low-cost hash function, destination service port-modulo-64, which effectively distributes pcbs across the algorithm's 64 hash chains. Hashing on local service port is attractive because well-known services (assigned local port numbers up to 1024) will be evenly distributed when bound to wildcard pcbs with only local port number specified. Clearly, TCP/IP hosts that provide UDP-based services will benefit from this scheme; other TCP/IP hosts can also benefit, depending on their network applications' bindings. Table 4 summarizes single list versus multiple hash chain lengths at simulation start up.

Server	Single List length	Min. chain length	Max. chain length
cleon	327	1	8
gaia	266	2	7
lennon	414	4	8
siwenna	17	1	2

Table 4. List vs. chain lengths (using 64 hash chains) at simulation start up.

3.4 As demultiplexing gets faster, in_pcblookup overhead becomes significant.

Each in_pcblookup call requires 12-14 assembly language instructions for parameter setup and a sub-routine jump (depending on optimization). When execution costs are as low as those shown in Table 3, the function call becomes substantial overhead. For example, on our *lennon* server, when a simple linear list search is the chosen demultiplexing algorithm, the in_pcblookup call overhead is less than 1% of total demultiplexing cost; call overhead accounts for as much as 27% of total cost when the multiple hash chain algorithm is used instead.

Greater efficiency can be obtained via in-line iteration through the pcb list(s).[4] Thus, all averages we report may be further reduced by 12-14 assembly language instructions for each in_pcblookup invocation if in-line list/chain iteration is implemented.

3.5 The two-stage algorithm can speed up demultiplexing, but results are not consistent.

The two-stage algorithm separates fully specified pcbs from wildcard pcbs so separate lookup strategies can be applied. Recall that this algorithm had the following features:

1. The wildcard pcb store uses a non-caching multiple hash chain algorithm.
2. The fully-specified pcb store uses a 1-cache multiple hash chain algorithm.
3. The fully-specified pcb store is searched before the wildcard pcb store so an exact match (if one exists) can be found early in the search.
4. We only use 64 hash chains for each store and use a *local service port modulo-64* hash function.

Table 5 shows that the two-stage algorithm can also speed up UDP demultiplexing substantially, though not without exception and not as much as the simple non-caching hash

[4] The precedent is already set: Net/3 and SunOS 4.X demultiplex arriving multicast and broadcast packets using an in-line while-loop rather than a function call.

chain solution discussed in the previous section. The primary additional cost this two-stage algorithm incurs is (1) cache management cost, (2) a NULL pointer test for entering the second stage, and (3) an additional `in_pcblookup` invocation. Thus, little can be done about (1) and (2), but in-line list iteration of the two-stage algorithm can reduce the instruction cost averages shown in Table 6 by as much as 28 instructions.

Figures 4 through 7 compare (for each server trace) the two-stage algorithm's cumulative distribution curve with those of `in_pcblookup`, one-behind cache, and the non-caching multiple hash chain algorithm. The hash chain algorithms clearly out-perform typical UDP implementations with one exception. In Table 5 and Figure 7, our *siwenna* server results show that the search space partitioning benefit is overwhelmed by the two-stage algorithm's overhead liability. This results when a host, *siwenna* in this case, has few pcbs, wildcard or otherwise. Thus, one or more pcb hash chains (especially one that is preferentially referenced) and the original single list may be close in length. In such cases, hash function execution, address indirection, and `in_pcblookup` function invocation liabilities may exceed shorter pcb list benefits. (Note, however, that a two-stage algorithm using in-line hash chain iteration may reverse this result.)

Server	2-stage Multi-Hash (Avg.)	90-percentile	Speedup over linear search
cleon	114.7	122	7.7
gaia	117.9	122	7.1
lennon	123.4	125	10.2
siwenna	106.7	107	0.86

Table 5. Mean instructions per UDP pcb lookup, 90-percentile, and speedup for the modified multiple hash chain algorithm.

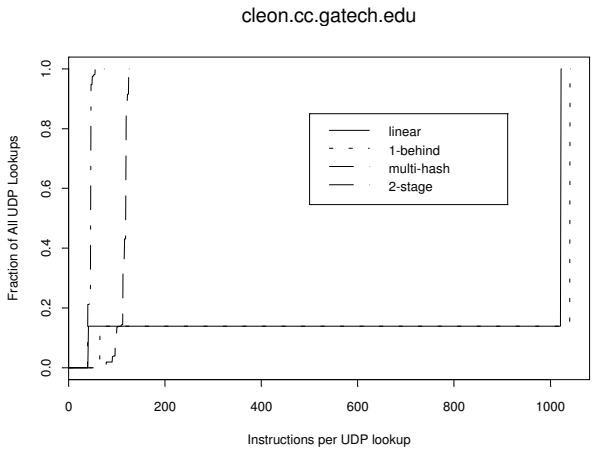

Figure 4 Cumulative distribution of various algorithms' for *cleon* (xterm & other services).

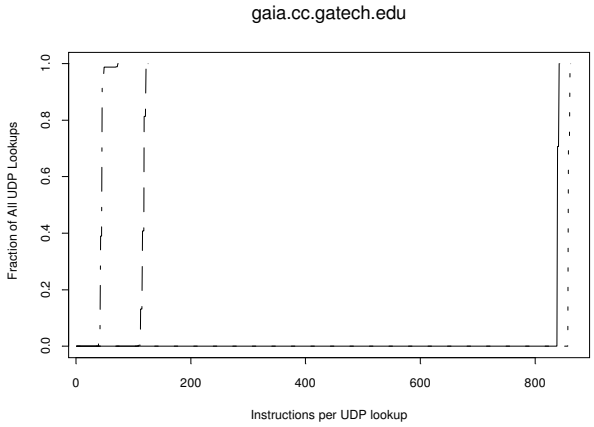

Figure 5 Cumulative distribution of various algorithms' for *gaia* (xterm & other services).

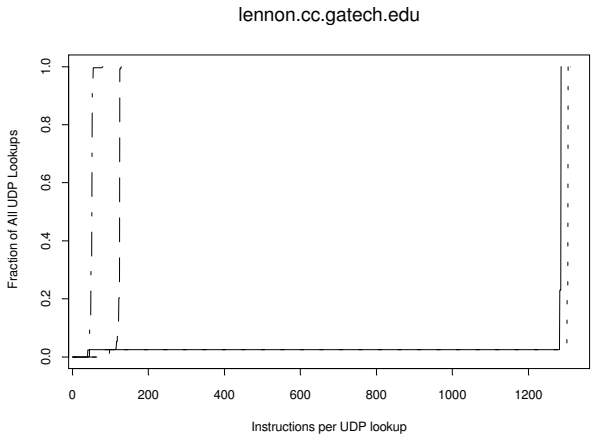

Figure 6 Cumulative distribution of various algorithms' for *lennon* (nfs, smtp, & other services).

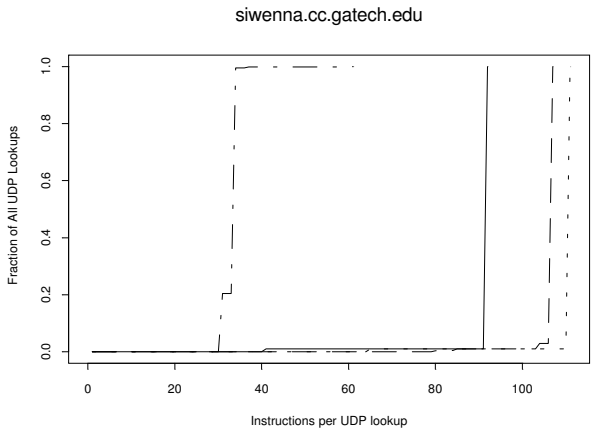

Figure 7 Cumulative distribution of various algorithms' for *siwenna* (nfs & nntp services).

4. How does demultiplexing speedup impact overall in-bound UDP packet processing?

We attempted to speed up UDP demultiplexing because we ultimately wanted to make overall datagram processing more efficient than current BSD derived implementations. We define overall datagram processing as the kernel actions necessary to deliver a UDP datagram's payload from kernel buffers (copied there by the network interface card) to the correct destination process' buffer. Figure 8 shows the primary Net/3 function calls that make up total packet processing, given an Ethernet NIC. [Wri&Stev95] Many function calls are not shown; they do not represent major processing steps.

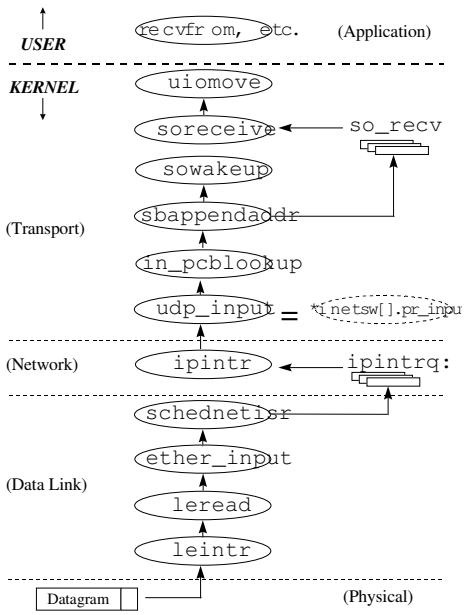

Figure 8. Net/3 functions executed to deliver an arriving UDP datagram to its destination pcb.

A natural "next step" for our work (a step we have not completed) is to apply our experimental process to Figure 8's routines to obtain precise results. In the interim, we can make a few simple assumptions and use past findings to extrapolate overall in-bound packet processing improvements. While informal, this approach puts demultiplexing speedup in the overall packet processing context.

Partridge and Pink showed that the checksum calculation (`in_cksum` in Net/3) accounted for 8.4% of total IP and UDP packet processing time for a 512-byte UDP datagram. [Part&Pink93] This includes four `in_cksum` invocations (for the IP header and the entire datagram on *send* and *receive*.) Our experiments show that, when optimally compiled on a SPARC architecture, the four `in_cksum` invocations require 1680 assembly language instructions to process a 512-byte UDP datagram.[5] Therefore, if we assume:

- the fraction of process time is equivalent to fraction of instructions executed;
- a UDP datagram send and receive require an equal number of instructions;

then total packet processing requires approximately 10,000 instructions for an arriving 512-byte UDP datagram. Thus, we obtain the overall packet processing speedups shown in Table 7.

Server	No-cache savings over Net/3 (Avg Instr)	No cache multi-hash speedup	2-stage savings over Net/3 (Avg Instr)	2-stage multi-hash speedup
cleon	861	1.094	792	1.086
gaia	813	1.088	740	1.079
lennon	1222	1.139	1150	1.129
siwenna	77	1.007	4	1.000

Table 6. Overall packet processing speedup when using the multiple hash chain algorithms versus Net/3's one-behind cache.

Though informal, the Table 7 results suggest that in-bound packet processing processing cost would be reduced by more than 12% for 512-byte UDP datagrams arriving at our *lennon* server (if packets were demultiplexed using the non-caching multi-hash algorithm rather than the Net/3 one-behind cache.) This performance gain is particularly important, since lennon is primarily used as a server. (More than 96% of its arriving UDP datagrams are destined for server applications.)

5. Conclusions

Recent proposals (including our own past work) have focused almost exclusively on TCP demultiplexing speed up. Such approaches include passing 32-bit pcb identification parameters as a TCP connect-time option [Huitema95]; *source hashing*, which allows direct access to various information associated with general packet processing [Chan&Varg95]; Mentat, Inc.'s *streams*-implemented TCP/IP, which demultiplexes incoming packets in IP rather than TCP or UDP [Mentat93]; and our own multiple hash chain recommendations [Dix&Cal96]. We know of little else done recently to address UDP demultiplexing efficiency since Partridge and Pink's effort. [Part&Pink93]

In this work, we turned our attention to UDP demultiplexing. We corroborated past findings when we

[5] A single 20-byte IP header checksum calculation required 87 instructions while the 512-byte UDP datagram checksum required 753 instructions. We made two simplifying assumptions: (1) No datagram word spans two *mbufs*; (2) the buffered datagram always begins on a word boundary.

showed that conventional UDP demultiplexing is notoriously inefficient, even with the one-behind cache strategy. We also showed, however, that simple UDP and host features could be used to design algorithms that result in substantial application-independent speedup. As a result, we offer protocol implementers several recommendations concerning UDP:

- Use a non-caching multiple hash chain solution if the local host has few arriving packets destined for client applications.
- If the fraction of UDP clients implemented with `connect` (and their traffic) is high enough, then a two-stage multiple hash chain solution may be most suitable.
- When devising a multiple hash chain solution, use a simple, "cheap" hash function and a relatively small number of hash chains (e. g., 64).
- If optimum UDP performance is desired, use in-line pcb list/chain iteration to save function call overhead normally incurred when `in_pcblookup` is invoked.

The long-term solution for demultiplexing is not clear. Meanwhile, our conclusions and recommendations provide simple, server-independent solutions that yield performance gains at such high levels that, for some servers at least, additional enhancements may be unnecessary.

Acknowledgment - The authors thank Dan Forsyth for assisting with collecting the packet traces.

References

[Chan&Varg95] Girish P. Chandranmenon and George Varghese, Trading packet Headers for Packet Processing, SIGCOMM '95, 1995.

[Clark89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June, 1989.

[Com&Stev91] Douglas E. Comer and David L. Stevens, Internetworking with TCP/IP Volume II: Design, Implementation, and Internals, Prentice Hall, Inc., 1991.

[Dix&Cal96] Joseph T. Dixon and Kenneth L. Calvert, Increasing Demultiplexing Efficiency in TCP/IP Network Servers, International Conference on Computer Communications and Networks, October, 1996.

[GIT-CC-96-08] Joseph T. Dixon and Kenneth L. Calvert, Increasing Demultiplexing Efficiency in TCP/IP Network Servers, Technical Report #: GIT-CC-96-08, Georgia Institute of Technology, 1996.

[Huitema95] Christian Huitema, Multi-homed TCP - IETF Draft, Network Working Group, May, 1995. *This is a work in progress.*

[Hen&Pat96] Hohn L. Hennessy and David A. Patterson, Computer Architecture - A Quantitative Approach 2nd Edition, Morgan Kaufman Publishers, 1996.

[McK&Dove92] Paul E. McKenney and Ken F. Dove, Efficient Demultiplexing of Incoming TCP Packets, ACM SIGCOMM '92, August, 1992.

[Mentat93] Mentat TCP/IP Design Overview (extracted from Mentat TCP/IP Internals Manual), Mentat, Inc., Los Angeles, CA., July, 1993.

[Mogul92] Jeffrey C. Mogul, Network Locality at the Scale of Processes, ACM Transactions on Computer Systems, May, 1992.

[Part&Pink93] Craig Partridge and Stephen Pink, A Faster UDP, IEEE/ACM Transactions on Networking, August, 1993.

[Stevens90] W. Richard Stevens, UNIX Network Programming, Prentice Hall, Inc., 1990.

[Wri&Stev95] Gary R. Wright and W. Richard Stevens, TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley Publishing Company, 1995.