Trading Packet Headers for Packet Processing

Girish P. Chandranmenon and George Varghese

***Abstract*— In high speed networks, packet processing is relatively expensive while bandwidth is cheap. Thus it pays to add information to packet headers to make packet processing easier. While this is an old idea, we describe several specific new mechanisms based on this principle. We describe a new technique, *source hashing*, which can provide $O(1)$ lookup costs at the Data Link, Routing, and Transport layers. Source hashing is especially powerful when combined with the old idea of a *flow ID*; the flow identifier allows packet processing information to be cached, and source hashing allows efficient cache lookups. Unlike Virtual Circuit Identifiers (VCIs), source hashing does not require a round trip delay for set up. In an experiment with the BSD Packet Filter implementation, we found that adding a flow ID and a source hash improved packet processing costs by a factor of 7. We also found a 45% improvement when we conducted a similar experiment with IP packet forwarding. We also describe two other new techniques: *threaded indices*, which allows fast VCI-like lookups for datagram protocols like IP; and a *Data Manipulation Layer*, which compiles out all the information needed for Integrated Layer Processing and scheduling into an easily accessible portion of each packet.**

1 Introduction

Networks are getting faster because of advances in transmission and switching. Processor and memory speeds are not increasing as fast as bandwidth increases [Par93]. Thus it makes sense to consider adding fields to packet headers to speed up packet processing. We propose three specific *new* mechanisms (source hashing, threaded indices, and Data Manipulation headers) based on this principle. Since protocol standards are currently in transition at all layers (e.g., IP version 6 [Hin94], multipoint transport protocols), we believe this is a good time to reexamine the use of additional fields for performance. Also, if some implementations disregard these extra header fields, this will only affect performance and not correctness.

Our first technique, *source hashing*, is a technique for adding an index field to protocol identifiers at all layers to reduce lookup costs and protocol processing. Source hash indices are similar to Virtual Circuit Identifiers (VCIs) in that they reduce lookup times from $O(\log(n))$ to $O(1)$, where n is the number of flows. However, they differ in three significant ways. First, and most importantly, *source hashing does not require a round trip to set up*, unlike VCIs. This latency reduction is especially significant for flows that have a relatively small number of packets to send. Second, a source hash is a consistent random label placed in the packet by the source; by contrast, a VCI is a deterministic label that is unique on every hop. Thirdly, source hashing applies to all layers (we provide example applications at the Data Link, Network, and Transport Layers), while VCIs are typically used at the network layer.

Source hashing is especially powerful when combined with the old idea of a Flow Identifier (flow ID). A Flow Identifier is a unique label that identifies a stream of packets that require identical processing. Processing costs can be reduced significantly by caching processing information on a per flow ID basis. Given such a processing cache, lookup costs become significant. Source hashing allows small cache lookup costs that do not degrade as the cache size grows and provides predictable performance independent of the way addresses or flow IDs are assigned. (To contrast with other schemes, in a simple linear list, cache lookup costs increase as the cache size increases; and the performance of the simpler hashing schemes depend strongly on the way addresses are assigned.) Source hashing also does not require special purpose hardware (e.g., Content Addressable Memories) and thus can easily be implemented in either hardware or software.

While the idea of a flow ID is an old one, it has been typically confined to the routing layer (e.g., VCIs in ATM, and flow IDs in IP version 6). We believe that the notion of a flow ID is very general and can be applied in some unusual contexts. For example, we show that the use of a flow ID and source hash can greatly speed up packet filter processing.[1] Our experiments with the BSD packet filter (Section 3.4) show a factor of 7 improvement in performance.

Source Hashing is essentially a technique to provide nearly perfect random hash indices without the run-time overhead of computing a good hash function. Although it provides expected $O(1)$ lookup times for all topologies and configurations, its guarantees are probabilistic. Threaded Indexing is a way to reduce the cost of destination address lookups in datagram networks to $O(1)$ in the worst case. Threaded indices at first appears to be similar to the notion of setting up Permanent Virtual Circuits (PVCs). However, there is a major difference. Threaded indices provide a per hop index for each *destination* in a datagram network; VCIs, however, provide a per hop index for each active *source-destination pair*. Thus using a PVC like solution in

Both authors are with the Department of Computer Science, Washington University in St. Louis.

This project was supported by a grant (NCR-940997) from the National Science Foundation.

Earlier versions of this paper appeared in [Var94] and [CV95].

[1] Packet filters are used to demultiplex incoming packets to destination processes.

a datagram network would be prohibitively expensive.

Our third idea is a proposal to add a *data manipulation* header to an easily accessible portion of each data packet. The header provides information required for data manipulation (e.g., destination buffer names, encryption keys) and dispatch (e.g., destination process IDs). This allows low level software or hardware to avoid scheduling overhead [vECGS92] and to do *integrated layer processing* [CT90], in which a large number of data manipulation operations are done in a single pass, without sequential parsing of intermediate layer headers. Essentially, the header contains information, compiled from several layer headers, that is crucial for data manipulation and dispatch. While all the information that we propose to add to the Data Manipulation header has been proposed before, we suggest integrating this information and placing the information (possibly redundantly) in a separate layer header.

The rest of this paper is organized as follows. In Section 2 we compare our ideas with previous work. In Section 3 we describe Source Hashing in detail. We provide a performance model in Section 3.2 that allows a simple comparison of source hashing with datagram and VCI lookup techniques. We also provide experimental results in Section 3.4. We report the quantitative improvements obtained by adding a source hash and a flow ID in two different applications: IP packet forwarding and BSD Packet Filter processing. Section 4 describes the Threaded Indexing technique and Section 4.1 evaluates it both qualitatively and quantitatively. Section 5 describes the proposed Data Manipulation Layer. We state our conclusions in Section 6.

2 Related Work

In Table 1 we compare existing lookup schemes with the source hashing and threaded index schemes that we propose in this paper. Our comparison is based on three metrics: lookup costs, delay to set up the indices, and memory requirements. We also compare the schemes qualitatively based on when the indices are set up. Keep in mind that our comparisons are not just for network layer applications (where there are multiple hops) but also for data link, transport, and application layer protocols (where there is only one hop).

Ordinary VC lookup takes $O(1)$ time but adds a round trip worth of latency for setup. By fast VC set up, we mean the notion that the initial round trip's worth of data is sent without a VCI; fast VC set up avoids the set up delay but increases the lookup times for the packets sent without a VCI. VCI lookup can sometimes be augmented by returning the VCIs on every hop before the entire virtual circuit is set up; this reduces the setup delay to a round trip time on a single hop. However, in data link or transport applications, where there is only a single hop, this scheme has no advantage over ordinary VCs. Permanent Virtual Circuits (PVCs) is a trivial method of reducing lookup costs. However, setting up PVCs between all pairs of possible source-destination pairs would be completely impractical for most real networks.

A much more interesting idea mentioned in [Par93] is the idea that on every hop, the source assigns the VCI for that hop. We call this source assigned VCIs. This is especially attractive in virtual circuit networks where switches are connected by point-to-point links; since the number of potential sources on each hop is only one, it does not cost any extra memory to have each source assign the VCI tables at the other end of the hop. However, consider a single hop transport protocol (e.g., an RPC transport) in which each destination has thousands of potential sources. If each source can have multiple flows as well, then it would be extremely expensive to have the destination reserve table space for each potential source so that the sources can allocate indices within the destination tables. This scheme is also infeasible if each hop is a multi-access link like an Ethernet or FDDI in which there are many possible sources on every hop.

On the other hand, source hashing provides $O(1)$ expected lookup time with no set up delay. We use "expected time" in a rigorous sense; the source hashing lookup guarantees are independent of assumptions about the probability distributions of addresses or the details of the particular topology used. This is in stark contrast to many standard hashing schemes like XOR folding [Jai92] that are strongly dependent on the way addresses are assigned in particular networks.[3] The memory requirements for source hashing are a constant factor (a factor of two should be sufficient for most purposes) larger than the memory required for ordinary VCs; this extra factor is used to make hash collisions a rare occurrence, and should not be a problem for most applications.

The IP version 6 proposal [Hin94] uses both the notion of a flow ID and (implicitly) the notion of a source hash. In IPv6, it is suggested that some of the bits in the flow ID be assigned randomly; this makes it more probable that flow IDs are unique after crashes, and the random bits can also be used as a hash index. Our work was independent of this proposal. More importantly, we have generalized the use of a flow ID and source hash as a useful tool at many layers (including transport and data link). We also have experimental results and performance models. Finally, the distinction between flow ID and source hash allows us to explore different ways of assigning source hash values.

Threaded indices provide $O(1)$ worst case lookup times. They are useful only in datagram networks. It requires memory proportional to the number of possible destinations. This is not a problem for most datagram routers since they have to keep state for every possible destination anyway. Also, the memory required for threaded indexing is much more modest than the memory required for PVCs! Note that threaded indices also differ from ordinary VCIs in that threaded indices are computed whenever the topology changes and not when data packets need to be sent

[2]RTT is the end to end round trip time

[3]It is easy to assign addresses or flow IDs so that XOR folding will lead to $O(n)$ expected lookup time; this is impossible to do for source hashing.

	Lookup Cost	Setup Delay	Memory	When Setup
Ordinary VC.	$O(1)$	1 RTT[2]	$O(f), f = \#\ of\ flows$	Data transfer time
Fast VC setup	Slow for packets in first RTT; $O(1)$ otherwise	None	$O(f)$	Data transfer time
Returning VC index per hop	$O(1)$	2 hop delays	$O(f)$	Data transfer time
Source assigned VCIs	$O(1)$	None	(number of sources) $*$(number of flows per source)	Data transfer time
Permanent VCs	$O(1)$	None	# of src-dst pairs	When topology changes
Source Hashing	$O(1)$ expected	None	$O(f) * constant$	Data transfer time
Threaded Indexing	$O(1)$	None	# of destinations;	When topology changes

Table 1: Comparison of various lookup schemes.

between a source and a destination. We give a comparison between threaded indices and a scheme called Pip [Fra93] later.

Data manipulations like copying, checksumming and encryption are expensive because they involve operating on all the bytes in a packet [CT90]. To avoid using the system bus multiple times, [CT90] suggests the use of *Integrated Layer Processing (ILP)*, in which many data manipulations are done in a single pass. Since the information required to guide these operations is often in different layers, it requires sequential parsing through intervening layer headers before this information can be found. Clearly a hand-crafted implementation (that combines the processing of all layers) has access to this information. However, it seems desirable to provide more structured access to this information.

Several other authors have suggested adding information to packet headers to make data manipulation (especially copying) and dispatch (finding and waking up the destination process) less expensive. For example, the VAX Clusters [KLS86] and AXON [SP90] protocols add destination memory buffer names to packets. Both the recent Active Messages [vECGS92] and Birrell and Nelson's original RPC paper [BN84] suggest adding the address of the destination process to each data packet so that the packet can quickly be dispatched. Abbot and Peterson [AP93] suggest keeping a pointer to the application data at the start of a packet; this allows low level software or hardware to separate out user data from protocol headers without parsing intervening protocol headers. Finally, [Fel93] proposes making packets (and data chunks within packets) self describing in order to handle out of order delivery of packets.

In Table 2 we compare these earlier ideas with our idea of a Data Manipulation Layer (DML) header. First, note that each of these earlier ideas provides some of the information required to make data manipulation and dispatch easier; second, note that the information in each case is added in an *ad hoc* basis to the concerned packet layer. We suggest integrating all the needed information and collecting it in a separate layer header that is between the Data Link and Routing headers. This allows low level software or hardware to access all the required information without parsing multiple protocol headers. We also show later that the information needed for this new header can be passed down to this layer from other layers in a structured fashion. In essence, the DML header is a compilation of information needed for data manipulation and dispatch. For example, hand-crafted versions of ILP [CT90] are difficult to change when individual layer implementations change; we believe the DML layer will make such changes much easier.

3 Source Hashing

As link speeds move to Gigabit speeds and higher, a number of optimizations will be necessary over and beyond the initial optimizations caused by restructuring poor implementations [Cla85]. Kay and Pasquale [KP93] point out that typical packet sizes in the Internet are small. Their study asserts that non-data touching overhead (i.e., functions required to process packet headers such as setting timers, doing lookups etc.) is significant, and a collection of optimizations for the various protocol functions are needed to improve performance for small packets.

One way to improve packet processing performance is to take advantage of the temporal locality of network packets — i.e., cache the results of prior packet processing. However, once we cache processing information, we need a mechanism for efficient cache lookup. Cache lookup can dominate the header processing time.

In order for caching to be useful for packet processing we need the concept of a Flow Identifier (flow ID). A flow ID is a unique label that identifies a stream of packets that require identical processing. For example, the IP version 6 proposal includes a flow ID in its routing header; the flow ID uniquely identifies packets between the same source and destination that require similar processing. Given a flow ID in the layer header, the processing device (a router in the case of IP) can cache the processing requirements for each flow ID. Then in most cases, processing reduces to looking up the cache using the flow ID as a key. If actual processing requirements are small or can be relegated to hardware,[4] the cache lookup costs will quickly become significant.

If state lookup is only a small fraction of packet processing, then Amdahl's law implies that improving lookup costs cannot significantly improve performance. However, once

[4]For example in IP, after the header is parsed and the next hop is identified, the actual processing requirements for a packet may only involve changing the TTL field, updating the IP header checksum and moving the packet to an output interface. The data movement can be done by DMA engines.

	dst memory address	dst proc ID	pointer to data in packet	self-describing fragments	encryption, other processing
VAX Clusters[KLS86]	√				
Axon[SP90]	√				
Active Messages[vECGS92]	√	√			
Abbot&Peterson[AP93]			√		
Birrell&Nelson[BN84]		√			
Chunks[Fel93]				√	
DML	√	√	√		√

Table 2: Comparison of various proposals to add information to speed up data manipulation and dispatch.

we cache processing information on a per flow ID basis, lookup costs become more significant. We provide more detailed quantitative results later.

The use of cache memory like Content Addressable Memories (CAMs) is probably the most efficient solution for the lookup problem. This solution is expensive, inconvenient, and not an option for software implementations. The conventional software approach is to use a very good hash function (one that generates hash indices as randomly as possible) on a portion of the packet, that is *unique* per flow.[5] Well known hash functions fall into two categories. Some hash functions are simple to implement (e.g., XOR folding) but they do not produce truly random indices and as a result their performance depends crucially on the way addresses are assigned in a network. Other hash functions (e.g., CRC) produce truly random indices but are hard to compute.

The role of the hash function is to provide a deterministic random number which then can be used to index into a table or cache. The more random the indices are, the smaller the chances of collisions. Instead of each consumer of the packet (e.g., routers, end nodes etc.) computing the hash function for each packet in order to generate a random index into their tables, we suggest that the source of the flow provide this index directly as a field in the protocol header. This removes the need for hash computation at the receiver and allows the hash index to be more random than any simple hash function that we know of. We illustrate this idea, which we call source hashing, with an example.

Consider the case of a router. For each packet that arrives at the router, the router has to lookup the destination address in its tables to find the port to forward the packet. Routing tables are usually implemented using sophisticated data structures (e.g., BSD4.4 uses a radix tree, many OSI implementations use a trie). Searches in these data structures are quite expensive if the number of entries is large. Thus it is often a good idea to cache the most recently used entries, If each packet carries a flow ID and a source hash, then the cached lookup can use the source hash index as described in the next paragraph (see Fig. 1). For a router, a flow ID can be either the destination address or a more general flow ID (as in IP version 6) that defines the quality of service required by the flow.

[5]Uniqueness over the network can be achieved in many different ways: the simplest is to add a unique source address and a source specific component that each source can ensure is unique.

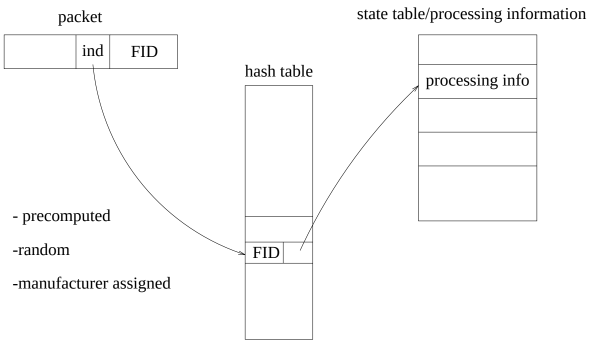


Figure 1: Source Hashing: the mechanism.

The first time a packet arrives with a flow ID FID and a source hash index ind, the router tries to lookup the hash table entry corresponding to the index ind. The router either finds that the entry is invalid or it contains a different flow ID. Then it processes the packet in the usual way, doing a long lookup using the routing table data structures and finds out which port the packet has to be forwarded to. This information is cached in the hash table in the entry corresponding to index ind. Then if the subsequent packets carry the same index ind and the same flow ID FID, these packets can be looked up in the hash table with very little cost.

What size source hash indices should be used? The simplest approach is to require sources to provide fairly large random indices (say 16 or 24 bits). Destinations that use smaller lookup tables can (assuming that the lookup table size is a power of 2) efficiently extract the right number of random bits — e.g., a destination with a lookup table of size 256 could use the low order 8 bits of the source random index.

In general, this combination of a flow ID and a source hash can be used at various layers. Table 3 describes examples at the Data Link, routing, and Transport Layers. For instance, for Data Link bridges the flow ID can be considered to be the Data Link destination address. For a timer-based Transport Protocol (used for RPC applications) the flow ID can be the destination process ID. An even more unusual application is described in Section 3.4 where we show how a flow ID can be used to speed up packet filter processing.

Layer	Application	FID
Data Link Layer	Bridging	Data Link Address
Network Layer	Routing	Flow ID
Timer-based Transport Protocols	RPC	Caller ID
User Level Protocols	Packet Filter	Flow ID

Table 3: Source Hashing: Applications.

3.1 Assignment Techniques

The source hash index is assigned by the source. The only condition to be satisfied is that it be the same for all packets in a flow. Source hash indices can be chosen in one of the following ways:

random: Each source chooses a random index at the start of a flow and uses the same index on all subsequent packets in the flow. Pseudorandom numbers will work well as long as each source uses a seed that includes its unique source ID. This introduces computation when a source starts a new flow. We can avoid this computation by constructing a long enough list of random numbers when the source first boots, and by using elements from this list in a round robin fashion.

precomputed: All machines using a specific protocol agree on using a particular hash function f. Sources compute $f(FID)$ when a flow FID first starts; the source then uses the resulting index on all subsequent packets. The only advantage of this over the random assignment technique is when multiple sources use the same flow ID (for example if the flow ID is a destination address). In the random index case, each source may choose a different index for the flow ID; this results in inefficient use of table memory at the receiver.

manufacturer assigned: Consider a flow ID that is assigned by a manufacturer (e.g., 6 byte Ethernet addresses). If the manufacturing process is changed to assign an extra two byte random index, then the resulting 8 byte address will automatically contain a 2 byte random index that is always associated with the 6 byte address. Any protocol (e.g., name servers, ARP protocols) has to be modified to pass the 8 byte address, including the random index. This technique can be used for static flow IDs like Ethernet and NSAP addresses. When it can be used it is superior to both random and deterministic assignment techniques because it has the best features of both (the index is truly random and yet all sources use the same index for a given flow ID). However, this technique is inapplicable for dynamically created flow IDs, especially at the network layer.

In summary, the primary advantage of source hashing over normal hashing is that the *receiver* of a packet *does not compute the hash index.* Computing a hash function was a trivial part of overall packet processing at low speeds; however, at higher (especially Gigabit) speeds the computation of the hash function is expensive in terms of either extra hardware or time.[6] A second advantage is *the hash indices are guaranteed to be more truly random than those computed by hash functions.* In ordinary hashing, there is a tradeoff between the complexity of the hash function and the randomness of the resulting indices: good hash functions based on CRC-like functions are expensive; cheaper hash functions typically are not as good or need to be tuned to the details of the particular addressing format. These problems are avoided with source hashing. We examine these issues in detail in Section 3.4 where we evaluate Source Hashing experimentally.

3.2 Performance Model

In this section we present a simple performance model for source hashing and illustrate its performance benefits. Let R be the end-to-end round trip time; let L be the time taken for a long lookup (e.g., the time taken for ordinary lookup without any indices); let s be the time required for a short lookup using an index; and n be the number of packets sent. We assume that the only significant delays are the propagation delay and the lookup costs. We assume only 1 hop between a source and a destination. The model can easily be generalized to multiple hops. Assuming that no packets are lost, we can compare the time taken for all n packets to be processed at the destination in the following three cases:

Normal Datagram Forwarding: The cost consists of the time taken to do long lookups for each datagram and a half round trip time for the first packet to reach the destination.[7] Total cost $= n * L + R/2$.

VC Forwarding: One round trip time is required for setting up the VCIs; all packets then take a short lookup time; once again a half round trip time is needed for the first packet. Total cost $= R + n * s + R/2$.

Datagram Forwarding with Source Hashing: Assuming there are no collisions, we take a long lookup time for the first packet, and short lookup times for all other packets. As usual we need a half round trip time for the first packet to reach the destination. Total cost $= L + (n-1) * s + R/2$.

Typically the round trip time is large as compared to long lookup time. When $n = 1$, i.e., there is only one packet to be transmitted, Source Hashing is as bad as normal datagram routing. When there are a large number of packets, source hashing is not much better than VC-routing. As shown in the figure (Fig. 2 – which is a plot of the performance model equations) source hashing is better than both VC-routing and datagram routing if the number of packets is more than 1. In the case of a flow with a large number of packets, VC-routing and Source Hashing are not too far apart. We believe there are several applications where the number of packets is somewhere in between.

[6] For example, DEC's GIGAswitch product, which is a multi-port FDDI bridge, uses a semi-custom chip to do hashing at 100 Mbps.

[7] Only the first packet takes this amount of time, since packets are assumed to be sent in a pipelined fashion.

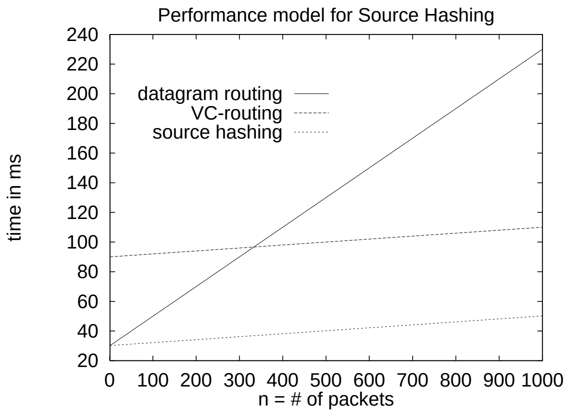


Figure 2: Source Hashing: a performance model. $R = 60ms$ (approximate coast to coast round trip latency), $s = 20\mu s, L = 200\mu s$.

These include applications like File Servers in which a small to medium number of data blocks are typically written or read occasionally. Often applications do not know in advance how many packets they will send in a flow; thus it is hard to decide whether it is worth setting up a VC[8] or just sending the packets without any set up. Source hashing avoids having to worry about this tradeoff; source hashing is better than either the VC or datagram models if a small to moderate numbers of packets are sent in each flow. Source hashing is also competitive with VCs if a large number of packets are sent in each flow.

3.3 Applications

We now discuss more details of the applications of source hashing that were outlined in Table 3.

Data Link Layer Data Link learning bridges should ideally forward data packets in the time required to receive a packet, which can be a few microseconds on many fast LANs. In this time, bridges need to lookup the destination address in order to forward the packet, and lookup the source addresses in order to learn. If we add a manufacturer assigned source hash index to the destination and source addresses, lookups can be simplified, and can even be done in hardware quite easily.

Network Layer We described this application in detail while introducing the idea of source hashing. (see Section 3) Once again, the idea is to use consistent flow IDs along with a random index, assigned by the source. The intermediate routers can make use of the random index in place of a hash function to reduce the lookup cost. We have also conducted experiments on this idea and the results are described in Section 3.4.

[8] In applications that exhibit considerable locality, it may be useful to keep a virtual circuit up after the data has been sent so that the circuit can be reused for other data to the same destination. However this strategy will not work for applications (e.g,. hypertext applications like Mosaic) that do not exhibit good locality of reference.

Timer-based Transport Protocols In Birrell and Nelson's RPC [BN84] each RPC call packet sent by the source carries a unique identifier (a unique network wide identifier of the process that initiates the call concatenated with a unique sequence number). Sequence numbers increase monotonically for each process. The RPCruntime at the destination maintains a state table giving the sequence number of the last call made by each caller. When a call packet is received, its call identifier is looked up in this table; if the sequence number is not greater than the last sequence number recorded for this process, the call packet is discarded as a duplicate. State information can be discarded at the receiver after a period equal to the maximum time a duplicate packet can live in the network. Thus if there is no information about a process in the state table, any new packet for that process is accepted and its sequence number must be recorded in the state table.

We can use source hashing to improve the lookup performance of such a transport as follows. First, we modify the lookup table at the receiver such that the lookup entries point to a *list of pointer blocks* much as in hashing with chaining [CLR90]. Each pointer block contains a pointer to a state table entry; state table entries contain a Process ID, the last sequence number for that process and any other state. A source process adds a consistent random index r with each of its call packets that is used to index the lookup table. Then the list of pointer blocks is searched to find an entry corresponding to the Process ID; if no entry is found, a new entry is made in the state table and a new pointer block is placed at the head of the list. This is very similar to doing hashing (with chaining) on the process ID except that the random index is a true random number chosen by the source.

User–level Protocols For implementing user–level protocols, the BSD4.4 operating system provides packet filters [MJ93] that allow demultiplexing of user level packets without extra costs for crossing kernel-user boundaries. Each packet that is received on a network interface is passed on to the series of filters that are attached to that interface. These filters are attached by user–level protocols using system calls. If a filter specification matches the packet, it is delivered to that process. Potentially, all filters could receive a packet.

In BSD4.4 the filters are processed serially. Our proposal is to add a flow ID and a source hash index (which can be placed in the special DML header that we propose later) to each packet. For every flow ID, we cache the processed information (the set of user–level processes that receive the packet) after the first packet. The remaining packets get delivered directly without going through the filtering process, if we get a cache hit. This can greatly reduce processing when there are a large number of packet filters.

We chose two of these applications — IP forwarding and the BSD Packet Filter (BPF) — to experimentally evaluate Source Hashing. The results are described in Section 3.4.

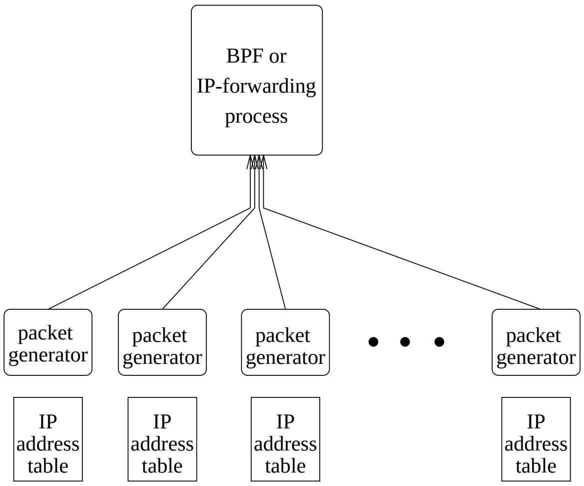


Figure 3: Source Hashing: Experimental setup.

3.4 Experimental Results

We conducted two different experiments (IP forwarding and BPF) to evaluate the performance benefits of source hashing and flow IDs. To separate out the effects of flow IDs from source hashing, we obtained three separate numbers: processing time without flow IDs and caching, processing time using flow IDs and normal hashing, processing time with flow IDs and source hashing. Both BSD Packet filter and IP forwarding implementations have tables (maintained as a linked list in the packet filter code, and as a radix tree in routing code) that can make use of source hashing to improve their lookups. In both experiments we extracted the relevant portions of the BSD4.4 kernel into user space and exercised the code to measure the performance differences.

The experimental setup in both cases is shown in the Fig. 3. Each of the packet generators set up a table containing two IP addresses (representing a source and a destination), a unique ID for each table entry, and an associated random index. The generators select the IP addresses from different sections of a file[9] of IP addresses. In effect, the generator modules simulate a set of workstations whose packets reach the router/packet filter.

The central module contains the code for the router or packet filter. We chose a configuration with 16 filters for the packet filter program and a configuration with 16 interfaces for the router. In IP forwarding we set up a routing table with at most 100 entries. In both cases we used the concatenation of source address, destination address and a unique ID assigned by the generator as the flow ID. Together they make up a flow ID that has 12 bytes. We used a cache with 65536 entries. The experiments were run on a Sun 4/370 (processor: sparc 1, OS: NetBSD/sparc current version) machine.

From the graphs in Fig. 4, it is obvious that caching the most recently used entries using a flow ID as a key gives us performance benefits. In particular, we get about 45% improvement in processing time in the case of IP forwarding, and approximately 6-7 fold improvement in processing time in the case of BSD packet filter.

[9]This file is a modified version of /netinfo/hosts.txt available at nic.ddn.mil.

Hashing mechanism	# of instructions	disadvantage
Source Hash	2	
XOR Hash	15	more collisions
CRC Hash	100	too complex

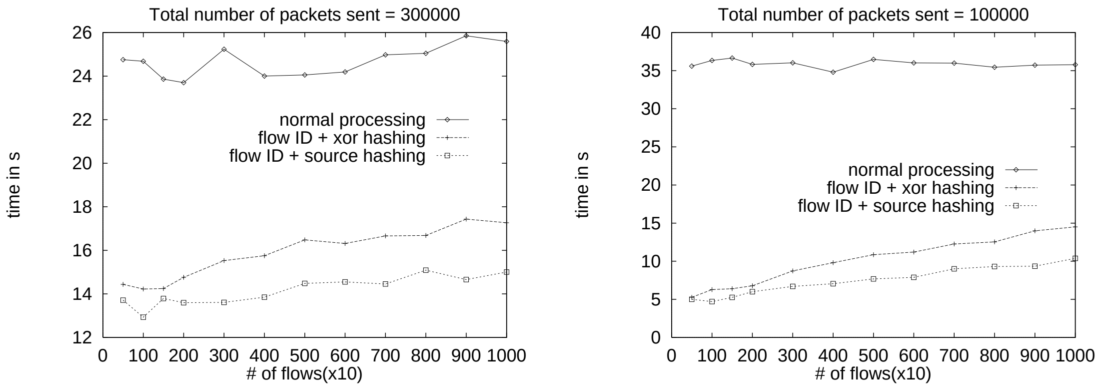
Table 4: Number of instructions required for different hash functions.

Given that caching is important, we turn to evaluating different lookup schemes. Most software cache lookup schemes are based on hashing. Evaluating different hash functions based on the experiments shown in Figure 4 was difficult because the time taken for the hash functions was quite small, and the measurement granularity was not fine enough. So we chose to compare them using the number of instructions required to compute the hash index in each case. This data is given in Table 4.

XOR folding works by taking XOR of all the bytes in the flow ID. If we do this 8 bits at a time, we obtain a reasonably random value. However, that limits us to a table size of 256 entries. If we do it 16 bits at a time, it is not sufficiently random. In CRC hashing we used CRC-8 on two halves of the unique identifier and concatenated the two to get a 16 bit hash index. (Even if we use CRC-16 or CRC-32 we believe the number of instructions will not be considerably smaller.) The instruction counts shown are dynamic counts — i.e., measured using `adb` single stepping through the program, compiled by `gcc-2.6.3` using its best optimization scheme. These include the instructions to compute the hash functions only. As we can see, Source Hashing needs the least number of instructions.

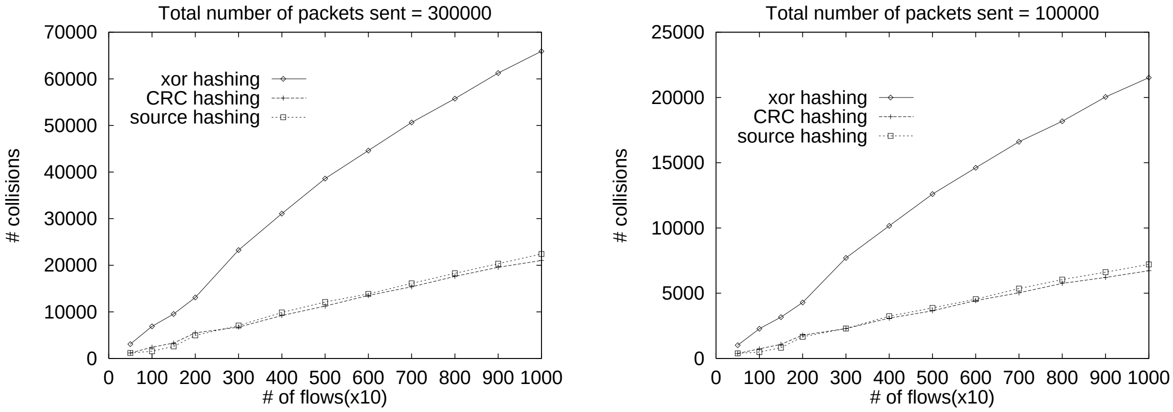
Let C denote the remaining processing time required beyond the computation of the hash index. This would include the memory lookup, comparison with the flow ID and any other packet processing. The performance improvement offered by source hashing will vary depending on the value of C, as dictated by Amdahl's law. Since XOR hashing has a fairly large number of collisions (see below) and CRC hashing is quite expensive, it seems reasonable to assign approximately 30 instructions for a hash computation other than source hashing. If $C = 30$, which may be possible if large parts of the other protocol processing are done in hardware, then source hashing will provide a factor of 2 improvement in performance over this hashing scheme.

For example, in BPF and IP forwarding, getting to the processing information takes only about 18 instructions after computing the hash index—see Section 4.1. In IP forwarding there will be an additional read and write of the TTL field. The remaining cost is of memory copy and we believe it is not appropriate to include the cost of memory copy in the per packet processing cost. Memory copy can be avoided by passing pointers around, or by relegating it to hardware. Thus (assuming a few more instructions to program the data movement hardware) a C value of 30 seems reasonable for the IP and BPF applications that we



(IP forwarding) (BPF)

Figure 4: Evaluating benefits of flows IDs and source hashing.



(IP forwarding) (BPF)

Figure 5: Comparison of Different Hashing Schemes.

have studied.

Probably the most important reason to use source hashing, however, is not the specific performance improvement but the predictability of its performance.

Another important way to evaluate the effectiveness of a hash function is to look at the number of hash collisions for a given table size. The plots in Fig. 5 show that source hashing is as random as CRC-hashing and needs only almost a tenth of the instructions that are needed to compute CRC. Simple hashing schemes like XOR-hashing lead to more collisions mainly because the IP addresses that collide at a router are mostly from the same subnet.

3.5 Subtle Issues in Assigning Flow IDs

After a crash at sender if we send packets with same flow ID as before but with different packet processing requirements, the router may keep processing new packets according to the old flow specification. To avoid this, as is done in transport protocols, we could **a)** either time out at the router periodically or **b)** add a random component to flow ID which makes it unlikely to have two identical flow IDs before and after a crash. Thus we have three components to a flow ID: one a component for space uniqueness (e.g, source address concatenated with a source specific flow number); second, a random part for time uniqueness after crashes; third, a random source hash index. In the proposal draft for IPv6 [Hin94] (work in progress), the latter two components are combined: we prefer to separate the two issues with the understanding that in optimizations (especially to save bits) the two random components may be combined. Note that it is important to align the source hash index at a word boundary within a packet; if not, the extraction of the source hash index may need more than two instructions.

4 Threaded Indexing

Threaded Indexing provides a mechanism for forwarding datagram flows as efficiently as connection oriented flows without the overhead of setting up a connection. The idea is as follows:

Exactly as in VC-routing, each packet carries with it an index which is an index into the routing table of the next hop. The difference lies in the process of setting up the indices. In VC routing, the VCIs are set up at data transfer time (analogous to run-time). Threaded indexing sets up an entry for every possible destination, and the indices are set up when the topology changes (analogous to

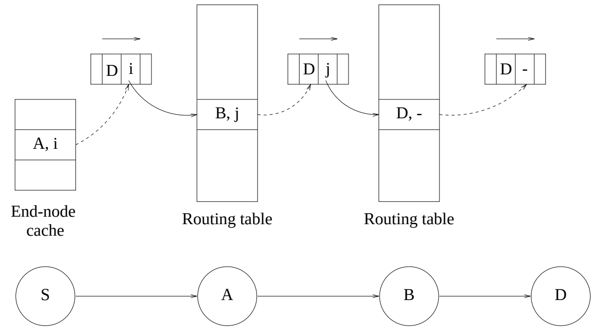


Figure 6: Threaded indices connecting routing entries in a path from a source S to a destination D through two routers A and B. The routing entry at each node contains the next hop node *as well* as an index (i.e., i, j) into the next hop node's routing entry.

compile-time). The differences between Threaded Indexing and PVCs have been listed in Table 1. PVCs need huge amounts of memory since a connection must be maintained between every possible source destination pair.

Once the tables are set up correctly, the datagram messages are routed as illustrated (Fig. 6). Before the source S can send a packet to the destination D it first sends a query to the next router A asking for D's index (which is i at node A). S then caches index i and uses it on all subsequent packets to D. When a packet from S arrives at A, A uses the index i to index into its lookup table. The table entry contains the next hop (i.e., B) as well as the index into B's table (i.e., j). The new index j now is placed in the packet before it is sent to B. The process of following threaded pointers continues until the last hop, where no index is required.

We give a quick example of how to modify an existing routing protocol to calculate threaded indices. Many routing protocols (e.g., RIP) are based on the Bellman-Ford algorithm [Tan81, Per92]. The modifications required for Bellman-Ford are illustrated in Figure 7. Normally each router sends its shortest cost to each destination D to all its neighbors. Router R calculates its shortest path to D by taking the minimum of the cost to D through each neighbor. The cost through a neighbor like A is just A's cost to D (i.e., 5) plus the cost from R to A (i.e., 3). In Figure 7 the best cost path from R to D is through router B. We modify the basic protocol so that *each neighbor reports its index for a destination in addition to its cost to the destination.* Then each router uses the index of the minimal cost neighbor for each destination. In Figure 7, R uses B's index (i.e., j) in its routing table entry for D. Any link state routing protocol [Per92] can be similarly modified by having each router send control packets listing its destination indices to all its neighbors.

4.1 Performance Model

In this section we provide a simple performance model for threaded indexing. This model is almost identical to that of source hashing described in Section 3.2. As before, R is the round trip time, s is the time required for a short lookup using an index, and n is the number of packets sent. Assuming one hop, the cost of threaded indexing is $n * s + R/2$ — the time for one short lookup for each packet and an additional half round trip time for the first packet. This curve, if plotted, is almost identical to that of source hashing in Fig. 2. This indicates that threaded indices work much better than VC-routing in cases where there are a small to moderate number of packets, since it saves the one round trip time of setting up the indices.

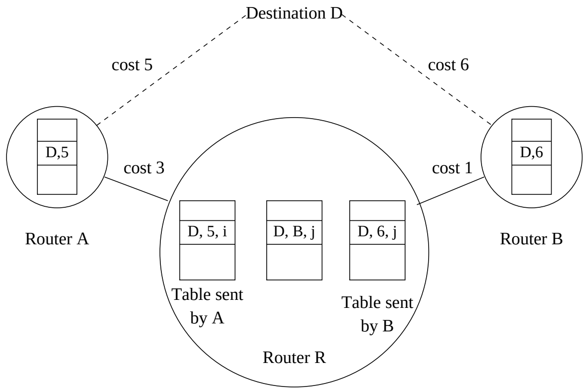


Figure 7: In Bellman-Ford, router R calculates its best path to destination D by choosing the neighbor (i.e., B) that yields the shortest cost ($6+1=7$) path to D. Each neighbor also passes an index into its routing table (e.g., j for B) together with its cost to D. R chooses its threaded index as the index (i.e., j) of its minimum cost neighbor (i.e., B).

scheme	# of instructions
XOR Hash Index	33
Source Hash Index	20
Threaded Indices	8

Table 5: Comparison of the number of instructions required to obtain routing information using different lookup schemes.

In Table 5, we compare source hashing, XOR-based hashing and threaded indexing based on the number of instructions that are needed to obtain the packet forwarding information for one packet. Instructions were counted using `adb`, single stepping through the program compiled by gcc-2.6.3. The number of instructions required to compute a hash index is reported in Table 4. Table 5 includes hash index computation and the additional steps required to get to the processing information for the packet.

Hash table based schemes have to compute the hash index, compare the key in the hash table to ensure that we have a hit, and then do a look up of the processing information by following a pointer into the processing information table. Since source hashing needs less instructions to compute the index, it requires less instructions overall than XOR based hashing. (Comparison of a 12 byte long flow ID with a table entry and chasing a pointer into a processing information table cost us 18 instructions.) In the case of threaded indices, we do not need to do the key match, nor do we have an additional pointer dereferencing. The threaded index is the index into the routing table. Thus the processing involved in the case of threaded indices consists of extraction of the index from the packet; looking up the corresponding entry in the routing table; obtaining two pieces of information – a port number to forward the packet to and a new label to be put on the packet header; and writing the new label in the packet header. This takes approximately 8 instructions.

If threaded indices are used for multicast forwarding on LAN links, the packet can contain only a single index which may be received by multiple routers. Thus all the routers on a LAN must agree on indices. This is more difficult than unique allocation within a router but it is still easier than unique global allocation of indices. One scheme would elect a single designated router on a LAN and have this router allocate indices and broadcast these indices to other routers.

Threaded indexing is preferable over source hashing wherever possible because (a) it requires less processing, (b) it requires less memory, and (c) it is deterministic. Hence it is best suited for hardware implementation. Source hashing, on the other hand, is more general, in the sense that it can be applied to a variety of applications where threaded indexing is not suitable. Threaded indexing is suitable only in cases where we can set up the indices ahead of time, whereas source hashing is usable in applications like packet filters, where setting up indices ahead of time seems infeasible.

A limitation of threaded indices is that it does not work well at hierarchy boundaries. However, it can be used within each level of hierarchy. Consider a Level 1 Router that sends a packet to a Level 2 Router that is looking up packets based on area addresses. To avoid a lookup at the Level 2 Router, the Level 1 Router has to keep an index for every possible area which would defeat the purpose of hierarchy! Assuming high performance is most critical for local communication, we suggest the following combination of source hashing and threaded indices.

The idea is to use source hashing for every packet that is sent outside the local area. We assume that each router has a cache of currently active destinations (that can be timed out) outside the router's level of hierarchy. If a source end station finds that a destination end station is outside its area (the source could find this by the same process it finds the threaded index for the first router), it adds a random index to the packet. Assume for simplicity, that we use a bit to distinguish between a random index and a threaded index. A router R getting a random index, applies the source hash algorithm to search its cache (as opposed to its threaded index forwarding table). If R does not find an entry in the cache for destination address D, R does a conventional lookup and adds a cache entry for D to the cache. Subsequent packets sent to D can then have fast lookups in the cache using the random index.

In summary, the idea is to use source hashing for all communication between areas, and to use threaded indexing only for local communication. This may be a useful combination, if local communication is important to optimize.

It is also important to compare threaded indices with a protocol called Pip proposed by Paul Francis [Fra93]. Pip is an internet protocol that tries to allow for maximum flexibility in addressing modes while not degrading packet processing speed. It does this by breaking hierarchical addressing information into a series of distinct fields (called FTIFs), and by including a pointer to the field that is active at any given time. This allows a router at level k of the hierarchy to determine the k-level component of the address without parsing the entire address. The FTIFs could be used for performance as well: the FTIF of Pip could be used as a direct index into a forwarding table.

The major difference between Pip and threaded indices is that in threaded indices the index (or FTIF in Pip) *changes on every hop* as in virtual circuits, while in Pip the index at a given level of hierarchy *is the same*. For example, with only one level of hierarchy, every Pip router would use the same index for a given destination which leads to a global index allocation problem. On the other hand, in threaded indices (as in VCs) indices need only be unique at a given router, and thus the index allocation problem is trivial. Also, the Pip paper [Fra93] talks only about the structure of addresses and not the modification to the routing protocol to support the structure, whereas we have described how to modify Distance Vector routing to implement threaded indices.

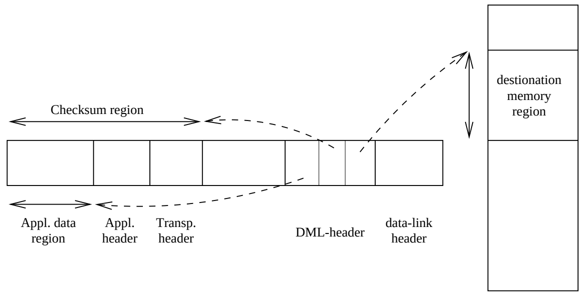


Figure 8: The Data Manipulation Layer header at the start of a packet can contain pointers that allow data to be manipulated while it is being received. A region consists of an offset and a length. The figure shows three potentially useful regions for the destination of user data (in memory), the region containing user data (in the packet) and the region (within the packet) covered by the transport checksum.

5 Data Manipulation Layer

We propose adding the information required for data manipulation and dispatch to a *Data Manipulation Layer header* that is placed in an easily accessible portion in the front of the packet. One possibility is to add this header directly after the Data Link header and before any Transport or Routing headers. This is convenient because the hardware or software driver that copies packets from the network to the host typically knows how to parse the Data Link header and hence can easily access the Data Manipulation Layer (DML) header. It is particularly convenient to add a DML header when a new Data Link protocol is being standardized; adding a DML header amounts to lengthening the Data Link Header.

A DML header (Figure 8) contains, among other things, a sequence of *regions*. A region consists of an offset, a length and a type field. The offset and length fields demarcate a sequence of bytes in the packet. The sequence of bytes represent the region of data to be manipulated (e.g., the transport data that is checksummed) or a region that provides information within the packet that is required for manipulation (e.g., a field that can be looked up in a table to yield an encryption key). The type provides information on the type of manipulation.

At the sending end system, it is easy to calculate the region descriptions with only a slight change to layer implementations. The idea is that the layer responsible for calculating a region passes down the region description (including the offset) to the next lower layer. Each layer n then adjusts the offsets of all regions created by higher layers (i.e., layers $n+1$ and above) to add in the length of the Layer n header. This simple structured technique of *accumulating offsets* deals with variable size headers and allows layer implementations to be changed easily.

We turn to specific data manipulations. Checksumming of transport layer data is used as an end-to-end check [SRC84] against undetected corruption. Checksumming only requires the checksum region to be demarcated. A reasonable solution for end-to-end data integrity and privacy is encryption at the transport layer. Encryption requires two regions: a region that can be used to extract the decryption key (when receiving a packet) and the actual region to be decrypted. For example, if each connection has a separate key, the decryption key can be found by extracting the connection ID and using that to lookup the decryption key.

It is desirable to avoid data copying by having packets copied directly from the network to their final destination in application space [KLS86, CJRS89, BP93]. It seems desirable to have two regions for data copying: a *memory destination* region which is a field that can be used to find the location in memory for the data bytes, and another region which gives the starting offset and length of the data within the packet.

Similarly, the DML header can contain information that helps to dispatch the packet to the final destination process. This can include a pointer to the message handler (as in Active Messages [vECGS92]) and flags that allow the source to control the rate at which interrupts occur at the destination. (The destination is interrupted for this packet only if certain bit of this flag is set.)

5.1 Advantages of a DML header

The data manipulation layer does allow a structured implementation of Integrated Layer Processing (ILP) [CT90] in which a number of data manipulations are combined. To avoid memory accesses, an ILP implementation attempts to read the data once from memory and keep the data within registers or cache while performing all the required data manipulations in one pass.

Clark and Tennenhouse [CT90] show a gain of 50% in packet processing for some experiments. ILP performance, however, depends on a number of factors including the machine architecture, ordering constraints among data manipulations [AP93] and the complexity of the data manipulation functions (e.g., DES encryption [GPSV91]). Diot and Braun [BD95] describe experiments in which the overall throughput improvement for an ILP implementation is only 10-20%. They also show that high cache hit rates can be achieved without integrating the several functions as long as the loops are processed close to each other. Thus the performance benefits of ILP are not completely clear.

It is important to note, however, that the advantages of a DML layer are not confined to ILP implementations. Even in a non-ILP implementation that needs to implement several data manipulations, placing information in the DML header allows hardware engines to manipulate the data without parsing headers. Similarly, a DML layer supports the basic notions of fast scheduling and avoiding data copies that are useful for most protocol implementations and have been justified before in active message [vECGS92] and VAX Cluster [KLS86] implementations.

If adaptor hardware is processing packets (we know of such a chip being designed at Washington University) to keep up with high speeds, then the main danger of parsing layer headers in hardware is that there are too many commonly used protocols, and hardware is too inflexible to deal with new protocols. One could implement packet filters in hardware, but interpreting multiple filters should increase cost or slow down processing. The DML layer, on the other hand, can be kept constant with a given data link and should be able to handle new higher layer protocols.

If the adaptor is processing packets in software or hardware, there is a minimum cost to skip past a layer header. To skip past a layer header and get to the next header, one has to process the length field (e.g., IP, TCP) and lookup the protocol ID field in order to know how to parse the next header. Using the same instruction counts we expect roughly 10 to 25 instructions per layer. In a general setting with a large amount of possible protocol clients at each layer, the cost should be closer to 25 because of the cost of a hash lookup of the Protocol ID field. Finally, when we get to final header, one has to deal with possible options to get to the final field. Thus the DML Layer can also save a significant amount of processing, especially in a diverse environment where there are multiple protocol stacks.

The regions in our Data Manipulation Layer header can be considered to be a generalization of the technique of [AP93] in which a single pointer (to the start of the application data) is added to a lower layer header. Our scheme is more general[10] because it allows a *set* of regions to be demarcated. We believe our structured technique of using accumulated offsets to calculate regions is also useful. On the other hand, we have not addressed a number of other issues relating to modularity and ordering constraints in ILP that are well treated in [AP93]. In [AP93], modularity is preserved by "automatically synthesizing the integrated implementation from independently expressed protocols." They use the idea of a word filter to accommodate different data units for different data manipulations, and a three

[10] and not much more expensive, because the main cost is in adding a new header. The number of extra bytes added should not be very significant.

stage model to handle ordering constraints. These are important ideas, and are *equally* applicable to protocols that use a Data Manipulation Header.

All the features described here need not be part of a given DML standard. It should be possible to implement subsets of DML functionality. For example, in a link interface chip that is being built locally, the chip separates out user data from protocol headers. The original version of the chip relied on fixed length (and special-purpose) protocol headers to allow the chip to easily find where the user data begins. The chip designers are contemplating the addition of a pointer to the user data after the Data Link header, thereby allowing the use of existing routing and transport layer headers that have variable lengths. Notice that in this example the information in the DML does not pertain to ILP.

6 Summary and Conclusions

This paper is based on the premise that packet processing is more expensive than network bandwidth. In this paper, we make two somewhat radical suggestions for additional packet fields to make packet processing faster. First, we propose adding additional index fields to protocol identifiers at all layers. Examples of such identifiers include connection identifiers, Network and Data Link addresses. We also propose the addition of a data manipulation layer to an easily accessible portion of each packet, so that implementors can do Integrated Layered Processing in a structured way.

Identifier index fields can be used in various ways to reduce lookups and packet processing. Perhaps the simplest idea is to standardize such fields but to let implementations decide which of several possible techniques they wish to use. We have proposed two new schemes *source hashing* and *threaded indexing* that use these indices. Both our techniques look superficially similar to VCIs in virtual circuit networks. The major difference is the absence of a round trip delay that is required to set up a traditional VCI. These differences are explored more carefully in Section 2 and summarized in Table 1.

It is hard to characterize the cost of lookups because they are so implementation dependent. For instance, hashing by exclusive-OR folding of address octets seems effective and easy to implement [Jai92]. In [CJRS89], the cost of hashing (in the best case) is taken to be 25 instructions. However, we believe the lookup techniques described in this paper are important because: **1)** Lookups are a critical part of cache processing and cache processing is becoming increasingly important for high speed networks **2)** Our techniques do not depend on address or traffic patterns and **3)** The lookup costs accumulated over several layers do add up.

Our Data Manipulation Layer is a generalization of many existing and successful ideas in reducing data processing. We have proposed that all the required information be collected in *a separate layer header* that is easily accessible to the adaptor or the lower level software to avoid the cost of the adaptor having to parse a number of layer headers in real time. The justification for integrated layer processing [CT90] and active messages [vECGS92] and avoiding data copies [KLS86] has been made before in quantitative terms. The Data Manipulation Layer only provides increased flexibility in obtaining this information as detailed in Section 5.1. We have not substantiated this idea by doing experiments as we have for source hashing; we leave this for future work.

In conclusion, we note that the current climate of transition (in which transport, routing, and data link protocols are changing in order to support new technologies and applications) makes this a good time to consider adding additional header fields. It is easy to add these headers gradually to existing implementations while remaining compatible with older implementations. For these reasons, we hope that there will be opportunities to apply our techniques to influence the new generation of protocols that are swiftly emerging.

Acknowledgments: Dave Clark independently invented source hashing. We would like to thank him, M. Shreedhar, Andy Fingerhut, and the anonymous reviewers for their valuable feedback and comments. We are grateful to Steve Deering for reminding us of the case of multicast forwarding; to Paul Francis for helping to compare threaded indices to Pip; and to Jon Crowcroft for pointing us to some missing references.

References

[AP93] M. Abbott and L. Peterson. Increasing network throughput by integrating protocol layers. *ACM Transactions on Networking*, 1(5), October 1993.

[BD95] T. Braun and C. Diot. Protocol implementation using integrated layer processing. In *Proceedings of the ACM SIGCOMM '95 Symposium*, pages 151–161, October 1995.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[BP93] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, February 1993.

[CJRS89] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6):23–29, June 1989.

[Cla85] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[CT90] D.D. Clark and D.L. Tennehouse. Architecural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[CV95] Girish P. Chandranmenon and George Varghese. Trading packet headers for packet processing. In *Proceedings of the ACM SIGCOMM '95 Symposium*, pages 162–173, October 1995.

[Fel93] D. C. Feldmeier. A data labelling technique for high-performance protocol processing and its consequences. *Computer Communications Review (SIGCOMM '93)*, 23(4):170–181, October 1993.

[Fra93] P. Francis. A near-term architecture for deploying pip. *IEEE Netowrk*, pages 30–37, May 1993.

[GPSV91] P. Gunningberg, C. Partridge, T. Sirotkin, and B. Victor. Delayed evaluation of gigabit protocols. In *Proceedings of 2nd MultiG Workshop*, June 1991.

[Hin94] R. Hinden. Editor, Internet Protocol, version 6 (IPv6) specification. *Draft (work in progress)*, October 1994.

[Jai92] Raj Jain. A comparison of hashing schemes for address lookups in computer networks. *IEEE Transactions on Communications*, COM-40(10):1570–1573, October 1992.

[KLS86] N.P. Kronenberg, H. Levy, and W.D. Strecker. VAXClusters: a closely coupled distributed system. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[KP93] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *Proceedings of the ACM SIGCOMM '93 Symposium*, pages 259–268, September 1993.

[MJ93] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Technical Conference Proceedings*, pages 259–269, Winter 1993.

[Par93] C. Partridge. *Gigabit Networking*. Addison-Wesley, Reading,MA, 1993.

[Per92] R. Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading,MA, 1992.

[SP90] J. P. G. Sterbernz and G. M. Parulkar. Axon: A high speed communication architecture for distributed applications. In *Proceedings of IEEE INFOCOMM '90, San Fransisco, CA*, pages 415–425, June 1990.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[Tan81] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N. J., 1981.

[Var94] George Varghese. Trading packet headers for packet processing. Technical Report WU94-16, Dept. of Computer Science, Washington University, 1994.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture.*, May 1992.