Memory-Efficient State Lookups with Fast Updates

Sandeep Sikka
Washington University
sikka@ccrc.wustl.edu

George Varghese
UCSD
varghese@cs.ucsd.edu

ABSTRACT

Routers must do a best matching prefix lookup for every packet; solutions for Gigabit speeds are well known. As Internet link speeds higher, we seek a scalable solution whose speed scales with memory speeds while allowing large prefix databases. In this paper we show that providing such a solution requires careful attention to *memory allocation* and *pipelining*. This is because fast lookups require on-chip or off-chip SRAM which is limited by either expense or manufacturing process. We show that doing so while providing guarantees on the number of prefixes supported requires new algorithms and the breaking down of traditional abstraction boundaries between hardware and software. We introduce new problem-specific memory allocators that have provable memory utilization guarantees that can reach 100%; this is contrast to all standard allocators that can only guarantee 20% utilization when the requests can come in the range $[1 \ldots 32]$. An optimal version of our algorithm requires a new (but feasible) SRAM memory design that allows shifted access in addition to normal word access. Our techniques generalize to other IP lookup schemes and to other state lookups besides prefix lookup.

1. INTRODUCTION

Internet usage has been expanding both because of a growing number of users and an increasing demand for bandwidth intensive data. To keep up with increased traffic, the speed of links in the Internet core has been increased to at least 622 Mbps, and vendors are working to build faster routers that can handle Gigabit and now even Terabit links. Thus there is a major need for high performance routers at speeds that keeps increasing. OC-192 (10 Gigabit) and possibly OC-768 (40 Gigabit) links will soon reach the marketplace, and router vendors (e.g., Cisco, Juniper, Extreme Networks) are already considering routers that target these higher speed links.

A router that forwards a message has two major tasks: first, looking up the message's destination address; and second, internally transferring the message to one of many possible output links. The second task is well understood with most vendors using fast busses or crossbar switches, and some exciting new switching technologies [MIM97, TCFF97]. In the last three years, several new solutions have appeared to the address lookup problem as well for Gigabit speeds [DBCP97, SV98, WVTP97]. However, our paper will argue that there are some new problems in the areas of memory allocation and pipelining that need to be solved when scaling such lookup schemes to OC-192 and higher speeds.

The design of such a high-end router will typically have the goal of forwarding minimum sized TCP packets (say 40 bytes, fifty percent of Internet packets are this size [TMW97]) at 10 Gbps rates (OC-192) or higher. Given 32 nsec to forward a minimum size packet and the fact that ordinary memory (called DRAM) takes 60-100 nsec nsec to do a single READ, a common approach is to design a custom chip to do IP lookups at each router port.[1] The prefix database can be stored in external SRAM. External SRAM is similar to CPU cache memory with 10-20 nsec access times. However, speed, pin count limitations, and the need for a smaller number of chips tend to favor the use of on-chip memory for prefix storage. Given that even aggressive manufacturing processes today can guarantee only 16 Mbits of on-chip memory on a custom chip, it appears necessary to use a compressed IP lookup data structure such as a Lulea trie [DBCP97] to store prefixes.

While IP lookup schemes with large update times are feasible, they require the use of two copies of the database: one for the copy being updated and the second for the copy used to forward packets. Since this cuts the utilization of the limited on-chip SRAM by half, it is preferable to have a scheme that allows incremental updates that only requires a small constant increase in memory to handle updates. Incremental updates may also be useful to quickly handle multicast routes, and the most extreme instabilities caused by backbone routing protocols [LMJ97]. Using say a compressed trie, incremental update can result in adding or deleting a prefix from a trie node, which in turn requires deallocating memory for the existing compressed trie node and allocating memory for another trie node of a slightly different size. If the trie nodes can be any size from (say) 1 to 32 words, such a scheme requires a memory allocator capable of allocating and deallocating chunks of memory in the range $[1, \ldots 32]$.

[1] We will argue that Content Addressable Memories (CAMs) are not adequate for backbone routers later.

Unfortunately standard memory allocators[2] do not guarantee good worst case utilization. It is possible for allocates and deallocates to break up memory into a patchwork of holes and allocated blocks. Specifically, the blocks are minimum sized, and the holes are just small enough not to be usable for a maximum size request. Thus only $\frac{1}{32}$ of chip memory can be guaranteed to be used.

If one ignores the allocator it is easy to show that 16 Mbit of on-chip memory can be used to support (say) 250,000 prefixes even in the worst-case. If one takes the allocator into account, the chip can only advertise 7500 prefixes. But Content Addressable Memory (CAM) vendors today are advertising worst-case numbers of 8000-128,000 prefixes (though some of the high end CAMs have not yet appeared in the market) with 15 nsec search times and single cycle updates. CAMs may still be too slow for the fastest backbone links, may have too small a worst-case number of prefixes for a backbone router, and cannot be used to integrate all of IP forwarding on a single chip. But it is important for a chip solution that competes with CAM solutions to have a large worst-case prefix database size it can advertise.

Besides the poor worst-case performance of standard allocators, there is a standard result [Rob71] which states that *no allocator* (that does not compact memory) can have a utilization ratio better than $\frac{1}{\log_2 W}$ (i.e., 20 percent if $W = 32$), where W is the largest possible allocation request. Since this is still unacceptable, in this paper we consider allocators that do compaction. Compaction refers to the moving of allocated blocks to increase the size of holes.

Compaction has two immediate problems. First, moving a piece of memory M requires correcting all pointers that point to M. Second, the existing literature on compaction is in the context of garbage collection (e.g., [Wil92, Bak78]), and tends to use *global* compactors that scan through all of memory in one compaction cycle. A plausible solution comes from real-time garbage collectors [Bak78] that interleave compaction with computation. But such collectors often break up memory into two "semi-spaces", and compact one space while allocating from the other. The use of semi-spaces forces derivative schemes to utilizations of 50 percent or less.[3] Thus one of the questions we ask in this paper is whether there exists a *local* compactor that compacts only a small amount of memory around the region affected by an update?

The answer to this question is yes. Our paper introduces two such local compaction schemes for the first time (to the best of our knowledge) in the literature. Our algorithms do $2kW$ units of compaction work in return for a utilization ratio of $\frac{k}{k+1}$, where k is a tunable parameter. By choosing say $k = 9$, a lookup chip needs to do only $18W$ compaction work to obtain 90% utilization. This in turn would allow the chip vendor to guarantee a worst-case of 250,000 prefixes for the lookup chip. Since an update must write and read W units in the worst case, this additional compaction work per update is a reasonable cost.

[2][WJNB95] contains an excellent survey of 30 years of research on memory allocators

[3]Semi-space and generational garbage collector schemes are discussed in more detail in the Related Work section.

Together with the use of pipelining, our IP lookup scheme *scales with memory speeds*, allowing the possibility of IP lookup schemes even for fast links without the use of CAM technology. CAMs have historically been slower and less dense than SRAM, and this trend appears likely to continue. Besides IP lookups, we also argue that our approach is beneficial for other state lookup tasks in networking such as bridge lookups, accounting lookups, flow lookups, and filter lookups. To test our ideas we implemented them in the context of a compressed trie IP lookup algorithm (a scheme very similar to one described by [Per99] and different from the Lulea [DBCP97] scheme) and report on our results for prefix utilization using real BGP traces that allow us to compare our problem-specific allocators with the best known general purpose allocators. We note that we can solve the first compaction problem of adjusting affected pointers only in the context of the tree-like data structures used in many networking lookup tasks.

Unfortunately, we show that the standard pattern of access to memory will force any scheme (that reads complete trie nodes) to have a utilization of at most 50% or to take twice the lookup time. This is because if a trie node straddles two memory words, the node lookup will require two memory accesses. By looking "under the covers" of an SRAM memory design, we show that a limited number of shifts can easily be designed (in a custom memory design for a custom chip) with a very small percentage increase in the number of column multiplexors. We show that adding two shifts with an increased memory width can yield 100% memory utilization. We also briefly comment on the interaction of pipelining with memory allocation.

The rest of this paper is organized as follows. In Section 2 we describe some background on IP Lookups, a model of a lookup chip and a sample IP algorithm. In Section 3 we describe previous work in IP lookups and memory allocation. In Section 4 we describe some common infrastructure. In Section 5 we describe our simplest memory allocation schemes based on Frame Compaction (LFC). In Section 6 we describe our second (and preferred) allocation scheme based on Segment Hole Compaction (SHC). We describe experiments to compare our IP lookup-specific allocation schemes to a benchmark best-fit allocator in Section 7. We return to IP lookups in Section 8, where we describe the performance of a lookup chip using our allocators, describe the need for a new form of shifted memory, and outline some of the problems with pipelining. We conclude in Section 9.

2. IP LOOKUPS

To motivate our discussion of issues that arise with doing IP lookups we describe key requirements and constraints (Section 2.1), describe a model of a lookup chip (Section 2.2), and then describe a sample IP lookup scheme (Section 2.3).

2.1 Requirements and Constraints

There are three three key requirements for lookup schemes, size, speed and dynamism.

(Size) Internet backbone routers have large databases (e.g., around 50,000 prefixes [Mer] today and increasing rapidly). After incorporating multicast addresses, multiple hops, host routes and growth, it is not unreasonable for a backbone router [J] to aim to support 150,000 to 250,000 prefixes.

Speed: Vendors are designing products for OC-12 rates (2.5 Gbps) and are looking ahead to OC-768 rates (close to 40 Gbps). Studies [TMW97] show that 50 percent of backbone traffic consists of 40 byte TCP acknowledgements. At OC-192 rates, a 40 byte packet must be processed in 32 nsec to provide wire-speed forwarding. While memory is considered "cheap", this applies only to off-chip DRAM (dynamic RAM) memory and its variants that have large densities but access times of around 60-100 nsec. While a 4 memory access lookup can be pipelined across 4 RAMBUS [IBM97] banks to provide a lookup every 60 nsec, external DRAM technologies cannot provide lookup times under 60 nsec for a single lookup.

To handle 30 nsec lookups, implementors must use either off-chip SRAM (Static RAM, 10-20 nsec), on-chip SRAM (1-5 nsec), or on-chip DRAM (10 nsec). Off-chip SRAM requires extra address and data pins especially for a pipelined lookup; if the lookup is pipelined using M stages, the lookup chip will require M sets of address pins (18-32 bits each) and M sets of data pins (typically greater than 32 bits each); this quickly becomes infeasible for large M. External SRAM is also around 10 times the price of external DRAM; thus reducing the amount of off-chip SRAM can result in overall cost savings. Embedded DRAM may not be fast enough for the highest speed applications. On the other hand, vendors like Texas Instruments and IBM offer 16 Mbits of on-chip SRAM today using half a (fairly large) die on a custom chip. ASICs offer even less on-chip SRAM. Thus we believe high-speed memory, whether on or off-chip, is worth managing efficiently. For our design center in the rest of the paper, we will use on-chip SRAM of 16 Mbits though our techniques apply to any situation where memory must be used efficiently.

Dynamism: IP prefixes are updated by the Border Gateway Protocol(BGP) [RL95], While update times can be two orders of magnitude slower than lookup times, there is still evidence that update times should be reasonably fast. For example, [LMJ97] reports unstable BGP implementations that require updates in the order of milliseconds. Multicast protocols like DVMRP and PIM may also add multicast routes at high rates. Given the use of damping timers to ameliorate BGP instabilities, a more compelling reason to have fast incremental updates is to *better utilize high speed memory*; a lookup scheme that requires the lookup structure to be completely rebuilt on an update will typically require two copies of the database. One copy is used by Search and one is used by Update; thus the memory utilization drops to no more than 50%.

While this paper concentrates on IP lookups we note that there are many other forms of lookup that have similar problems and for which the techniques in this paper could be useful. These include exact matching (e.g., for bridges, ARP caches, flow ID lookups) and packet classification using filters. All these applications can potentially have large, dynamically changing databases that need to be looked up at high speeds.

2.2 Lookup Chip Model

Based on the arguments above, Figure 1 describes a model of a lookup chip that does search and update. The chip has a Search and an Update process, both of which access a common SRAM memory that is either on or off-chip (or both). The Update Process allows incremental updates and (potentially) does a memory allocation/deallocation and a small amount of local compaction for every update. The actual updates can be done either completely on chip or partially in software (in which case the Update Process on chip is simpler or non-existent.) We assume that each access to SRAM is fairly wide, say 1000 bits, as is feasible today using a wide bus. We also assume that the search and update logic can process a large number of bits (say 500) in parallel in one memory cycle time.

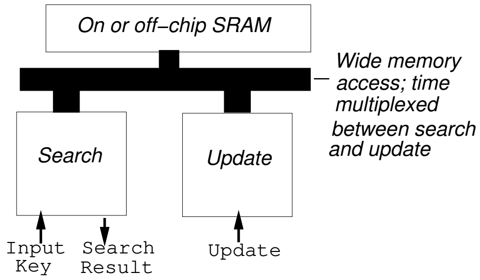

Figure 1: Model of a lookup chip that does search in hardware using a common SRAM memory that could be on or off-chip.

We assume that Search and Update share access to the common SRAM (that stores the lookup database) using time multiplexing. Thus the Search process is allowed S consecutive accesses to memory and then the Update Process is allowed K accesses to memory. If S is say 20 and K is say 1, this allows Update to periodically steal a cycle from Search while slowing down Search throughput by only a small fraction, and yet allow atomic updates.

The Chip has pins to receive inputs for Search (e.g., keys) and Update (e.g., update type, key, result), and can return search outputs (e.g., result). The model can be instantiated for various types of lookups including IP lookups (e.g,, 32 bit IP addresses as keys and Next Hops as results), bridge lookups (48 bit MAC addresses as keys and output ports as results), or filter matching [LS98] (e.g., packet headers as keys and matching filters as results).

We assume that each addition or deletion of a key can result in a call to deallocate a block, and to allocate a different size block. A memory allocator is a program that manages a fixed area of physical memory[4], and handle a stream of allocates and deallocates. We assume that each allocate request can be in any range from 1 to W memory words, that there is a total of M words that can be allocated. The goal of an allocator is to satisfy as many allocate requests as possible. We will evaluate allocators from a worst-case and average-case viewpoint. Let L denote the sum of the

[4]In software systems, the allocator often manages virtual memory; adding a level of indirection through a page table is too expensive for our purposes as it can slow down search by a factor of two. Thus our allocators manage physical memory within the SRAM.

allocated blocks in memory. We wish to know how high L can be with respect to M (ideally they should be the same) in the worst case, and in typical cases.

As a design center, we will assume that M is 16 Mbits. We will also assume that W is quite small (no more than 512 but more typically around 32 words) in order to allow the hardware to retrieve a complete block in one memory access.

2.3 Sample IP Lookup Algorithm

Backbone routers use something akin to telephone area codes to reduce forwarding table size; these "area codes" are known as prefixes. Prefixes are sequences of up to 32 bits. As an example, consider the IP lookup database consisting of the following 9 prefixes: P1 = 101*, P2 = 111*, P3 = 11001*, P5 = 0*, P6 = 1000*, P7 = 100000*, P8 = 100*, and P9 = 110*. A prefix like 100* matches all IP addresses that start with 100. An address that starts with 100000 matches P7, P6, P8, and P0 but P7 is the longest match. The longest matching prefix problem is to return the longest matching prefix of a 32 bit IP address.

Any IP lookup scheme that allows incremental updates and has good worst case guarantees on memory suffices for our purpose. Thus our ideas can be instantiated using Lulea compressed tries [DBCP97] or variable stride tries [SV98]. However, the Lulea scheme does not have a fast incremental update scheme and variable stride tries do not have deterministic bounds on memory utilization. While it is possible to modify these existing schemes to make them usable for our purposes, we prefer to illustrate our ideas using a scheme closely related to a scheme due to Perlman and described in her book [Per99]. Perlman's scheme does have worst case memory guarantees and fast incremental updates.

To understand our version of Perlman's scheme, we start with an expanded multibit trie [Per99, NK98, SV98]. Figure 2 shows a multibit trie for our example database using a trie that examines an address 3 bits at a time. Each array has 8 locations, each of which can contain a pointer to another trie node and a stored prefix. Thus the 100 entry in the root node points to all prefixes that start with 100, and also stores the prefix P8 = 100*. All prefixes are expanded [SV98] to lengths that are multiples of 3. Thus P5 = 0* expands to four 3 bit prefixes 000*, 001*, 010*, and 011*, and is stored in the first four locations of the root array.

We use a compressed trie data structure that is similar to Perlman's [Per99] but which is very different from the bitmap compressed structure used in the Lulea scheme [DBCP97]. The compressed trie version of the same database (Figure 3) replaces any expanded prefixes by a single prefix, and separates out pointer entries in the bottom portion of the node. The top portion of each node stores prefixes (e.g., P5 in the root) together with the bits (e.g., 0*) that extend the bit path to this node to form the prefix. Prefixes like P8 = 100* that also have an associated pointer are not stored in the root node, but are pushed down to the child node. Thus P8 is stored in the rightmost child node with only the bits *. There is no need for further bits to identify P8 = 100* because the path to the rightmost node uses the bits 100. The bottom portion of each node (e.g., the root has two pointers) contains pointers to other trie nodes, together

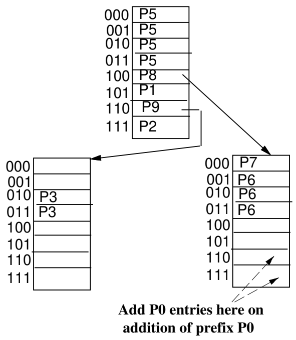

Figure 2: Expanded trie version of the example IP database. If we add prefix P0, the last two entries of the rightmost trie node must be changed because P0 expands into these entries.

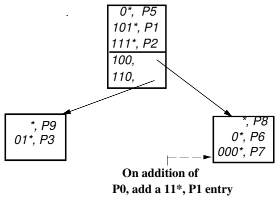

Figure 3: Compressed trie version of the example IP database compresses each trie node in Figure 2 by replacing all expanded prefixes within a node with a single prefix entry, and treating pointers separately.

with the bits that would have identified the location of the pointer in the original expanded node.

Figure 3 has less memory (10 words) than Figure 2 (27 words), but appears hard to search. When indexing into a compressed trie node N with a chunk C of bits, we first find the best matching prefix of C in the set of stored prefixes within N. We also look for any pointer associated with the bits of C in the bottom portion of the compressed trie node. These correspond to accessing a stored prefix and following a pointer in the corresponding expanded node.

However, these extra operations are not a problem in a hardware implementation that can read the entire compressed node within a single (wide) memory access. The needed operations can be performed using simple combinational logic. This clearly limits the size of each trie node access to be less than 8 bits. Doing trie search 4 bits at a time appears to be slower than the Lulea [DBCP97] scheme that searches in strides of 16, 8 and 8. However, our scheme requires only one memory access per node as opposed to two or three per node in Lulea. Thus the speeds are competitive. Both schemes can be pipelined for more speed.

More importantly, unlike the Lulea scheme it is easy to do incremental updates on compressed tries. For example, if we add the prefix P0 = 10011* to the example database of Figure 2, this will require modifying the last two locations of the rightmost trie node. Correspondingly, in Figure 3, we only add a single entry (11*, P0) to the rightmost compressed trie node. This requires incremental allocation to deallocate the old trie node (of size 3) and allocate a new node (of size 4).

Equally importantly, unlike the Lulea scheme, the worst case storage for N prefixes in a compressed trie (after one-way branches have been replaced by text strings) can be shown to be: $2N$ pointers of say 20 bits each (16 bit pointer plus 4 bits to identify which pointer), $2N$ text strings of 32 bits each, and N next hop pointers (20 bits each). This works out to a worst-case total of 124 bits per prefix. A careful look at the standard IP databases [Mer] however shows that 90% of the text strings are 4 bits or less; any (rare) text strings longer than this can be handled using extra nodes. Similarly the next hop pointer can often be relegated[5] to a single off-chip SRAM access whose address can be computed from the trie node in which the search terminates. This reduces the total to around 50 bits per prefix in on-chip SRAM.

Thus, with perfect memory allocation, a compressed trie can store 250,000 prefixes using $250000 * 50$ which is 12.5 Mbits. Thus storing 12.5 Mbits at 85% storage efficiency requires around 16 Mbits of on-chip memory which is just feasible today. However if storage was much more fragmented (say 20%), then the chip would only be able to handle only 25,000 prefixes. The bottom line is that *since on-chip/off-chip SRAM is limited, it is crucial to use a fast and efficient allocator to minimize total worst-case storage.*

3. PREVIOUS WORK

IP Lookups: Existing fast IP lookup schemes are either based on multibit tries [DBCP97, NK98, GKM98, SV98] or on binary search of hash tables [WVTP97]. Hashing schemes are used by some vendors at Gigabit rates but provide non-deterministic search times and require larger storage [WVTP97] than multibit trie solutions. This makes multibit trie solutions more attractive for schemes that require limited SRAM. CAM solutions are flourishing, especially for edge routers, with some vendors even announcing CAMs with 128,000 prefixes. However, most major backbone vendors (e.g., Cisco, Bay, Ascend, Juniper) we know of use special purpose hardware that implements some algorithmic solution (mostly based on tries) perhaps because they plan for even larger databases and because they prefer to integrate the entire forwarding algorithm into the chip that does lookups.

Caching 32 bit IP addresses has been traditionally considered to have poor hit rates [Par96] but recent interesting work [CP99] has shown the possibility of better hit rates. Despite this, the lack of clear tests on a large number of backbone traffic traces, the lack of determinism, vendor and ISP perception, and the need to receive and process packets at wire speeds while guaranteeing QoS [LS98], have made caching solutions unpopular so far in the backbone. We now discuss previous work in memory allocation.

Schemes that do not Use Compaction: The simplest allocator [SG97] maintains a linked list of allocated blocks and holes. An allocate request is satisfied by scanning the hole list for either the first hole that fits (*first fit*), the smallest hole that fits (*best fit*), or the first hole after the last allocated block that fits (*next fit*). On the other hand, the *buddy system* [Knu73] maintains holes using separate lists for hole sizes that are powers of two. For our purposes, the major difficulty with such schemes is their poor worst-case fragmentation properties. Figure 4 shows an example. First we allocate all of memory using size 1 requests. Next, we deallocate the last $W-1$ words starting at each location i such that $i \bmod W = 0$. This leaves the picture shown in Figure 4 where memory is now fragmented into allocated blocks of size 1, interleaved with holes of size $W-1$. If all future requests are of size W, the memory is unusable. However, only $\frac{1}{W}$ of memory is utilized. Note that compaction, which could easily cure the problem, is not allowed.

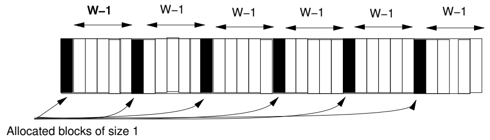

Figure 4: The worst case example of memory fragmentation resulting in $\frac{1}{W}$ memory utilization

The BSD 4.2 UNIX allocator reduces worst-case fragmentation using *segregated* free lists for sizes that are in powers of two but (unlike buddy systems) does not allow coalescing. This improves worst-case utilization ratio to $\frac{1}{2log_2W}$ but has a poor average utilization. Robson [Rob71, Rob77, Rob74] proved that *no allocator* that does not use compaction can have a utilization ratio better than $\frac{c}{log_2W}$, where c is between 1.25 and 1. Thus, among allocators that do not compact, the UNIX allocator is near optimal. Note that virtual memory (the classical solution to memory fragmentation in real computers) cannot help in our application because the extra level of indirection slows down search by a factor of two; caching virtual to physical translations cannot avoid this overhead because the "main" memory on chip is what CPUs use for fast cache memory!

Schemes that use Compaction: Even if $W = 8$, conventional allocators have an unacceptable worst-case utilization ratio of $\frac{1}{3}$. Thus we are forced to examine compaction. In the literature [Wil92], memory compaction is mostly associated with garbage collection. There are two major problems with standard real-time copying collectors [Bak78] for our application. First, garbage collectors solve the pointer adjustment problem by leaving a *forwarding pointer* at a relocated block B that points to the new location of B. The use of forwarding pointers can slow down search by a factor of two (each memory access may go through a forwarding pointer) in the worst-case.

Even if we adapted copying collectors using our solutions to pointer adjustment (Section 4), a second problem is that the utilization of copying collectors can only approach fifty percent. This is because memory is divided into two halves (*semi-spaces*) where each space must be large enough to contain the entire data structure [Bak78]. There are also generational garbage collectors [LH83, Wil92] that use multiple spaces, but these only help reduce the average compaction work without improving worst-case fragmentation.

4. COMMON INFRASTRUCTURE

Since compressed lookup structures require allocation and deallocation of variable amounts of storage, we describe fast memory allocation algorithms that provides guarantees on worst case memory usage. To do so, we first describe some common infrastructure used by all our allocators.

[5] While most previous work in IP lookups assumes a small 20 bit value for next hop, the actual next hop information used in real routers is much larger because of the need to handle load splitting and adjacencies on LANs. Thus it seems best to leave the final next hop lookup to external SRAM.

Locating Holes: First, to identify holes and block boundaries, we use two tag bits per memory word. In our applications, a memory word is > 24 bits; thus tag overhead is $< 8\%$. 00 denotes a free word, 01 denotes a word that is allocated and is the start of a block, 10 denotes a word that is allocated and is the end of a block.

To handle an allocate request of size i, our allocator will search for the smallest size hole of size $\geq i$. (For $W \leq 256$, this can be handled in 1 clock cycle by simple combinational logic that operates on a W size bit map, where W is the max request size.) To do so, for each $i, 2 \leq i \leq W$, we link every hole of size i into a doubly linked list. For $i < W$, list i contains only holes of exactly size i, but list W contains all holes of size *greater than or equal to* W. Free list pointers for each i are placed in the storage within a hole and have no extra overhead. The found hole is removed from the head of its queue and the remainder is added back to the appropriate queue. On a deallocate of a block B, the appropriate bits of the words of B are cleared and the hole is coalesced with any existing holes on either side in $2W$ time.

Pointer Adjustment: To improve memory utilization, our schemes compact memory by moving an allocated block B to a new location that starts at say P. Thus all memory locations in the application data structure that point to B (which we will call the *parents* of B) must be adjusted to point to P before search can proceed. The main problem is to efficiently locate parents without searching all of memory.

Fortunately, many networking data structures (e.g., all forms of tries, binary search trees, hash tables) have the simple property that *each block has at most one parent.* The simplest way to locate parents is to add a *parent* field to the start of each block that points to the (single) parent pointer location. This can be updated when the parent changes. If B is moved to B', the parent location is adjusted to point to B'. All children of B (at most W of them) also must change their parent fields to B'. This adds no storage overhead if the parent pointers are kept in a software copy of the data structure and not in on-chip memory. The *parent* field can be entirely avoided (even in the software copy) if block B stores a unique key that can be searched for to find the parent of B. Since search takes only a few memory accesses this will not slow down compaction appreciably.

But there are also networking applications such as filter search (e.g., [SVSW98]) where each node can have multiple parents. To allow this, we can allow a block to contain multiple (say up to P) parent pointers. To allow each block to contain exactly as many parent pointers as it needs, the unused tag bit combinations are used to denote a "parent pointer". Thus the first words of a block can have at most P parent pointers. Next, we increase the maximum allocation request from W to $W + P$. Increasing the maximum allocation size only increases the amount of compaction additively. The overhead for any size P is at most 50%. This is because if the original memory required for the data structure was say L, there can be at most L parents. Thus there can be a total of L parent pointers, leading to a total of at most $2L$ memory, a 50% overhead.

5. FRAME BASED ALLOCATORS

In this section, we describe the first of our two new memory allocator designs. We emphasize the following differences from standard memory allocators.

i) Compaction possible: To avoid the Robson bound, our allocators do local compaction. We can do this unlike say Unix's Malloc because we have specific applications that use only a finite number of parent pointers.

ii) No garbage collection: Our networking applications explicitly do deallocates. Our allocators compact only to gain worst-case utilization, not to identify garbage.

iii) Finite time: Since our major application is search, we cannot lock out search while compacting all of memory. Similarly, we cannot use forwarding pointers because they can slow search by a factor of two. We also prefer not to waste half the memory as a temporary area to build a compacted database while search works on the other half. Thus our allocators can only afford a small amount of compaction after each update.

iv) Local compaction: Our allocators only compact in the neighborhood of the last update as opposed to using a global compaction sweep. More precisely, a simple measure of locality is the size of the neighborhood (which must include the allocated or deallocated node) which a compaction algorithm reads or writes. If this size is only a constant factor larger than the maximum node size, then we say the algorithm does *local compaction.* By contrast, real-time garbage collectors that use global sweeps are not local and only approach 50% utilization. Generational collectors are less global but not as local as ours, and do not help worst case utilization.

We now describe a simple frame based allocator in this section and describe a more sophisticated allocator in the next section.

5.1 Frame Based Schemes and Lazy Frame Compaction

To show how simple local compaction schemes can be, we first describe an extremely simple scheme that does minimal compaction and yet achieves 50% worst-case memory utilization. We then extend this to what we call *Lazy Frame Compaction (LFC)* whose utilization can be tuned to approach 100%.

In Frame Merging, we divide all M words of memory into $\frac{M}{W}$ *frames*[6] of size W. Frame Merging seeks to keep the memory utilization at least 50%. Thus all utilized frames should be at least 50% full. If $\frac{M}{W}$ is large, we can relax this a little (and still achieve close to 50% utilization) if *we allow one flawed frame that is non-empty but less than 50% full.* In summary, Frame Merging maintains the following simple invariant: all but one unfilled frame is at least 50% full. If so, and $\frac{M}{W}$ is much larger than 1, this will yield a guaranteed utilization of almost 50%.

[6]Assume M divides W; if not, any remaining memory of size at most W can be ignored if M is much greater than W

To maintain the invariant, Frame Merging keeps one additional pointer to keep track of the current flawed (i.e., non-empty but less than 50 percent utilized) frame, if any. Allocate and Deallocate requests are handled using the infrastructure to find holes described in Section 4. The only restriction we add is that all holes must be contained within a frame; we do not allow holes to span frames. Now, an Allocate could cause a previously empty frame to become flawed if the allocation is less than $\frac{W}{2}$. Similarly, a deallocate could cause a frame that was filled more than 50% to become less than 50% full. For example, a frame that contains two blocks of size 1 and $W-1$ could have a utilization of $\frac{1}{W}$ if the block of $W-1$ is deallocated. This would cause two frames to become flawed, which would violate the invariant.

The simple trick to maintain the invariant is as follows. Assume there is already a flawed frame F and a new flawed frame F' appears on the scene. Then we merge the contents of F and F' into F, leaving F' empty, maintaining the invariant. This is clearly possible because both frames F and F' were less than half full, and thus can be merged into a single frame by compacting the allocated blocks within F and moving the allocated blocks within F' to the end of the remaining free space in F. An example of merging is depicted in Figure 5.

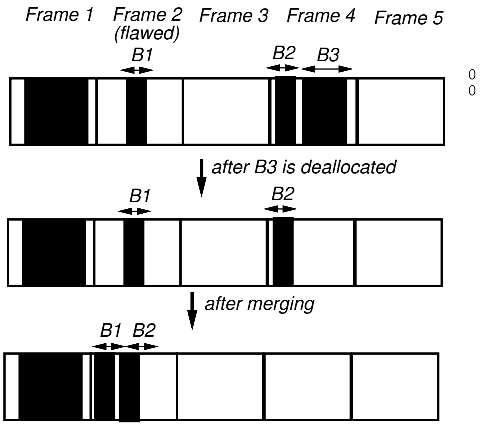

Figure 5: Example of merging. The top picture shows a picture of memory divided into five frames before block B3 is deallocated. Only Frame 2 is flawed. The middle picture shows memory after B3 has been deallocated causing Frame 4 to also become flawed. The bottom picture shows the situation after the contents of Frame 4 (i.e., B2) are merged into Frame 2, leaving only Frame 2 flawed.

Frame Merging is reminiscent of the merging of B-tree nodes used in B-trees [CLR90] except that Frame Merging abstracts the merging technique for use in a general memory allocation context. The worst-case utilization of Frame Merging can be improved as follows. We increase the frame size to kW, and require that at most one frame has utilization less than $\frac{k}{k+1}$, where k is a *compaction parameter* that can be used to tune the algorithm. Increasing k improves utilization but linearly increases the compaction work.

However, an even simpler scheme is as follows: *only compact within frames.* If after compaction, a frame is still useless to satisfy further allocate requests, then the frame must have a hole of at most $W-1$. Thus if an allocate request cannot be handled, every frame has a hole of size at most $W-1$. Since each frame is of size kW, this leads to a worst-case utilization ratio of at least $\frac{k-1}{k}$. We can also be lazy about compacting until an allocate request cannot be satisfied, by placing any frame that has more than W total free memory words (but less than W contiguous free space) in a *collapsible* frame list. We call this scheme Lazy Frame Compaction (LFC).

6. SEGMENT-HOLE COMPACTION

Lazy Frame Compaction (LFC) does not compact across frames. If there are two adjacent frames with $W-1$ free space in each frame, LFC will give up on both frames as neither is collapsible. However, a hole of size $\geq W$ could be produced if a scheme were to compact across frames. Thus, while LFC guarantees a good worst-case utilization, we might suspect that its average memory utilization may only be slightly better than its worst-case utilization. Do we care about the average memory utilization? If we recall the motivation and model of Section 2.2, we would like our lookup chip to guarantee a large number of keys in the worst case. However, it would be nice to also be able to handle an even larger number of keys in the "typical case".

Let us call each maximal sequence of contiguously allocated blocks a *segment.* In other words a segment consists of a number of blocks packed together without any intervening holes, and which cannot be extended to form a larger segment. Intuitively, for any value of parameter k, we would like any segment to be at least kW in size (this would rule out examples like Figure 4). We can relax this condition a little bit and require the following Either-or invariant: *either* each segment is at least kW in size *or* the hole immediately following the segment is at least W in size. The only exception we make is for the last segment that may not have a following hole.

The Either-Or invariant still guarantees a good worst-case utilization of $\frac{k}{k+1}$. Suppose an allocate request cannot be satisfied. Thus all holes must be of size $W-1$ or less. But in that case, all blocks preceding these holes (with the pesky exception of a hole at the start of memory) must be of size at least kW, where k is once again a compaction parameter. Thus memory is a patchwork of segments of size $\geq kW$ interleaved with holes of size $\leq W-1$. Thus the utilization is very nearly $\frac{k}{k+1}$.

Figure 6 shows a simple example of a state of memory satisfying this invariant. The first hole is of arbitrary size; the first segment is of size $> kW$ and the second hole is of size $> W$ (although this is not needed, it is not prohibited). The second segment is of size $> kW$ which allows the third hole to be of size $< W$. The third segment is of size $< kW$ but the fourth hole is fortunately $> W$ ($\geq W$ would have sufficed). The figure also shows that we do not place any restriction on the last segment. If M is much larger than kW, the arbitrary size of the first hole and the last block contribute edge effects that only marginally affect worst-case utilization.

Allocates can affect the invariant by allocating within a hole that was $\geq W$ and making the hole $< W$ while its preceding segment is $< kW$. Deallocates can affect the invariant by

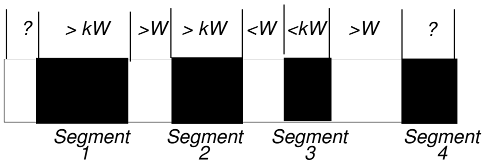

Figure 6: Segments and holes satisfying the Either-or Invariant. Note that a segment can contain several blocks all packed together without any holes in between. The invariant ensures that either a segment is "large" or is followed by a large enough hole.

deallocating within a segment that was $\geq kW$, splitting the segment into pieces that do not satisfy the invariant. While the exact algorithm to maintain the invariant has a number of cases, the essential intuition can be had by looking at the example in Figure 7.

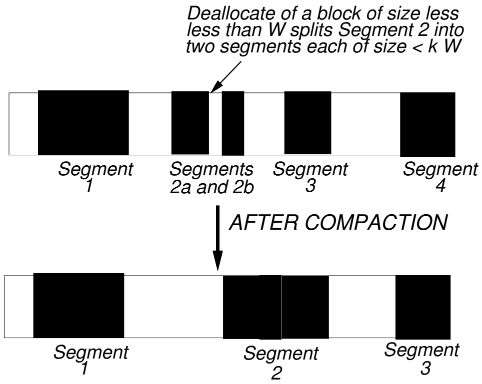

Figure 7: Maintaining the Either-or invariant. The top picture shows the memory of Figure 6 after a deallocate in the middle of Segment 2 splits segment 2 into two segments we call 2a and 2b such that neither satisfies the invariant. The invariant is restored by sliding both segments 2a and 2b in front of the old Segment 3. This results in only three total Segments that are renumbered as shown in the bottom picture.

The top of Figure 7 represents the same example as in Figure 6 after a deallocate has been done in the middle of Segment 2. This splits Segment 2 into two small segments, say Segment 2a and 2b that are both of size $< kW$. Segment 2a could now violate the invariant if the hole between 2a and 2b is $< W$. But Segment 2b can also violate the invariant because the hole following the old Segment 2 (see Figure 6) was $< W$. The easiest way to restore the invariant is to slide the concatenation of the two "flawed" segments 2a and 2b to the end of the hole before the old Segment 3. This results in the picture shown on the bottom of the figure. Since there are now only three segments we have renumbered the segments.

Thus the base intuition is as follows: when a segment becomes "flawed", we simply slide the segment down to merge with the next segment to the right. It should now be clear why we do not require the last segment to be $\geq kW$.

Finding whether a segment is flawed requires an examination of only kW tag bits (to check if the segment is of size $< kW$) and a further W bits (to check if the following hole is of size $< W$). The actual sliding process costs at most kW. It is also easy to see that deallocates can cause at most two segments to become flawed, and allocates can cause at most one segment to become flawed. The worst case work is $(2k+1) \cdot W$.

7. EXPERIMENTAL RESULTS

We already know the worst-case performance of our allocators. However, the theoretical analysis does not provide answers to the following questions.

Q1. What is the average case performance of the two allocators? How much compaction do they do on average, and (more importantly) what is their average memory utilization. If the average memory utilization is 100% and worst-case utilization is (say) 85%, that implies a 15% increase in the number of prefixes (keys) that can be handled by the lookup chip. Also, what advantages does the slightly more complex Segment-Hole compaction have over Frame Based compaction? What value of the compaction parameter k should we choose?

Q2. How much better do our allocators do than the best standard allocators on an actual application? A recent paper [J97] shows that good allocators (e.g., address-ordered best-fit) do quite well on actual memory traces. However, these studies apply only to standard benchmark processor applications and not to router lookup schemes. Although our schemes do offer a large worst-case improvement, it would be nice to also know whether they improve average performance compared to standard allocators.

To answer these questions, we implemented the Lazy Frame Compactor (LFC) and the Segment-Hole Compactor (SHC) and tested their average performance using our sample IP lookup algorithm and BGP traces (which cause prefix deletions and additions which in turn lead to allocates and deallocates). We also implemented a benchmark best-fit allocator that never compacts memory. To go beyond IP lookups, we also implemented a simple hashing lookup application that uses our allocators and could be used in (say) an ARP cache. We now provide more details.

Benchmark Allocator: The benchmark allocator we implemented uses exactly the same infrastructure described in Section 4 to keep track of holes (a list for all sizes $< W$ and a single list for holes of size $> W$) and uses tag bits in order to keep track of allocated words in memory. Given an allocate request of size n, as in Segment-Hole compaction, the benchmark allocator finds the smallest size hole that satisfies the request. It deletes the first hole on this list, allocates n memory words from this hole, and returns the leftover hole to the appropriate list. Given a deallocate request, the allocator resets the tag bits for these memory words. It then coalesces the newly created hole with any adjacent holes, and inserts the resulting hole into the appropriate list. Unlike our other allocators, the benchmark allocator *never compacts memory.* Thus it represents the class of *best-fit* allocators.

Best-fit allocators have been known to offer good memory usage and cause the least fragmentation among conventional allocators [J97] and thus provide a good point of comparison. Actually address-ordered best-fit does even better [J97] but address ordering would require sorting the hole lists which would slow down allocates and deallocates (and hence inserts and deletes) considerably. Recall that the worst-case for all conventional allocators, including this benchmark allocator, does not meet our performance requirements. However, it is useful to compare average performance.

Lookup Data: For both the IP lookup and hashing applications, we used data from the Mae-East database [Mer] on 1/20/99 at 10:19:10 hrs. The database had 42732 IP prefixes. For the IP lookup application, we used a BGP trace obtained used the Route Tracker tool [Mer2]. Since the tool does not let the user save updates to a file we used the system call trace facility `ktrace` on a NetBSD machine. The resulting trace had duplicate insertions and deletions presumably due to repeated BGP updates or due to system call behavior. We eliminated these duplicate prefix updates by not allowing a prefix already present to be inserted again, and not allowing an absent prefix to be deleted from the data structure. We extracted this data for 05/20/99, 00:00:00 hrs to 08:30:00 hrs.

Experimental Metrics: Average performance includes two metrics, memory utilization and compaction. First, consider average memory utilization. Computing the average memory needed is not trivial because (especially for the trace experiments) the size grows and falls as prefixes are deleted and added. We wish to find the minimum amount of memory M needed to satisfy a run of an allocator on the trace. We know that M must be larger than the peak size of the actual data structure (say M_l). However, it could be larger because the allocators are not perfect. However, it cannot be larger than that caused by worst case memory utilization (say M_h). We then do binary search between M_l and M_h by doing repeated runs of the same trace until we find the smallest value M of memory between M_l and M_h such that all allocates are satisfied. We compute the average memory utilization as M_l/M (because the allocator required M words of memory when a perfect allocator with 100% utilization would require only M_l words).

Although it is of secondary importance, we also measure compaction. If compaction is done by a software process other than the lookup chip, decreasing compaction work allows the route processor to have more time for more directly useful functions such as fast route updates. We measure compaction work by the number of words of memory written or read during any update operation. We measure the *average compaction*, the total compaction work divided by the number of updates processed.

Hash Application: Besides IP lookup, we used a hash lookup application that provides deterministic hashing. If the maximum number of keys that collide in a hash bucket is W, a W size block containing all the keys that hash into a bucket can be retrieved in a single memory access. For example, if there at most 4 collisions (Figure 8), we can allocate the four keys in a single size 4 block. The naive method would require allocating a size 4 block for each non-empty bucket. Clearly, this can be avoided using dynamic memory allocation. If only a few buckets have 4 collisions and the other buckets have only 1 collision, the naive scheme will require 4 times the amount of storage.

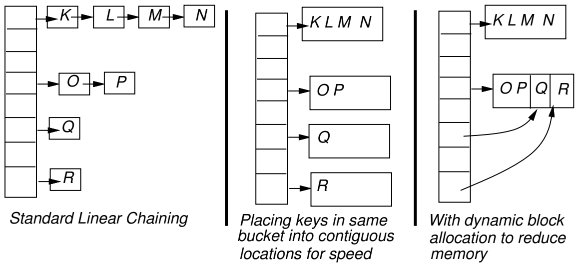

Figure 8: The figure shows standard hashing with linear chaining (left). To improve speed in terms of memory access, multiple entries that hash together in a bucket can be placed in a contiguous block of memory (middle). Dynamic allocation can be used to avoid allocating every block to be the size required to handle the worst case number of collisions (right).

7.1 Results

For lack of space we only provide a few sample curves and the highlights of our experimental results. To test compressed tries, we built two tries. The first is a 8/8/8/8 trie, and the second is a 12/4/4/4/4/4 trie. The 8/8/8/8 trie traverses an IP address 8 bits at a time. The second trie traverses an IP address 12 bits at the first step, and 4 bits at each subsequent step. For the first set of IP lookup experiments, we used a snapshot of the Mae-East database. The database had 42732 prefixes. Building the 8/8/8/8 trie required 74226 allocates and 42731 deallocates. Building the 12/4/4/4/4/4 trie required 87695 allocates and 42731 deallocates.

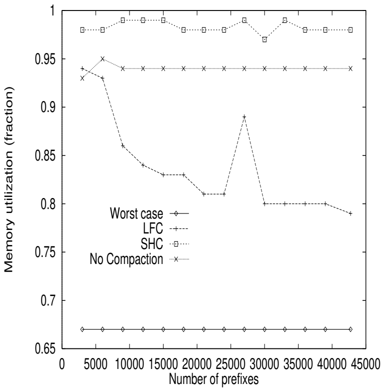

Figure 9: Mae-East: Memory profile of the 8/8/8/8 trie, $k = 2$

Figure 9 and Figure 10 show the memory utilization[7] for Mae-East with the two tries. To plot multiple points, we calculate the memory utilization after inserting the first 2000 prefixes, the next 2000 prefixes etc., until all 42732 prefixes are inserted. We plot the memory utilization for all three

[7]Recall this is the ratio of the minimum possible memory required to hold the data structure (ignoring the allocator) at any point to the actual minimum memory (taking into account the allocator). The latter memory is calculated by binary search to find the smallest memory size at which no allocate requests fails.

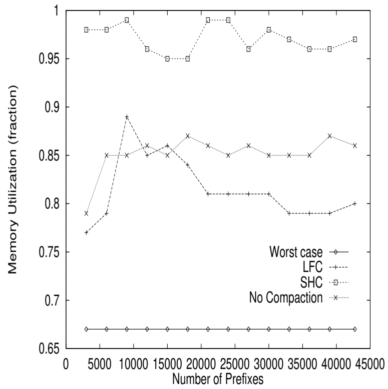

Figure 10: Mae-East: Memory profile of the 12/4/4/4/4/4/ trie for $k = 2$

allocators: Lazy Frame Compactor (LFC), Segment Hole Compactor (SHC) and the benchmark allocator. Both figures use the low compaction factor $k = 2$; thus the worst-case utilization for SHC is $k/k + 1 = 66\%$ which is also plotted for easy reference.

Notice from Figure 9 and Figure 10 that all allocators do significantly better than the worst case of 66%. The benchmark allocator has a low of just below 95% for the 8/8/8/8 trie but falls to below 80% for the 12/4/4/4/4/4 trie. LFC is almost consistently worse than both allocators (although it is significantly better than the worst case predicted by the compaction parameter k) falling to below 80% after the 8/8/8/8 trie is completely built. On the other hand, SHC gets consistently over 95%. It is easy to increase the utilization of LFC by increasing the compaction parameter k to say 8. For $k = 8$, using the same database the utilization of the benchmark remains constant (we have omitted the graphs for lack of space) but LFC does better than the benchmark (over 90% for both tries). For $k = 8$, SHC does even better (over 98%).

In general, in all the experiments we performed SHC does consistently better than both the benchmark and LFC allocators, *achieving consistently over 95% even for low values of k.* It is interesting to also compare the compaction work done by LFC and SHC. For the 8/8/8/8 trie described above and for $k = 2$, the worst case compaction per update is $2kW$ which was equal to 988 words. However, the average number of words compacted per update by LFC was only 0.18 while the average for SHC was 25.10. Moving to $k = 8$, the worst case compaction goes up to 2964 words but the average number of words compacted per update by LFC was only 0.5 while the average for SHC goes up to 94. Thus the average number of words compacted by both SHC and LFC is significantly less than the worst case but LFC's compaction work is much less than SHC.

The Mae-East table building experiment is clearly a special case that can happen when say a router first boots up. Table building provides very stylized allocates and deallocates with the deallocates closely following the allocates in a regular pattern as prefix insertions cause nodes to expand. Thus we should also examine more arbitrary inserts and deletes that occur as routes go up and down in a backbone router.

Thus for the second set of experiments, we used a BGP trace (see description earlier) to construct each of the two tries. We only show results for the 12/4/4/4/4/4 trie. The memory utilization results are shown in Figure 11. In this figure, we plot the utilization at 5 points for every 1/5th of the trace. When the x-axis says number of prefixes, it means the number of prefixes processed so far (added or deleted) in the trace. The graph is similar to Figure 10 except that Figure 10 uses a static database while Figure 11 uses a dynamic database. Note also that in this dynamic graph the total memory used (not shown) grows and shrinks as prefixes are added and deleted.

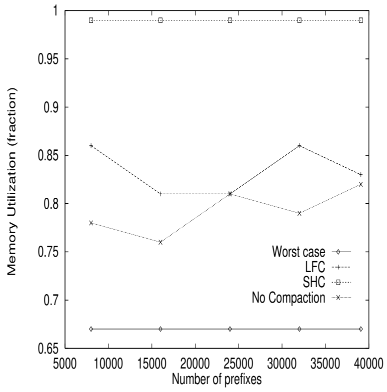

Figure 11: BGP trace: Memory profile of the 12/4/4/4/4/4/ trie for $k = 2$

The most interesting observation from Figure 11 is that it is fairly consistent with the other graphs, with SHC doing close to 100% while the benchmark allocator falls at some point to nearly 75%. Once again, LFC is sometimes better and sometimes worse than the benchmark, but all allocators are significantly better than the 66% worst case predicted by the compaction parameter. The results for compaction were also similar with LFC compacting 0.6 words per update on average, while SHC compacted 21 words per update on average for $k = 2$. Other experiments provided similar results.

So far we have only considered IP lookup applications; it is natural to ask whether our results would change for other applications. Thus we also tested the hash function application using IP addresses. For simplicity, we obtained these IP addresses from the Mae-East snapshot by padding prefixes with 0's. The hash function chosen uses an array size of 8192. The memory utilization is shown in Figure 12 for compaction parameter $k = 2$. The results are qualitatively similar, but the smaller variability in node sizes seems to make all the allocators behave very well

Based on these results we offer preliminary answers to the two questions we raised earlier.

A1. The average case memory utilization of SHC was always over 95 % for every experiment we conducted even for very low values of the compaction parameter. Thus for an IP lookup application, one can use a low value of k (say 4) that guarantees 80% utilization but for which we can expect an SHC allocator to allow 15% more prefixes (than the worst case would predict.) LFC on the other hand does worse

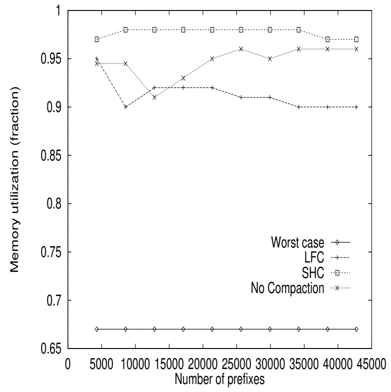

Figure 12: Mae-East: Memory profile of the 8192 array size hash function for $k = 2$

than the benchmark for low values of k. LFC does better for higher values of k but is always inferior to SHC. The average compaction work performed by SHC is an order of magnitude less than the worst case compaction work but is still worse than LFC. However, since the average compaction work is small compared to the work required anyway (by the route update process) to write the fields in the trie nodes, this seems unimportant. Thus, despite its slight increase in complexity we believe SHC is a better allocator. This is possibly because it can compact across frame boundaries and so produces fewer but larger holes. Only if compaction work were a major factor, would LFC (with a high value of k) be preferred to SHC.

A2. The benchmark allocator does very well on our experiments achieving over 75%. We believe our schemes are better than the benchmark allocators for three reasons. First, our allocators provide a guaranteed level of performance which is important when comparisons are made to technologies like CAMs that provide tight guarantees. We also only used a single BGP trace; we leave to future work the task of checking whether the benchmark can do worse on other traces. Second, our SHC allocator does better on average (over 95% even for low values of k) than the benchmark (as low as 75% in our trace study). Third, our allocators can be *tuned* by adjusting k if the observed utilization is found to be inadequate, unlike the benchmark.

8. BACK TO IP LOOKUPS

Since our original focus was IP lookups, we revisit the implications of the results of the last few sections for a lookup chip. Suppose the lookup chip uses 16 Mbits of on-chip SRAM with a 4 nsec access time (easily feasible today for a custom chip), then the 6 level 12/4/4/4/4/4 trie will require 6 memory accesses (of width less than 500 bits) which yields a 24 nsec lookup. This allows wire speed forwarding at OC-192 rates. Using the earlier number we computed of 50 bits per prefix for our sample IP lookup scheme, 250,000 prefixes requires 12.5 Mbits at minimum to just store the data structure.[8]

[8]We assume the parent pointers used for compaction are stored off-chip. For example, if update is done entirely in software, then the route processor could keep parent pointers in its copy of the trie.

Using a compaction parameter of $k = 5$ yields a worst-case memory utilization of $k/(k+1) = 83.3\%$. Assuming a 5% overhead for tag bits (recall we needed these in Section 4 for finding holes), we get an overall utilization of $83.333 * 0.95$ which is 79%. Since 79% of 16 Mbits is smaller than 12.5 we can expect to fit 250,000 prefixes in this memory. Our average case experiments for the same value of $k = 5$ using SHC show the actual memory utilization is actually closer to 98%. Thus we can expect to store 15% more prefixes in practice, which would allow 287,500 prefixes. This provides an extra cushion for unexpected growth, or allows the use of on-chip SRAM for other purposes such as filters or accounting.

For update times, assume the update process gets a memory access every 20 memory accesses, and each memory access is W bytes wide. Counting the worst case sequence including the time to locate parent nodes, we estimate 4520 memory accesses for an update. Incorporating the slow access for update and using a memory access time of 4 nsec we get a worst case update time of 3.164 msec. This seems more than adequate at present, and the average time is much better. However, two problems remain which we now discuss.

8.1 The Memory Access Problem

The reader may have noticed the following problem. Our sample IP lookup scheme requires a W word memory access, but our allocators do not guarantee to layout trie nodes within W bit word boundaries. For example, in Figure 13, we see a node that straddles two W bit boundaries. It is easy to show that any allocator that does not allow nodes to straddle W bit boundaries can guarantee only a little over 50% utilization (consider a series of allocate requests of size $W/2 + 1$). But for conventional memories, if a node straddles two W bit boundaries, the only way to access the entire node (needed for compressed tries) is to make two memory accesses. This would slow down speed by a factor of two!

The simplest way to avoid this problem is to use a scheme like [SV98] or a version of [DBCP97] (suitably modified) which also use variable size trie nodes but only require a 1 word memory access for lookup. However, there is also a simple trick that avoids this dilemma using a new memory design that allows shifted access. In Figure 13, for example, we show a shifted access that allows a W bit READ to start at any position $k \cdot W/2$ for any k, In other words, we can read from bit position 0 to W as usual but we can also read from $W/2$ to $3W/2$. Since all on-chip SRAM memory on a cutom design is generally custom designed, it is possible to design a new form of SRAM memory that allows shifted access.

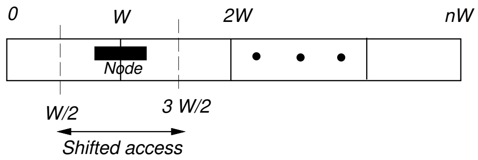

Figure 13: Shifted memory access allows a node that straddles W bit word boundaries to be read in one memory access at a small cost in column multiplexors.

First, most memory is internally designed in terms of large rows which are first decoded (to select the row) and then the appropriate bits within the row are chosen. Allowing nodes to straddle rows seems to be very hard, and is fortunately not needed because typical row sizes are larger than the kW needed for high utilization.[9] However, allowing a simple $W/2$ shift within a row is easy. This is because each bit in a row can only go to two output positions. For example, bit $W/2$ in Figure 13 can either go to output bit position $W/2$ (bit $W/2$ in normal access) or to output position 0 (first bit in shifted access). This only requires a small change in the column multiplexors; memory designers we have consulted estimate such extra column multiplexor wires will only add about 5% extra logic.

Finally, it is easy to see that using $2W$ size memory access (twice the size of a node) plus one shifted access can allow any node to be read out in one memory access. Using shifted access reduces memory density by say 5%; using twice the access width required only means that we must examine one less bit at each node. For example, if a 1000 bit access allowed 5 bit access at each node, we may have to settle for 4 bit access instead. This will increase overall lookup time by only 1 memory access for typical cases, and is a good tradeoff.

8.2 The Pipelining Problem

A 12-4-4-4-4-4-4 compressed trie can do an IP lookup in 6 memory accesses which is around 24 nsec using 4 nsec SRAM. To obtain a faster lookup that scales with memory speeds, we need to pipeline the trie to obtain a lookup every memory access (4 nsec). There are several new problems created by pipelining. We briefly describe one problem: the interaction of pipelining with memory allocation.

The simplest scheme is pipelining by height. Thus the pipeline would have S stages, where S is the tree height: each stage has some logic and some SRAM memory to store trie nodes. The root of the trie is assigned to stage 1, all children of the root are in stage 2; in general, nodes of height i are assigned to stage i. An address to be looked up first flows through stage 1 which passes a pointer to stage 2 along with the address; while the address is in Stage 2, a second address can enter Stage 1, creating a pipeline.

Unfortunately, each of the stage memories must be strictly partitioned so that each stage only accesses its own memory. This is because of the difficulty in building large multiport memories[10]. It appears that we must *statically* divide the available on-chip memory (say 16 Mbits) among the S stages.

Unfortunately, with height pipelining, the amount of memory allocated to a stage can vary drastically depending on the keys entered into the trie. This is because the trie is not a balanced tree whose memory needs at each height are predictable. With the exception of Stage 1 which contains the root, for any i, one can find a set of prefixes where most of the trie nodes are at height i. In other words, suppose the non-pipelined trie requires M memory in the worst case; then for every $i \neq 1$, there is some set of prefixes where Stage i of the height pipelined trie requires $\approx M$ memory.

Since we are forced to do static memory partitioning this condemns us to allocate M memory for almost every stage i. However, this is clearly wasteful because the total amount of memory required across all stages for any database is only M. With S stages, this would be a factor S waste of memory. We leave solutions for this problem to future work.

9. CONCLUSIONS

This paper describes an IP lookup scheme that can scale with memory access speeds while allowing tight guarantees on the number of prefixes that can be supported and providing fast updates times. To do so, we introduced a set of locally compacting allocators that can be tuned to obtain close to 100% utilization of the limited on-chip SRAM needed for high speed lookups. Such compacting allocators are, we believe, a crucial component of *any* IP forwarding chip that supports close to 250,000 prefixes and has fast update times. Such performance figures are beyond the reach of today's CAM technology. Without using compacting allocators, an IP forwarding chip must either use two copies of the database (in which it case it can only guarantee half the number of prefixes and have update times in the order of seconds) or use an incremental scheme with a conventional allocator (in which case the guaranteed number of prefixes goes down by an order of magnitude).

Our locally compacting allocators only compact in the vicinity of the last update. We have shown that the average performance of one of our allocators, the Segment-Hole allocator does much better than a benchmark Best-Fit allocator on average, is tunable, and provides good worst case memory utilization guarantees.

Our compacting allocators are problem-specific unlike general purpose allocators like Best-Fit. However, our allocators work for any application for which each allocated node is only pointed to by a small number of "parents". While we have only emphasized applications like lookup and hashing that have a single pointer, it can also be used for applications with a small number of parents. We conjecture that other applications that use limited fast memory (cache memory for software, or SRAM for hardware) can also benefit from local compaction. Perhaps local compaction could become part of the "bag of tricks" available to systems implementors.

While our sample IP lookup scheme requires a shifted word access to access entire trie nodes, this need for an unconventional memory design can be avoided by using more standard IP lookup schemes like Lulea tries [DBCP97]. However, such compressed trie schemes need to be modified to allow incremental updates in order not to limit memory utilization to 50%. Finally, we note that pipelining is often cited in earlier work as a simple means for making lookup speeds scale with memory speeds (e.g., [SV98]). However, we have shown that pipelining adds new memory allocation problems of its own. While there are good solutions to this important problem, we leave details for a future paper.

[9] One would have to modify the SHC scheme to only maintain the invariant within each row of memory.

[10] Register files in CPUs are multiport memories but are much smaller in size that the amount of SRAM memory (16 Mbits) we need.

10. ACKNOWLEDGEMENTS

The compaction problem was originally suggested to us by Will Eatherton and Zubin Dittia of Growth Networks. Will Eatherton provides an alternate (and elegant) solution for compaction in his thesis [Eat99] that works well for small size granularities. We are also grateful to the anonymous referees, and to our shepherd Nick McKeown who helped us find an appropriate title. We also thank John Holst for helping assure us that a shifted memory design was feasible.

11. REFERENCES

Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

T. Chiueh and P. Pradhan. High Performance IP Routing Table Lookup using CPU caching. *Proc IEEE Infocom 99*, April 1999.

R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. *RFC 2205*, September 1997.

DegerMark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Proceedings SIGCOMM 97*, September 1997.

A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a fair queuing algorithm. *Proceedings of Sigcomm 89*, (September), 1989.

W. Eatherton Hardware-Based Internet Protocol Prefix Lookups *M.S. Thesis, Washington University, Department of Electical Engineering*, May 1999.

Pankaj Gupta, Steven Lin, and Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. *Proceedings of IEEE Infocom 98*, April 1998.

Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, December 1997.

Juniper Networks M20 Product Web Page. *http://www.juniper.net/products/m20-l2.htm.*

D. E. Knuth. The Art of Computer Programming, volume 1. Addison-Wesley Publishing Company, Stanford University, 2nd edition, 1973.

H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. *Proceedings of SIGCOMM 97*, October 1997.

T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of SIGCOMM 98*, October 1998.

B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multi-way and multicolumn search. In *Proceedings of IEEE Infocom 98*, April 1998.

Merit. Routing table snapshot at the Mae-East NAP. *http://www.merit.edu/ipma/routing_table/.*

Merit. Route Tracker Tool *http://www.merit.edu/ipma/tools/.*

N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz. The Tiny Tera: a packet switch core. *IEEE Micro*, January 1997.

Music Semiconductors. What Is a CAM (Content Addressable Memory)? Application Brief AB-N6. *http://www.music-ic.com.*

S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. *Proceedings of IEEE Broadband Communications 98*, (April), 1998.

C. Partridge. Locality and route caches. *In NSF Workshop on Internet Statistics Measurements and Analysis*, February 1996.

R. Perlman. *Interconnections, Bridges and Routers.* Second Edition, Addison-Wesley, 1999.

Y. Rechter and T. Li. *A Border Gateway Protocol 4 (BGP-4) RFC 1771.* 1995.

J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the Association for Computing Machinery*, 18(3):416–423, July 1971.

J. M. Robson. Bounds for some functions concering dynamic storage allocation. *Journal of the Association for Computing Machinery*, 21(3):491–499, July 1974.

J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, August 1977.

IBM. *64Mb Direct Rambus SDRAM Advance Datasheet.* December 1997.

IBM. *Double Data Rate SDRAM, IBM0664404ET3A, 128k x 8 x 4 banks.* IBM, August 1998.

A. Silberschatz and P. B. Galvin. *Operating Systems Concepts.* Addison-Wesley Publishing Company, 5th edition, November 1997.

V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, November 1998.

V. Srinivasan, G. Varghese, S. Suri, and M. Waldgovel. Fast and scalable layer four switching. In *Proceedings of SIGCOMM*, October 1998.

J. Turner, T. Chaney, A. Fingerhut, and M. Flucke. Design of a Gigabit ATM switch. In *Proceedings of Infocom 97*, March 1997.

T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

K. Thomson, G. J. Miller, and R. Wilder. Wide-area traffic patterns and characteristics. *IEEE Network*, December 1997.

Torrent. *Torrent Systems, Inc.* http://www.torrent.com, 1998.

Paul R. Wilson. Uniprocessor garbage collection techniques. In *Springer-Verlag Lecture Notes in Computer Science*, number 637, pages 1–42. Springer-Verlag, September 1992.

P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Springer-Verlag Lecture Notes in Computer Science*, number 986, pages 1–116. Springer-Verlag, September 1995.

M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM 97*, October 1997.

C. D. Waitzman, Partridge, and S.E. Deering. Distance vector multicast routing protocol. *RFC 1075*, November 1988.