High-Performance IP Routing Table Lookup Using CPU Caching

Tzi-cker Chiueh Prashant Pradhan

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

chiueh, prashant @cs.sunysb.edu

Abstract

Wire-speed IP (Internet Protocol) routers require very fast routing table lookup for incoming IP packets. The routing table lookup operation is time consuming because the part of an IP address used in the lookup, i.e., the network address portion, is variable in length. This paper describes the routing table lookup algorithm used in a cluster-based parallel IP router project called *Suez*. The innovative aspect of this algorithm is its ability to use CPU caching hardware to perform routing table caching and lookup directly by carefully mapping IP addresses to virtual addresses. By running a detailed simulation model that incorporates the performance effects of the CPU memory hierarchy against a packet trace collected from a major network router, we show that the overall performance of the proposed algorithm can reach 87.87 million lookups per second for a 500-MHz Alpha processor with a 16-KByte L1 cache and a 1-MByte L2 cache. This result is one to two orders of magnitude faster than previously reported results on software-based routing table lookup implementations. This paper also reports the performance impacts of various architectural parameters in the proposed scheme and its storage costs, together with the measurements of an implementation of the proposed scheme on a Pentium-II machine running Linux.

1 Introduction

Each IP[1] routing table entry [2] logically includes the following fields: a *network mask*, a *destination network address* and an *output port identifier*. Given an IP packet's destination host address, the network mask field of an IP routing table entry is used to extract the destination network address, which is then compared to the entry's destination network address field. If they match, this routing table entry is considered a potential candidate. Logically the destination host address is compared against each and every routing table entry this way. Finally, the routing table entry candidate with the longest network address field wins, and the packet is routed via the output port specified in this entry to the corresponding next-hop router. If none of the routing table entries match the incoming IP packet's destination host address, the packet is forwarded to a default router.

The network mask field in each routing table entry potentially extracts a different number of the most significant bits from an IP address as the network address. Therefore, IP routing table lookup is essentially searching for the longest prefix match for a given destination IP address in the routing table. Instead of sequential scanning, existing IP routing software such as that in BSD Unix, builds an index tree to speed up the lookup process by avoiding visiting unrelated routing table entries. However, even with indices, software-based routing table lookup operation still can not run at wire speed. For example, assume each packet is 1000 bits and the link speed is 1 Gbit/sec, wire-speed routing table lookup means 1 million lookups per second per input port. Only until recently can custom silicon-based routers achieve such performance, let alone pure software solutions.

This paper describes the IP routing table lookup algorithm used in a high-performance software-based IP router project called *Suez*. Because the target hardware platform is a general-purpose CPU, the algorithm is designed specifically for efficient software-only implementation. The key observation of this algorithm is that routing table lookup is a search of the mapping from destination IP addresses to output ports, and the CPU cache on modern microprocessors is designed to facilitate a similar process. Therefore, by treating IP addresses as virtual memory addresses, one can exploit CPU caching as a hardware assist to speed up routing table lookup significantly. In the ideal case, if each routing table lookup corresponds to one virtual address lookup, it takes only one L1 cache access, and hence only one cycle, to do a routing table lookup. However, in practice, more complex machinery is required to twist the CPU cache as a routing table lookup cache, because the "tags" of IP addresses are of variable length, whereas existing CPU cache hardware only supports fixed-length tags. The routing lookup algorithm in Suez carefully maps between IP addresses and virtual addresses to ensure that routing lookups are performed at a rate close to what is dictated by the processor speed.

Section 2 reviews previous work on IP routing table lookup and caching to contrast *Suez*'s approach with theirs. Section 3 presents in detail how *Suez*'s routing table caching and lookup algorithm is architected to exploit CPU caching. Section 4 describes the methodology used to evaluate the performance of the proposed routing table lookup algorithm. Section 5 presents

[1]Unless explicitly indicated otherwise, the term "IP" refers to Internet Protocol Version 4.

[2]Each entry also includes a next-hop router's address and an expiration timer, but they are ignored in this paper.

the measured and simulated performance results and their detailed analysis. Section 6 describes our implementation of the algorithm on a sample architecture and operating system. Section 7 concludes this paper with a summary of the main results of this work, and a brief outline of on-going work.

2 Related Work

The most popular search data structure for the longest prefix string match problem is the bit-by-bit Patricia trie [1]. A similar technique called reduced radix tree [11], has been implemented in the 4.3 version of Berkeley Unix [2]. Pei et al. [3] compared the VLSI implementation cost and the performance of content-addressable memory (CAM)-based and trie-based solutions to the routing table lookup problem. McAuley and Francis [4] proposed several fast routing table lookup solutions for variable-length hierarchical network address based on binary and ternary CAMs. Knox and Panchanathan [5] described a multiprocessor-based routing table lookup solution based on linear pipelined array implementation of the radix searching algorithm. More recently, Waldvogel et al. [9] developed a lookup scheme using multiple hash tables, each based on a distinct prefix length. The worst-case lookup time is shown to be $\log_2(No.\ of\ address\ bits)$. This work also introduced a Mutating Binary Search technique to further reduce the required number of hash table lookups. Degermark et al. [10] developed a compact routing table representation to ensure the entire representation be fit within typical L2 caches. They estimated each lookup can be completed within 100 instructions using 8 memory references. Compared to previous schemes, *Suez*'s routing table organization is much simpler and the lookup operation is thus more efficient. Unlike tree-based search algorithms, *Suez*'s routing table lookup algorithm does not require backtracking to support longest prefix match.

Another work in [13] addresses routing table lookup in hardware by using a large amount of inexpensive DRAM to implement a two-level flat lookup table and pipelining the accesses. In contrast, the routing lookup algorithm in Suez relies solely on standard processor memory and caching hardware and is a software implementation based on exploiting the address and page management hardware /software in commodity PCs.

None of previous works have reported detailed delay measurements that included CPU caching effects. Another way to speed up routing table lookup is to cache the lookup results. Feldmeier [6] studied the management policy for the routing-table cache, and showed that the routing-table lookup time can be reduced by up to 65%. Chen [7] investigated the validity and effectiveness of caching for routing-table lookup in multimedia environments. Estrin and Mitzel [8] derived the storage requirements for maintaining state and lookup information on the routers, and showed that locality exists by performing trace-driven simulations of an LRU routing table lookup cache, for different conversation granularities. None of the previous works exploited the CPU cache available in modern processors, as in *Suez*. In addition, we believe *Suez*'s routing table lookup scheme is the first that integrates routing table caching and searching in a single algorithm. Also, our work includes detailed simulation and implementation measurements of the algorithm against the routing table and real traffic traces collected from a major network router.

3 Routing-Table Lookup Algorithm

A major design goal of the *Suez* project is to demonstrate that general-purpose CPU can serve as a powerful platform for high-performance IP routing. Therefore, *Suez*'s routing table lookup algorithm heavily leverages off the cache memory hierarchy. The algorithm is based on two data structures, a destination host address cache (HAC), and a destination network address routing table (NART). Both are designed to use CPU cache efficiently. The algorithm first looks up the HAC to check whether the given IP address is cached in the HAC because it has been seen recently. If so, the lookup succeeds and the corresponding output port is used to route the associated packet. If not, the algorithm further consults the NART to complete the lookup.

3.1 Host Address Caching

Typically multiple packets are transferred during a network connection's lifetime, and the network route a connection takes is relatively stable. Therefore the destination IP address stream seen by a router exhibits temporal locality. That is, the majority of the routing table lookups are serviced directly from the HAC. Therefore, minimizing the HAC hit access time is crucial to the overall routing table lookup performance.

Rather than using a software data structure such as a hash table, *Suez*'s HAC is architected to be resident in the Level-1 (L1) cache at all time, and to be able to exploit the cache hardware's lookup capability directly. As a first cut, the 32-bit IP addresses can be considered as 32-bit virtual memory addresses and simply looked up in the L1 cache. If they hit in the L1 cache, the lookup is completed in one CPU cycle; otherwise an NART lookup is required. However, this approach exhibits several disadvantages. First, there is much less spatial locality in the destination host address stream compared to the memory reference stream in typical program execution. As a result, the address space consumption/access pattern is going to be sparse, which may lead to a very large page table size and excessive TLB miss rate and/or page fault rate. Second, unless special measures are in place, there is no guarantee that HAC is L1 cache-resident all the time. In particular, uncoordinated virtual-to-physical address mapping may result in unnecessary conflict cache misses, because of interference between HAC and other data structures used in the lookup algorithm. Third, the cache block size of modern L1 cache is too large for the HAC. That is, due to the lack of spatial locality in the network packet reference stream, individual cache blocks tend to be under-utilized, leading to zero or one HAC entry in the cache block most of the time. Consequently, the overall caching efficiency is less than what it should be for a fixed cache size.

To address these problems, *Suez*'s HAC lookup algorithm takes a combined software/hardware approach. To reduce virtual address space consumption, *Suez* only uses certain portion of each IP address to form a virtual address, and leaves the remaining bits of the IP address as tags to be compared by software. This approach makes it possible to restrict the number of virtual pages reserved for the HAC to a small number.

To ensure that the HAC is always L1 cache-resident, a portion of L1 cache is *reserved*. For the purpose of exposition, let's assume that the page size is 4KBytes, the L1 cache is 16KBytes direct mapped with a block size of 32 bytes and the first 4KBytes of the cache are reserved for the HAC. This means that given a physical address, the 9 bits from bit 5 to bit 13 of the physical address will be used to identify a cache block. Since the page size is 4KBytes, the physical page number for this address overlaps with these 9 bits in its last two bits. Thus, the physical addresses that get mapped to the first 4KBytes of the cache would be the ones that lie in physical pages whose page numbers are a multiple of 4.

Now, to prevent other data structures from polluting the portion of L1 cache reserved for HAC, all of the other physical pages whose physical page number is an integral multiple of 4 should be marked as *uncacheable* at system startup, so that they would never be brought into L1/L2 cache at run time. This would ensure that each HAC access is always a cache hit and hence completes in one cycle. Thus by reserving one virtual page for the HAC and mapping it to a 4-KByte physical page whose physical page number is an integral multiple of 4, if we remap IP addresses to virtual addresses lying within the HAC, we can ensure that all IP addresses be checked against the HAC, and these checks are guaranteed to involve only L1 cache accesses. There is thus a performance tradeoff between HAC hit ratio and the utilization efficiency of the L2 cache. Larger the percentage of L1 cache reserved for HAC, the lesser is the percentage of L2 cache usable for other purposes, for example, NART search. However, for a given HAC size, say one page, the utilization efficiency increases as the L1 cache size increases.

Finally, to improve the cache utilization efficiency, a software-driven set-associative caching mechanism is adopted. The fact that the HAC address space is mapped to a single physical page means IP addresses with different "virtual page numbers" can now co-reside in the cache block. Therefore, each cache block may contain multiple HAC entries. The cache hardware first identifies the block of interest, and the software examines each HAC entry of that block in turn until a hit or exhaustion. To exploit temporal locality, the software starts the examination of each cache block from the last HAC entry it accesses in the previous visit. Because associativity is implemented in software, HAC hit time is variable, and HAC miss time increases with the degree of associativity. The performance tradeoff in choosing the degree of associativity then lies in the gain in higher HAC hit ratios versus the loss in longer miss handling time.

Assume the L1 data cache is 16KBytes direct-mapped and physically addressed, with 32-byte blocks. Also assume that the virtual memory page size is 4 KBytes, and each HAC entry is 4 bytes wide, containing a 23-bit tag, one unused bit, and an 8-bit output port identifier. Therefore each cache block can contain at most 8 HAC entries. Finally, assume one quarter of the L1 cache, i.e., 128 out of 512 cache sets, is reserved for HAC.

Assume that the HAC page is allocated virtual page number V_{HAC}. Given a destination IP address, DA, the $DA_{5,11}$ [3] portion is extracted and used as an offset into the HAC page to form

[3]We will use the notation$X_{n,m}$ to represent a bit subsequence of X ranging from the n-th bit to the m-th bit.

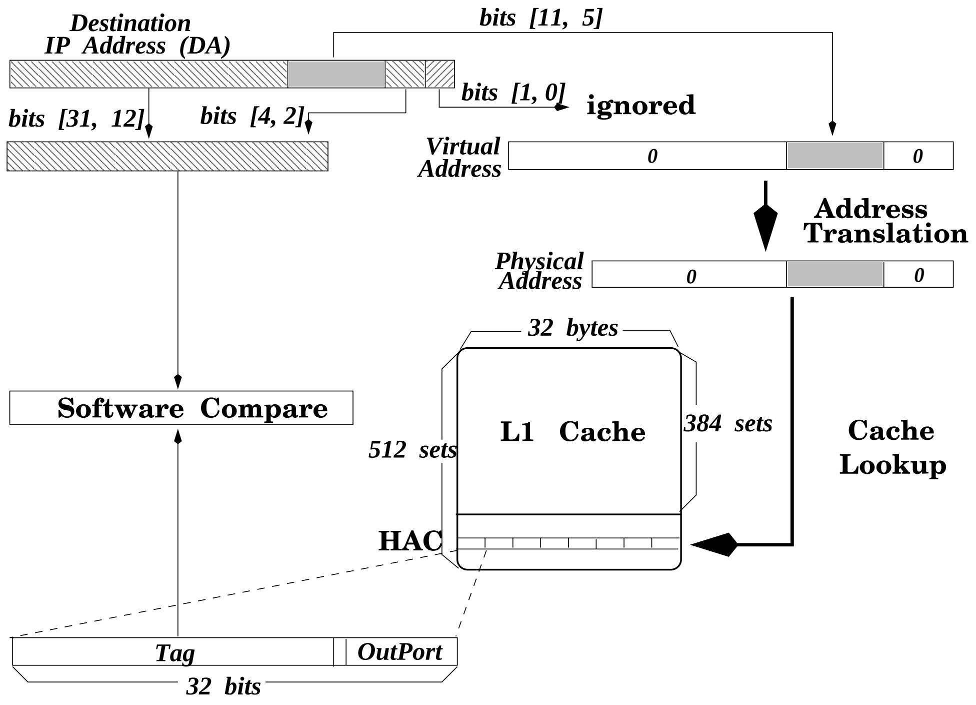

Figure 1: *The data flow of HAC lookup. Here we assume that L1 cache is direct mapped with 512 32-byte cache sets, among which 128 cache sets are reserved for HAC. Each HAC entry is 4 bytes wide. The search through the HAC entries within a cache set is performed by software.*

a 32-bit virtual address $VA = V_{HAC} + DA_{5,11}$. The virtual address thus formed fetches the first word, CW, of the cache set specified by $DA_{5,11}$. Meanwhile, a 23-bit tag, DAT, is constructed from a concatenation of $DA_{12,31}$ and $DA_{2,4}$. Since realistically none of the routing tables contain network addresses with prefix lengths longer than 30, the last two bits of the destination host address are ignored while forming the tag.

The first word of each HAC cache set is a duplicated HAC entry that corresponds to the HAC entry last accessed in the previous visit to the cache set. If DAT matches $CW_{9,31}$, the lookup is completed and $CW_{0,7}$ is the output port identifier. Otherwise, the software search continues with the second word of the cache set and so on, until a hit or the end of the set.

The way VA is formed from DA means that only one virtual page is allocated to hold the entire HAC. As a result, only one page table entry and thus one TLB entry is required. Such a setup guarantees that the performance overhead in HAC lookup due to TLB misses is negligible.

After the lookup is completed, a new HAC entry for the destination IP address is constructed and a word from the corresponding cache set is allocated for it, if such an HAC entry does not exit yet. Finally, this HAC entry is copied to the first word of the cache set if necessary. When a cache word is chosen for replacement, the associated HAC entry is gone. Because the HAC contents are generated dynamically from looking up the NART, it is safe to over-write HAC entries without keeping a copy of them. Figure 1 illustrates the flow of the HAC lookup process.

The following code segment shows the exact instruction sequence that implements the HAC lookup algorithm. The input is DA, a 32-bit destination IP address, and the output is the corresponding 8-bit output port identifier, $OutPort$. We assume that the CPU is three-way superscalar, and the load delay stall is one cycle.

```
Initialization:
$Mask1 \leftarrow 00000000000000000000111111100000$;   $Mask2 \leftarrow 11111111111111111111100000000000$;
$Mask3 \leftarrow 00000000000000000000000000011000$;   $Mask4 \leftarrow 00000000000000000000000011111111$;
$Mask5 \leftarrow 11111111111111111111110000000000$;   $Count \leftarrow 0$;

1.  $MAR \leftarrow DA \&\& Mask1$;  $TR1 \leftarrow DA \&\& Mask2$;  $TR2 \leftarrow DA \&\& Mask3$;
2.  $CW \leftarrow Mem[MAR]$;  $TR2 \leftarrow TR2 << 7$;
3.  $DAT \leftarrow TR1 \,||\, TR2$;  $MAR0 \leftarrow MAR$;
4.  $Match \leftarrow DAT\ nxor\ CW$;  $Mem[MAR0] \leftarrow CW$;  $MAR \leftarrow MAR + 4$;
5.  $If(Match \geq Mask5)\ goto\ DONE$;  $CW \leftarrow Mem[MAR]$;  $OutPort \leftarrow CW \&\& Mask4$;
6.  $If(Count < 7)\ goto\ 4$;  $Count \leftarrow Count + 1$;
7.  $Perform\ NART\ lookup$;  $Count \leftarrow 0$;
8.  $DONE$;
```

When the program control jumps to `DONE`, $OutPort$ contains the output port identifier to route the IP packet in question. While searching through the HAC entries in the cache set, the first word of the cache set is speculatively loaded with the current HAC entry that is being examined (the second instruction at the 4-th cycle). The above algorithm gives a 5-cycle best-case latency for an IP address lookup, and takes 3 cycles for each additional HAC entry access in the cache set. So it takes $5 + 3 * 7 + 1 = 27$ cycles to detect a HAC miss. By pipelining the lookups of consecutive IP packets, i.e., overlapping the 5-cycle work of each lookup, the best-case throughput can be improved to 3 cycles per lookup. Because HAC is guaranteed to be in the L1 cache, the 3-cycle per lookup estimate is exact, because the HAC lookup itself is a hit.

3.2 Network Address Routing Table

If the HAC access results in a miss, a full-scale network address routing table lookup is required. The guiding principle of *Suez*'s NART design is to trade table size for lookup performance. This principle is rooted in the observation that the L2 cache size of modern microprocessors is comparatively large for storing routing tables and is expected to continue increasing over time. For example, one MByte of L2 cache is a norm in mid-range PCs today. On the other hand, the routing table of an Internet backbone router has about 100,000 entries. Assume that each entry takes 8 bytes, then the entire routing table takes only 800 KBytes, i.e., comfortably fits into a PC's L2 cache. Of course, such a comparison is over-simplified because the 800 KBytes estimate assumes a sequential search procedure, and thus does not include any search data structure to speed up the lookup process.

As mentioned in the Introduction section, NART lookup is difficult to speed up because given an IP address, it is not possible to determine a priori its corresponding network address, which is the most significant N bits of the IP address, where N is unknown beforehand. With the increasing threat of the IP address depletion, a technique called Classless InterDomain Routing (CIDR) is currently in use that allows more efficient allocation of contiguous address chunks in the IP address space. In contrast to the original Class A/B/C scheme, in which N can take only one of three possible values: 8, 16, and 24, CIDR allows N to take any value (realistically, from 1 to 30). This generality complicates routing table lookup because it significantly increases the number of possible network addresses.

NART Construction

Let us classify the network addresses in an IP routing table into three types according to their length: smaller than or equal to 16 bits (called Class AB), between 17 and 24 bits (called Class BC), and between 25 and 30 bits (called Class CD). To represent a given routing table, *Suez*'s NART includes three levels of tables: one *Level-1 table* and one *Level-3 table*, but a variable number of *Level-2 tables*. Entries in these tables are denoted as Level1-Table[.], Level2-Table[.], and Level3-Table[.], respectively. The Level-1 table has 64K entries, each of which is 2 bytes wide, and contains either an 8-bit output port identifier or an offset pointer to a Level-2 table. These two cases are distinguished by the entry's most significant bit. Each Level-2 table has 256 entries, each of which is 1 byte wide, and contains either a special indicator (11111111) or an 8-bit output port identifier. The Level-3 table has a variable number of entries, each of which contains a 4-byte network mask field, a 4-byte network address, and an 8-bit output port identifier. The number depends on the given IP routing table and is typically small.

To convert an IP routing table to the NART representation, Class AB addresses are processed first, then Class BC addresses, followed by Class CD addresses. Each Class AB network address, NA, of length L in the routing table takes 2^{16-L} entries of the Level-1 table starting from Level1-Table[$NA_{0,L-1} * 2^{16-L}$], with each of these entries filled with NA's output port identifier, as specified in the IP routing table. After all Class AB addresses are processed, the unfilled Level-1 table entries are filled with the *default* output port identifier.

For each Class BC address, NA, of length L in the IP routing table, if Level1-Table[$NA_{L-16,L-1}$] contains an 8-bit output port identifier, a Level-2 table is created; Level1-Table[$NA_{L-16,L-1}$] is put aside as OPI and changed to be an offset that can be used to compute the base of the newly created Level-2 table; every entry in the Level-2 table is initialized to OPI. If Level1-Table[$NA_{L-16,L-1}$] already contains an offset to a Level-2 table, no table is created. In both cases, 2^{24-L} entries of the Level-2 table, starting from Level2-Table[$NA_{0,L-17} * 2^{24-L}$], are filled with NA's output port identifier as specified in the IP routing table.

For each Class CD address, NA, of length L in the IP routing table, if Level1-Table[$NA_{L-16,L-1}$] contains an 8-bit output port identifier, a Level-2 table is created; Level1-Table[$NA_{L-16,L-1}$] is put aside as $OPI1$ and changed to be an offset that can be used to compute the base of the newly created Level-2 table; and every entry in the Level-2 table is initialized to $OPI1$. If Level1-

Table[$NA_{L-16,L-1}$] already contains an offset to a Level-2 table, this Level-2 table is used in the next step. If Level2-Table[$NA_{L-24,L-17}$] is *not* 11111111, Level2-Table[$NA_{L-24,L-17}$] is put aside as $OPI2$ and is changed to 11111111; and a new Level-3 table entry with a network mask of 11111111111111111111111100000000, a network address of $NA_{L-24,L-1}$, and an output port identifier of $OPI2$ is created. Regardless of the content of Level2-Table[$NA_{L-24,L-17}$], a Level-3 table entry with a network mask same as NA's network mask, a network address of NA and an output port identifier the same as NA's output port identifier, is added to the global Level-3 table.

NART Lookup

Given a destination IP address, DA, the most significant 16 bits, $DA_{16,31}$, are used to index into the Level-1 table. The exact address used to index the Level-1 table is $Base1 + 2 * DA_{16,31}$, where $Base1$ is the base address of the Level-1 table. If DA has a Class AB network address, one lookup into the Level-1 table is sufficient to complete the routing table search.

If DA has a Class BC address, Level1-Table[$DA_{16,31}$] contains an offset from which the base of the Level-2 table is computed. The corresponding Level-2 table's base is $Base2 + 256 * Level1 - Table[DA_{16,31}]$, where $Base2$ is the base for an address region specifically reserved for all Level-2 tables. If Level2-Table[$DA_{8,15}$] is not 11111111, the lookup is completed. Otherwise the network address of DA is a Class CD address, and a search through the Level-3 table is required.

We chose to use a single global Level-3 table, rather than multiple Level-3 tables like Level-2 tables, because it reduces the storage space requirement and because it allows Level-2 table entries to specify up to 255 output port identifiers by not requiring them to have offsets to Level-3 tables. The search through the Level-3 table is based on $DA_{0,31}$, one after another. Because the Level-3 table entries are sorted according to the length of their network addresses, the linear search can stop at the first match. As will be shown in the Section 5, the performance overhead of this linear search is insignificant. Figure 2 illustrates the data flow of the NART lookup process.

The following code segment shows the exact instruction sequence that implements the NART lookup. Again the input is DA, a 32-bit destination IP address, and the output is the corresponding 8-bit output port identifier, $OutPort$. We assume that the CPU can issue three instructions at a time, and the load delay stall is one cycle.

```
Initialization:
$Mask1 \leftarrow 11111111111111110000000000000000$;   $Mask2 \leftarrow 00000000000000001111111100000000$;
$Mask3 \leftarrow 00000000000000000000000011111111$;   $Mask4 \leftarrow 00000000000000001000000000000000$;
$Mask5 \leftarrow 00000000000000000111111111111111$;

1.  $Index1 \leftarrow DA \&\& Mask1$;   $Index2 \leftarrow DA \&\& Mask2$;
2.  $Index1 \leftarrow Index1 >> 14$;   $Index2 \leftarrow Index2 >> 7$;
3.  $MAR \leftarrow Base1 + Index1$;
4.  $MDR \leftarrow Mem[MAR]$;    /* MDR is the indexed Level-1 Table entry */
```

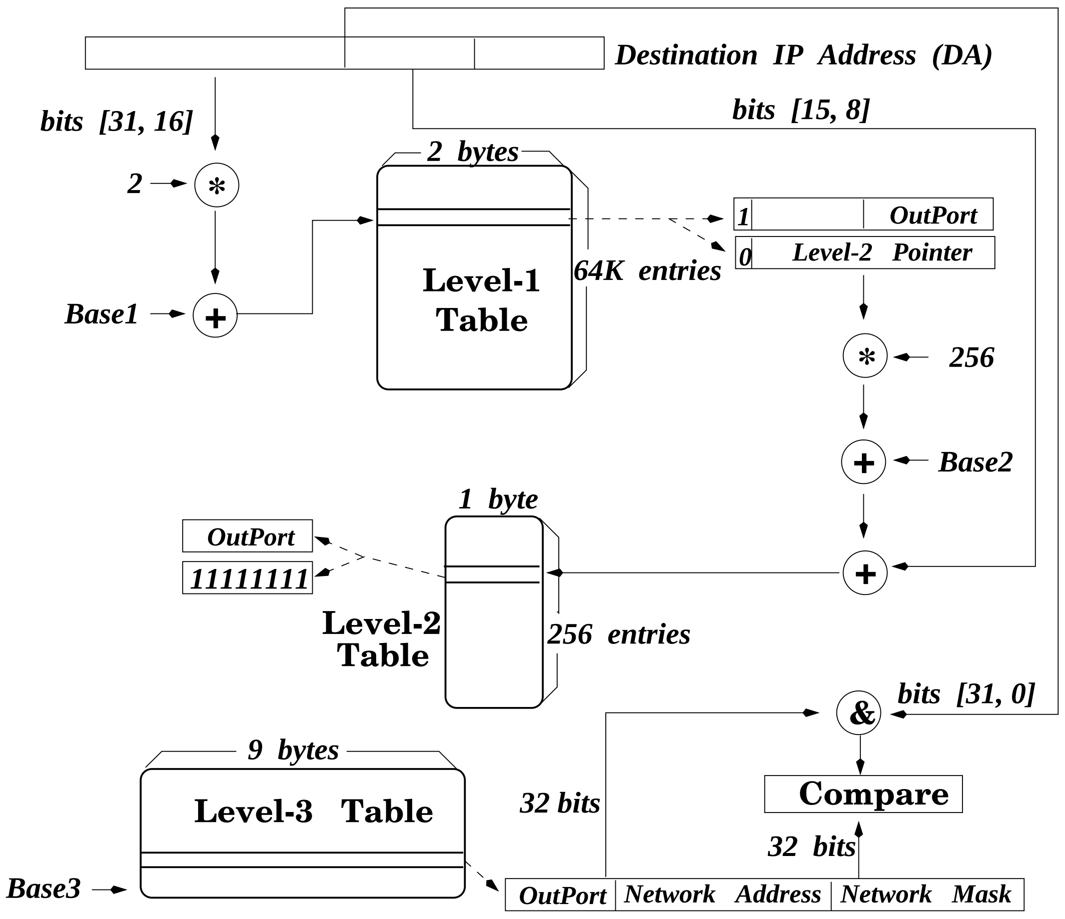

Figure 2: *The data flow of the NART lookup. It starts with the Level-1 table, and if necessary looks up a Level-2 table pointed by the Level1-Table[$DA_{16,31}$]. If the chosen Level-2 table entry still can not resolve DA, then a sequential search through the Level-3 table is needed. Note that NART has only one Level-1 table and one Level-3 table, and many Level-2 tables.*

```
5.  /* load delay slot */
6.  $Flag \leftarrow MDR \; \&\& \; Mask4$;  $OutPort \leftarrow MDR \; \&\& \; Mask3$;  $Offset \leftarrow MDR \; \&\& \; Mask5$;
7.  $If(Flag) \; goto \; DONE$;  $Offset \leftarrow Offset << 8$;  $TBase \leftarrow Base2 + Index2$;
8.  $MAR \leftarrow TBase + Offset$;
9.  $MDR \leftarrow Mem[MAR]$;  /* MDR is the indexed Level-2 Table entry */
10.  $MAR \leftarrow Base3$;
11.  $OutPort \leftarrow MDR \; \&\& \; Mask3$;
12.  $If(OutPort \; != \; Mask3) \; goto \; DONE$;
13.  $NMask \leftarrow Mem[MAR]$;
     /* NMask is the indexed Level-3 Table entry's network mask field */
14.  $MAR \leftarrow MAR + 4$;
15.  $Tag \leftarrow NMask \; \&\& \; DA$;  $NA \leftarrow Mem[MAR]$;
     /* NA is the indexed Level-3 Table entry's network address field */
16.  $MAR \leftarrow MAR + 5$;  $MAR1 \leftarrow MAR + 4$;
17.  $If(Tag \; == \; NA) \; goto \; 14$;  $NMask \leftarrow Mem[MAR]$;
18.  $MDR \leftarrow Mem[MAR1]$;
19.   /* load delay slot */
20.  $OutPort \leftarrow MDR \; \&\& \; Mask3$;
21.  $DONE$;
```

If DA's network address is a Class AB address, the lookup takes 7 cycles to complete. If DA's network address is a Class BC address, the lookup takes 12 cycles to complete. If DA's network address is a Class CD address, the lookup takes $20 + 4 * K$ cycles to complete, where K is the number of Level-3 table entries the algorithm has to examine before the first match. Unlike HAC lookup, these cycle estimates may not be precise because there is no guarantee that the memory accesses in NART lookup always result in cache hits.

4 Evaluation Methodology

4.1 Trace Collection

We chose to use a trace-driven simulation methodology to evaluate the performance of *Suez*'s routing table lookup algorithm. Although there are several IP packets traces available in the public domain, none of them are suitable for our purposes for two reasons. First, they were mostly captured before 1993, when WWW started to take off, and thus may not be representative of today's network traffic. Second, all of these traces have been "sanitized", i.e., IP addresses replaced with unique integers. While this is fine for fully associative cache simulation, it is completely unusable for set-associative cache simulation and NART lookup simulation.

As a result, we decided to collect a packet trace from the periphery link that connects the Brookhaven National Laboratory (BNL) to the Internet via ESnet at T3 rate, i.e., 45 Mbits/sec.

This link is the only link that connects the entire BNL community to the outside world. The internal backbone network of BNL is a multi-ring FDDI network, while the periphery router is a Cisco 7500 router. The trace was collected by a TCPdump program running on a Pentium-II/233 MHz machine, which snoops on a mirror port of a Fast Ethernet switch that links BNL's periphery router with ESnet's router. Therefore the packet collection is completely un-intrusive. The first 80 bytes of each Ethernet packet are captured, compressed, and stored to the disks in real time. The packet trace is collected from 9AM on 3/3/98 to 5PM on 3/6/98. The total number of packets in the trace is 184,400,259, with no packet loss reported by TCPdump.

4.2 Architectural Assumptions

The performance metric of interest is million lookups per second (MLPS). We have built a detailed simulator that incorporates the HAC and NART lookup algorithms, as well as the effects of L1 cache, L2 cache and main memory. However, the effect of the TLB is not modeled. This should not affect HAC lookup performance because the HAC costs only one TLB entry and thus should be TLB-resident all the time. Unless specified otherwise, the L1 cache is direct mapped, its size is 16 KBytes and its cache block size is 32 bytes; the L2 cache is direct mapped, its size is 1 MBytes and its cache block size is 32 bytes. The first 4 KBytes of the L1 cache are reserved for HAC. The CPU memory hierarchy also impacts the performance of the NART lookup delays, because the Level-1, Level-2, and Level-3 tables may be spread across L1 cache, L2 cache and main memory. Depending on which level of the memory hierarchy satisfies the memory accesses during the NART search process, the total lookup delay varies accordingly. In this study, we assume that the access time for L1 cache hit is 1 CPU cycle, the total access time for L1 cache miss and L2 cache hit is 6 CPU cycles, and the total access time for L1 cache miss, L2 cache miss and main memory hit is 30 CPU cycles.

Source	**No. of Entries**	**AB**	**BC**	**CD**	**NART Size (Bytes)**	**Efficiency**
Merit	39,680	14.2%	85.8%	0.03%	843,444	42.3%
IANA/ISI	171,938	5.9%	79.1%	15.0%	886,023	174.6%
BNL	50,523	12.7%	87.2%	0.1%	905,195	50.2%
Trace	*X*	78.7%	21.29%	0.01%	*X*	*X*

Table 1: *The distribution of Class AB, BC, and CD network addresses in static routing tables and the collected dynamic packet trace. X means not applicable.*

5 Performance Evaluation

5.1 Network Address Distribution

To determine the storage cost of NART tables, we converted the routing tables of two major routers, one from BNL and one from Merit [12], into the NART representation. We also did the conversion for the entire IP network address space as reported by IANA/ISI up until 11/1997. The results are shown in Table 1. Each row shows the total number of entries in each routing table, the NART representation size, and the storage efficiency, which is the ratio of the given flat IP routing table size and the NART table size. Here we assume each routing table entry costs 9 bytes : 4 bytes each for network address and network mask, and 1 bytes for thge output port. For Merit and BNL, the storage efficiency is around 50%, which means that NART costs twice as much as the flat IP routing table representation to speed up the lookup process. However, NART is actually more efficient than the flat IP routing table representation in the case of IANA. This is so because fewer entries in the NART tables are wasted as more routing table entries are to be represented and because flat routing table entries are 9 bytes whereas level one and level two entries are 2 and 1 byte respectively.

Table 1 also shows the distribution of Class AB, BC, and CD addresses. Class BC addresses seem to be dominant in all routing tables. Although there is a significant percentage of Class CD addresses in the IP network address space (IANA), the percentage of Class CD addresses is insignificant in operational routers' routing tables, because Internet routing tends to be performed in a hierarchical fashion. Note that there is an anomaly in that the NART size for IANA is actually smaller than that for BNL. The reason is that the IANA address allocation sample that we could find contained only addresses allocated within the US. Allocation data in other regions was not accessible at that time.

5.2 Results and Analysis

Table 2 shows the overall performance of *Suez*'s routing table lookup algorithm for a 500-MHz Alpha processor with a 16-KByte L1 cache and 1-MByte L2 cache, as the HAC's degree of associativity varies. The HAC size is fixed at 4 KBytes. Because the first HAC entry in each cache set is replicated for performance optimization, 7-way and 3-way logical set associativities actually correspond to 8-way and 4-way physical set associativities. For 1-way and 2-way set associativities, however, this optimization is not applied. The cache set size is the degree of physical associativity multiplied by 4 bytes. In the best case, the proposed scheme can perform one lookup every 5.69 CPU cycles, or 87.87 million lookups per second. This performance level is one to two orders of magnitude better than previous reported results on software-based IP routing table lookup implementations!

Although larger degrees of associativity increase the HAC hit ratio, the best performance occurs when the degree of associativity is 1. The main reason is that set associativity is implemented in software in this algorithm, which entails two performance overheads. First, with a large set-associative cache, the access time for HAC hits at the end of the cache sets is not necessarily shorter than NART lookup time. That is, the time to match an IP address at the 8-th HAC entry of a 8-way HAC's cache set may be longer than looking it up in the NART directly. Second, large set associativities increase the HAC miss detection time, thus adding a higher fixed overhead for *every* NART lookup. In the BNL trace, because the HAC hit ratio is already high (95.22%) when it is direct mapped, the additional gain in HAC hit ratio is not sufficient to offset the additional overhead due to software-implemented set associativity. The reason that the overall performance of the 3-way case is slightly better than that of the 2-way case, despite lower HAC hit ratio and higher HAC miss detection time, is because the average hit access time for the 3-way case is shorter, owing to the duplication optimization.

Degree of Logical Associativity	**MLPS**	**HAC Hit Ratio**
7 (8)	60.49	97.87%
3 (4)	77.09	96.95%
2 (2)	71.92	97.21%
1 (1)	87.87	95.22%

Table 2: *The performance comparison in MLPS (Million Lookups Per Second) among HACs with the same total size (4 KBytes) but varying degrees of logical set associativity, ranging from 7, 3, 2 to 1. The parenthesized numbers are physical set associativities. The hit ratio for the 2-way case is higher than the 3-way case because the former does not waste HAC entries in duplication optimization.*

Duplication	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Hit Time**
Yes	78.73%	3.02%	3.04%	3.06%	3.04%	3.04%	3.01%	3.03%	7.55
no	14.09%	14.62%	14.04%	14.15%	14.45%	14.84%	13.79%	*X*	13.99

Table 3: *The comparison of the distributions of HAC hits among the blocks within a cache set between the cases with and without duplication optimization. The degree of associativity is 7. X means not applicable. The average hit access time is measured in numbers of CPU cycles.*

We conjecture that it is worthwhile to put in the first HAC entry in each cache set the matched HAC entry when the cache set was last accessed. That is, the first HAC entry is a duplication of some other HAC entry in the set. Table 3 demonstrates that this is indeed the case, by showing the distributions of HAC entry hits within a cache set for a logically 7-way cache with and without duplication optimization. With duplication optimization, most HAC hits go to the first entry in the cache sets, thus significantly reducing the average hit access time. Without duplication optimization, HAC hits are evenly spread among the blocks in the cache sets, and thus lead to longer average hit access time. The overall lookup performances are 60.69 MLPS and 34.29 MLPS for these two cases.

Increasing the reserved HAC size in L1 cache improves the HAC hit ratio, but potentially can hurt the NART lookup performance because it reduces the effective L2 cache size available to NART. Table 4 shows the overall lookup performance when a different portion of a 16-KByte L1 cache is reserved for the HAC. Surprisingly, the gain in HAC hit ratio due to larger HAC size outweighs the NART performance loss due to lower L2 cache utilization. The explanation for this behavior is that the working set in the NART lookup is fully captured even when only half of the L2 cache, 500 KBytes, is available, which corresponds to the case in which the HAC size is 8 KBytes. When the reservation for HAC is zero, it means that all lookup requests go to NART directly. The performance for this case, 30.2 MLPS, is still respectable, and reflects the intrinsic performance of the NART itself.

HAC Size	**MLPS**	**HAC Hit Ratio**
8 KBytes	68.86	98.78%
4 KBytes	60.49	97.89%
0 KBytes	30.2	0%

Table 4: *The performance comparison among HACs with the same degree of set associativity (7) but different sizes, 8 KBytes vs. 4 KBytes.*

We also hypothesize that unlike instruction/memory reference traces during program execution, the packet trace exhibits much less spatial locality. We ran the packet trace against a direct-mapped 4-KByte cache using different cache block sizes, from 1 byte to 32 bytes. Table 5 shows that as the cache block size decreases, the hit ratio indeed increases. This result verifies the lack of spatial locality in the packet trace. By comparing the hit ratios of the case in Table 2 when the cache set size is 32 bytes and the case in Table 5 when the block size is 32 bytes, it is clear that software set associativity improves the cache hit ratio significantly, yet another evidence of the lack of spatial locality in the packet trace.

1 Byte	**4 Bytes**	**16 Bytes**	**32 Bytes**
97.41%	94.02%	86.40%	80.64%

Table 5: *The hit ratios for a direct-mapped 4-KByte cache using different cache block sizes. As the cache block size decreases, the hit ratio increases. This verifies the conjecture that the packet trace lacks spatial locality.*

6 Implementation

The simulation results reported in the previous section assume a RISC architecture, which typically provides a large number of general purpose registers, and a direct-mapped L1 cache. The pseudo-code sequence presented earlier makes the best use of RISC processors' superscalarity with techniques such as software pipelining, i.e., overlapping the execution of consecutive iterations in a loop. In addition, it is critical to apply the performance optimization that ensures that HAC is always L1-cache resident. However, we chose to prototype the proposed routing-table lookup algorithm on a Pentium-II machine running Linux 2.0.30 operating system, because we did not have access to a RISC machine with OS source code at the time of writing. The current implementation was not hand-coded to minimize the instruction cycle count. We just relied on the compiler's optimization capabilities. Due to the small number of registers (7) provided by Pentium-II, the generated instructions were not parallelizable since adjacent instructions tend to have data dependencies. Moreover, the L1 and L2 caches in Pentium-II are set associative rather than direct mapped. This means that the idea of reserving a portion of the L1-cache specifically for HAC by manipulating cacheability bits in the page table entries is not completely effective. The final difference between Pentium-II and RISC processors is that the branch misprediction penalty seems to be fairly high. As a result, the best-case scenario, where the HAC hit occurs at the very first entry, takes 23 cycles rather than 5 cycles. We plan to hand-code the entire HAC and NART code in the future work.

Almost all modern processor architectures support a *page cache disable* flag in the page table entry which when set, disables caching for pages corresponding to the given page table entry. Assume the HAC is allocated a physical page whose page number is a multiple of 4. Any other virtual page that is mapped to a physical page whose page number is a multiple of 4 is made non-cacheable by turning the page cache disable bit on in its page table entry. Note that typically the instruction cache and data cache are separate, so there is no contention between data and instruction references. Hence, we do not have to make any of the code pages in the kernel non-cacheable.

However, since Pentium-II uses a set-associative L1 cache, the set of pages that contend with HAC are no longer just those pages whose page number is a multiple of 4. In fact, if the L1 cache is 4-way set associative and has 16 KBytes, the 4-KByte HAC would be spread across the entire L1 cache and would face contention from all the pages in the address space. Therefore the only benefit of disabling the caching for pages whose page number is a multiple of 4 is to reduce but not eliminate the cache contention.

The results of running the trace through the prototype implementation on a Pentium-II 233 MHz machine with a 16-KByte 2-way set associative L1 cache, and 512-KByte 4-way set associative L2 cache for various HAC configurations are shown in Table 6. The timing measurements were made using the cycle counter provided by the Pentium architecture. The best-case performance, 8.04 MLPS, is about an order of magnitude slower than the simulation result, because of the high cycle count for the HAC hit case and the slower clock rate (233 MHz) compared to the assumed one (500 MHz).

The importance of the duplication optimization is again demonstrated by comparing the MLPS numbers for the 7-way case and for the case where the tags for all 8 blocks are different. The former case gives 4.95 MLPS (as in the table) whereas the latter case gives 3.14 MLPS, due to the

Degree of Logical Associativity	**MLPS**
7 (8)	4.95
3 (4)	6.05
2 (2)	5.51
1 (1)	8.04

Table 6: *The performance comparison in MLPS (Million Lookups Per Second) for the implementation on Pentium-II. As in the simulation, the 2-way case loses to the 3-way case because of higher hit time, since the duplication optimization is not applicable in the 2-way case.*

higher hit time. Finally, we measured the no-HAC case. In this case, the HAC is turned off and all pages are cacheable. Every lookup goes through the NART search only. This gives an MLPS of 5.2, which is actually better than some of the configurations with the HAC in place. The reason for this behavior is that the HAC hit cycle count in those cases is actually higher than the cycle count required for NART lookup itself.

7 Conclusion

This paper describes the design and detailed simulation results of a high-performance software-based IP routing table lookup algorithm that exploits CPU caching aggressively. The algorithm uses a portion of the CPU cache to support destination host address caching, and resorts to a table-driven lookup procedure to resolve the routing decisions. This table-driven lookup procedure is simple and fast, and does not require backtracking to support longest prefix match. Our empirical data shows that although the associated storage cost is higher than other schemes, it is still within the typical L2 cache size on current-generation PCs and workstations. Through a detailed performance model that includes the effects of the CPU memory hierarchy, this work demonstrates that the proposed routing table lookup algorithm can achieve up to 87.87 and 30.2 million lookups per second with and without host address caching, respectively, on a 500-MHz DEC Alpha system with 1 MByte of L2 cache. This is at least an order of magnitude faster than any empirical performance results on software-based routing table lookup implementations reported in the literature.

Currently we are implementing the proposed algorithm on the *Suez* prototype, which is based on 233-MHz Pentium-II processors interconnected by Myrinet networks. An important issue that the current implementation is addressing is the routing table update cost, which is not discussed in this paper. We intend to collect route change statistics to evaluate the performance cost associated with run-time NART reorganization. Finally, with the advent of IPv6, CIDR may not be as important any more. However, the basic idea of HAC and NART is equally applicable to IPv6. The only difference is that for HAC, more number of comparisons need to be done for tag matching and the number of levels, as well as the level boundaries, in the NART may need to be adjusted, depending on the density distribution in the IPv6 address space.

Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc. The authors would also like to thank Mike O'Connor and Frank Lepera in Brookhaven National Lab in helping us collecting the packet trace used in this research, and Jeff Mogul in providing help to run our simulator against un-sanitized DEC traces in the early stage of this research. Prashant Pradhan is partially supported by a Symbol Fellowship.

References

[1] Szpankowski, W., "Patricia tries again revisited," Journal of the Association for Computing Machinery, vol.37, no.4, p. 691-711.

[2] Sklower, K., "A tree-based packet routing table for Berkeley UNIX," Proceedings of the Winter 1991 USENIX Conference, p. 93-103, Dallas, TX, USA 21-25 Jan. 1991.

[3] Pei, T.-B.; Zukowski, C., "Putting routing tables in silicon," IEEE Network, vol.6, no.1, p. 42-50, Jan. 1992.

[4] McAuley, A.J.; Francis, P., "Fast routing table lookup using CAMs," IEEE INFOCOMM '93, p. 1382-91 vol.3, San Francisco, CA, USA 28 March-1 April 1993.

[5] Knox, D.; Panchanathan, S., "Parallel searching techniques for routing table lookup," IEEE INFOCOMM '93, p. 1400-5 vol.3, San Francisco, CA, USA 28 March-1 April 1993.

[6] Feldmeier, D.C., "Improving gateway performance with a routing-table cache," IEEE INFOCOMM '88, p. 298-307, New Orleans, LA, USA 27-31 March 1988.

[7] Chen, X., "Effect of caching on routing-table lookup in multimedia environments," IEEE INFOCOMM '91, p. 1228-36 vol.3, Bal Harbour, FL, USA 7-11 April 1991.

[8] Estrin, D.; Mitzel, D.J., "An assessment of state and lookup overhead in routers," IEEE INFOCOMM '92, p. 2332-42 vol.3, Florence, Italy 4-8 May 1992.

[9] Waldvogel, M.; Varghese, G.; Turner, J.; Plattner, B., "Scalable High Speed IP Routing Lookups," ACM SIGCOMM Computer Communication Review, Vol. 27, No. 4, p. 29-36, October 1997.

[10] Degermark, M.; Brodnik, A.; Carlsson, S.; Pink, S., "Small Forwarding tables for Fast Routing Lookups," ACM SIGCOMM Computer Communication Review, Vol. 27, No. 4, p. 3-14, October 1997.

[11] Doeringer, W.; Karjoth, G.; Nassehi, M., "Routing on Longest Matching Prefixes," IEEE Transactions on Networking, Vol.4, No.1, Feb.96.

[12] Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project. http://nic.merit.edu/ ipma.

[13] Gupta, P.; McKeown, N.; Lin, S., "Routing Lookups in Hardware at Memory Access Speeds," IEEE INFOCOMM, April 1998, San Francisco.