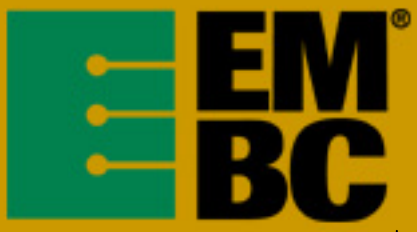

Networking

software benchmark data book

www.eembc.org

An Industry-Standard Benchmark Consortium

Table of Contents

Networking Version 2.0 Benchmark Name: IP Reassembly

Highlights

- **Based on NetBSD kernel code**

Application

When an IP packet is too large to fit within the Maximum Transfer Unit (MTU) of the egress interface, it can no longer be transmitted as a single frame. Rather, the IP packet must be fragmented and transmitted in multiple frames. Dealing with reassembly, the process of reassembling the IP fragments to form the original packet, can place significant resource requirements on systems.

Additionally, with the increasingly heterogeneous networking environment of LANs and WANs, fragmentation becomes increasingly likely.

Benchmark Description

Based on the NetBSD kernel code, the IP Reassembly benchmark simulates the processing performed to handle reassembly. The benchmark simulates the arrival of a large number of IP fragments of varying lengths – ranging from those requiring just a single mbuf to those requiring a cluster. The degree of fragmentation, the number of fragments per IP packet, the arrival order of the fragments, and the number of packets being reassembled in parallel is configured to represent different networking environments.

When a packet "arrives" it is checked for basic correctness. Its packet ID, source, and destination parameters are compared with those of all the packets waiting for reassembly. If the fragment corresponds to a new packet, a new queue is started to hold the additional fragments that will be required to reassemble the packet. If the fragment belongs to a packet reassembly effort already in progress, then the doubly linked list which forms the reassembly queue is traversed to determine where this fragment belongs in the packet. Each fragment contains offset information indicating its relative position to the start of the packet. The new fragment is then inserted into the linked list at the appropriate position, and because fragment overlap is possible, it may be necessary to trim (or even dequeue) adjacent fragments. A check is subsequently made for complete reassembly. If, with the addition of the current fragment, reassembly is complete, fragment concatenation is undertaken and the reassembled packet is passed up the stack.

Analysis of Computing Resources

1. Tests data caches – large memory requirements test cache size and replacement algorithms, while aggressive pointer chasing tests latency.
2. Linked list traversal tests processors' ability to perform loads and compares and stresses processors' branch prediction logic and ability to recover gracefully from misprediction.

Special Notes NetBSD is a secure and highly portable UNIX-like operating system available for many platforms, from high-end servers to embedded and handheld devices. The core routines in this benchmark are based on the networking functionality in the NetBSD operating system.

Optimizations Allowed

Out of the Box / Standard C
Full Fury / Optimized

- For Out of the Box, you may not change the algorithm nor the C code except for the Test Harness, or to get the code to compile. All compiler-related changes must be documented and must not have a performance impact.
- For Out of the Box, if the C compiler can schedule for any additional hardware without code changes, these are allowed. ASM statements are not allowed. All optimized libraries must be part of the standard compiler package, and/or available to all customers
- You may use Test Harness Regular or Test Harness Lite. You may not create your own Test Harness.
- For Optimized, you may re-write the basic algorithm as long as the output is unchanged.
- For Optimized, you may re-write the code in assembler.
- For Optimized, you may use publicly available optimized libraries, use hardware-assist if it is on the same processor as that being benchmarked, and/or inline the code.
- You may not assume that the data files shipped with the benchmark are the only data files that will be run through the certification run by EEMBC. However, the distributions will be as specified above.

An Industry Standard Benchmark Consortium

Networking Version 2.0

Benchmark Name: IP Packet Check

Highlights

- **Simulates a network router with four data sets to compare memory bus effects**
- **Focuses on checksum calculations and logical compare operations**

Application

The IP Packet Check benchmark performs a subset (essentially the IP Header Validation) of the network layer forwarding function of the Internet protocol suite as specified in RFC1812, "Requirements for IP Version 4 Routers" which can be found at http://www.faqs.org/rfcs/rfc1812.html. The benchmark provides an indication of the potential performance of a microprocessor in an IP router system.

A TCP/IP router normally examines the IP protocol header as part of the switching process. It generally removes the Link Layer header from a received message, modifies the IP header, and replaces the Link Layer header for retransmission. In this benchmark, the Link Layer header has already been removed and will not be replaced, i.e. all processing is done at Layer 3, on the assumption that lower level functions are handled by hardware or an interrupt service routine.

Benchmark Description

The benchmark simulates a router with four network interfaces. It initializes a buffer of programmable size (512 KB, 1 MB, 2 MB and 4 MB for reporting purposes) with IP datagrams. The header is always the minimum 20 bytes and is made up random characters except in the byte positions to be checked (IP version, checksum, and length). A checksum for the IP header is calculated and stored in each datagram. Errors are introduced in certain headers and an error count is logged. Datagrams are allowed to be aligned on the best natural boundary of the microprocessor and padding is added between them.

As a benchmark, the IP packet size is chosen randomly to be either 46 bytes (small packets) or 1500 bytes (large packets) in size. Packet receipt is simulated by creating a dummy store queue of 512 KB (approximately 370 packets), 1 MB (~720 packets), 2 MB (~1400 packets) or 4 MB (~2800 packets) outside of the timing loop. One timed iteration of the benchmark consists of processing each packet header pointed to by the receive queue and moving the descriptor to a holding queue. Results are reported in iterations per second for each of the buffer sizes but can be equated to packets per second by the conversion of ~660, ~1320, ~2640 or ~5280 packet headers checked per iteration, respectively.

Two descriptor queues are created with a pointer to the next descriptor and a pointer to the datagram header. One queue is called the receive queue (`rx_queue` in the code) and the other queue is the holding queue (`hold_que` in the code). IP datagrams are often stored like this in actual systems using descriptors that are separate from the datagram. A descriptor has a next

member that allows it to be put in a linked list and a pointer to a datagram.

Benchmark Description (continued)

As each datagram is processed by the benchmark algorithm it is removed from the receive queue and placed in the holding queue. Processing consists of:

1. Checking that the packet length is large enough to hold the minimum length legal IP datagram (>=20 bytes).
2. Checking that the IP checksum is correct (a bad packet counter is incremented if the checksum is not correct)
3. Checking that the IP version number is 4
4. Checking that the IP header length field is large enough to hold the minimum length legal IP datagram (20 bytes = 5 words)
5. Checking that the IP total length field is large enough to hold the IP datagram header, whose length is specified in the IP header length field

Cache route lookup, the routing decision, and test for local delivery, which would normally be a part of packet routing are not implemented in Version 1.0 of this benchmark. Cache route lookup, however, is implemented in the EEMBC Route Lookup benchmark and those results can be combined with IP Packet Check to get a better indication of microprocessor performance in an IP router system.

A single iteration of the benchmark is complete when the receive queue of packet descriptors is empty. At the end of one iteration, the receive queue and the holding queue are switched allowing the next iteration to execute with a full receive queue.

Analysis of Computing Resources

The IP Packet Check benchmark performs integer math on 16 bit unsigned quantities (the checksum calculation) and shift and logical compare operations (the IP version number and length checks). These operations and accessing the data from memory are primarily what is tested by this benchmark. Though the buffer sizes in memory are large, the checksum and verification process is only over the IP headers, which tend to take up residence in cache; therefore even at the largest buffer sizes, this benchmark has a high cache hit rate for microprocessors with 32KB of Data Cache. (The headers and packet descriptors for a 1MB buffer come very close to fitting in 32 KB of L1.) The code size is trivial and easily fits in even a small L1 Instruction Cache.

Special Notes

Do not directly compare the results of IP Packet Check benchmark to EEMBC Networking Version 1 Packet Flow benchmark. Even though the two benchmarks test the same function, the algorithm was changed in IP Packet Check to allow a user specified alignment without impacting the number packets processed.

Networking Version 2.0

Benchmark Name: IP Network Address Translator (NAT)

Highlights

- **Based on NetBSD kernel code**
- **Stresses data cache efficiency and latency**

Application

Basic Network Address Translation (NAT) is a method by which an Internet router maps IP addresses from one group to another, transparent to end users. NAT is traditionally required when a network's internal IP addresses cannot be used outside the network, either because they are not globally unique, or for privacy reasons.

A NAT router, residing on the border between two networks, translates the addresses in the IP headers so that when the packet leaves one network and enters another, it can be correctly routed. For egress packets, the source address is mapped to a globally unique external network address, while, for ingress packets, the destination address is mapped from the external address to the relevant address in the private network. IP header checksums (and UDP and TCP checksums if applicable) are also updated to reflect the address translation.

Benchmark Description

The dataset for the NAT benchmark focuses on the handling of egress packets. When a packet "arrives," initial processing ascertains what action, if any, needs to be undertaken. The NetBSD NAT implementation uses a 128-entry hash table to hold information about current connections. By using the source address, destination address, protocol, and ports (if applicable) of the packet, the system computes an offset into the hash table. If this entry in the hash table relates to the current packet, the packet belongs to a "connection" that is already established and the packet processing is undertaken as dictated by the NAT table entry. If the packet doesn't belong to a current connection, the list of NAT rules are searched to ascertain if a rule exists for the packet handling. If a rule exists for this "connection" (rules are specified during an initialization phase before the benchmark is started), the system creates an entry in the hash table for this "connection" to accelerate future handling of packets for this connection.

If the packet is determined to correspond to a NAT entry, the source address of the packet is altered as stipulated by the pertinent rule. The IP header checksum is then fixed to reflect this modification. Additionally, if the packet is a TCP packet, the TCP checksum is also updated to reflect the modification in source address. The translated packet is then sent onward.

Analysis of Computing Resources	Aggressive pointer chasing tests cache latency. Hash table searching tests processors' ability to perform loads and compares and stresses processors' branch prediction logic and ability to recover gracefully from misprediction.
Special Notes	P. Srisuresh, “IP Network Address Translator (NAT) Terminology and Considerations,” RFC2663, August 1999. P. Srisuresh et al, “Traditional IP Network Address Translator (Traditional NAT),” RFC3022, January 2001.

Networking Version 2.0 Benchmark Name: OSPF

Highlights

- **Benchmarks Potential Performance of Routers**

Application

The OSPF (Open Shortest Path First)/Dijkstra benchmark implements the Dijkstra shortest path first algorithm, which is widely used in routers and other networking equipment.

Benchmark Description

The Dijkstra algorithm finds the shortest, or least cost path, from a specific router (called the source) to all other routers that the source knows about. It builds a table of nodes where each node is a router. Each node has one or more “arcs” where each arc is a directed (one way) link to another node. These arcs represent links between routers. Each arc has a cost value that represents the 'value' of the link. The lower the cost number, the more desirable it is to use the link.

The Dijkstra algorithm starts at a source (or root) node. It then computes the best-case cost, or shortest route of all the other nodes in the network in relation to the source node.

There are two tables, `arc_base` and `node_base`. Each table is initialized before the benchmark starts and then reinitialized after each iteration of the benchmark, so that each iteration does exactly the same thing.

Instead of building a predefined route, the standard method in this benchmark builds the routing tables dynamically.

Analysis of Computing Resources

The benchmark repeatedly walks the list that is used to hold the nodes. Consequently, a processor's load-use latency and its ability to handle frequent CTI (control transfer instructions) operations are an important factor in this benchmark.

Special Notes

1. Do not directly compare Version 2.0 results to results of Version 1.The dataset in Version 2.0 has been significantly changed from the Version 1 implementation to improve the quality and real-world nature of this benchmark.

Networking Version 2.0 Benchmark Name: QOS

Highlights

- **Based on NetBSD kernel code**

Application

This benchmark simulates the processing undertaken by bandwidth management software used to “shape” traffic flows to meet Quality of Service (QoS) requirements. The system paces the delivery of the packets to the desired speed, based on a set of predefined rules. This shaping is achieved via the use of a variant of the Weighted Fair Queuing (WFQ) algorithm. Random Early Detection (RED) queue management is also supported to provide flow control.

Benchmark Description

The overall structure for the QoS system is as follows (largely based on documentation provided with the Dummynet QoS system):

In the QoS system, egress packets are selected based on rules established during the initialization phase, and passed to two different objects: “pipe” or “queue.”

A queue is just a queue with configurable size and queue management policy. It is also associated with a mask (to discriminate among different flows), a weight (used to give different shares of the bandwidth to different flows) and a pipe, which essentially supplies the transmit clock for all queues associated with that pipe.

A pipe emulates a fixed-bandwidth link, whose bandwidth is configurable. The "clock" for a pipe is incremented every iteration in the benchmark. A pipe is also associated with one (or more, if masks are used) queue, where all packets for that pipe are stored.

The bandwidth available on the pipe is shared by the queues associated with that pipe (only one in case the packet is sent to a pipe) according to the WF2Q+ scheduling algorithm and the configured weights.

Egress packets are stored in the appropriate queue, which is then placed into one of a few heaps managed by a scheduler to decide when the packet should be extracted. The scheduler is run once per iteration, and grabs queues from the head of the heaps when they are ready for processing.

There are three data structures defining a pipe and associated queues:

1. `dn_pipe`, which contains the main configuration parameters related to bandwidth
2. `dn_flow_set`, which contains WF2Q+ configuration information
3. `dn_flow_queue`, which is the per-flow queue (containing the packets)

Multiple `dn_flow_set` can be linked to the same pipe, and multiple `dn_flow_queue` can be linked to the same `dn_flow_set`. All data structures are linked in a linear list which is used for housekeeping purposes.

Benchmark Description (continued)

During configuration, the `dn_flow_set` and dn_pipe structures (a `dn_pipe` also contains a `dn_flow_set`) are created and initialized.

At runtime, packets are sent to the appropriate `dn_flow_set` (either WFQ ones, or the one embedded in the `dn_pipe` for fixed-rate flows), which in turn dispatches them to the appropriate `dn_flow_queue` (created dynamically according to the masks).

The transmit clock for fixed rate flows (`ready_event()`) selects the `dn_flow_queue` to be used to transmit the next packet. For WF2Q, `wfq_ready_event()` extracts a pipe which in turn selects the right flow using a number of heaps defined into the pipe itself.

The current dataset sends packets directly to a pipe and utilizes fixed rate flows.

Analysis of Computing Resources

QoS is a memory intensive benchmark – large memory requirements test data cache misses, while long sequences of dependent loads test memory latency.

Networking Version 2.0

Benchmark Name: Route Lookup

Highlights

- **Benchmarks Potential Performance of Routers**

Application	This benchmark is a distillation of the fundamental operation of IP datagram routers: receiving and forwarding IP datagrams.
Benchmark Description	All IP routers keep a table that allows it to lookup IP addresses and determine to which port an incoming IP datagram should be forwarded. This benchmark implements an IP lookup mechanism based on a Patricia Tree (or trie). The Patricia tree data structure is a type of binary, compact tree that allows fast and efficient searches with long or unbounded length strings. The number of search steps is bounded by the length of the search key e.g. 32-bit IPv4 addresses. The benchmark builds a tree from the IP address data supplied in *route.txt* (200 routes). After the tree is initialized, the benchmark calls the function *pat_search()* for each IP address in *lookups.txt* (2000 lookups). One pass through this data is regarded as a single interation.
Analysis of Computing Resources	The benchmark repeatedly walks through the trie. Consequently, a processor's load-use latency and its ability to efficiently handle frequent CTI (control transfer instructions) operations are an important factor in this benchmark.

Networking Version 2.0

Benchmark Name: Transmission Control Protocol (TCP)

Highlights

- **Captures most frequently used and most processing-intensive portion of RFC793 protocol**
- **Simulates TCP traffic characteristics in real networks**
- **De-couples processor speed from any randomness in TCP operation**
- **Uses three different data sets to simulate a variety of workloads**

Application

The ability of an embedded processor to handle Transmission Control Protocol (TCP) layer processing is an important consideration for avoiding bottlenecks in network equipment designs. Unlike ATM and some other network protocols that are mainly processed by network processors, ASICs, or specialized hardware blocks directly attached to general purpose processors, the TCP layer is often processed by the CPUs in general-purpose processors. The interest of benchmarking TCP performance on embedded general-purpose processors has increased with the connection of more and more embedded devices to the network. The flexibility of TCP is such that it is used in wireline and wireless applications.

The ISO reference model is commonly used when discussing protocol layering. This model depicts the TCP layer as sitting on top of the Internet Protocol (IP) layer and under the application layer. The function of IP is to provide a means of transferring TCP segments over inter-connected networks. IP has unique addressing information for each network element, and data communication is based on routing that provides best effort service to TCP and other transmission control layer protocols like UDP.

In contrast to IP, TCP service is a reliable, connection-oriented byte stream service. It typically interfaces with an unreliable network layer protocol. Unlike other connection-oriented protocols that are based on a reliable network layer, TCP has to implement a more complex transmission control scheme to overcome these seemingly contradictory philosophies between protocol layers.

The basic operation of TCP can be broken down into the following six areas:

1. Basic data transfer
2. Reliability
3. Flow control
4. Multiplexing
5. Connections
6. Precedence and security

Application

The core of the TCP protocol is to transfer data between two connection endpoints. Like data processing in most of the network protocols, large data blocks are chopped into optimized sizes (as deemed by TCP) and encapsulated in a TCP segment. Communications in TCP involve both data and control operations. Comparing data processing with other protocols, the biggest difference with TCP is a mandatory checksum across the entire segment. This is because TCP provides reliable communication service on top of an unreliable IP layer. For the same reason, TCP requires fairly complex control and signaling to achieve reliability, efficiency, and connection management. Compared with IP, data operations are simpler but are more expensive in terms of performance. The cost associated with the data block size is linear in most of the cases. (For example, computing IP style checksum and memory copy.) The benchmark captures all the costly data manipulations while some of the complex but rarely used control logic can be omitted.

Benchmark Description

This benchmark implementation captures the most frequently used and processing-intensive portion of the protocol described in RFC793. The benchmark measures the data and buffer management performance, which is common and expensive in TCP implementations. Also, because this benchmark targets embedded general-purpose processors, the execution environment should match code size and memory scale. Typically, execution environments include a reasonably-sized memory and high-performance RTOS with shared kernel and user addressing spaces. The scope of this benchmark does not include measuring overall network performance.

EEMBC's TCP benchmark follows these general requirement guidelines:

Accurate – the benchmark captures all major TCP operations in terms of processing cost

Realistic – the benchmark simulates TCP traffic characteristics in real networks

Deterministic – the benchmark de-couples processor speed from any randomness in TCP operation

Simplistic – the benchmark implementation allows for a simplified TCP implementation with reasonable assumptions

Application protocols that use bulk transfer contribute 90% of the traffic in terms of number of bytes but represent only about half the packets. The TCP benchmark is designed to be flexible enough to capture processor performance for both transfer types.

Benchmark performance metrics include:

1. Complete event-driven TCP state machine, connection management signaling
2. Transient behavior in short TCP conversations
3. Buffer management – Data manipulation in both ingress and egress directions
4. Queue management – Send queue, unacknowledged queues in egress direction
5. Separate re-entrant client-server task with context switching
6. Basic flow control
7. Multiple data stream (phase II)
8. Configurable packet size distribution for different traffic patterns

RFC793 requirements that are not included in benchmark include:

1. Real-time timer related – RTT estimation and update, RTO
2. Exception handling, out-of-order delivery, duplicates and lost packets

TCP behavior varies dramatically between different applications. Packet sizes, conversation length, and queue depth can all affect processing in different ways. To cope with different scenarios, the benchmark is configurable. In addition, the following four standard test cases were designed based on representative statistical data.

Parameters/Tests

The application data block processed between the client and server is constructed as a ring or circular buffer based on the segment size, and number of packets in the workload.

1. Bulk data transfer test uses maximum TCP segment size (which is typically used in FTP data channel)
2. Jumbo test uses MTU sizes, and numbers of packets applicable to a Gigabit Ethernet Backbone.
3. Mixed packet sizes is an average case Standard Ethernet – mixture of activity

Analysis of Computing Resources

Each workload simulates network traffic using the following steps:

1. Initiate server task.
2. Insert network channel effect
3. Initiate client task
4. Insert network channel operations of client.

This workload is repeated until all client connections are closed.

Networking Version 1.1

Benchmark Name: Packet Flow

Highlights

- **Simulates a network router with three data sets to compare memory bus effects**
- **Focuses on checksum calculations and logical compare operations**

Application

The Packet Flow benchmark performs a subset (essentially the IP Header Validation) of the network layer forwarding function of the Internet protocol suite as specified in RFC1812, "Requirements for IP Version 4 Routers" which can be found at http://www.faqs.org/rfcs/rfc1812.html. The benchmark provides an indication of the potential performance of a microprocessor in an IP router system.

A TCP/IP router normally examines the IP protocol header as part of the switching process. It generally removes the Link Layer header from a received message, modifies the IP header, and replaces the Link Layer header for retransmission. In this benchmark, the Link Layer header has already been removed and will not be replaced, i.e. all processing is done at Layer 3, on the assumption that lower level functions are handled by hardware or an interrupt service routine.

Benchmark Description

The benchmark simulates a router with four network interfaces. It initializes a buffer of programmable size (512KB, 1MB, and 2MB for reporting purposes) with IP datagrams. The header is always the minimum 20 bytes and is made up random characters except in the byte positions to be checked (IP version, checksum, and length). A checksum for the IP header is calculated and stored in each datagram. Errors are introduced in certain headers and an error count is logged. Datagrams are allowed to be aligned on the best natural boundary of the microprocessor and padding is added between them.

As a benchmark, the IP packet size is chosen randomly to be either 46 bits (small packets) or 1500 bits (large packets) in size. Packet receipt is simulated by creating a dummy store queue of 512KB (approximately 660 packets), 1MB (~1320 packets), or 2MB (~2640 packets) outside of the timing loop. One timed iteration of the benchmark consists of processing each packet header pointed to by the receive queue and moving the descriptor to a holding queue. Results are reported in iterations per second for each of the buffer sizes but can be equated to packets per second by the conversion of ~660, ~1320, or ~2640 packet headers checked per iteration, respectively.

Two descriptor queues are created with a pointer to the next descriptor and a pointer to the datagram header. One queue is called the receive queue (rx_queue in the code) and the other queue is the holding queue (hold_que in the code). IP datagrams are often stored like this in actual systems using descriptors that are separate from the datagram. A descriptor has a next member that allows it to be put in a linked list and a pointer to a datagram.

As each datagram is processed by the benchmark algorithm it is removed from the receive queue and placed in the holding queue. Processing consists of:

6. Checking that the packet length is large enough to hold the minimum length legal IP datagram (>=20 bytes).
7. Checking that the IP checksum is correct (a bad packet counter is incremented if the checksum is not correct)
8. Checking that the IP version number is 4
9. Checking that the IP header length field is large enough to hold the minimum length legal IP datagram (20 bytes = 5 words)
10. Checking that the IP total length field is large enough to hold the IP datagram header, whose length is specified in the IP header length field

Cache route lookup, the routing decision, and test for local delivery, which would normally be a part of packet routing are not implemented in Version 1.0 of this benchmark. Cache route lookup, however, is implemented in the EEMBC Route Lookup benchmark and those results can be combined with Packet Flow to get a better indication of microprocessor performance in an IP router system.

A single iteration of the benchmark is complete when the receive queue of packet descriptors is empty. At the end of one iteration, the receive queue and the holding queue are switched allowing the next iteration to execute with a full receive queue.

Analysis of Computing Resources

The Packet Flow benchmark performs integer math on 16 bit unsigned quantities (the checksum calculation) and shift and logical compare operations (the IP version number and length checks). These operations and accessing the data from memory are primarily what is tested by this benchmark. Though the buffer sizes in memory are large, the checksum and verification process is only over the IP headers, which tend to take up residence in cache; therefore even at the largest buffer sizes, this benchmark has a high cache hit rate for microprocessors with 32KB of Data Cache. (The headers and packet descriptors for a 1MB buffer come very close to fitting in 32KB of L1.) The code size is trivial and easily fits in even a small L1 Instruction Cache.

Networking Version 1.1 Benchmark Name: OSPF

Highlights

- **Benchmarks Potential Performance of Routers**

Application

The OSPF (Open Shortest Path First)/Dijkstra benchmark implements the Dijkstra shortest path first algorithm, which is widely used in routers and other networking equipment.

Benchmark Description

The Dijkstra algorithm finds the shortest, or least cost path, from a specific router (called the source) to all other routers that the source knows about. It builds a table of nodes where each node is a router. Each node has one or more “arcs” where each arc is a directed (one way) link to another node. These arcs represent links between routers. Each arc has a cost value that represents the 'value' of the link. The lower the cost number, the more desirable it is to use the link.

The Dijkstra algorithm starts at a source (or root) node. It then computes the best-case cost, or shortest route of all the other nodes in the network in relation to the source node.

There are two tables, *arc_base* and *node_base*. Each table is initialized before the benchmark starts and then reinitialized after each iteration of the benchmark, so that each iteration does exactly the same thing.

Instead of building a predefined route, the standard method in this benchmark builds the routing tables dynamically.

Analysis of Computing Resources

The benchmark repeatedly walks the list that is used to hold the nodes. Consequently, a processor's load-use latency and its ability to handle frequent CTI (control transfer instructions) operations are an important factor in this benchmark.

Networking Version 1.1

Benchmark Name: Route Lookup

Highlights

- **Benchmarks Potential Performance of Routers**

Application

This benchmark is a distillation of the fundamental operation of IP datagram routers: receiving and forwarding IP datagrams.

Benchmark Description

All IP routers keep a table that allows it to lookup IP addresses and determine to which port an incoming IP datagram should be forwarded. This benchmark implements an IP lookup mechanism based on a Patricia Tree (or trie). The Patricia tree data structure is a type of binary, compact tree that allows fast and efficient searches with long or unbounded length strings. The number of search steps is bounded by the length of the search key e.g. 32-bit IPv4 addresses.

The benchmark builds a tree from the IP address data supplied in *route.txt*. After the tree is initialized, the benchmark calls the function *pat_search()* for each IP address in *lookups.txt*. One pass through this data is regarded as a single interation.

Analysis of Computing Resources

The benchmark repeatedly walks through the trie. Consequently, a processor's load-use latency and its ability to efficiently handle frequent CTI (control transfer instructions) operations are an important factor in this benchmark.