Master thesis
Evaluation of an LC-trie algorithm for IP-address lookups

Majid Zandieh

December 1999

Uppsala University

Supervisor
Bengt Ahlgren
Swedish Institute of Computer Science, SICS
Box 1263, SE-164 29 Kista
Sweden

Examiner
Mats Björkman
Department of Computer Systems, DOCS
Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
Abstract

The growth of the Internet in recent years has led to an enormous increase of the number of routing table entries. Address tables in IP routers require efficient and compact implementation to allow fast lookup of IP addresses. One solution for fast address lookup in software is to use the LC-trie data structure. The search depth for the LC-trie increases slowly as function of the number of entries.

This master thesis discusses the performance of the fast address lookup in the LC-trie algorithm. The main focus of this master thesis is to use the instruction set simulator, SimICS for performance evaluation of the address lookup in the LC-trie algorithm. The address lookup is performed for 100000 addresses in a LC-trie. The results are measured in terms of number of memory accesses and number of executed instruction per address lookup.
Preface

This master thesis is the final part of my education at the Uppsala University that leads to the degree of Master of Science in Scientific Computing. The work was performed at the Swedish Institute of Computer Science, SICS, Sweden.

I would like to thank my supervisor at SICS Bengt Ahlgren whose assistance has been essential for this master thesis. I would also like to thank Prof. Gunnar Karlsson at Royal Institute of Technology, KTH, Peter Magnusson, Ian Marsh and other members of CNA lab who have supported me with their expertise and useful comments during the work with this master thesis.
# Contents

1 Introduction  
1.1 Background 1  
1.2 SimICS 2  
1.3 Aim of the master thesis 2  
1.4 Organization of the master thesis 2

2 Routing and address lookup  
2.1 Internetwork 3  
2.1.1 Routers in general 3  
2.1.2 Routing 3  
2.1.3 Routing protocols 4  
2.2 Routing table 4  
2.3 Packet forwarding 5  
2.3.1 Address Lookup 5  
2.3.2 Representation of forwarding table 6  
2.3.2.1 Path-compression 7  
2.3.2.2 Level-compression 8  
2.3.2.3 LC-trie algorithm 8  
2.3.2.4 Array representation of the LC-trie 9  
2.3.2.5 The Search operation in the LC-trie 10

3 Performance evaluation and tuning using SIMICS  
3.1 Instruction set simulation 12  
3.2 Starting SimICS for simulation 12  
3.2.1 Compiling of the source code 12  
3.2.2 Start plain SimICS 13  
3.2.3 GDB SimICS 13  
3.3 Loading extension and data caches in SimICS 13  
3.3.1 Loading extension 14  
3.3.2 Configuration of data caches (generic-cache) 14  
3.4 Loading and running the program 14  
3.5 How to use SimICS for performance debugging 15  
3.5.1 Performance analysis 15  
3.5.2 Disassembling instructions 16  
3.5.3 Processor statistics 17  
3.5.4 Control of the program execution 17

4 Implementation and performance analysis  
4.1 Preparing the source code 19  
4.1.1 compile the source code 20  
4.1.2 The Routing tables and the Traffic files 20  
4.1.3 Preparations for running SimICS 20  
4.2 Running SimICS for Performance debugging 21  
4.2.1 Finding the virtual addresses 21  
4.2.2 Setting break points 22  
4.3 Performance analysis 22  
4.3.1 Using the profiling statistics 25  
4.4 Result and conclusion 27
5 References

Appendix A  
Appendix B  
Appendix C
1 Introduction
In this chapter a brief overview of the bottleneck problem in the router and the LC-Trie algorithm [1] is given as well as a description of the aim of this thesis.

A router is a device that chooses different paths for the network packets, based on the addressing of the IP (Internet Protocol) frame it is handling. Different routes connect to different networks. The router will have more than one address, as each route is part of a different network. One of the most fundamental operations in a router is the routing table search process. The router must search a forwarding table using the IP destination address as its key, and determine which entry in the table represents the best route for the packet towards its destination.

IP addressing is based on the concept of hosts and networks. A host is essentially anything on the network that is capable of receiving and transmitting IP packets on the network, such as a workstation or a router. The hosts are connected together by one or more networks. An IP address is 32 bits wide. It is composed of two parts: the network number, and the host number. By convention, it is expressed as four decimal numbers separated by periods, such as "200.1.2.3" representing the decimal value of each of the four bytes. Valid addresses thus range from 0.0.0.0 to 255.255.255.255, a total of about 4.3 billion addresses.

1.1 Background
The growth of the Internet in recent years has led to an increase (for example 40K entries) of the number of routing table entries. Further on with upgrade from IPv4 to IPv6 will increase the size of the address field from 32 bits to 128 bits, with network prefixes up to 64 bits in length. And the ever-expanding number of networks and hosts on the Internet is pushing routing table sizes higher and higher. The address lookup must be performed fast even though the routing tables are large. The rapid growth of the Internet traffic as well demands higher performance of the network. The development of the transmission technology has moved the bottleneck in the network from the link to the routers where the address lookup in the forwarding table has the key role. There are different proposals in order to take care of this kind of problem e.g. by improving and developing algorithms implemented in software or hardware.

The performance and efficiency of a router depends to a large extent on the speed of the routing table address lookup. The LC-Trie algorithm used by G. Karlsson and S. Nilsson [1] is a recent technique that is able to support Gb/s throughput. A search operation is performed fast and efficiently in the LC-Trie algorithm, which is a result of the path compression (each internal node with only one child is removed) and level compression (replacing the $i$ highest complete levels of the binary trie with a single node of degree $2^i$) of the original binary trie.
Since the path compression shortens the search path and the level compression decreases the size of trie, the search operation is performed faster because the needed number of memory accesses is reduced for searching the forwarding table.
1.2 SimICS
SimICS [2] is a system level architecture simulator developed at SICS [3]. SimICS supports unix emulation and gathers statistics of instruction cache and execution profiling. SimICS is used in the performance analysis of the address lookup performed in the LC-Trie algorithm.

1.3 Aim of the master thesis
The goals of this master thesis are to evaluate the performance of the LC-Trie algorithm with respect to the number of instructions, memory references and cache behavior performed during address lookups, and to evaluate the possibilities to enhance the performance of the address lookup in the LC-Trie algorithm by using the results from the simulation.

1.4 Organisation of the master thesis
The rest of the Thesis is organised as follows. In chapter 2 the routing principles are explained. In chapter 3 an introduction about how SimICS is used as a performance debugger is given and finally in chapter 4 the results of the performance analysis and conclusions are discussed.
2 Routing and address lookup
The purpose of this section is to review some terms and principles of IP-routing [4], [5], [6], and discuss the LC-trie algorithm as a solution to the bottleneck problem in routers during address lookup.

2.1 Internetwork
An internetwork can be described as a number of different networks connected by several intermediate networking devices functioning as one large network. When implementing an internetwork, connectivity, reliability, network management and flexibility must be considered in order to establish an efficient internetwork. In Figure 1 an example of an internetwork is illustrated.

![Figure 1: An internetwork created by connecting different network technologies.](image)

2.1.1 Routers in general
A router handles the task of forwarding IP-packets and gathers information of the network topology. The topology of the network routing is discovered by using a routing protocol and the topology is used when the routing table is calculated. To determine the optimal path to a destination, routing-algorithm uses routing table. Every routing algorithm builds and maintains these tables with different route information. When an incoming packet arrives into a router the router checks the destination address of the packet and associates this address with next-hop address in the routing table; this operation is known as address lookup.

2.1.2 Routing
Routing is the selection of paths for packets. A router’s two main functions are determination of the optimal routing path and the transport of the packets through the network. There are several activities involved in these operations such as buffering, scheduling, switching and address lookup.
2.1.3 Routing protocols
Routers use routing protocols to maintain information about network topology. The most used protocols are Routing Information Protocol (RIP) and Open Shortest Path First (OSPF). RIP is the old routing protocol used in TCP/IP where the whole routing table is sent during a routing update. The new routing protocol used in the TCP/IP is OSPF, which sends only the last changes in the routing table. For routing between autonomous systems, an external routing protocol, like External Gateway Protocol (EGP) or Border Gateway Protocol (BGP) is used.

2.2 Routing table
The core of every router is the routing table. Routers do routing lookup in the routing table to determine the forwarding address, which results in the next-hop address on the path towards the destination. Each entry of the routing table for IP addresses has two fields: an address prefix and a next-hop address. The address prefix represents a group of addresses and consists of a network identifier field (an IP address) and a prefix length. It is not necessary that the network identifier should be the same for one address in all other routers. The next-hop field defines how the packet should be forwarded. It is necessary for a core router in the Internet to recognise all network identifiers, which is why the routing table in core routers has no default entry and it is the reason that the core routing table tends to be large. In case there is no matching prefix in the routing table most routers have a default route. This entry has a prefix of zero size for matching all addresses.
Figure 3: An example of a routing table for TCP/IP.

Figure 3 illustrated an example routing table. The prefix 193.52.7.0/24 in the table, which is a 24 bits long prefix, represents all IP addresses with the first 24 bits equal 193.52.7.0. Packets with a destination address matching this prefix are routed next to 192.7.2.1.

2.3 Packet forwarding

The process that moves packets from the incoming port to the outgoing port of a router is called forwarding (see Figure 4). This process consults the forwarding table information, which is an optimised representation of the routing table used for the actual address lookup. The performance of the router depends to a large extent on how fast it does the address lookup during the forwarding process.

Figure 4: Forwarding table and routing table in the IP-packet forwarding.

2.3.1 Address Lookup

Address lookup is based on the longest prefix match in the forwarding table. In the forwarding table prefixes are network identifiers which are stored in binary strings that has a variable length from 8 to 32 bits in IPv4. The result of this operation is a next-hop address that should be used for the packet. The next-hop address is used to find the physical-link address for the next downstream router when the interconnection is via a shared-medium network. Address lookup algorithms are usually organised and based on either hash functions or tree techniques.
2.3.2 Representation of forwarding table

The network prefix data or other variable length binary data is presented by a trie. A trie is a tree data structure, where each binary string or elements (Figure 5.a), is represented by a leaf in a tree structure. The value of the string corresponds to the path from the root of the tree to the leaf (Figure 5.b). A left branch denotes 0 and a right branch denotes 1. When increasing the number of the nodes this representation become inefficient as the trie needs a large memory space and the average depth of the tree increases linear as a function of the string size, which causes a longer search time in the trie.

**Figure 5.a**

<table>
<thead>
<tr>
<th>nbr</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>00101</td>
</tr>
<tr>
<td>3</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
</tr>
<tr>
<td>5</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>10100</td>
</tr>
<tr>
<td>8</td>
<td>101001</td>
</tr>
<tr>
<td>9</td>
<td>10101</td>
</tr>
<tr>
<td>10</td>
<td>10110</td>
</tr>
<tr>
<td>11</td>
<td>10111</td>
</tr>
<tr>
<td>12</td>
<td>110</td>
</tr>
<tr>
<td>13</td>
<td>1101000</td>
</tr>
<tr>
<td>14</td>
<td>1101001</td>
</tr>
</tbody>
</table>

**Figure 5.b**: Binary tree representation (breadth first order).

There are several methods to solve these problems such as AVL-tree, balanced tree or LC-trie techniques. The last mentioned method, the LC-trie algorithm, modifies the binary tree by path- and level compression to a more compact trie with fewer levels and a more space efficient structure.

The main purpose of this algorithm is to make the routing table as small as possible, which should make it possible to take advantages of faster caching techniques. It is desirable in order to develop a space efficient structure for representation of the forwarding table, Figure 6, which leads to less and faster memory accesses during lookups i.e. fast address lookup.
Cache with the last recently used routes speeds up the address lookup.

2.3.2.1 Path-compression
The path-compression technique shrinks the average depth of the trie. When using the path-compression method each internal node with only one child is removed, i.e. *sparsely populated* parts of the trie are compressed. The number of bits that have been skipped on each path is stored as the skip value in the corresponding node. The total number of nodes in a path-compressed binary trie is exactly $2n-1$, where $n$ is the total number of leaves in the trie. The path-compressed binary tree, Figure 7, also known as the Patricia tree, is a well-known method to decrease the search cost [7] in the binary trie.

The significant effect of the path-compressed binary trie is the overall size reduction.

**Figure 6**: Cache with the last recently used routes speeds up the address lookup.

**Figure 7**: The path-compressed trie of the binary trie showed in Figure 6.
2.3.2.2 Level-compression
The second compression technique used in the LC-trie data structure is Level compression. Level compression makes it possible to compress the most densely populated parts of the trie and decrease the size of the Patricia trie. The idea is that on each subtrie replace recursively the $i$ highest complete levels of the binary trie with a single node of degree $2^i$ [1]. In Figure 8 the level-compressed trie is shown. The compressed levels are marked by shadowed rectangles in Figure 7.

![Figure 8: The level-compressed trie of the trie showed in Figure 7.](image)

A level-compressed trie, LC-trie, is a multi-digit [8] trie with following properties:
- the degree of the root is $2^i$, where $i$ is the smallest number such that at least one of the children becomes a leaf;
- each child is a level-compressed trie [8].

If the $i$ highest levels of the trie are complete but level $i+1$ is not complete, the $i$ highest levels are replaced by a single node of degree $2^i$ in a top down operation.

The expected average [8] of the depth of a LC-trie for an independent random sample with a density function that is bounded from above and below is:

$$
\begin{align*}
\Theta(\log^* n) & \text{ if } n > 1 \\
0 & \text{ else }
\end{align*}
$$

$log^* n = 1 + log^*(log n)$, $log^* 1 = 1$.

2.3.2.3 LC-trie algorithm
The forwarding table consists of:
- The LC-trie structure
- The base vector
- The next-hop table
- The prefix vector

The LC-trie structure is represented by an array. Each entry in the array represents a node in the trie. Each external node (leaf) of the LC-trie contains pointers into a base vector.

The base vector is the largest part in this structure and contains all complete strings (string size = 32-bits). Each entry in the base vector contains complete strings, one pointer to the next-hop table and one pointer to the prefix table. The next-hop-table is
an array where all possible next-hop addresses are stored. The prefix table contains information about strings that are proper prefixes of other strings, and the reason why the prefix table is needed is that at the internal nodes of the LC-trie do not contain pointers to the base vector. As a result of optimizing the trie some of the information in the trie is removed. But the search operation needs to compare the search IP-address (key) with complete IP-addresses somehow. The complete addresses are stored in a base vector and from each external node (leaf) there is a pointer into this vector. Each entry in the prefix table contains a number that indicates the length of the prefix; this number as in the base vector is not necessarily stored explicitly.

The base vector is used in the first step of the search operation. If the search address is found in the base vector table the corresponding next-hop address is used. If a match does not occur during the first step the information in the prefix table is used and the search routine checks the entries in the prefix table for a less specific match.

2.3.2.4 Array representation of the LC-trie
The array representation of the LC-trie (Figure 8) is showed in Figure 10. Using consecutive memory is a way to reduce the size of the data structure. Each node, Figure 9, which is 32 bits, is stored in an array (Figure 10), which makes it possible to use only one pointer to the leftmost child instead of using a set of children pointers in each node. Each node is represented by three numbers: the first 5-bits represent the branching factor, the next 7-bits the skip value and the last 20-bits is a pointer to the leftmost child node in the trie.

![Array representation of the LC-trie](image)

Figure 9: The LC-trie node with 32 bits.

The branching factor, \( k \), (the number of the descendants of the node) is a number of power of 2, \( 2^k \), where \( k \) by using 5-bits can represent the maximum branching of \( 2^{31} = 2.147483\times10^9 \). The skip value (7 bits) is the number of skipped bits at the node that represents values in the range from 0 to 127. The pointer to the leftmost child (20 bits) makes it possible to store at least \( 2^{19} = 524288 \) strings.
As an example of an array representation of the LC-trie, when traversing the LC-trie in breadth first order, the root of the LC-trie (Figure 8), node number zero, is stored at the entry number zero in the array (Figure 10). The root node has $8 = 2^3$ descendants or branches which means the branching factor is $3, k=3$. The skip value at root node is 0. The pointer at this node points to the leftmost child, because the leftmost child is an internal node, which is node number one (the branching factor $k \geq 1$).

Entry number 1 contains node number 1. This node in the LC-trie has $2 = 2^1$ branches which means that the branching factor is $k=1$. The skip value is zero at this node. The pointer points to the leftmost child (k=1) which is the internal node number 9.

The number of bits that should be skipped during a search operation.

If the node is internal, $k \geq 1$: A pointer to the leftmost child.
If the node is a leaf, $k=0$: A pointer to the base vector.

Figure 10: The Array representation of the LC-trie in Figure 8, where each entry represents a node. (k= branching factor).

As an example of an array representation of the LC-trie, when traversing the LC-trie in breadth first order, the root of the LC-trie (Figure 8), node number zero, is stored at the entry number zero in the array (Figure 10). The root node has $8 = 2^3$ descendants or branches which means the branching factor is $3, k=3$. The skip value at root node is 0. The pointer at this node points to the leftmost child, because the leftmost child is an internal node, which is node number one (the branching factor $k \geq 1$).

Entry number 1 contains node number 1. This node in the LC-trie has $2 = 2^1$ branches which means that the branching factor is $k=1$. The skip value is zero at this node. The pointer points to the leftmost child (k=1) which is the internal node number 9.

The entry number 9 in the array contains node number 9 which is a leaf. The branching factor at a leaf is zero. The skip value at node number 9 is zero. Because the node is a leaf (k=0) the pointer points to the base vector where the string 0 is stored.

2.3.2.5 The Search operation in the LC-trie
Let S be the binary string searched for and let EXTRACT (S, k, m) be a function that returns the number given by the m-bits starting at position k in S.

The tree is represented by an array T[i].

Step 1- Start the search at the root node in the tree, root = T[0].
Step 2- Skip “skip value”-bits in the search key.
Step 3- If the node is a leaf (branching factor = 0) then the corresponding pointer points at the base-vector which contains the complete string and denotes the address. Else extract k-bits (the branch factor) from the search key S, and add the value of these bits to the search key pointer and then go to the new entry and continue with Step 2.
Step 4- Compare the found key with the search key. If they match, return the next-hop address, else use the prefix vector for a less specific match and then go to the next-hop vector.

If a match occur one memory lookup is needed for every node traversed (level), and two additional memory accesses for the base vector and the next-hop (considering the size of the next-hop table the lookup in this table is fast). But if the searched string is not found in the base vector table one additional memory lookup is performed in the prefix table.
3 Performance evaluation and tuning using SIMICS

The purpose of this section is to explain how to use SimICS when studying address lookup in the LC-trie algorithm.

3.1 Instruction set simulation

Instruction set simulation is a powerful tool for performance debugging and analysis of programs in different environments. An instruction set simulator runs the programs by simulating the effect of each instruction on a target machine, one instruction at a time, which is also called program-driven simulation.

Each performed address lookup needs a number of memory references. The memory access is generally one of the most important time-consuming operations. Therefore it is necessary to study how the address lookup in LC-Trie algorithm uses the available environment. In the performance analysis the behaviour of the instruction cache miss and hit rates and the translation look aside buffers are significant. SimICS is the available and suitable tool for this performance analysis.

SimICS is an instruction-set simulator developed at the Swedish Institute of Computer Science (SICS). SimICS is able to support one or multiple SPARCv8 processors, physical address spaces, system level calls and emulation of the SunOS 5.x operating system for direct analysis of user-level programs. SimICS enables the programmer to analyse both debugging and performance profiling i.e. it can profile data and instruction cache misses, translation look-aside buffer misses (TLB), virtual memory events and instruction counts. SimICS emulates the SunOS 5.x kernel by explicitly emulating the program’s system calls, which includes support for multitasking as well as multiprocessing. This Unix emulation mode can be disabled, in which case SimICS will emulate the target machine at the system architecture level (sun4m) allowing operating system code to run unmodified. The core of SimICS is a threaded-code interpreter that executes programs by running a central fetch-decode-execution loop. SimICS interface is command-line oriented and by using it as a back-end to GDB provides a source code debugging environment.

3.2 Starting SimICS for simulation

3.2.1 Compiling of the source code

The GCC compiler was used to compile the source code. Three important tasks to perform before compilation of source code are:

1- Choose static linking
2- Set the optimisation level
3- Set the debugging flag
3.2.2 Start plain SimICS
SimICS is started with the "simics" command. It is possible to start SimICS in command line mode or script mode.

```
1001 scheutz $ simics

+------------+      Copyright 1998 by Virtutech, All Rights Reserved
| Virtutech   |      Copyright 1991-1997 by SICS, All Rights Reserved
| SimICS/V8   |      Version: Alpha .93 (Mon Dec 14 13:22:00 CET 1998)
+------------+      Variant: (TRANS) (GCC 2.7)
www.simics.com      Processor: 'Sparc V8 (v1.0)'

Type 'license' for details on warranty, copying, etc.
Type 'readme' for further information about this version.

SimICS log file opened as '.simics-log'
SimICS>
```

Figure 11: The information generated when SimICS is started.

Each time we run SimICS a log file “.simics-log” is generated by SimICS as well as the normal output from the program (Figure 11). The "simics-log" contains all the commands given to SimICS during the runtime and can be used as a script file to SimICS. The default name of the script file is “simics” and if the script file already exists SimICS reads this file. The “-n” flag after the start command "simics" tells SimICS to ignore the default script file "simics". The "-x" flag enables SimICS to start with a different script file than the default script file.

3.2.3 GDB SimICS
The SimICS distribution includes a modified version of GDB, the GNU debugger, called "gdb-simics" which can support running SimICS as a back-end. The modified GDB is able to run as a front-end to SimICS. Any command that GDB does not understand is passed to SimICS. GDB is run as a front-end to SimICS by using the command “gdb-simics”. In this mode SimICS reads the script file "gdb-simics" instead of "simics". The target is chosen by the command “target simics”. The “target simics” command tells GDB to start a background SimICS process. The communication between GDB and the SimICS process (SimICS backend) is done via a pipe. By using "sim" before a command the user can ensure that SimICS will handle this command.

3.3 Loading extensions and data caches in SimICS
SimICS can be extended at run-time by using the “load-object <extension>” command. This command uses the Solaris 2.x support for dynamically loadable modules. Examples of modules that can be loaded into SimICS are sunos, new cache hierarchies (super-sparc or generic-cache), TLB simulator and devices.
3.3.1 Loading extensions
For running SimICS in user-mode the command “load-object sunos” is used. This extension provides emulation for the SunOS 5.x binary interface and allows normal Solaris binaries to be run directly on SimICS.
In the SimICS distribution two memory hierarchy extensions are included: supersparc and generic-cache. The “supersparc” extension provides simulation of on-chip data and instruction caches. The Super SPARC chip has the following cache configuration:

- 16 kbyte data \(\Rightarrow\) 32-byte lines, 4-way set associative.
- 20 kbyte instruction \(\Rightarrow\) 64-byte lines, 5-way associative.

This cache extension supports a uniprocessor and does not simulate coherency. With the command: “load-object supersparc” this hierarchy is simulated.
The default cache hierarchy in SimICS is "generic-cache", which provides unified (data and instruction cache) support. This data cache supports multiple processors and is easy to configure.

3.3.2 Configuration of data caches (generic-cache)
The data cache can be configured dynamically by setting the following cache parameters: associativity, line sizes, number of lines and miss penalty values.
For example set the generic cache parameters to:

```
$simcacheassoc = 1
$simcachelinecount = 16384
$simcachelinesize = 64
```

Figure 12: SimICS commands.

Then initiate this configuration by the “init” command and it is possible to simulate a 1 Mbyte direct-mapped unified cache with 64-byte lines.

3.4 Loading and running the program
The “load-unix” command fetches a program binary into the memory. The command "load-unix" followed by the program name and arguments list loads a program and its arguments into the simulated memory. The user should remember to define all arguments to the program in closed quotation marks: "load-unix <"program binary name""> <"argument 1 argument 2...">.
For example load-unix "trietest" "routing-table traffic-file", if the marks are missing SimICS will not be able to read the arguments routing-table and traffic-file.

SimICS is a system-level simulator, which means it is able to run multiple processes simultaneously. The system call "_exit" forces SimICS to clean up after the process and restore the allocated memory regions for the corresponding process. Used by its own the exit call will result in SimICS losing the statistics, so before the system call "exit" is reached the execution should be stopped by setting a breakpoint with the "sysbreak" command. The simulation of the program in SimICS is run with the "c"
command which is an abbreviation for "continue". By the "sim help <argument>" command the user can be sure that the command is passed to SimICS. In case the "help <argument>" command is used the command is passed to the debugger and then if it fails the command is passed to SimICS.

3.5 How to use SimICS for performance debugging

In order to improve a program the first goal in performance debugging is to locate the most time consuming part or parts of the program. The instruction cache hit/miss ratio and the translation look-aside buffer are the most important events to examine in performance debugging.

3.5.1 Performance analysis

A useful command for performance analysis is "prof-weight", which gives statistics such as instruction cache hit and miss ratio, and the tlb-misses for the most expensive parts of the code. Further more the "prof-weight" command gives the physical and virtual addresses of these events which is used as a map for performance debugging. Before using this command the weight parameter should be set. The command "prof-info" shows the weight parameters, which are different in different cache hierarchies. For example we can get information about the weight parameters which are active in different cache hierarchies as is illustrated in Figure 13 and Figure 14 by using SimICS command "prof-info". The number of active profilers in the super-sparc is 8 and in generic-cache is 4.

### Figure 13: The command prof-info is used for a list of active profilers. Each column explains the active profiler and the corresponding weight parameter.

| Column 1: Instruction cache misses caused by program line ($SIM_SS_INSTR_MISS_WEIGHT = 0.0000) |
| Column 2: Cache misses (writes) caused by program line ($SIM_SS_WRITE_MISS_WEIGHT = 0.0000) |
| Column 3: Cache misses (reads) caused by program line ($SIM_SS_READ_MISS_WEIGHT = 0.000000) |
| Column 4: TLB misses passed on to Unix emulation ($SIM_TLB_MISS_WEIGHT = 0.000000) |
| Column 5: Number of (taken) branches *to* the code block ($SIM_TO_WEIGHT = 0.000000) |
| Column 6: Number of (taken) branches *from* the code block ($SIM_FROM_WEIGHT = 0.000000) |
| Column 7: Count of instruction execution (based on branch arcs) ($SIM_PC_WEIGHT = 0.000000) |
| Column 8: Number of addresses from which instructions have been fetched ($SIM_INSTR_WEIGHT = 0.000000) |

### Figure 14: The number of profiler is 4 in generic-cache.
For example to obtain statistics of the write miss rate of the instruction cache the corresponding parameter must be set as is illustrated in Figure 15.

```
(gdb-simics) SSIM_SS_INSTR_MISS_WEIGHT =1
-> 1
```

**Figure 15:** The weight assigned to each profiler value is set by environment variables. In this example weight value is set to 1.

The “prof-weight < block size > < top count >” command has two arguments; the block size and the top count. The <top count> parameter is the number of memory blocks to list, default top count is set to 10. The optional <block size> parameter is the chunk size over which to aggregate values, the default value is set to 4.

In the example shown in Figure 16 the profiling statistics of the top 5 blocks, each of size 64 bytes is generated. In this profiling result the number of the most instruction cache misses and where (the physical and virtual addresses) they occurred is explained.

```
(gdb-simics) prof-weight 64 5
Weighted profiling results:

<table>
<thead>
<tr>
<th>Physical</th>
<th>Virtual</th>
<th>( source )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00004600 0x00010600 (pid 1001)</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>0x00004900 0x00010900 (pid 1001)</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>0x00004b00 0x00010b00 (pid 1001)</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>0x00004d00 0x00010d00 (pid 1001)</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>0x00004f00 0x00010f00 (pid 1001)</td>
<td>2.00</td>
<td></td>
</tr>
</tbody>
</table>

Sum: 10.00 (2%)
Not shown: 450.00 (98%)
System total: 460.00
```

**Figure 16:** The result of the profiling for instruction cache misses.

The marked column in Figure 16 shows the number of instructions cache misses which occurred at each block; in this example they are 2. Each block is distinguished by different physical and virtual address intervals. Totally in the top 5 blocks 10 instruction have misses occurred which is only 2% of the totally 460 instruction misses. The 450 or 98% of instruction misses are not shown in this example of profiling.

### 3.5.2 Disassembling instructions

As described in 3.5.1 a map is established of the memory in order of the top five blocks where the most instruction misses occur. By using the address information (Figure 16) it is possible to look closer at each memory block. By using the “x <virtual address>” command it is possible to disassemble the contents of these memory blocks. The correct “x” command syntax in SimICS is “… s x <virtual address>”, whilst in GDB the syntax it is “x /<8i> <virtual address>”.

16
The pipe communication between SimICS-backend and GDB might get out of synch and to remedy this problem use the "flush" command is shown in Figure 17.

```plaintext
(gdb-simics) help flush
Try to clean up connection with SimICS.
SimICS and GDB communicate over pipes (with SimICS started with the ‘-backend’ flag). Also, ctrl-c (interrupt) is passed along via a memory-mapped file. This asynchronous setup sometimes causes either SimICS or GDB to be confused. The ‘flush’ command does various things in an attempt to clean up the communication. If you ever notice gdb-simics printing strange things, such as incomplete output from SimICS commands, then try 'flush'. Note that 'flush' is *always* harmless, so try it whenever something strange happens.
(gdb-simics)
```

**Figure 17:** Flush is used to avoid trouble between SimICS and GDB

The other commands, which are used for disassembling, are “list<argument>” and “list-det<argument>” commands. The argument that follows these commands is a virtual address, a line number interval or a function name. The “list-det” command is more useful and flexible than the “list” command because the information generated by the “list-det” command covers a wider area of information and provide the same information as ”list” command plus we are able to see the source code.

### 3.5.3 Processor statistics

The “pstats<cpu number>” or more exactly "print-statistics" command is an important command which produces various and useful statistics of the simulation. If no argument is given, general statistics about the current CPU is printed. Statistics such as instruction cache hit and miss rates, tlb miss rate and number of instructions executed by the program which are usable in the performance analysis.

### 3.5.4 Control of the program execution

Break points can be set to control the program execution in a particular part of the program. The particular line number of the program and the corresponding virtual address must also be known. The needed information is obtained in several steps. The command "list<function name>" serves as a guide to find the program line in the program. The command: “list-det<program line interval>” is used to find out the virtual address of the program line.

In SimICS a breakpoint is set after a certain numbers of executed instructions by “sim-break<number of instruction>”. Another possibility is to set a watch-point. SimICS supports breakpoints with the more general watch points for any combination of the operations read, write or instruction fetch for any set of memory addresses, Figure 18. To set a watch-point the "watchpoint<address><length><rwx>" command is used.
'wp' is an alias for 'watchpoint':
Add memory watchpoint on virtual address <argument>
Usage: watchpoint <address> [length [r][w][x]]
Adds a breakpoint on memory accesses (reads, writes or execute) to the
specified address. <i>length</i> defaults to 4. Once inserted, a watchpoint
will cause execution to stop immediately prior to any (program) access that
touches the watched memory (be it reading, writing, or executing).
Default effect is to break on all memory accesses. You can optionally specify
a subset, by adding any combination of "r", "w", and "x" for Read, Write, and
Execute operations. Thus, "wx" adds watchpoint for writes and execute only.

**Figure 18:** The SimICS manual provides a detailed information for each command.
For example the watch-point “wp 0x12794 4 x” command adds a watch-point for
execution on the actual address. By using the ‘watch-point-info’ command, Figure
19, it is possible to get a list of the watch-points and their properties.

```
(gdb-simics) watchpoint-info
Memory watchpoints (including breakpoints) for node 0:
Reads (physical addresses):
Writes (physical addresses):
Execute (physical addresses): 0x00006774 - 0x00006777
```

**Figure 19:** The watch point information.
4 Implementation and Performance analysis

This section describes how SimICS was used in the performance debugging of the LC-trie address lookup program. The purpose of this analysis was to find out how efficient the address lookup in the LC-trie data structure is performed and calculate the required number of memory accesses for each address lookup. Using SimICS allows the number of instructions and memory lookups to be calculated for each address lookup. Additionally SimICS can produce statistics for the number of accesses to the data and instruction cache.

4.1 Preparing the source code

The LC-trie method is implemented by Gunnar Karlsson and Stefan Nilsson [1]. The source code [1] is implemented in C and it is made available for the public by the authors. Before using the LC-trie program in SimICS it was necessary to modify the program. Those parts of the program that did not participate in the address lookup and had other functions in the program were removed. To make it possible to use the program for measurement it was necessary to find the particular part of the program where the address lookup is performed. The following code in Figure 20 belongs to the part of the program where the address lookup is performed in the LC-trie.

```c
/********** search **********/
  s = testdata[k];
  node = table->trie[0];
  pos = GETSKIP(node);
  branch = GETBRANCH(node);
  adr = GETADR(node);
  while (branch != 0) {
    node = table->trie[adr + EXTRACT(pos, branch, s)];
    pos += branch + GETSKIP(node);
    branch = GETBRANCH(node);
    adr = GETADR(node);
  }
  /* was this a hit? */
  bitmask = table->base[adr].str ^ s;
  if (EXTRACT(0, table->base[adr].len, bitmask) == 0) {
    res = table->nexthop[table->base[adr].nexthop];
    goto end;
  }
  /* if not look in the prefix tree */
  preadr = table->base[adr].pre;
  while (preadr != NOPRE) {
    if (EXTRACT(0, table->pre[preadr].len, bitmask) == 0) {
      res = table->nexthop[table->pre[preadr].nexthop];
      goto end;
    }
    preadr = table->pre[preadr].pre;
  }
  res = 0; /* not found */
end:
/********** End search **********/
```

**Figure 20:** The part of the source code where the address lookup is performed.
4.1.1 compile the source code
For compiling the program the GNU compiler, GCC, was used. The source files (qsort.c clock.c trie.c triestest.c Good_32bit_Rand.c) was included. Two flags were set: The optimisation flag, which was set at level 4, and the debugging flag.

4.1.2 The Routing tables and the Traffic files
The routing tables used here, “FUNET, MaeEast and MaeWest”, Figure 21, are the same routing tables used by **Gunmar Karlsson** and **Stefan Nilsson** in their work [1].

<table>
<thead>
<tr>
<th>Site</th>
<th>Routing entries</th>
<th>Next-hops</th>
<th>Number of entries</th>
<th>Av. depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Trie</td>
<td>Base</td>
</tr>
<tr>
<td>FUNET</td>
<td>41578</td>
<td>20</td>
<td>128865</td>
<td>39765</td>
</tr>
<tr>
<td>Mae East</td>
<td>38367</td>
<td>59</td>
<td>114319</td>
<td>36859</td>
</tr>
<tr>
<td>Mae West</td>
<td>15022</td>
<td>57</td>
<td>81817</td>
<td>14621</td>
</tr>
</tbody>
</table>

**Figure 21:** The LC-trie statistic for different routing tables.

Since the actual traffic corresponding to these tables is not available, the traffic is permuted randomly by using the existing entries in the actual routing. These traffic files contain 100000 IP-addresses.

4.1.3 Preparations for running SimICS
In case “gdb-simics” is used for access to SimICS it is necessary to choose SimICS as a target for GDB by the “target simics” command which instructs GDB to start a background SimICS process. The extension module, sunos, is loaded by the “load-object sunos” command. The cache hierarchy “ssparc-cache” is chosen which is more suitable for this work because simulation of on-chip data and instruction caches are needed (see and compare in Figure 13 and Figure 14). To load the object code, “triestest”, into simulated memory the “load-unix triestest "funet.table"” command is used. The “funet.table” i.e. the routing file containing a description of an IPv4 routing table was given as parameter to “load-unix”. Each line of the file contains three numbers: bits, len and next in decimal notation. Bits is the bit-pattern, len is the length of the entry and next is the corresponding next-hop address. To prevent memory reset it is necessary to stop the system call “exit” by “sysbreak _exit” command before starting the simulation, else the profiling information is lost before we can use it. Figure 22 shows the commands, which are used for running gdb-simics.

```plaintext
(gdb-simics) target simics
(gdb-simics) load-object sunos
(gdb-simics) load-object ssparc-cache
(gdb-simics) load-unix "triestest" "funet.table"
(gdb-simics) sysbreak _exit
```

**Figure 22:** The needed command for starting the simulation.
4.2 Running SimICS for Performance debugging

At this point it is possible to run gdb-simics by using the “c” i.e. the continue command. The problem here was that it is not possible to reset the profiling information (not in this version). It is desirable to restore profiling information for partial profiling of the source code. Because of this restriction in this version of SimICS the user should isolate the part of the source code they are interested in. Profiling data is collected before and after the address lookup has taken place and a difference in the gathered statistics is calculated.

4.2.1 Finding the virtual addresses

For setting breakpoints or watch points the virtual addresses of the particular parts of the source code is needed. The profile information in SimICS is kept on an assembler-line granularity and for providing more detailed information we can disassemble the code or use the GBD command “list-det”. Here the “list-det” command is used to find out the virtual addresses of those lines in the source code situated before and after the lookup operation is performed. By using the “list-det 280,285” command, which results in the following SimICS output, it is possible to find the virtual addresses for the corresponding program lines of interest for this analysis. The result shown in Figure 23 gives us the needed virtual addresses for setting break points.

```
(gdb-simics) list-det 280,285
280
281       // fprintf(stderr, "Function search START\n");
282
283          0 0 0 0 1 8 8  run(testdata, ntraffic, repeat,
284          table, FALSE, 1, verbose);
285 0x12774 [0x00006774]:  0 0 0 0 1 0 0 1 st %i3, [ %sp + 0x5c ]
286 0x12778 [0x00006778]:  0 0 0 0 0 0 1 1 mov %i1, %o0
287 0x1277c [0x0000677c]:  0 0 0 0 0 0 0 1 1 mov %l0, %o1
288 0x12780 [0x00006780]:  0 0 0 0 0 0 0 1 1 mov %l6, %o2
289 0x12784 [0x00006784]:  0 0 0 0 0 0 0 0 1 1 mov %l2, %o3
290 0x12788 [0x00006788]:  0 0 0 0 0 0 0 1 1 call 0x121c8 [0x0000678c] <run>
291 0x1278c [0x0000678c]:  0 0 0 0 0 0 0 1 1 call 0x121c8 [0x00006790] <run>
292 0x12790 [0x00006790]:  0 0 0 0 0 0 0 1 1 mov 1, %o5
293 0x12794 [0x00006794]:  0 0 0 0 1 0 0 5 fprintf(stderr, "Function search END\n");
294 0x12798 [0x00006798]:  0 1 0 1 0 0 0 0 1 1 sethi %hi(0x45c00), %o0
295 0x1279c [0x0000679c]:  0 0 0 0 0 0 0 1 or %o6, 0x210, %o0 ! 0x45e10
296 0x127a0 [0x000067a0]: 
297 0x127a4 [0x000067a4]: 
298
```

Figure 23: The result provides the needed virtual addresses.

At the line 282 (Figure 23) the first eight columns contains the total profiling statistic (the "prof-info" describes these columns, see even chapter 3) for the following C code in the source code. The following eight rows (until line 283) show the same statistic as in line 282 but for every single instruction plus the virtual and the physical addresses.
4.2.2 Setting break points
By using the SimICS output from section 4.2.1 two virtual addresses were found. Watch-points were set at the addresses \texttt{0x12774} and \texttt{0x12794}. The command lines used in GDB-SimICS are shown in Figure 24.

\begin{verbatim}
(gdb-simics) wp 0x12774 4 x
(gdb-simics) wp 0x12794 4 x
\end{verbatim}

\textbf{Figure 24:} The command “wp” is an alias for watch point.

4.3 Performance analysis
The analysis begins by running SimICS until the first watch-point. The “pstats” command is used to extract performance statistics. (The result of the operation is saved and shown in appendix B part 1.) At this step SimICS provides statistics until the first watch-point (\texttt{0x12774}), which is before the lookup is performed. The program executes until the second watch point (\texttt{0x12794}) is reached and the output contains the same format of data. This part of the output shows statistic for the source code from the beginning until the second watch-point (appendix B part 2). The difference between these is the statistic of the part of the program where “address lookup” is performed. The result is shown in Figure 25 (appendix B part 3). In addition to the FUNET table The same test is performed with two different routing tables, the Mae East and the Mae West.

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
Site & TLB misses & Memory & Instruction cache & Data cache & Number of instructions \\
\hline
 & & Read op. & Write op. & Hit & Miss & Read miss & Write miss & \\
FUNET & 79464 & 479790 & 41710 & 3291 & 8 & 135789 & 72 & 3094415 \\
Mae East & 73505 & 434878 & 38471 & 2872 & 8 & 105002 & 49 & 2744596 \\
Mae West & 18446 & 167143 & 15051 & 742 & 8 & 35399 & 15 & 1035025 \\
\hline
\end{tabular}
\caption{The memory and cache performance for 100000 address lookups.}
\end{table}

Figures 26 to 29 shows diagrams for memory and cache performance for the FUNET routing table. The test traffic is generated randomly. The total number of memory references for each address lookup is 5,2 (see Figure 25), where 4,8 or 92\% of the references are memory read operations and only 0,4 or 8\% of the references are memory write operations (Figure 26).

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{memory_statistic.png}
\caption{Memory statistic for 100000 address lookups.}
\end{figure}
The hit and miss rates for memory read and writes operations are shown in Figure 27 and Figure 28.

![Data cache read hit/miss rate](#)

**Figure 27**: Data cache read performance.

The average data cache read miss rate is 28% (72%, average hit rate) and the average write miss rate is 0.17% (99.83%, average hit rate).

![Data cache write performance](#)

**Figure 28**: Data cache writes performance.

The highest number of data cache misses during address lookup occurred at the first step of the address lookup in the LC-trie structure, when the search mechanism traversed the LC-trie. The LC-trie is built out of the base vector entries and the base vector is the largest structure in the LC-trie algorithm. While traversing the trie the search performs memory access for each node in the trie that has to be traversed where each memory access on average cause 0.7 data cache misses and 0.3 tlb-misses (appendix B line 391). The next largest number of data cache and tlb-misses occurred when the search mechanism examined if there was a hit, and if that was the case accessed the base vector in order to return the next-hop address (appendix B line 398). The search operation needed to find out if there really was a hit causes on average 0.85 memory references which results in 0.4 data cache misses and 0.3 tlb-misses per address lookup. One last memory lookup is performed when the next-hop table is accessed in order to return the next-hop address, which results in 0.4 data cache misses and 0.2 tlb-misses.

If there were no hits the prefix vector should be searched for the best prefix match (appendix B line 400). During the search in the prefix table for the best prefix match
the memory is accessed on average 0.4 times and the number of cache misses and tlb misses compared with the hit case are neglectable.

In Figure 29 instruction cache performance is shown for 100000 address lookup. The instruction miss rate is 0.00% which is a high performance for the instruction cache.

![Instruction cache performance](image)

**Figure 29:** Instruction cache performance for 100000 address lookups.

Figures 30 to 32 show the comparison between translation look-aside buffers, tlb, and cache performance when different routing tables were used. The number of tlb-misses increases as a function of the entries in the routing table. The number of entries in the FUNET and the Mae East routing table is close to each other (41578 and 38367) but in Mae West the number of the entries are less than half of the earlier mentioned routing tables.

![TLB-misses per address lookup](image)

**Figure 30:** Average number of TLB-misses per address lookup when different routing tables are used.

The average number of data cache misses in different tests is shown in Figure 32. The average number cache misses is 1.4 per address look up when FUNET is used, 1 when Mae East is used and 0.4 when Mae West is used.
The cache hierarchy used in this simulation was “super sparc”. The processor has two on-chip caches, an instruction cache and a data cache. The data cache is 16 Kbytes, 4-way associative with 32 bytes long cache lines, and the instruction cache is 20 Kbytes, 5-way associative with 64 bytes cache lines. As we can see cache performance is close to optimal. The data cache write hit rate is 100% whilst the data read hit rate is 72%. In this simulation the translation look-aside buffer, tlb, has 64 entries. The average number of tlb-misses is 0.8 for FUNET, 0.7 for Mae East and 0.2 for Mae West per address lookup Figure 30.

The high frequency of misses concerning tlb causes a large number of memory accesses.

### 4.3.1 Using the profiling statistics

By using the profiling statistics from “list-det 378,410”, saved in appendix B, we can provide more detailed information about which instruction is used and how often it is invoked and finally calculate the number of memory accesses performed for 100000 address lookups.

For example the profiling result (appendix B) provides the statistics for different instruction where the load instruction is one of these instruction.

```
386  1 0  97  3564  41709  0  83418   2 node = t->trie[0];
0x11754 [0x00005754]:  1 0  31  2352  41709  0  41709 1 ld [ %o1 ], %o5
0x11758 [0x00005758]:  0 0  66  1212  0  0  41709 1 ld [ %o5 ], %g3
```

**Figure 33:** A part of the profiling result from appendix B.
At line 386 the operation “node = t->trie[0];” requires two memory accesses and the load operation is executed 41709 times during 100000 lookups, which means that for each address lookup in this part of the program (line 386) 0.42+0.42=0.84 memory accesses are performed. By adding the number of the memory accesses for each line the total number of memory accesses performed for each address lookup is calculated. The results of these calculations for different routing tables are shown in Figure 34.

![Figure 34: Test result for different routing table.](image)

In the case the test traffic file is generated at random.

Each address lookup is performed in average by 4.36 memory accesses and 30.9 instructions when FUNET was used. The same tests for the Mae East and the Mae West result in 3.96 respective 1.51 memory accesses and the average number of performed instructions are 27.4 respective 10.4 per address lookup.

When the trace traffic corresponding to the FUNET routing table is used the average number of memory accesses per address lookup is 3.96 and number of instruction is 63.6 (see Figure 35).
The result of this study shows that each address lookup is performed at a maximum of 3.96 memory accesses and at a minimum of 1.51 memory accesses depending on which of the routing tables that is used. By using the result (appendix B) from SimICS we can see, Figure 36, that each time (at line 398, 399 and 400) when the base vector is invoked it causes a large number of read cache misses. Even the number of tlb-misses is too high at these lines. This high rate of cache misses is proportional to the size of routing table entries. But at line 400 there is a noticeable decrease of the number of cache read misses and tlb-misses, where the decrease of tlb-misses is at the same rate as the increase of the number of next-hop table entries in different routing tables.

**Figure 35:** FUNET routing table is tested by using different traffic.

**4.4 Result and conclusion**

The result of this study shows that each address lookup is performed at a maximum of 3.96 memory accesses and at a minimum of 1.51 memory accesses depending on which of the routing tables that is used. By using the result (appendix B) from SimICS we can see, Figure 36, that each time (at line 398, 399 and 400) when the base vector is invoked it causes a large number of read cache misses. Even the number of tlb-misses is too high at these lines. This high rate of cache misses is proportional to the size of routing table entries. But at line 400 there is a noticeable decrease of the number of cache read misses and tlb-misses, where the decrease of tlb-misses is at the same rate as the increase of the number of next-hop table entries in different routing tables.

**Figure 36:** In the Mae East and Mae West at line 399, 400 the number of tlb-misses and cache read misses is zero.
Cache statistics shows an acceptable performance for address lookup in the LC-trie structure but the number of tlb-misses is higher than it should be. A closer look to these tlb-misses reveals where and when the most number of misses occur. The highest number of tlb-misses occurs at the lines 391 and 398, where the traversing in the trie and base vector lookup is performed. The translation look-aside buffer function is to improve the performance of translation of the virtual addresses into physical addresses by caching technique (Appendix A), and the size of this table (or cache) is an important parameter in the tlb performance. Each of these structures, trie and specially the base vector are large and the tlb-table is segmented, which require continuously updating of the tlb-table and that is the reason why the occurrence of tlb-misses is much too high. By increasing the number of the entries in the tlb-table the number of tlb-misses can be kept down efficiently.

![Figure 36: Number of tlb-misses decrease by increasing number of tlb entries.](image)

The high number of tlb-misses decreases by increasing the number of the entries in translation look-aside buffer. When tlb has 128 entries the number of tlb-misses were neglected small and with 256 entries these misses were practically eliminated. A similar solution to this problem is to change the associatively for each entry in the tlb instead of changing the number of entries in tlb, in Figure 36 these statistics is illustrated.
5 References

URL: http://www.it.kth.se/~gk/publications.html
URL: http://www.nada.kth.se/~snilsson/public/code/router

URL: http://www.sics.se/simics/

[3] Swedish Institute of Computer Science, SICS.
URL: http://www.sics.se/cna


URL: http://www.cisco.com/


Appendix A

The translation Look-aside Buffer, TLB

The number of instructions per TLB miss indicates the frequency of misses to the address translation cache. These misses demand more CPU-time.
### Appendix B

**Random traffic test.**

```c
(gdb-simics) list-det 381,410
381 int pos, branch, adr;
382 word bitmask;
383 int preadr;
384
385 /* Traverse the trie */
386 10 97 3564 41709 0 83418 2 node = t->trie[0];
0x11754 [0x00005754]: 10 0 97 2352 41709 0 41709 l ld [%o1], %o5
0x11758 [0x00005758]: 06 66 1212 0 0 41709 l ld [%o5], %e3
387 00 0 0 0 0 0 83418 2 pos = GETSKIP(node);
0x11760 [0x00005760]: 00 0 0 0 0 0 41709 l srl %e3, 0x16, %e0
0x11764 [0x00005764]: 00 0 0 0 0 0 41709 l and %e2, 0x1f, %e3
388 00 0 0 0 0 41709 1 branch = GETBRANCH(node);
0x11768 [0x00005768]: 00 0 0 0 0 0 41709 l srl %e3, 0x1b, %e0
0x11770 [0x00005770]: 00 0 0 0 0 0 41709 1 or %e2, 0x3ff, %e2 ! 0x3ffff
0x00408ffc <traffic.11+3707431>
389 00 0 68242 34276 59048 0 604542 6 while (branch != 0) {
0x11774 [0x00005774]: 00 0 0 0 0 0 41709 l cmp %e0, 0
0x11778 [0x00005778]: 00 0 0 0 0 0 41709 l be 0x117c0 [0x000057c0] <find+108>
0x1177c [0x0000577c]: 00 0 0 0 0 0 41709 l and %e3, %e2, %e0
0x11780 [0x00005780]: 01 0 0 0 0 0 41709 l mov 0x20, %e4
0x11784 [0x00005784]: 00 0 0 0 0 0 41709 l mov %e2, %e1
390 00 0 0 0 0 0 208545 5 pos += branch + GETSKIP(node);
0x11788 [0x00005788]: 00 0 0 0 0 0 41709 l cmp %e0, 0
0x117774 [0x00005774]: 00 0 0 0 0 0 41709 l cmp %e0, 0
391 00 0 554 4 0 55648 8 if (EXTRACT(0, t->base[adr].len, bitmask) == 0)
0x117a0 [0x000057a0]: 00 0 0 0 0 0 100757 l sll %e4, %e3, %e2
0x117a4 [0x000057a4]: 00 0 0 0 0 0 100757 l sll %e3, 0x16, %e2
0x117a8 [0x000057a8]: 00 0 0 0 0 0 100757 l sll %e3, 0x1b, %e0
0x117ac [0x000057ac]: 00 0 0 0 0 0 100757 l add %e2, %e2, %e2
392 00 0 0 0 0 403028 4 pos += branch + GETSKIP(node);
0x117a0 [0x000057a0]: 00 0 0 0 0 0 100757 l srl %e3, 0x16, %e2
0x117a4 [0x000057a4]: 00 0 0 0 0 0 100757 l srl %e3, 0x1b, %e0
0x117a8 [0x000057a8]: 00 0 0 0 0 0 100757 l srl %e3, 0x1f, %e2
0x117ac [0x000057ac]: 00 0 0 0 0 0 100757 l srl %e3, 0x3f, %e2
393 00 0 0 0 0 100757 1 node = t->trie[adr + EXTRACT(pos, branch, s)];
0x117b0 [0x000057b0]: 00 0 0 0 0 0 100757 l srl %e3, 0x1b, %e0
394 00 0 0 0 0 100757 l add %e3, %e1, %e0
395 00 0 0 0 0 59048 302271 
396 0x117b4 [0x000057b4]: 00 0 0 0 0 0 100757 l cmp %e0, 0
0x117b8 [0x000057b8]: 00 0 0 0 0 0 100757 l bne 0x11788 [0x00005788] <find+52>
0x117bc [0x000057bc]: 00 0 0 0 0 0 59048 100757 l add %e3, %e1, %e0
0x117c [0x000057c0]: 01 0 0 0 0 0 40426 34401 0 166836 4 /* Was this a hit? */
0x117c0 [0x000057c0]: 10 31 45 0 0 41709 l ld [%o1 + 8], %e0
0x117c4 [0x000057c4]: 00 0 0 0 0 0 41709 l sll %e2, 4, %e3
0x117c8 [0x000057c8]: 00 40395 34356 0 0 41709 l ld [%o0 + %e3], %e2
0x117cc [0x000057cc]: 00 0 0 0 0 0 41709 l xor %e2, %e4, %e3
397 00 0 0 0 0 554 4 0 41709 sll %e2
0x117c0 [0x000057c0]: 00 0 0 0 0 0 41709 l ld [%o0 + 4], %e3
0x117d0 [0x000057d0]: 00 0 0 0 0 0 41709 l mov 0x20, %e2
0x117d4 [0x000057d4]: 00 0 0 0 0 0 41709 l mov 0x20, %e2
0x117dc [0x000057dc]: 00 0 0 0 0 0 41709 l sub %e2, %e3, %e2
```

31
0x117e0 [0x000057e0]: 0 0 0 0 0 0 41709 l srl %o3, %g2, %g2
0x117e4 [0x000057e4]: 0 0 0 0 0 0 41709 l cmp %g2, 0
0x117e8 [0x000057e8]: 0 0 0 0 0 40539 41709 l bne,a 0x1180c [0x0000580c]
0x117ec [0x000057ec]: 0 0 554 4 0 1170 1170 l d [ %o0 + 8 ], %o0

400 0 0 19597 135 40539 40539 81078 2 return t->nexthop[t->base[adr].nexthop];
0x117f0 [0x000057f0]: 0 0 0 0 0 40539 1 b 0x117fc [0x0000580c] <find+168>
0x117f4 [0x000057f4]: 0 0 19597 135 40539 40539 1 ld [ %o0 + 0xc ], %g2
401
402 /* If not, look in the prefix tree */
403 preadr = t->base[adr].pre;
404 0 0 4 33 2340 1170 4680 4 while (preadr != NOPRE) {
}
0x117f8 [0x000057f8]: 0 0 0 0 0 1170 1 ld [ %g2 + 8 ], %g2

407 0 0 2 0 0 0 18 1 l d [ %g2 + 4 ], %o0
408 0 0 0 0 0 18 54 3 }
409
410 /* Debugging printout for failed search */
(gdb-simics)
Appendix B, Part 1

Random traffic test

Statistics for cpu 0
Statistics vectors: (raw data)

User mode:
- 5402 tlb misses passed on to OS emulation
- 25324495 memory read operations
- 10376971 memory write operations
- 27 (internal) intermediate text pages allocated
- 2284 simulated physical pages allocated
- 203516761 number of instructions
- 475135 i_cache_hit
- 454 i_cache_miss
- 118334 d_cache_read_hit
- 478560 d_cache_read_miss
- 91844 d_cache_write_hit
- 172972 d_cache_write_miss
- 649484 d_cacheReplacement

Supervisor mode:
(Note: only non-zero statistic vector values are shown.)

Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)

Memory statistics, user mode:
- Memory reads: 25324495 (70.93%)
- Memory writes: 10376971 (29.07%)
- I/O: 0 (0.00%)
- Total accesses: 35701466

Data cache performance, user mode: (ignores I/O accesses)
- Read miss rate: 1.890% (478560/25324495)
- Write miss rate: 1.667% (172972/10376971)
- Total miss rate: 1.825% (651532/35701466)

Memory statistics, supervisor mode:
- Memory reads: 0 (NaN%)
- Memory writes: 0 (NaN%)
- I/O: 0 (NaN%)
- Total accesses: 0

Data cache performance, supervisor mode: (ignores I/O accesses)
- Read miss rate: NaN% (0/0)
- Write miss rate: NaN% (0/0)
- Total miss rate: NaN% (0/0)

Instruction cache performance, both modes:
- Op fetches: 203516761
- Miss rate: 0.000% (454/203516761)
- Number of cycles executed (CPU 0): 203516761

Exception frequencies (global count):
- [5] 1014 Window_Overflow
- [6] 1013 Window_Underflow

Total: 2027 exceptions.

Profiler totals: (this may take a while, you can interrupt with Ctrl-C)
- Instruction cache misses caused by program line --> 454
- Cache misses (writes) caused by program line --> 172972
- Cache misses (reads) caused by program line --> 478560
- TLB misses passed on to Unix emulation --> 5402
- Number of (taken) branches *to* the code block --> 40764947
- Number of (taken) branches *from* the code block --> 40764947
- Count of instruction execution (based on branch arcs) --> 203516761
- Number of addresses from which instructions have been fetched --> 410
Appendix B,  Part 2

Random traffic test
Statistics for cpu 0
Statistics vectors: (raw data)
User mode:
84866  tlb misses passed on to OS emulation
25804285  memory read operations
10418681  memory write operations
27  (internal) intermediate text pages allocated
2284  simulated physical pages allocated
206611176  number of instructions
478426  i_cache_hit
462  i_cache_miss
156338  d_cache_read_hit
614349  d_cache_read_miss
93055  d_cache_write_hit
173044  d_cache_write_miss
785345  d_cache replacement

Supervisor mode:
(Note: only non-zero statistic vector values are shown.)
Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)
Memory statistics, user mode:
Memory reads: 25804285 (71.24%)
Memory writes: 10418681 (28.76%)
I/O: 0 (0.00%)
Total accesses: 36222966
Data cache performance, user mode: (ignores I/O accesses)
Read miss rate: 2.381% (614349/25804285)
Write miss rate: 1.661% (173044/10418681)
Total miss rate: 2.174% (787393/36222966)
Memory statistics, supervisor mode:
Memory reads: 0 (NaN%)
Memory writes: 0 (NaN%)
I/O: 0 (NaN%)
Total accesses: 0
Data cache performance, supervisor mode: (ignores I/O accesses)
Read miss rate: NaN% (0/0)
Write miss rate: NaN% (0/0)
Total miss rate: NaN% (0/0)
Instruction cache performance, both modes:
Op fetches: 206611176
Miss rate: 0.000% (462/206611176)
Number of cycles executed (CPU 0): 206611176
Exception frequencies (global count):
[ 5] 1014  Window_Overflow
[ 6] 1013  Window_Underflow
Total: 2027 exceptions.
Profiler totals: (this may take a while, you can interrupt with Ctrl-C)
Instruction cache misses caused by program line --> 462
Cache misses (writes) caused by program line --> 173044
Cache misses (reads) caused by program line --> 614349
TLB misses passed on to Unix emulation --> 84866
Number of (taken) branches *to* the code block --> 41075442
Number of (taken) branches *from* the code block --> 41075442
Count of instruction execution (based on branch arcs) --> 206611176
Number of addresses from which instructions have been fetched --> 416

Appendix B, Part 3
Random traffic test
User mode:
79464 tlb misses passed on to OS emulation
479790 memory read operations
41710 memory write operations
3094415 number of instructions
3291 i_cache_hit
8 i_cache_miss
135789 d_cache_read_miss
72 d_cache_write_miss
135861 d_cache_replacement

Supervisor mode:
Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)

Memory statistics, user mode:
Memory reads: 479790 (92%)
Memory writes: 41710 (8%)
Total accesses: 521500

Data cache performance, user mode: (ignores I/O accesses)
Read miss rate: 28% (135789/479790)
Write miss rate: 0.17% (72/41710)
Total miss rate: 26.1% (135861/521500)

Instruction cache performance, both modes:
Op fetches: 3094415 --> 31 instructions/lookup
Miss rate: 0.000% (8/3094415)
Number of cycles executed (CPU 0): 3094415
Appendix C

Trace traffic test

(gdb-simics) list-det 378,410

nexthop_t find(word s, routtable_t t)
{
    node_t node;
    int pos, branch, adr;
    word bitmask;
    int preadr;

    /* Traverse the trie */
    node = t->trie[0];

    while (branch != 0) {
        pos = GETSKIP(node);
        branch = GETBRANCH(node);
        adr = GETADR(node);

        bitmask = t->base[adr].str ^ s;
        pos += branch + GETSKIP(node);

        /* Was this a hit? */
        bitmask = t->base[adr].str ^ s;
    }

    /* Was this a hit? */

399 0 0 0 0 38470 270453 8 if (EXTRACT(0, t->base[adr].len, bitmask) == 0)
0x117d0 [0x000057d0]: 0 0 0 0 0 0 38470 1 add %o0, %g3, %o0
0x117d4 [0x000057d4]: 0 0 0 0 0 0 38470 1 ld [%o0 + 4], %g3
0x117d8 [0x000057d8]: 0 0 0 0 0 0 38470 1 mov 0x20, %g2
0x117dc [0x000057dc]: 0 0 0 0 0 0 38470 1 cmp %g2, %g3, %g2
0x117f0 [0x000057f0]: 0 0 0 0 0 0 38470 1 cmp %g2, 0
0x117f4 [0x000057f4]: 0 0 0 0 0 0 37307 38470 1 bne,a 0x1180c [0x0000580c]
0x117ec [0x000057ec]: 0 0 0 0 0 0 0 38470 1 mov 0x20, %g2
0x11800 [0x00005800]: 0 0 0 0 0 0 0 37307 38470 1 bne,a 0x1180c [0x0000580c]
0x11808 [0x00005808]: 0 0 0 0 0 0 0 37307 38470 1 ld [%o0 + 8], %o0
400 0 0 0 2 37307 37307 74614 2 return t->nexthop[t->base[adr].nexthop];
0x117f8 [0x000057f8]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0
0x1180c [0x0000580c]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0
0x11814 [0x00005814]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0
0x11818 [0x00005818]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0
0x1181c [0x0000581c]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0
401 /* If not, look in the prefix tree */
402 preadr = t->base[adr].pre;
403 0x11820 [0x00005820]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
404 0x11824 [0x00005824]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
405 0x11828 [0x00005828]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
406 0x1182c [0x0000582c]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
407 0x11830 [0x00005830]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
408 0x11834 [0x00005834]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
409 0x11838 [0x00005838]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
410 0x1183c [0x0000583c]: 0 0 0 0 0 0 0 0 0 0 0 clr %o0, 1, %g2
411 /* Debugging printout for failed search */
412 (gdb-simics)
Appendix C, Part 1

Trace traffic test
(gdb-simics) pstats
Statistics for cpu 0
Statistics vectors: (raw data)
User mode:
5163 tlb misses passed on to OS emulation
39424713 memory read operations
18338148 memory write operations
27 (internal) intermediate text pages allocated
2246 simulated physical pages allocated
322026172 number of instructions
440324 i_cache_hit
448 i_cache_miss
145441 d_cache_read_hit
428061 d_cache_read_miss
92109 d_cache_write_hit
180334 d_cache_write_miss
606347 d_cache_replacement
Supervisor mode:
(Note: only non-zero statistic vector values are shown.)
Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)
Memory statistics, user mode:
Memory reads: 39424713 (68.25%)
Memory writes: 18338148 (31.75%)
I/O: 0 (0.00%)
Total accesses: 57762861
Data cache performance, user mode: (ignores I/O accesses)
Read miss rate: 1.086% (428061/39424713)
Write miss rate: 0.983% (180334/18338148)
Total miss rate: 1.053% (608395/57762861)
Memory statistics, supervisor mode:
Memory reads: 0 (NaN%)
Memory writes: 0 (NaN%)
I/O: 0 (NaN%)
Total accesses: 0
Data cache performance, supervisor mode: (ignores I/O accesses)
Read miss rate: NaN% (0/0)
Write miss rate: NaN% (0/0)
Total miss rate: NaN% (0/0)
Instruction cache performance, both modes:
Op fetches: 322026172
Miss rate: 0.000% (448/322026172)
Number of cycles executed (CPU 0): 322026172
Exception frequencies (global count):
[  5] 1144 Window_Overflow
[  6] 1143 Window_Underflow
Total: 2287 exceptions.
Profiler totals: (this may take a while, you can interrupt with Ctrl-C)
Instruction cache misses caused by program line --> 448
Cache misses (writes) caused by program line --> 180334
Cache misses (reads) caused by program line --> 428061
TLB misses passed on to Unix emulation --> 5163
Number of (taken) branches *to* the code block --> 66167166
Number of (taken) branches *from* the code block --> 66167166
Count of instruction execution (based on branch arcs) --> 322026172
Number of addresses from which instructions have been fetched --> 407
Appendix C, Part 2

Trace traffic test
(gdb-simics) pstats
Statistics for cpu 0
Statistics vectors: (raw data)
User mode:
44392 tlb misses passed on to OS emulation
40527767 memory read operations
18438148 memory write operations
27 (internal) intermediate text pages allocated
2246 simulated physical pages allocated
328383680 number of instructions
442392 i_cache_hit
456 i_cache_miss
238119 d_cache_read_hit
453075 d_cache_read_miss
92710 d_cache_write_hit
180350 d_cache_write_miss
631377 d_cacheReplacement
Supervisor mode:
(Note: only non-zero statistic vector values are shown.)
Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)
Memory statistics, user mode:
Memory reads: 40527767 (68.73%)
Memory writes: 18438148 (31.27%)
I/O: 0 (0.00%)
Total accesses: 58965916
Data cache performance, user mode: (ignores I/O accesses)
Read miss rate: 1.118% (453075/40527767)
Write miss rate: 0.978% (180350/18438148)
Total miss rate: 1.074% (633425/58965916)
Memory statistics, supervisor mode:
Memory reads: 0 (NaN%)
Memory writes: 0 (NaN%)
I/O: 0 (NaN%)
Total accesses: 0
Data cache performance, supervisor mode: (ignores I/O accesses)
Read miss rate: NaN% (0/0)
Write miss rate: NaN% (0/0)
Total miss rate: NaN% (0/0)
Instruction cache performance, both modes:
Op fetches: 328383680
Miss rate: 0.000% (456/328383680)
Number of cycles executed (CPU 0): 328383680
Exception frequencies (global count):
[5] 1144 Window_Overflow
[6] 1143 Window_Underflow
Total: 2287 exceptions.
Profiler totals: (this may take a while, you can interrupt with Ctrl-C)
Instruction cache misses caused by program line --> 456
Cache misses (writes) caused by program line --> 180350
Cache misses (reads) caused by program line --> 453075
TLB misses passed on to Unix emulation --> 44392
Number of (taken) branches *to* the code block --> 66828529
Number of (taken) branches *from* the code block --> 66828529
Count of instruction execution (based on branch arcs) --> 328383680
Number of addresses from which instructions have been fetched --> 413
Appendix C, Part 3
Trace traffic test

User mode:
39256 tlb misses passed on to OS emulation
1103054 memory read operations
100000 memory write operations
6357508 number of instructions
2068 i_cache_hit
8 i_cache_miss
25014 d_cache_read_miss
16 d_cache_write_miss
25030 d_cache_replacement

Supervisor mode:
Analysis of SuperSparc cache simulation for CPU 0:
(Dcache: 16 kbyte, 4-way set associative, 32-byte lines;
Icache: 20 kbyte, 5-way set associative, 64-byte lines)

Memory statistics, user mode:
Memory reads: 1103054 (91.7%)
Memory writes: 100000 (8.3%)
Total accesses: 1203054

Data cache performance, user mode: (ignores I/O accesses)
Read miss rate: 2.26% (25014/1103054)
Write miss rate: 0.016% (16/100000)
Total miss rate: 2.08% (25030/1203054)

Instruction cache performance, both modes:
Op fetches: 6357508 => ca. 64 instructions/lookup
Miss rate: 0.000% (8/6357508)
Number of cycles executed (CPU 0): 6357508