Final thesis

Potential for increasing the size of NETSim simulations through OS-level optimizations

by

Kjell Enblom,
Martin Jungebro

LITH-IDA-EX--08/001--SE

2008-01-18

Final thesis

Potential for increasing the size of NETSim simulations through OS-level optimizations

by

Kjell Enblom,
Martin Jungebro

LITH-IDA-EX--08/001--SE

Supervisor : **Tomas Abrahamsson**
FJI/K
at Ericsson AB

Examiner : **Christoph Kessler**
Dept. of Computer and Information Science
at Linköpings universitet

Abstract

English

This master's thesis investigates if it is possible to increase the size of the simulations running on NETSim, Network Element Test Simulator, on a specific hardware and operating system. NETSim is a simulator for operation and maintenance of telecommunication networks.

The conclusions are that the disk usage is not critical and that it is needless to spend time optimizing disk and file system parameters. The amount of memory used by the simulations increased approximately linear with the size of the simulation. The size of the swap disk space is not a limiting factor.

Svenska

Detta exsamensarbete undersöker om det är möjligt att öka storleken på simuleringskörningar av NETSim, Network Element Test Simulator, på en specifik hårdvaru- och operativsystemsplattform. NETSim är en simulator för styr och övervakning av telekomnätverk.

Slutsatserna är att diskanvändandet inte är kritiskt och att det är onödigt att ägna tid åt att optimera disk- och filsystemsparametrar. Minnesutnyttjandet ökar approximativt linjärt med storleken på simuleringarna. Storleken på swapdisken är inte någon begränsande faktor.

Keywords : NETSim, simulating UMTS networks, OS parameters, Linux, Ericsson AB

Acknowledgements

We would like to thank those at the NETSim department at Ericsson in Linköping, especially Daniel Wiik, David Haglund and our tutor Tomas Abrahamsson.

Chapters 2.7, 2.9, 3.2.3, 3.2.5 is written by Kjell Enblom, chapters 2.5, 2.6, 3.2.1, 3.2.2 is written by Martin Jungebro and all other chapters by Kjell Enblom and Martin Jungebro.

Contents

Chapter 1

Introduction

This master's thesis investigates if it is possible to increase the size of the simulations running on NETSim, Network Element Test Simulator, on a specific hardware and operating system.

1.1 Overview over the report

This chapter introduces the background and purpose of this master thesis.

Chapter 2 introduces an overview over cellular networks, NETSim, the operation and maintenance system OSS, the load generator, RAID, confidence interval, file systems, I/O scheduling algorithms, virtual memory and swap.

Chapter 3 describes the test environment and the test runs.

Chapter 4 presents the test results.

Chapter 5 discusses the results and presents our recommendations.

The main target group for this thesis is NETSim personel at Ericsson AB. Recommended previous knowledge for readers is a masters degree in computer science or similar.

1.2 Background

Since 1993 Ericsson is developing the simulation tool NETSim. NETSim is a simulator for testing telecommunication networks. NETSim simulates parts of or whole UMTS networks, Universal Mobile Telecommunications System networks from an operational and maintenance perspective. The users are primarily other divisions at Ericsson AB.

NETSim is a resource demanding program and claims expensive computers to run large simulations, about 5000 cellular network nodes. 5000 nodes covers a small country. The network nodes are base stations phone switches, network routers etc.

Since 1993 more network elements has been developed and the need for running larger simulations has increased.

1.3 Purpose and problem description

NETSim is resource demanding and needs expensive computers for running large simulations, about 5000 nodes, called NEs, Network Elements, in NETSim. And soon customers needs to run even larger simulations, about 2 to 3 times larger. Therefore Ericsson wants to know if it is possible to increase the number of network elements, NEs, on the existing hardware without increasing the number of computers.

One possible way to do it is to optimize the NETSim code. Another possible way is to do performance tuning in the operating system environment. In this thesis we are studying the operating system environment. To study

possible ways to optimize the source code of NETSim is something that should be done but that is outside the scope of this master's thesis.

The goal of this thesis is to ascertain if it is possible to increase the size of the running simulations with 50% more NEs on existing hardware with operating system environment tuning.

1.4 Items to tune

After a discussion with our supervisor and seven other developers we came to the conclusion that this list is what is most interesting to study within the scope of the master's thesis. The items to study and the three categories is mainly Ericsson's priority order. All technical terms and abbreviations will be defined later in this thesis.

1.
   - Compare NETSim and databases on RAID 0 with NETSim and databases running without RAID.
   - Compare 32 bit and 64 bit SSL esock.
   - Compare Kpoll and Epoll in SSL esock.
   - Compare SSL with and without a crypto accelerator card
2.
   - Compare the reiserfs filesystem with other file systems, ext3, xfs and jfs.
   - Study different I/O scheduling
   - Study different values for swappiness
   - Study different VM parameters
3.
   - Compare the standard Linux kernel, 2.6.8, with a newer kernel.
   - Compare 32 bit and 64 bit Erlang

This is what Ericsson and we think can give most performance on the operating system level.

Chapter 2

Theoretical background

This section describes the theoretical background of UMTS networks, NET-Sim, OSS, the load generator, RAID, Confidence Interval, File systems, I/O Scheduling algoriths and Virtual memory and Swap.

2.1 Overview over UMTS networks

The parts of a UMTS network are core, GSM, Global System for Mobile, and UTRAN, UMTS Terrestrial Radio Access Network [1].

A UMTS network is divided in two main parts the BSS, Base Station System, and the switching system. The BSS is the lower part elements of figure 2.1 and the switching part is the upper part elements of the figure. The left part of the figure is the GSM network and the right part is the UTRAN network.

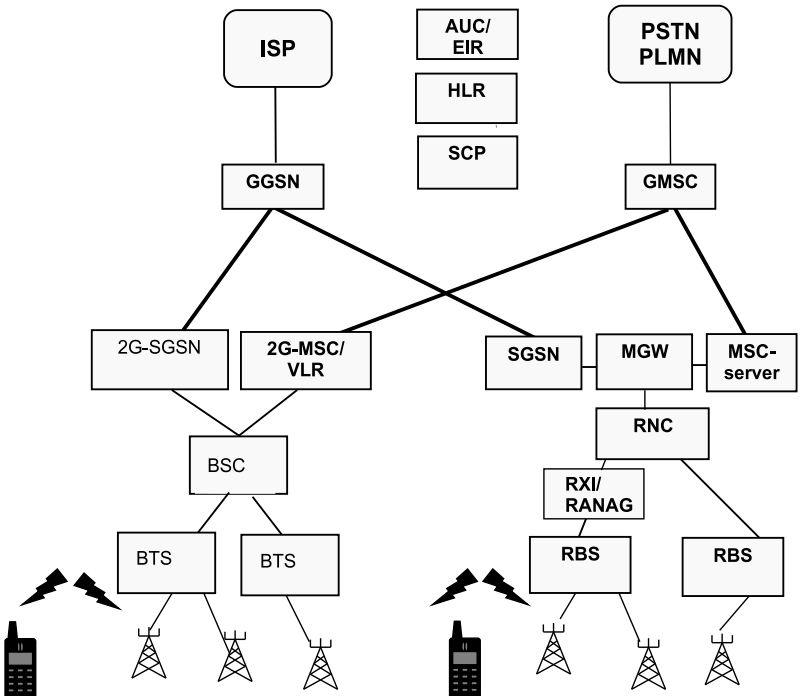

Figure 2.1: UMTS Network.

The Base Station System is responsible for the radio functions in the network. In GSM the functions is divided in BTS, Base Transceiver Stations and BSC, Base Station Controllers. BTS is the unit providing radio communication in each radio cell. BSC is controlling a set of BTSs, see figure 2.1. The most important task is utilization of the radio resources. The BSC uses traffic information to balance the temporary traffic load between its cells. They are expensive and therefore there are not so many BSCs in a UMTS network [2].

The MSC, Mobile Service Switching Center, is part of the Switching System, see figure 2.1. The MSC controls calls to and from the PSTN, Public Switched Telephone Network, and within and between PLMN, Public Land Mobile Networks. A national or international transit is called a GMSC,

Gateway Mobile Service Switching Center [2].

In UTRAN the BSS consists of RNC, Radio Network Controller, and RBS, Radio Base Station.

The main function of the Node B (or RBS in Ericsson terminology) is to perform the air interface L1, level 1, processing (channel coding and interleaving, rate adaption, spreading, etc.). It also performs some basic Radio Resource Management operation as the inner loop power control. It logically corresponds to the GSM Base Station. see figure 2.1 [3].

RNC performs cell resource allocation, radio resource management, system information broadcasting and hand over. Described in Gunnarsson et al. "The mobiles, the base stations and their radio resources are all controlled by the radio network controller (RNC)" [4].

RXI, is an ATM and IP router [5, 6].

The SGSN, Serving GPRS Support Node, stores information about mobile nodes visiting its network, copy of the visiting user's service profile, as well as more precise information on th UE's, User Equipments, location within the serving system. The MSC and SGSN servers determine what media gateway functions and resources are required by the call/session and controls them via the gateway control protocol. The Ericsson SGSN server determines and controls end-user Internet protocol services and mobility management [6, 3].

GGSN, Gateway GPRS Support Node, provides the interface between mobile networks and the Internet or corporate intranets. Consequently, the GGSN usually incorporates a firewall. Incoming data packets are packed in a special container by the GGSN and forwarded over GTP, the GPRS Tunnel Protocol, to the SGSN [7].

HLR, Home Location Register, stores the locations of mobile stations. It contains subscription information and information about which MSC area the mobile station is within at a given moment. This information is needed to make it possible to set up a call to a mobile station [2].

EIR, Equipment Identity Register, stores hardware numbers of the authorized mobile stations and information about them. This information is used to prevent not type-approved equipment from accessing the network, for example stolen mobile station equipment. MSC is connected to EIR and uses the information from EIR to check the validity of the mobile station equipment [2].

SCP, Service Control Point, is detecting and handling IN services, Intelligent Network services, like local number portability, routing calls to the location that is closest to or most convenient for the calling party, Premium Rate services etc. [8].

2.2 NETSim

NETSim is a simulator developed by Ericsson. NETSim is written in the Erlang programming language and is currently running on Suse Linux and on Sun Solaris. NETSim simulates GSM, UTRAN or combined UMTS networks. It simulates the operation and maintenance behavior of telecommunication networks. It does not simulate data traffic in the net, telephone traffic, GPRS data etc.

NETSim can be combined with a real UMTS network. An O&M, Operation and Maintenance, operator can in that way perform O&M on both real NEs, Network Elements, and on simulated NEs. It is also possible to train on simulated NEs simulating not yet developed NEs [1].

NETSim can be used [9]:

- for testing O&M part of the network elements in different types of testing.
- as a substitute when real nodes are too expensive or are not developed yet.
- to simulate erroneous behaviour in a node.
- to simulate the behaviour of a real network for training of O&M system users.
- for installation and delivery tests of O&M systems.

In NETSim a simulation consists of one or more networks that each contains a number of network elements, usually 100 – 300 NE per network. To load, start and stop simulations one can either use the command line interface, netsim_shell, or the graphical user interface, netsim_gui, see figure 2.2.

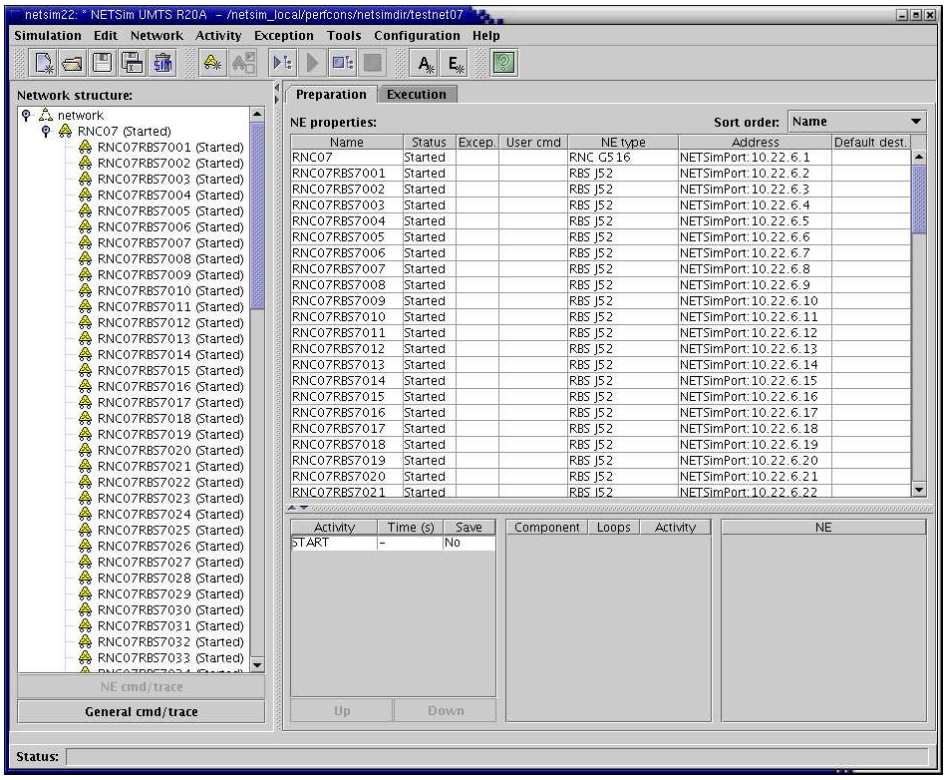

Figure 2.2: NETSim graphical user interface.

In figure 2.2 we can see a running network simulation with 100 NEs. In the simulation there is one RNC, RNC07, and 98 RBS, RNC07RBS7001-RNC07RBS7098, and one hidden RXI. The RBS's and the RXI are all connected to the RNC called RNC07.

NETSim consists of the following blocks; a super-server that takes care of starting and registering other nodes, a coordinator that handles the user interfaces, server(s) that each of them handles execution of 1 to MaxPerServer NEs (in our simulations 1 to 32), NME, NETSim Management Extension, that handles communication to monitor and administration through SNMP, Simple Network Management Protocol, and error logger that takes care of all error logging [10].

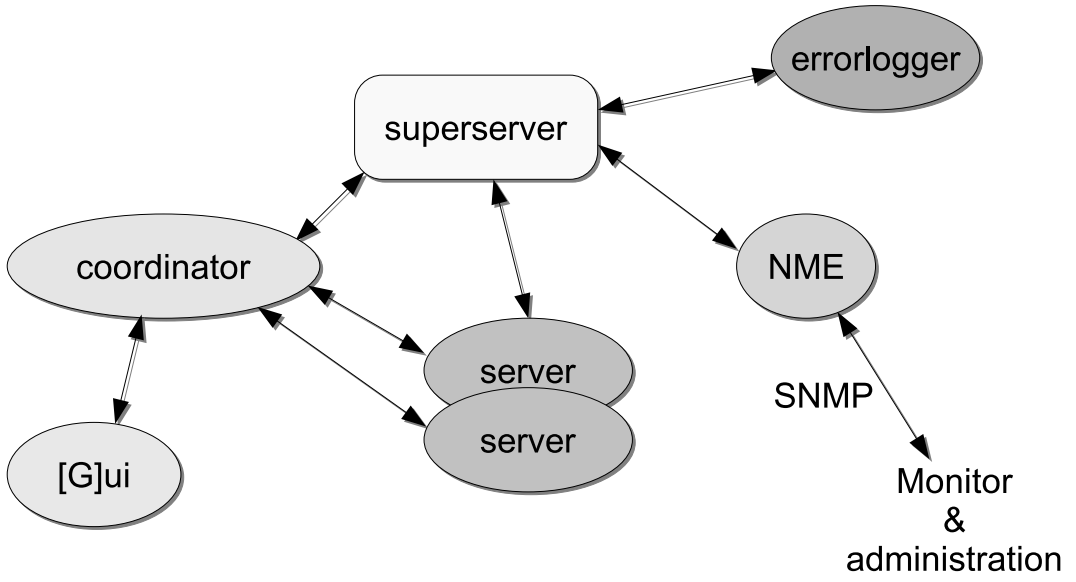

Figure 2.3: NETSim block diagram.

Many of the NEs communicate using Corba over SSL, Secure Sockets Layer, and some of the AXE based NEs use X25. Different protocols used for O&M traffic are used by NETSim. Examples of such protocols are MTP, Message Transfer Protocol, Corba, telnet, SSH, FTP, File Transfer Protocol, and SFTP, Secure File Transfer Protocol. Alarms from NEs that communicate using Corba are sent with SSLIOP, Secure Socket Layer Inter-ORB Protocol, to the O&M system.

The O&M systems sends operation and maintenance commands, config-

urations, software upgrades etc, to NEs and NEs sends alarm messages, notifications and produces performance measurements to the O&M system [1].

The NEs use one IP address each in a simulation [1].

NETSim uses databases, one for storing global data, one database for each running simulation and one database per NE. During simulation runs the NEs have data stored in a database on disk and a working copy in primary memory. When an NE is started it reads in data from disk to its working area and when it is stopped it saves data to the database on disk.

NEs are restarted every time they perform a SWUG, SoftWare UpGrade. SWUG in NETSim is a simulation of a software upgrade. The NEs gets an upgrade control file, where the upgrade control file contains information about where the new software file is and what method should be used to collect it. The simulated NEs in NETSim gets the new software file and drops it, without installing it (whereas a real NE would, of course, use it for installation). The NEs stores attributes in an MO tree, Managed Objects tree, within a MIB tree, Management Information Base. The MO tree is a data structure for storing information, attributes, about the NEs, and for manipuling hardware (in real NEs), software and configurations. The information is stored in the internal nodes and in the leafs. Example of attributes is fan speed for a fan. The number of MO's and MO attributes for an RNC, an RBS and an RXI are shown in table 2.1 [1, 11].

	Number of MO	Number of MO attributes
RNC	$\sim 5 * 10^4$	$\sim 2 * 10^6$
RBS	$\sim 4 * 10^2$	$\sim 4 * 10^3$
RXI	$\sim 2 * 10^4$	$\sim 3 * 10^5$

Table 2.1: Number of MO's and MO attributes in some NEs.

NETSim is distributing simulated NEs over all CPU's in a system via the server nodes [10]. The algorithm is:

1. If the number of server nodes are fewer than the number of CPU's in the system start a new server node on a new CPU.
2. If number of NEs is equal to max number of NEs per server node start a new server node.
3. Else select the least occupied server node for the new NE.

2.3 OSS

OSS, Operations Support System, is an O&M system. The operations of OSS are [2]:

- Cellular network administration
- Configuration management
- Software management
- Hardware management
- Fault management
- Performance management
- Security management

In large Public Land Mobile Network, the amount of network data is huge. There may be thousands of NEs and each NE has more than a hundred NE parameters controlling and defining neighbour relations between NEs and the behaviour of the NE itself.

When changes are introduced, it is important that the new data, for example a NE parameter in a new NE, does not disturb the other NEs or introduce unexpected behaviour into the network. Checking all the parameters involved is a tedious task. Cellular Network Administration provides support for changes in the cellular network [2].

Configuration management includes the following functions: presenting managed objects and their parameters, adjusting the database with network configuration data, importing and exporting configuration data, support for taking objects in or out of operation, initiating tests on objects, planning and introducing new sites and reconfiguring objects [2].

The software management is used to store, control and upgrade central function software and transceiver software [2].

The hardware management provides the operator with a register of all installed hardware from which he or she can get an overview of hardware products on the sites. This information is important for many reasons, for example to handle spare parts or to trace hardware units of a specific update revision [2].

OSS has a fault management that covers the following types of problems: network element generated alarms, data link generated alarms, externally generated alarms i.e. alarms originated from alarms in the buildings where the OSS is installed and OSS internal generated alarms [2].

There are three performance management functions within OSS: performance management statistics which can be used to continuosly evaluate the overall performance of the cellular network, statistical reports which is a set which focus on the data used for managing, planning and engineering a cellular network, performance management traffic recording which is a general data collection tool for the radio path that can be used to survey limited network areas and to verify NE and locating changes [2].

The OSS provides mechanisms for handling authority and access and it is used among other things for delegating management tasks to specific departements [2].

The data OSS collects is statistics and NE data. OSS transfers the data from the NEs with FTP or SFTP. Example on statistics is the number of phone calls, number of lost calls etc. OSS also subscribes for notifications and alarm messages from the NEs. Alarms and notifications are sent with SSLIOP, Corba over SSL. NEs send notifications when an attribute value in

the MO tree is changed, and when the MO tree is changed, an attribute is deleted or a new attribute is added. The O&M systems sends configuration data, commands and software upgrades to the NEs over SSLIOP [1].

2.4 The load generator

The load generator imitates some of the load that OSS puts on NETSim. The load generator performs performance monitoring, PM, by collecting data from the NEs. It also sees to that the NEs change their attribute values in the MO tree and sends an AVC notification, Attribute Value Change, and sees to that NEs changes the MO tree and sends topology change notifications.

Another important thing is alarms. In a simulation with a load generator 10 alarms / second are sent from NEs to the load generator. The alarms are sent from NEs in the first simulated network. 10 alarms / second is a standard value at Ericsson.

The load generator also does topology synchronization, and attribute synchronizations for all MO trees for all NEs. A topology synchronization followed by an attribute synchronization is also called nesync. During the topology synchronization the load generator makes a snapshot of the NEs MO tree.

All the above is background load. Background load 1 is defined as average alarms + average attribute change + performance monitoring + NE restarts. Background load 2 is defined in the same way as background load 1 except NE restarts. Another task the load generator can do is SWUG, SoftWare UpGrade, on NEs [12].

2.5 RAID

Redundant Arrays of Independent Disks, RAID works with three key concepts, mirroring, striping and different error correction techniques. Mirroring needs two or more physical disks, and copies each block to all disks at the same offset. Striping need two or more physical disks and saves data blocks or blocksegments in a round robin fashion. Error correction or detection techniques are implemented by parity check and Reed-Solomon code. RAID can be implemented in hardware or software. In hardware implemented RAID, a RAID controller performs parity calculations, and management of the disks. Hardware implementations do not add any extra processing time to the CPU and present the RAID system as a logical disk to the operating system. Software RAID provides an abstraction layer between the logical disk and disk controller. There are six main standard RAID levels, RAID-0, RAID-1 and RAID-3 to RAID-6, standardized by SNIA , Storage Networking Industry Association.

2.5.1 RAID-0

Striped set of disks without parity, needs two or more disks, see figure 2.4. RAID-0 provides improved bandwidth in reading and writing large files, because all included disks read and write at the same time. For files smaller than the blocksize all disks can read and write independent of each other. [13]

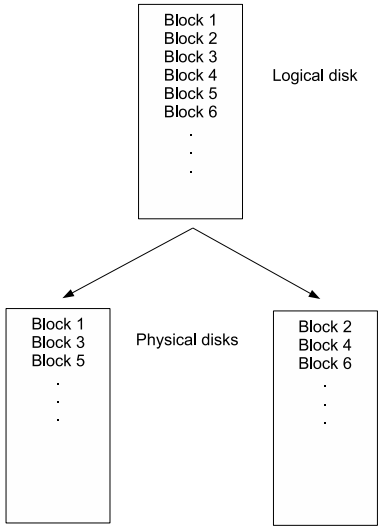

Figure 2.4: RAID 0.

2.5.2 RAID-1

Mirrored set of disks needs two or more disks, see figure 2.5. RAID-1 writes at the same speed as a single disk but reading can be done independently on all disks increasing bandwidth and decreasing seek time. [13]

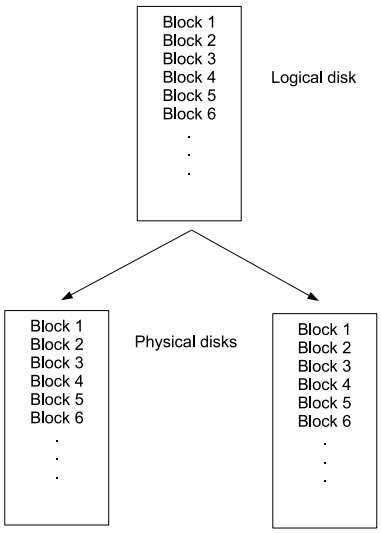

Figure 2.5: RAID 1.

2.5.3 RAID-3

Byte level striped set of disks with parity on a dedicated parity disk, see figure 2.6. RAID-3 needs at least three disks. All disks apart from the parity disk read at the same time. And all disks including the parity disk write at the same time. [13]

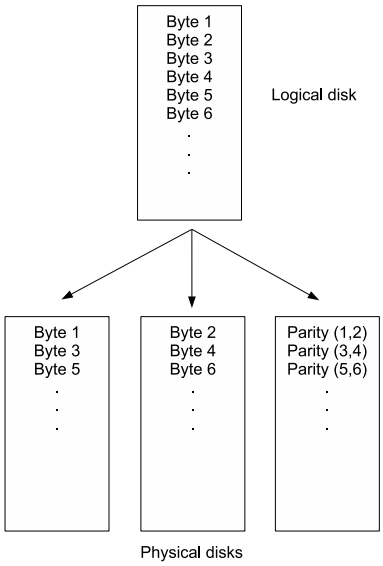

Figure 2.6: RAID 3.

2.5.4 RAID-4

Block level striped set of disks with parity on a dedicated parity disk, see figure 2.7. RAID-4 needs at least three disks. It has the same advantage as RAID-0 but the parity disk can be a bottleneck, because after all write operations new parity need to be calculated and written down. [13]

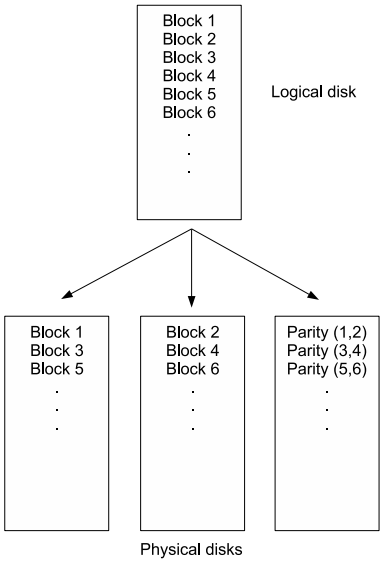

Figure 2.7: RAID 4.

2.5.5 RAID-5

Block level striped set with distributed parity, see figure 2.8. RAID-5 needs at least three disks. It has the same advantage as RAID-4 and because the parity is distributed between all disks we avoid the parity disk bottleneck. [13]

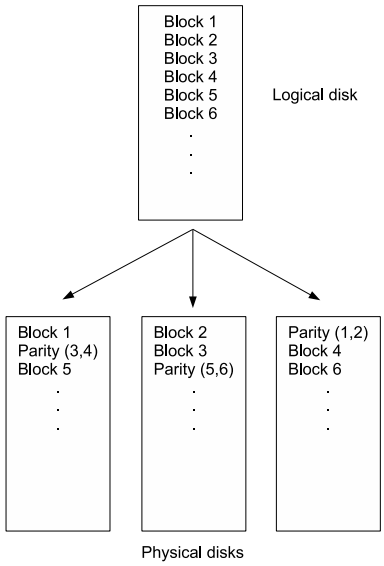

Figure 2.8: RAID 5.

2.5.6 RAID-6

Block level striped set of disks with double distributed parity, see figure 2.9. RAID-6 needs four or more disks. It is the same as RAID-5 but double parity, using parity and Reed-Solomon code, orthogonal dual parity or diagonal parity. [13]

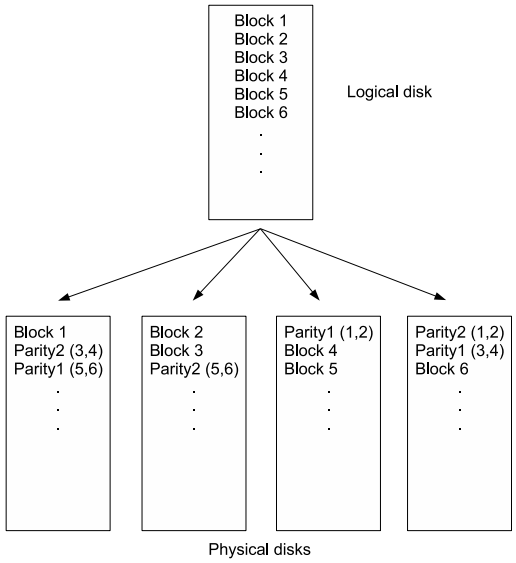

Figure 2.9: RAID 6.

2.6 Confidence Interval

Estimating a parameter by a single value is sometimes not precise enough. Instead we use an interval called confidence interval. Confidence interval is an interval that covers an unknown parameter with probability $1 - \alpha$. $1 - \alpha$ is called confidence level and should be as large as possible, in most cases 0.95, 0.99 or 0.999. Confidence interval is calculated from n samples, with chosen confidence level $1 - \alpha$.

I_m = Confidence Interval = $(\overline{x} - t_{\alpha/2}(f)d, \overline{x} + t_{\alpha/2}(f)d)$
$d = s/\sqrt{n}$
$f = n - 1$
s = sample standard derivation = $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$

$x_1, x_2, ...; x_n$ = the samples of the parameter.
$\overline{x}$ = arithmetic mean of the parameter.
n = number of samples.
$t_{\alpha/2}(f)$ = Student's t-distribution.

To get good precision we need the interval to be sufficiently small. To achieve that we need adequate number of samples, a rule of thumb is if we wants to halve the interval size, we need to quadruple the number of samples. [14]

2.7 File systems

This section describes some of the Linux filesystems.

In Linux there is a virtual filesystem, VFS, that is "a kernel software layer that handles all system calls related to a standard Unix filesystem" [15]. It provides a common interface to several kinds of filesystems. The filesystems supported by the VFS can be grouped into three main classes, disk-

based filesystems, network filesystems and special filesystems, see figure 2.10. Some examples of disk based filesystems are: ext2, ext3, Reiserfs, XFS, JFS, VFAT, NTFS, ISO9660 CD-ROM filesystem etc. Examples of network based filesystems are NFS, Coda, AFS, CIFS, and NCP. A typical example of a special filesystem is the proc filesystem [15].

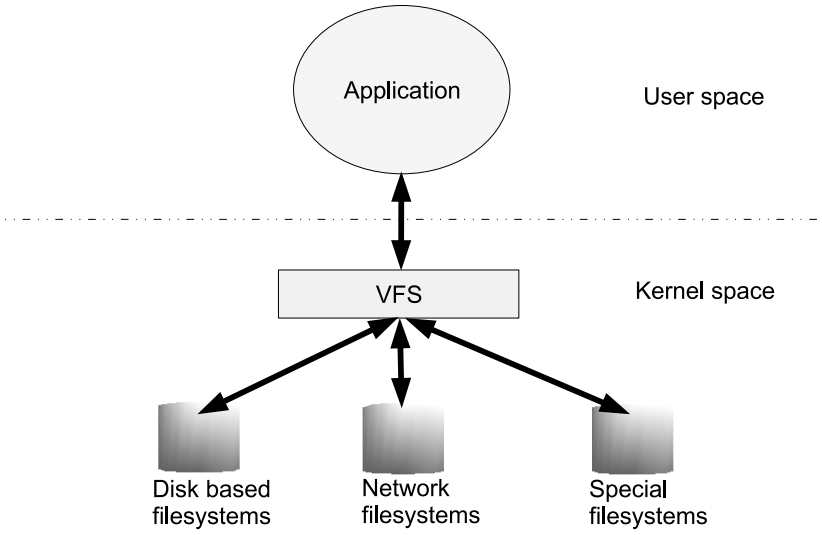

Figure 2.10: VFS role in file handling.

The disk based Linux file systems are quite different from each other but they all have in common that they implement a few POSIX APIs. The original Linux file system, ext, was developed from the minix file system. The ext filesystem was further developed to ext2. Later journaling was added to the file system that became ext3 [15].

Hans Reiser developed a filesystem called Reiserfs [16].

Silicon Graphics ported their filesystem XFS and IBM ported their filesystem JFS to Linux in the late 1990's.

Common to all disk based filesystems described here is that they all have

a superblock that contains information about the filesystem and inodes, where files and directory are represented persistently by inodes. "Each inode describes the attributes of the file or directory and serves as the starting point for finding the file or directory's data on disk" [15]. [15, 16, 17, 18]

In the VFS there are two caching mechanisms, dentry cache and inode cache [15].

Every time a new file object is read from disk a dentry object is created. The dentry object "stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file" [15]. The dentry objects are cached in a dentry cache in RAM. Further accesses to the dentry data "can then be quickly satisfied without slow access to the disk itself" [15]. Dentry cache "speeds up the translation from a file pathname to the inode of the last pathname component" [15].

When a file is opened an inode object is created in the inode cache, which stores corresponding inode information from disk [15].

Because these caches are in the VFS layer they are independent of the underlying filesystems. The caches works in the same way independent of file system type [15].

2.7.1 B+tree

B+trees is a tree data structure used by the filesystems Reiserfs, XFS and JFS. B+trees provides faster lookup, insertion, and delete capabilities than traditional filesystems with a linear structure. In B+trees the data are sorted in a balanced tree with data stored in the leaf nodes. The time complexity to search for data in a linear structure is O(n) and for a B+tree it is $O(log\ n)$. B+trees are for that reason more efficient than a linear data structure [19].

2.7.2 ext3

The ext3 filesystem is essentially the same as the ext2 filesystem with journal file added. This section describes ext2 and ext3 data structures and ext3 journaling.

The ext3 partition is split into a boot block and n block groups, see figure 2.11. The boot block is reserved and is not managed by the ext3 filesystem [15].

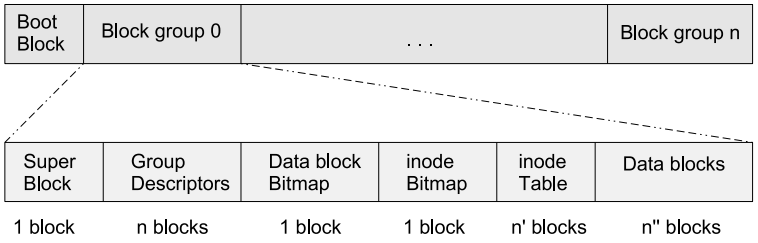

Figure 2.11: Ext3 partition layout.

The layout of each block group is shown in the lower part of figure 2.11. The superblock contains information about the filesystem, the total number of inodes, filesystem size in blocks, number of reserved blocks, free block counter, free inodes counter, block size, volume name etc. The first superblock, the superblock in block group 0, is the primary and is used by the filesystem. All other superblocks are spare and are only used by the filesystem check program [15].

The group descriptors in each block contains information about the number of free blocks in the group, number of free inodes in the group, number of directories in the group, the block number of the first inode table block, the block number of the inode bitmap and the block number of the block bitmap. Free blocks, free inodes and used directories are used when allocating new inodes and data blocks. They determine the most suitable block to allocate for each data structure. The two bitmaps contains 0's and 1's

corresponding to the inodes and data blocks in the block group. 0 means that the inode or the block is free and 1 that it is used. Each bitmap, that must be stored in a single block, describes the state of 8192, 16384 or 32768 blocks depending on the blocksize, 1024, 2048 or 4096 bytes. A small block size is preferable when the average file length is small because "this leads to less internal fragmentation–that is, less of a mismatch between the file length and the portion of the disk that stores it" [15]. Larger block sizes "are usually preferable for files greater than a few thousand bytes because this leads to few disk transfers, thus reducing system overhead" [15].

"The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes." [15] "All inodes have the same size: 128 bytes." [15] The inode contains the file attributes, file type and access rights, file owner, file length in bytes, last access time, time that the inode was last changed, time that the file contents last was changed, time of file deletion, the group that the file belongs to, hard link counter (number of filenames associated with the inode), number of datablock of the file, pointers to the file data blocks, etc [15].

The data blocks contain the contents of the files and directories [15].

"Ext2 implements directories as a special kind of file whose data blocks store filenames together with the corresponding inode number" [15], see figure 2.12. The directory entry parts are: inode number, length of the record, length of the filename, file type and the filename. The length of a directory entry is a multiple of 4 and, if necessary the filename are padded with null characters at the end, see figure 2.12. The record length can be interpreted as a pointer to the next record. The base address of the record plus the record size gives the address to the next directory entry [15].

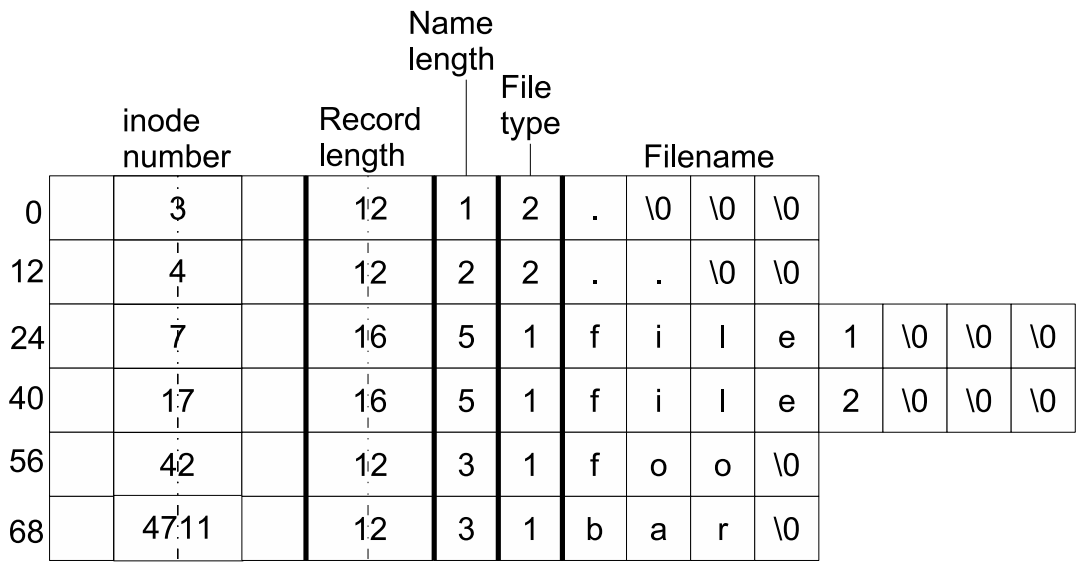

Figure 2.12: An example of a ext3 directory.

Because the files are stored in a linear structure in the directory the time-complexity for searching for a file in a directory in ext3 is $O(\mathrm{n})$, where n is the number of files in the directory.

Nonempty regular files consist of a group of datablocks. In the indexnode there is an array of 15 components (default value) that contains block numbers to logical disk blocks of the file, see figure 2.13 [15].

The first 12 components in the array, 0 – 11, yield the block numbers corresponding to the first 12 logical blocks of the file [15].

The 12th points to a block called indirect block, that represents a second-order array of blocks. "They correspond to the file block numbers ranging from 12 to $b/4+11$, where b is the filesystem's block size" [15]. Each block number is stored in 4 bytes therefore we divide by 4 [15].

The 13th component in the array contains the block number to an indirect block containing a second-order array of block numbers. The logical blocks pointed to by the second-order array contains the logical blocks $b/4+12$

to $(b/4)^2 + (b/4) + 11$ of the file [15].

The last index, 14, uses a triple indirection. The fourth-order arrays store the logical blocks from $(b/4)^2 + (b/4) + 12$ to $(b/4)^3 + (b/4)^2 + (b/4) + 11$ of the file [15].

This structure is an unbalanced tree with a shallow depth for the first part of the file and a larger depth for the end of the file. The worst case search complexity is $O(\mathrm{d})$, where d is the depth of the tree, se figure 2.13 [19].

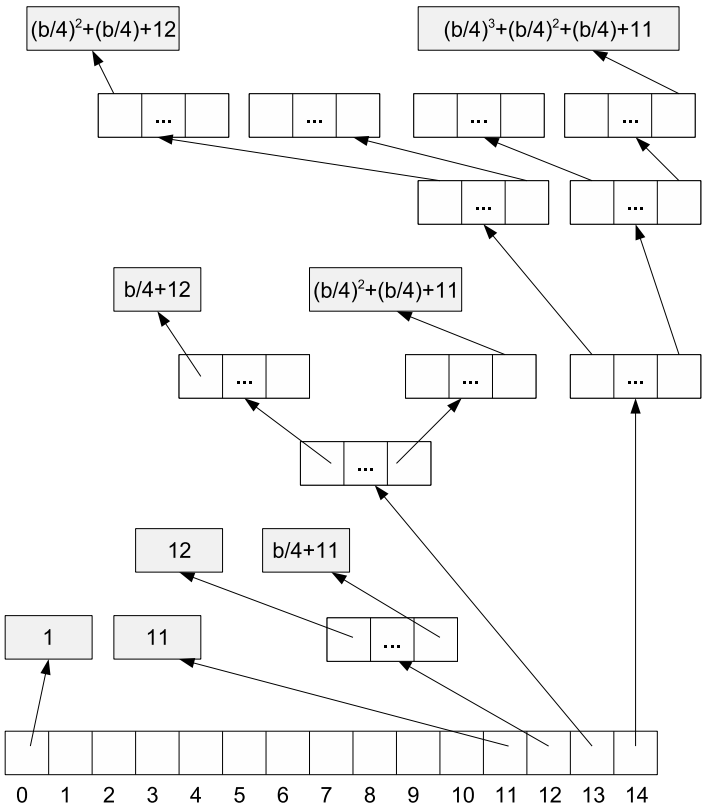

Figure 2.13: Data structures to address the file's data blocks.

When ext3 allocates logical blocks for a file it preallocates up to eight adjacent blocks. Ext3 also tries to allocate new blocks for a file near the

last block of the file. If that is not possible the filesystem searches for free blocks in the same block group that includes the file's inode. As a last resort the filesystem allocates free blocks from other block groups. Both these allocation methods reduces file fragmentation [15].

What distinguishes ext3 from its precursor ext2 is that ext3 has a special file, a journal file. The main idea behind journaling is to perform each high-level change to the filesystem in two steps. "First, a copy of the blocks to be written is stored in the journal; then, when the I/O data transfer to the journal is completed (in short, data is committed to the journal), the blocks are written in the filesystem." [15] When done the copies in the journal are discarded. If the system fails before the blocks are completely committed to the journal then, at recovery when the e2fsck program runs, it ignores them. If the system fails after the blocks are committed to the journal and before the blocks are written to the filesystem then at recovery the e2fsck program copies the blocks from the journal and writes them into the filesystem. In that way data can be lost but the filesystem is always consistent [15].

For efficiency reasons, "most information stored in the disk data structures of an ext2 partition are copied into RAM when the filesystem is mounted" [15].

2.7.3 Reiserfs

This section describes Reiserfs version 3. Reiserfs is a journaled file system and it uses balanced trees to store files and directories [16].

A reiserfs partition starts with 64 KB unused disk space, reserved for partition labels or boot loaders. "After that follows the superblock." The superblock contains information about the partition such as the block size, the number of free blocks, and the block number of the root and journal nodes. "There is only one instance of the superblock for the entire partition" in Reiserfs [16].

Directly following the superblock are n bitmap blocks, each mapping k blocks. "One bitmap block can address $(8 * blocksize)$ blocks." [16] If a bit is set in the bitmap block it indicates that the block is in use, a zero bit indicates that the block is free. The blocks follow the bitmap blocks [16].

There are 4 types of items in Reiserfs, stat data items, directory items, indirect items and direct items. The stat data item contains information about the file or directory such as file type, permissions, number of hard links, id of the owner, group id, the file size, last access time, the time that the file contents last was changed, the time the inode (stat data) was last changed, and an offset from the beginning of the file to the first byte of direct item of the file. For small files, the direct items contains the entire file, "all the necessary other information can be found in the item header and the corresponding stat item for the file" [16]. Larger files have pointers to indirect items which points to the blocks that belongs to the file. "Larger files are composed of multiple indirect items." [16] [20, 16]

Directory items describes a directory. Directories with few files uses one directory item and directories with many files will span across several directory items. The directory items contains headers and filenames, see figure 2.14. The directory headers have a hash value of the filename (called offset), an object id of the referenced item's parent directory, object id of the referenced item, offset to the name within the item, an unused bit and a bit indicating if the entry should be visible or not. "The file names are simple zero-terminated ASCII strings." [16] The hash value is used to search for file and directory names, and the items are sorted by the offset value [16].

Figure 2.14: A directory item in Reiserfs.

"The Reiser file system is made up of a balanced tree" [16], a B+tree. "The tree is composed of internal nodes and leaf nodes" [16], see figure 2.15. Each object, file, directory, or stat item, "is assigned a unique key, which can be

compared to an inode number in other file systems". "The internal nodes are mainly composed of keys and pointers to their child nodes." "Except for indirect items all the data is contained within the leaf nodes." [16] Each node is contained in a disk block. The leafs have level number 1 in the tree and the root node has the highest level [16].

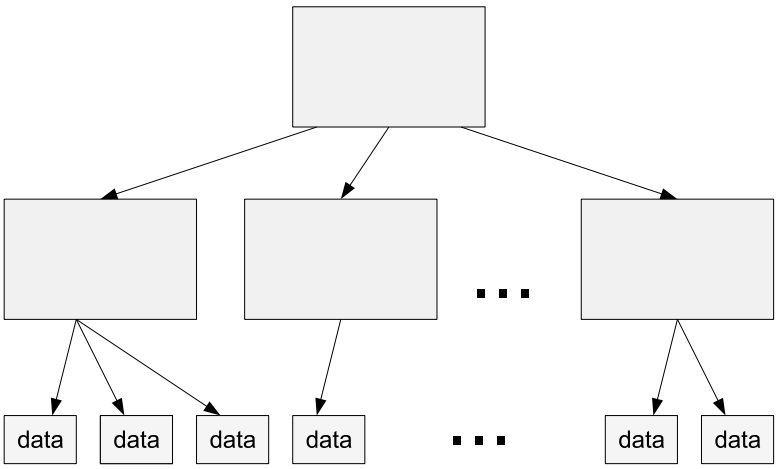

Figure 2.15: Reiser file system B+tree.

The keys are used to uniquely identify items, to find them in the tree and to achieve local groupings of items that belong together. The keys consists of four objects, the directory id, the object id, an offset and a type. Two objects in the same directory have the same directory id. When two keys are compared they are first compared by their directory id, and if they are equal by their object id and so on. For large files the Reiser file system uses multiple keys where the offset in the keys denotes the offset in bytes of the file [16].

When Reiserfs places nodes of the tree on the disk, it searches for the first empty block in the bitmap which it finds by starting at a location of the left neighbor of the node in the tree ordering, and moving in the last moved direction. This method was experimentally found by the developers to be better than "taking the first non-zero entry in the bitmap" [20], or

"taking the entry after the last one that was assigned in the direction last moved" [20], or "starting at the left neighbor and moving in the direction of the right neighbor" [20].

2.7.4 XFS

The XFS file system is a journaled file system. The XFS file system uses B+trees for almost all file system data structures [21].

"XFS is modularized into several parts" [21]. The space manager is a central and important module which manages the file system free space, the allocation of inodes, and the allocation of space within individual files. "The I/O manager is responsible for satisfying file I/O requests" [21]. "The directory manager implements the XFS file system name space." [21] The buffer cache is used by all these parts to cache contents of frequently accessed blocks in memory for efficiency reasons. The transaction manager is used by other pieces of the file system so all metadata updates can be atomic. [21].

The XFS partition is split into a number of equally sized chunks called AG, Allocation Groups. "Each AG has the following characteristics:" [17] "A super block describing overall filesystem info" [17] , "free space management" [17] and "inode allocation and tracking" [17]. The disk layout structure of an Allocation Group are shown in figure 2.16. The superblock in the the first Allocation Group is the primary and is used by the filesystem. All other superblocks are spare and are only used by xfs_repair. The first AG maintains global information about free space across the filesystem and total number of inodes. "Having multiple AGs allows XFS to handle most operations in parallel" [17].

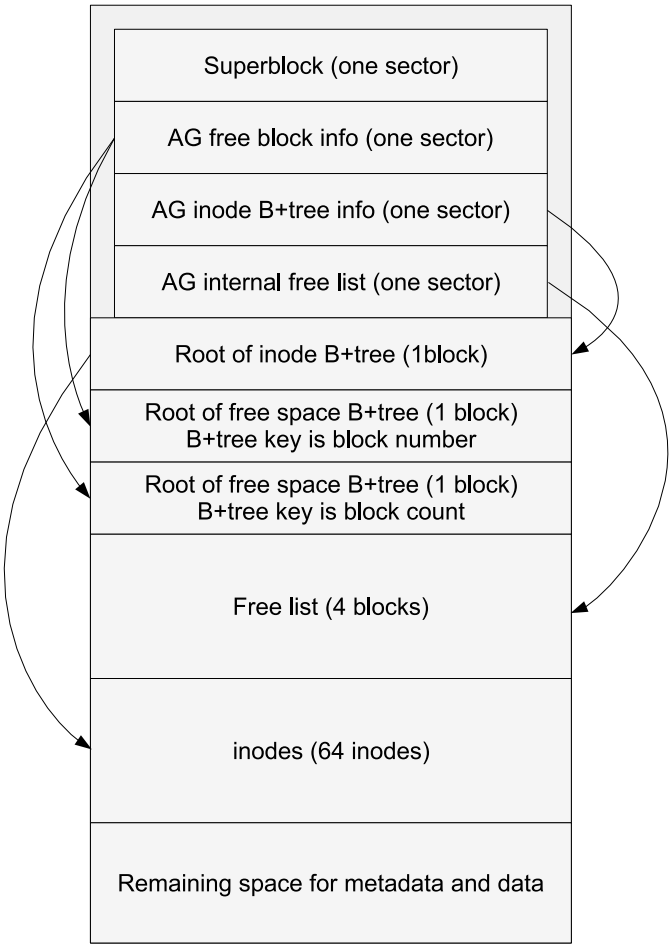

Figure 2.16: XFS AG layout.

The superblock contains information about the filesystem, the block size, total number of blocks available, the first block number for the journaling log, the root inode number in the inode B+tree, the number of levels in the inode B+tree, the AG relative inode number most recently allocated, the size of each AG in blocks, the number of AGs in the filesystem, the number of blocks for the journaling log, the underlying disk sector size, the inode size in bytes, number of inodes per block, name for the filesystem, etc [17].

XFS uses two B+trees to manage free space. One B+tree tracks free space by block number and the second B+tree by the size of the free space block. "This scheme allows XFS to quickly find free space near a given block or of a given size." [17] "The second sector in an AG contains the information about the two free space B+trees" [17].

Inodes are allocated in chunks where each chunk contains 64 inodes. A B+tree is used to track the inode chunks of inodes as they are allocated and freed. The inodes contain almost the same information as in the ext3 filesystem. XFS uses extents where the extents specify where the file's actual data is located within the filesystem. "Extents can have 2 formats" [17]: for small files the "extent data is fully contained within the inode which contains an array of extents to the filesystem blocks for the file" [17], for larger files the "extent data is contained in the leaves of a B+tree" [17] where the "inode contains the root node of the tree" [17], see figure 2.17 [17].

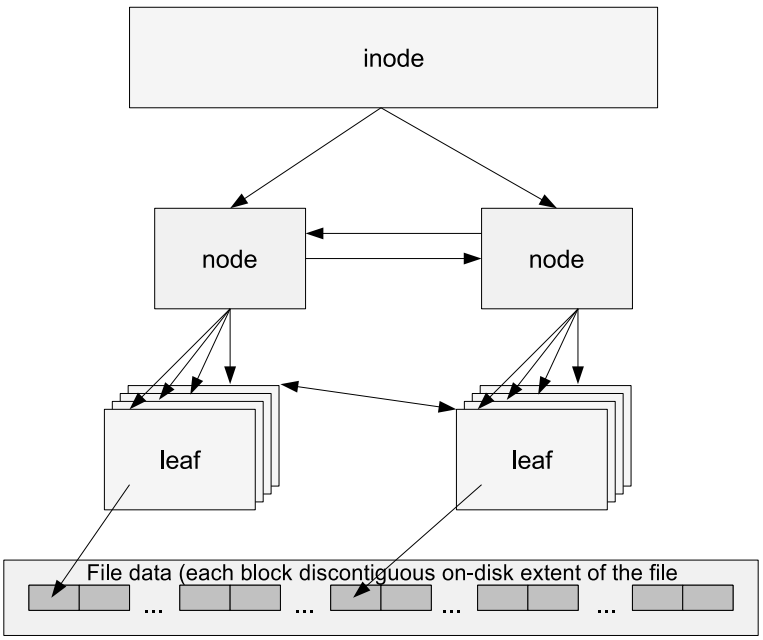

Figure 2.17: XFS file data structure.

"The data fork contains the directory's entries and associated data" [17] and the format of the entries can be one of 3 formats. For small directorys with few files the "directory entries are fully contained within the inode" [17]. For medium size directorys the "actual directory entries are located in another filesystem block" [17] and "the inode contains an array of extents to these filesystem blocks" [17]. For large directorys with many files the "directory entries are contained in the leaves of a B+tree" [17]. "The inode contains the root node of the tree" [17]. The number of directory entries that can be stored in an inode depends on the inode size, the number of entries, the length of the entry names and extended attribute data [17].

In the XFS filesystem the journaling is implemented to journal all metadata. First XFS writes the updated metadata to an in-core log buffer and then asynchronously writes the log buffers to the on-disk log [21].

2.7.5 JFS

The journaled file system, JFS, was developed by IBM for the IBM AIX Unix and later ported to Linux [18].

A JFS filesystem is built on top of a partition. The partition can be a physical partition or a logical volume. The partition is divided in a number of blocks where each block has the size *blocksize*. "The partition block size defines the smallest unit of I/O" [18]. There is one aggregate in the partition and it is wholly contained within the partition. "A fileset contains files and directories." [18] A fileset forms an independently mountable subtree. "There may be multiple filesets per aggregate" [18], see figures 2.18 and 2.19. [18, 22]

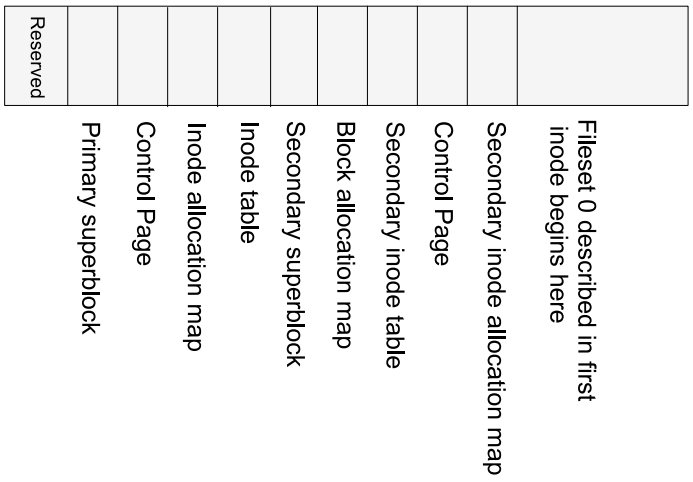

Figure 2.18: JFS aggregate with two filesets.

A fileset has a Fileset Inode Table and a Fileset Inode Allocation Map. The Inode Table describes the fileset-wide control structures. The Fileset Allocation Map contains allocation state information on the fileset inodes as well as their on-disk location. The first four, 0–3, inodes in the fileset are used for "additional fileset information that would not fit in the Fileset Allocation Map Inode in the Aggregate Inode Table" [22], root directory

inode for the fileset, inode for the Access Control List file. “Fileset inodes starting with four are used by ordinary fileset objects, user files, directories, and symbolic links” [22].

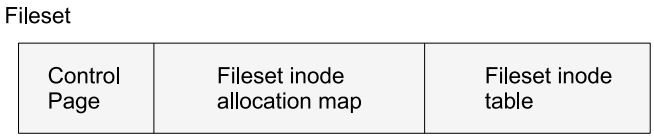

Figure 2.19: A JFS fileset.

The Aggregate superblock maintains information about the entire file system and includes the following fields: size of the file system, number of data blocks in the file system, a flag indicating the state of the file system, allocation group sizes. The secondary superblock is used if the primary superblock is corrupted [22].

The Aggregate inode table contains inodes describing the aggregate-wide control structures. The secondary Aggregate Inode Table contains replicated inodes from the inode table. The inodes in the inode table are critical for finding file system information therefore are they duplicated. The Aggregate inodes are used for describing the aggregate disk blocks comprising the Aggregate Inode Map, describing the Block Allocation Map, describing the journal log, describing bad blocks, and there is one inode per fileset [22].

The Aggregate Inode Allocation Map contains allocation state information on the Aggregate inodes as well as their on-disk location. The secondary Aggregate Inode Allocation Map describes the Secondary Aggregate Inode Table [22].

The Block Allocation Map describes the control structures for allocating and freeing aggregate disk blocks within the aggregate. It is used to track the freed and allocated disk blocks for an entire aggregate [22].

JFS allocates inodes dynamically in contiguous chunks of 32 inodes in a inode extent. This allows inode disk blocks to be placed at any disk ad-

dress, which decouples the inode number from the location. It eliminates the static allocation of inodes when creating the filesystem. "File allocation for large files can consume multiple allocation groups and still be contiguous" [22]. "With static allocation the geometry of the file system implicitly describes the layout of inodes on disk; with dynamic allocation separate mapping structures are required." [22] The Inode Allocation Map provides with the function of finding inodes given the inode number. "The Inode Allocation Map is a dynamic array of Inode Allocation Groups (IAGS)." [22] The mapping structures are critical to the JFS integrity [22].

For directories two different directory organizations are provided. The first is used for small directories, up to 8 entries, and stores the directory contents within the directory's inode (except . and .. which are stored in separate areas of the inode). This eliminates the need for separate block I/O and separate block storage. "The second organization is used for larger directories and represents each directory as a B+tree keyed on name" [18].

Allocation groups divide the space in an aggregate into chunks where allocation policies try to cluster disk blocks and disk inodes for related data to achieve good locality. Files are often read and written sequentially, and the files in a directory are often accessed together [22].

"A file is represented by an inode containing the root of a B+tree which describes the extents containing user data." [22] The B+tree is indexed by the offset of the extents. "A file is allocated in sequences of extents." [22] An Extent is a variable length sequence of contiguous aggregate blocks allocated to a JFS object as a unit. The extents are defined by two values, its length, measured in units of aggregate block size, and its address (the address to the first block of the extent). The extents are indexed in a B+tree [23, 22].

"JFS uses a transaction-based logging technique to implement recoverability." [23] JFS journals the file system structure to ensure that it is always consistent. "User data is not guaranteed to be fully updated if a system crash has occurred" [23]. When the inode information is updated for some inodes the inode information is written to the inode extent buffer. Then the record is written. "After the commit record has actually been written

(I/O complete), the block map and inode map are updated as needed, and the tlck'ed meta-data pages are marked 'homeok' " [23].

2.8 I/O Scheduling Algorithms

In Linux there are 4 algorithms for handling disk I/O requests, the noop elevator, the complete fairness queueing elevator, the deadline elevator and the anticipatory elevator. The heuristics is reminiscent of the algorithm used by elevators when dealing with request coming from different floors to go up or down. "The elevator moves in one direction and when it has reached the last booked floor the elevator changes direction and starts moving in the other direction" [15].

In the Linux 2.6 kernel up to 2.6.17, including the 2.6.8 used in our tests, the anticipatory elevator is the default algorithm. In 2.6.18 the kernel developers changed algorithm to the complete fairness queueing elevator as the default algorithm [15, 24].

The Noop elevator is the simplest algorithm of the four algorithms. There is no ordered queue and new requests are added either at the end or at the front of the dispatch queue. "The next request to be served is always the first in the dispatch queue" [15].

The Deadline elevator algorithm uses four queues and a dispatch queue. Two queues are sorted queues for read and write requests ordered according to their initial sector number. Two queues are deadline queues including the same read and write requests sorted according to their deadlines. The expire time for read requests is 500 milliseconds and for write requests 5 seconds. Read requests are privileged over write requests. The deadline ensures that the scheduler looks at a request if it has been waiting a long time [15].

"When the elevator replenishes the dispatch queue it determines the data direction of the next request and if there are both read and write requests to be dispatched, the elevator chooses the read direction, unless the write

direction has been discarded too many times" [15].

The elevator also checks the deadline queue relative to the chosen direction. If a request in the queue is elapsed, the elevator moves that request to the tail of the dispatch queue. It also moves a batch of requests taken from the sorted queue starting from the request following the expired one [15].

If there are no expired requests the elevator dispatches a batch of requests starting with the request following the last one taken from the sorted queue. When the tail of the sorted queue is reached the search starts again from the top [15].

The anticipatory elevator uses a dispatch queue, two deadline queues and two sorted queues. The algorithm is an evolution of the Deadline elevator. The two sorted queues are one for read requests and one for write requests. The two deadline queues, one for read requests and one for write requests, are sorted according to their "deadlines". The request are the same as in the two sorted queues [15].

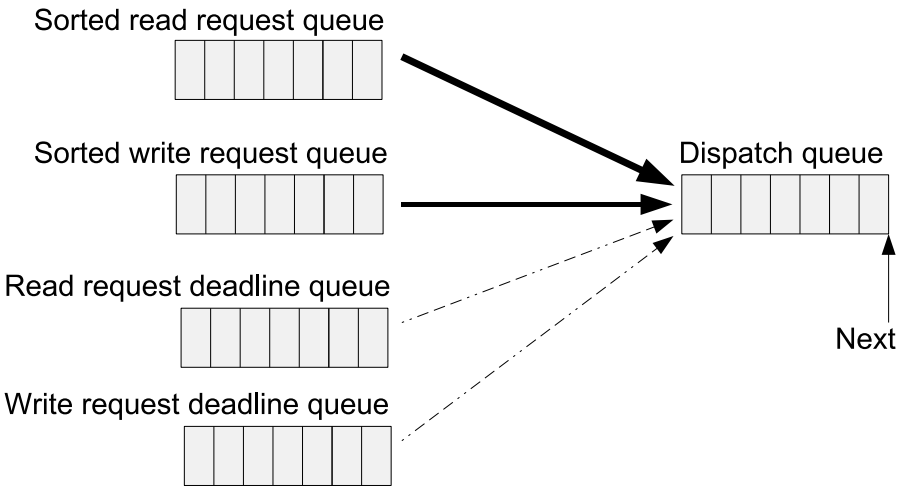

Figure 2.20: The anticipatory elevator.

The deadline queues are implemented to avoid request starvation, which occurs when the elevator policy for a very long time ignores a request be-

cause it handle other requests that are closer to the last served one. A request deadline is an expire timer that starts when the request is queued. The default expire time for read requests is 125 milliseconds and for write requests is 250 milliseconds. The I/O scheduler keeps scanning the two sorted queues, alternating between read and write request but giving preference to the read ones and moves the requests to the dispatch queue. The scanning is sequential, unless a request expires. The scheduler looks at the deadline queues and checks if there are any requests to expire, then it handles them and moves them to the dispatch queue, see figure 2.20.

If a request behind another one is less than half the seek distance of the request after the current position the algorithm chooses that one instead. This forces a backward seek of the disk head [15].

The anticipatory elevator saves statistics about the patterns of I/O operations. The elevator tries to anticipate requests based on the statistics. Right after dispatching a read request that triggered by some process, the anticipatory elevator checks if the next request in the sorted queue comes from the same process. If not the elevator looks at the statistics about the process and decides if its likely that the process will issue another read request soon, then it stalls for a short period of time [15].

The CFQ, complete fairness queueing, elevator uses a dispatch queue and a hash table with queues. The default number of queues in the hash table is 64. The hash function converts the thread group identifier of the current process into the index of a queue. The thread group identifier is usually corresponding to the PID, Process ID. The elevator scans the I/O input queues in a round-robin fashion and refills the tail of the dispatch queue with a batch of requests [15].

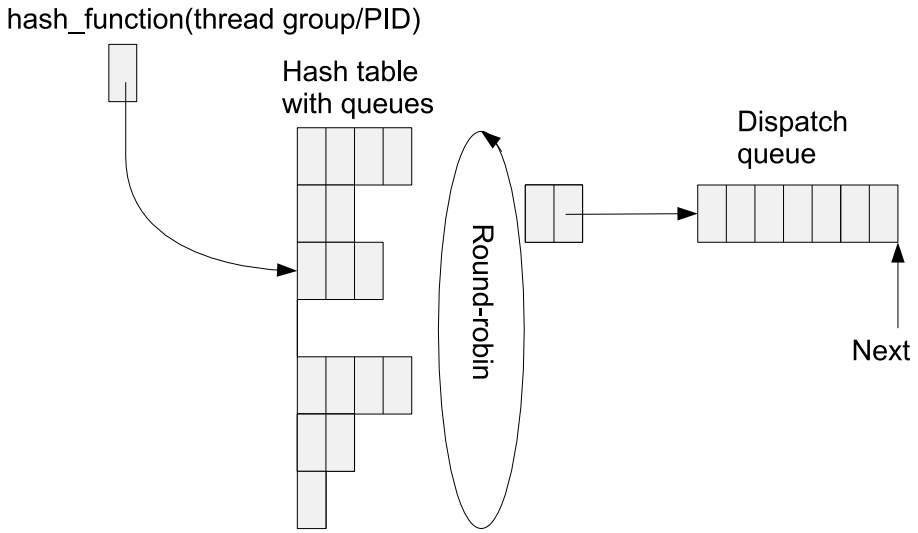

Figure 2.21: CFQ elevator.

2.9 Virtual Memory and Swap

"Virtual memory is the separation of user logical memory from physical memory." [25] "This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available" [25], see figure 2.22 [25].

"Virtual memory is commonly implemented by demand paging."[25] "When we want to execute a process, we swap it into memory." [25] "Rather than swapping the entire process into memory, however, we use a lazy swapper." [25] "A lazy swapper never swaps a page into memory unless that page will be needed". [25]

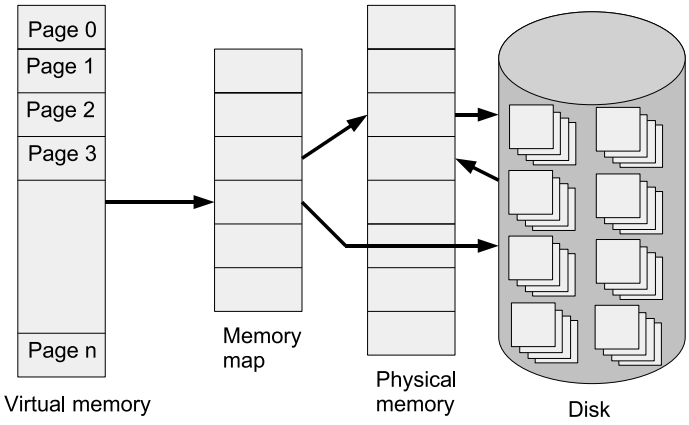

Figure 2.22: Virtual memory.

The virtual memory is mapped to physical memory and to swap space (usually a hard disk). When a process is started parts of the program, or the whole program, is loaded into the physical memory. When we start executing a process the operating system sets the instruction pointer to the first instruction of the process. If that page is not in physical memory a page fault occurs and the page is brought into memory. "After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory." [25] "In this way, we are able to execute a process, even though portions of it are not (yet) in memory." [25] When a process tries to allocate more memory it is given the memory if there is enough free physical memory. If there is no free page frame the OS uses a page-replacement algorithm to select a victim page. If the victim page has been changed it is first written to swap space before it is reused. If a process accesses a swapped-out page a page fault occurs and the page is swapped in. When swapping in a page it might be neccessary to first swap out another page [25].

Virtual memory in Linux is implemented by demand paging. It uses the PFRA, Page Frame Reclaiming Algorithm. The main goal of the algorithm is to pick up page frames and make them free. PFRA handles page

frames in different ways depending on what content they have. We can distinguish between unreclaimable pages, swappable pages, syncable pages and discardable pages. For unreclaimable pages reclaiming is not allowed or needed. Example of such pages are reserved pages, pages dynamically allocated by the kernel, pages in the kernel mode stacks of the processes and temporarily locked pages. Swappable pages are anonymous pages in the user mode address area (user mode heap and stack) and mapped pages from the tmpfs filesystem. When a swappable page is reclaimed the kernel saves the page contents on the swap area first. Syncable pages must be synchronized with their image on disk before they can be used. Discardable pages are unused pages included in memory caches and unused pages of the dentry cache (cache of links between file names and index nodes). The discardable pages can be used without saving the page contents first [15].

When swapping out a page the kernel tries to store it in contiguous disk slots to minimize disk seek time when accessing the swap area. If more than one swap area is used the kernel starts to use the area with the highest priority. "If there are several of them, swap areas of the same priority are cyclically selected to avoid overloading one of them." [15] If there is no free slot in the highest priority areas the search continues with the swap areas that have a priority next to the highest one, and so on [15].

In many computer architecture and in the 80x86 architecture the page frame memory have hardware constraints that limit the way page frames can be used. There are in particular two hardware constraints that the Linux kernel must deal with in the 80x86 architecture [15]:

- "The Direct Memory Access (DMA) processors for old ISA buses have a strong limitation: they are able to address only the first 16MB of RAM." [15]
- "In modern 32-bit computers with lots of RAM, the CPU cannot directly access all physical memory because the linear address space is to small." [15]

To cope with these limitations the memory in the Linux 2.6 kernel, the

physical memory is partitioned into three zones. In the 80x86 architecture the zones are [15]:

ZONE_DMA Contains frames of memory below 16MB

ZONE_NORMAL Contains frames of memory from 16MB to 896MB

ZONE_HIGHMEM Contains frames of memory from and above 896MB

In the Linux kernel all memory pages belonging to the User Mode address space and to the page cache are grouped into two lists, the active list that tends to include the pages that have been accessed recently and the inactive list that tends to include the pages that have not been accessed for some time. The function refill_inactive_zone() moves pages from the active list to the inactive list. The function must not be too aggressive and move a large number of pages from the active list to the inactive list. In that case the system performance will be hit."On the other hand, if the function is too lazy, the inactive list will not be replenished with a large enough number of unused pages" [15]. In that case the PFRA will fail in reclaiming memory.

In the kernel there are a group of kernel threads called pdflush. The pdflush systematically scan the page cache looking for dirty pages, pages that has been changed, to flush [15].

2.9.1 Swappiness

The refill_inactive_zone() function uses swap tendency which regulates its behaviour. "The swap tendency value is computed by the function as follows" [15]:

"swap tendency = mapped ratio / 2 + distress + swappiness" [15]

"The mapped ratio value is the percentage of pages in all memory zones that belong to User Mode address spaces (sc->nr_mapped) with respect to the total number of allocatable page frames. A high value of mapped_ratio

means that the dynamic memory is mostly used by User Mode processes, while a low value means that it is mostly used by the page cache." [15]

"The distress value is a measure of how effectively the PFRA is reclaiming page frames in this zone; it is based on the scanning priority of the zone in the previous run of the PFRA" [15]. 12 is the lowest priority and 0 is the highest. The distress value is computed based on the zones previous priority, see table 2.2 [15].

Zone's previous priority	12..7	6	5	4	3	2	1	0
Distress value	0	1	3	6	12	25	50	100

Table 2.2: Distress value.

Swappiness is a user defined constant which may be tuned by the system administrator by writing in the /proc/sys/vm/swappiness file or by using the sysctl command (issuing the sysctl() system call). The swappiness value is usually set to 60 [15].

"Pages will be reclaimed from the address space of processes only if the zone's swap tendency is greater or equal to 100." [15] With a swappiness value of 0 the PFRA never reclaims pages in the User Mode address space unless the zone's previous priority is zero, which is an unlikely event. With a swappiness value of 100 the PFRA reclaims pages in the User Mode address space at every invocation [15].

2.9.2 vfs_cache_pressure

The /proc/sys/vm/vfs_cache_pressure, and sysctl vm.vfs_cache_pressure, control the tendency of the kernel to reclaim the memory which is used for caching of directory and inode objects.

"At the default value of vfs_cache_pressure=100 the kernel will attempt to reclaim dentries and inodes at a "fair" rate with respect to pagecache and swapcache reclaim. Decreasing vfs_cache_pressure causes the kernel

to prefer to retain dentry and inode caches. Increasing vfs_cache_pressure beyond 100 causes the kernel to prefer to reclaim dentries and inodes." [26]

2.9.3 dirty_ratio and dirty_background_ratio

With these parameters it is possible to control the syncing of dirty data.

The /proc/sys/vm/dirty_ratio, and the sysctl vm.dirty_ratio, "contains, as a percentage of total system memory, the number of pages at which a process which is generating disk writes will itself start writing out dirty data" [26].

The /proc/sys/vm/dirty_background_ratio, and the sysctl vm.dirty_background_ratio, "contains, as a percentage of total system memory, the number of pages at which the pdflush background writeback daemon will start writing out dirty data" [26].

2.9.4 min_free_kbytes

The /proc/sys/vm/min_free_kbytes, and sysctl vm.min_free_kbytes, is used to force the Linux VM to keep a minimum number of kilobytes RAM free. "The VM uses this number to compute a pages_min value for each lowmem zone in the system. Each lowmem zone gets a number of reserved free pages based proportionally on its size." [27]

2.9.5 page-cluster

"The Linux VM subsystem avoids excessive disk seeks by reading multiple pages on a page fault. The number of pages it reads is dependent on the amount of memory in your machine." [27]

"The number of pages the kernel reads in at once is equal to $2^{page-cluster}$. Values above 2^5 don't make much sense for swap because we only cluster

swap data in 32-page groups." [27]

The value is defined bye the /proc/sys/vm/page-cluster, and the sysctl vm.page-cluster. [27]

2.9.6 dirty_writeback_centisecs

The /proc/sys/vm/dirty_writeback_centisecs, and sysctl vm.dirty_writeback_centisecs, controls how often the pdflush writeback daemon will wake up and write old data out to disk. "This tunable expresses the interval between those wakeups, in 100'ths of a second." [26].

Chapter 3

Realization

The goal of this thesis is to examine if it is possible to increase the size of the running simulations with 50% more NEs on existing hardware with operating system environment tuning. To see if it was possible we needed to run test runs and measure the test runs by comparing them with a reference test.

We decided to test one parameter at a time and see which of them made improvements.

3.1 The test environment

The test environment consisted of 2 computers, one for the NETSim simulations, running Suse Linux 9.2, and one for the load generator, running Solaris 9, and a local network between them. The NETSim computer was an HP DL145 G1 with two single core AMD Opteron 2.4 GHz CPUs, 8GB RAM and two Ultra SCSI disks. Unfortunately they were not the only computers on the local network. However the network load was low and

should therefore not have affected on our test runs. We wanted to test only the simulations, not the combination of simulation and load generator, therefore the need of two computers. We wanted to be independent of external servers, so we used local user accounts. The test environment is a standard test environment for NETSim.

The computer running NETSim was reinstalled before the test runs with a standard installation of Suse 9.2. The installation of Linux was automated with autoyast/jumpstart and the installation of NETSim was automated with shell scripts which give that the NETSim computer's software and configuration always was exactly the same for all test runs.

Our test nets consisted of 100 NEs, Network Elements, where 98 NEs was RBS's, one RNC and one RXI.

To find out where the limit for the computer running the simulations was we ran simulations with 8, 9, 10 and 11 nets. During these test runs and all the rest of the test runs we supervised the load generator by measuring the alarm rate, PM, AVC, nesync and SWUG with the program moodss, see figure 3.1. Moodss is a opensource software which can be found at http://moodss.sourceforge.net/ . Moodss is frequently used at NETSim at Ericsson.

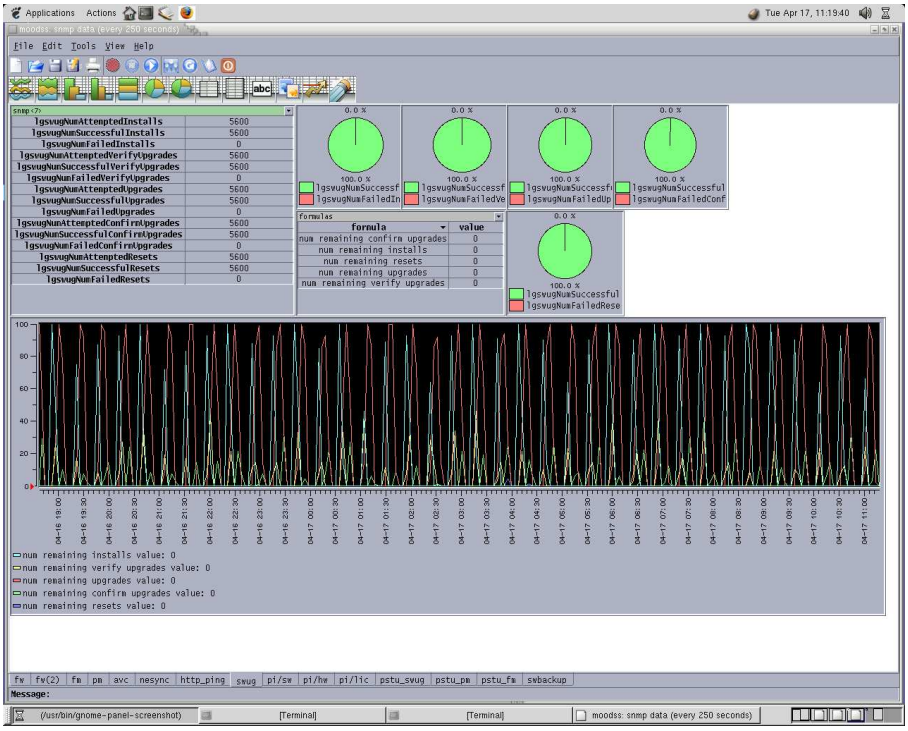

Figure 3.1: Moodss supervising the load generator.

We also supervised the two computers load, memory usage and swap usage with moodss, see figure 3.2.

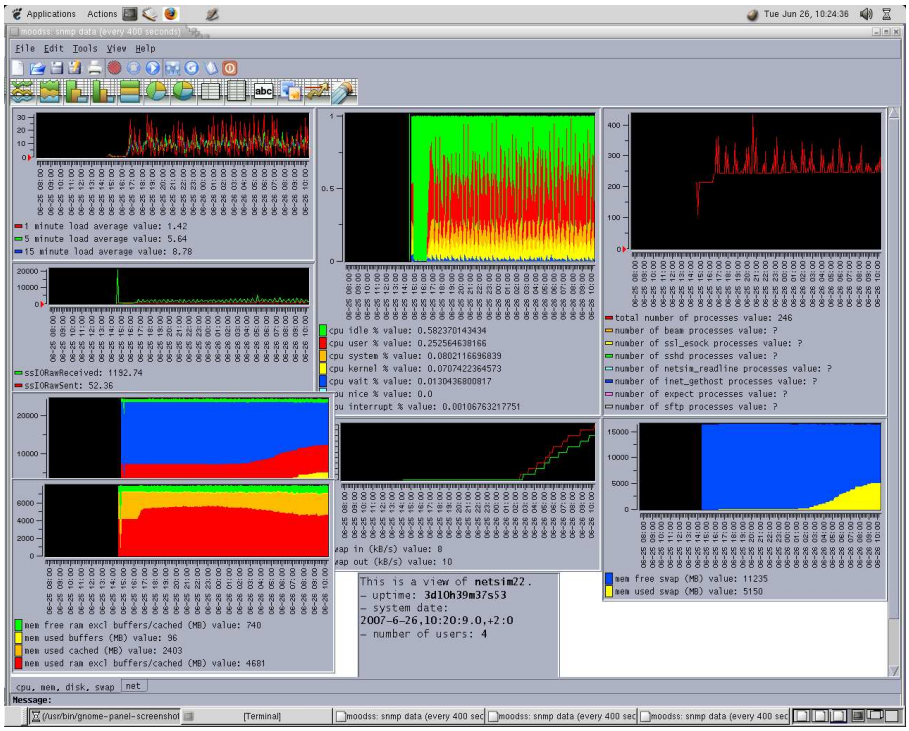

Figure 3.2: Moodss supervising a computer running NETSim.

Apart from monitoring the load, memory usage and swap usage with moodss we did run extra tests to study the processor usage and network usage.

The maximum response time is 2 minutes for all of the above. This 2 minute limit is defined by OSS. At 11 nets the time limit was exceeded with some errors as a consequence. Exceeding the time limit is an error. The conclusion was that 10 nets with 1000 NEs was the upper limit for this computer without optiminations applied.

The test run with 10 nets then became our reference run. All parameters had their default values in this test. Thus the goal of this thesis is to examine if it is possible to run NETSim with 15 nets on the computer we

used for the NETSim tests.

When changing a parameter on the computer running the simulations the computer was first reinstalled. The only exception was for the virtual memory parameters which could be changed without reinstalling or reboot the computer.

All NETSim files and load generator temporary files were located on the same local file system during all our tests. Load generator temporary files is the part of the load generator that resides on the NETSim computer.

3.1.1 Test runs and measurement of IDL methods

The test runs were realized by starting NETSim and starting the test nets and then running background load 2 (see page 15) plus software upgrades. All tests took at least 15 – 17 hours because it took at least 15 – 17 hours for the swap to level out. Before we could start the test runs the computer was reinstalled, NETSim was reinstalled and the test nets were reinstalled and then started. Reinstalling the computer and NETSim and starting all the nets took 1 – 2 hours. This gives a total of 16 – 19 hours for each test run.

The load generator logs times and response times for the IDL, interface descriptions language,methods in log files on the computer running the load generator. These include the three IDL methods below. The load generator only writes to the logs when something happens. That means that there can be different amounts of measure points for two test runs. The unit for the logged response times is milliseconds.

The response times of three IDL methods were measured from the load generator logs, basic_get_MO_containment_short, get_MO_attributes, and basic_create_MO. These three make most load on the computer of all IDL methods. The basic_get_MO_containment_short is part of a topology sync. Topology sync is where the load generator makes a snapshot of the whole MO tree. The get_MO_attributes is part of an attribute sync, where sev-

eral MO's with several IDL methods are synced by the load generator. The basic_create_MO is done as a part of the SWUG where the NEs collect an Upgrade Control File, an XML file, who parse the XML file. The XML file contains information about the software upgrade such as where to find the software file and how to get it.

Mean value calculations with sliding window

To be able to interpret the test results we did a mean value calculation of the response times by dividing them in two intervals and calculate one interval at a time. Without doing that the oscillation of the response times for the IDL methods were too fast to see anything and it was impossible to draw any conclusions.

We divided the two test runs to be compared into intervals, where the number of intervals were equal to the number of SWUG cycles. A SWUG cycle is from the start of a SWUG to the start of the next SWUG. The intervals from the two compared runs where synced with the SWUG cycles by using the start and end times from the SWUG cycles, see figure 3.3.

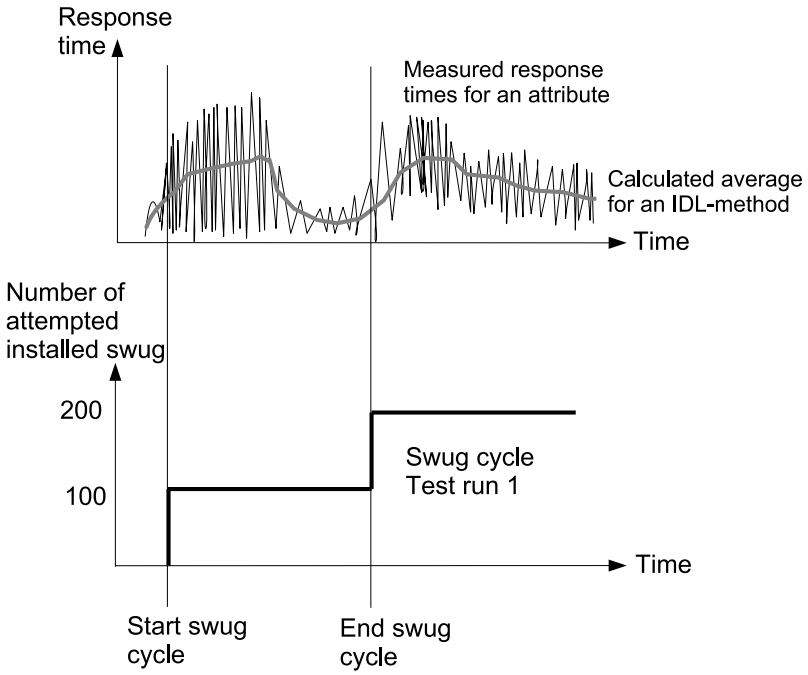

Figure 3.3: Mean value calculation of response times.

Each SWUG cycle was divided into equal partial intervals to get an equal

number of mean values for the two test runs. Otherwise the width of the two calculated mean value graphs would differ and it would be impossible to compare them. Further we used a sliding window and calculated the mean value for the whole window and then moved the window forward and calculated a new mean value and continued so to the end of the SWUG cycle, see figure 3.4. The calculation then continued with the next SWUG cycle.

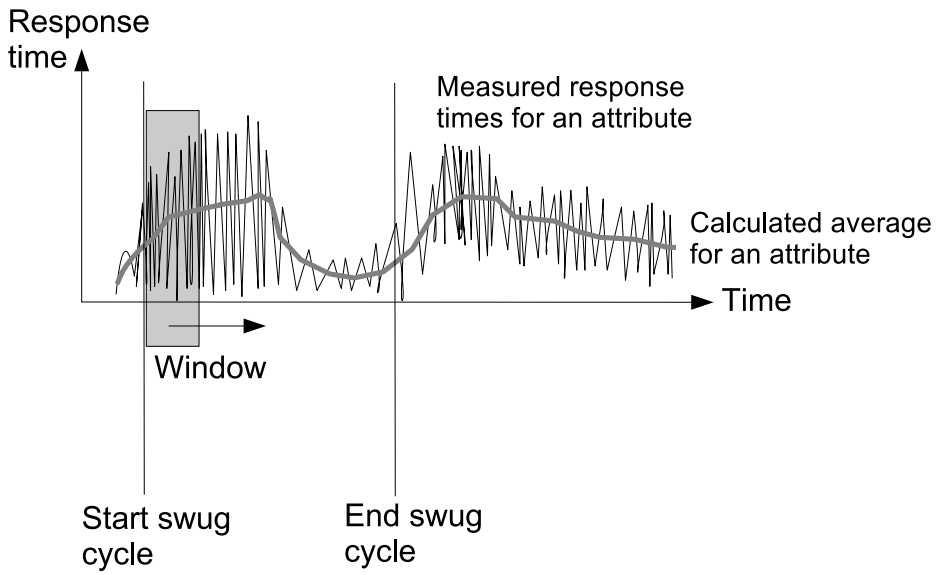

Figure 3.4: Mean value calculation with a sliding window.

The two mean value curves were then plotted into one graph to a file with gnuplot. Three graphs, one for each of the three IDL methods, were created.

Mean value calculations

The next thing we did was to calculate one mean value for each SWUG cycle and plot them in a graph and calculated the least square for a third degree function [28, 29] with gnuplot and plotted in the same graph.

```
f(x) = a1*x*x*x + b1*x*x + c1*x + d1
g(x) = a2*x*x*x + b2*x*x + c2*x + d2
fit f(x) "file1" via a, b, c
fit g(x) "file2" via d, e, h

plot f(x) title "˜s" with lines lt 9, g(x) title "˜s" with lines lt -1
```

In separate graphs the mean values were complemented with whisker bars with median, quartile 1 and 3 and min95- and max95- values. 1st and 3'd quartile is the median of the lower respectively upper half of the data, where the median of the whole set is not included. Max95 and Min95 is the greatest respectively smallest value of the data, not included 5% of the outliners. Outliners are the 5% extremest values.

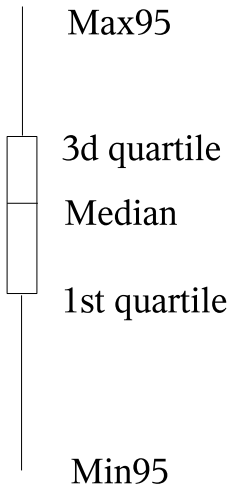

Figure 3.5: A whisker bar.

The graphs show the deviation for the measured IDL methods in one test run. See figure 3.6 for an hypothetic example.

These two graphs were easier to calculate, took less time to calculate and gave the same result as the first type with sliding window.

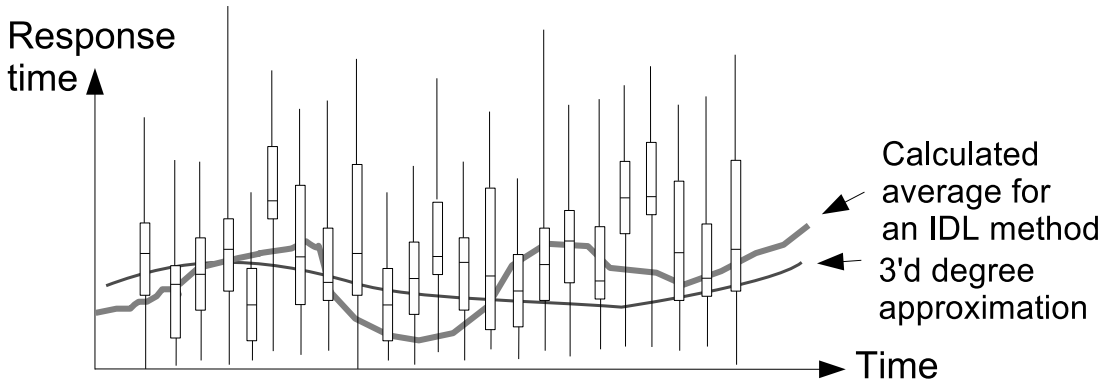

Figure 3.6: Mean value calculation with whisker bars and least square for a third degree function.

Times for SWUG's

SWUG's are heavy and cause a lot of load on the NEs that are upgraded during the software upgrade. It is therefore interesting to study the time it takes to do all SWUG's. We complemented with studying and comparing the time it took for one SWUG cycle from start to nesync at the end of the SWUG cycle, see figure 3.7.

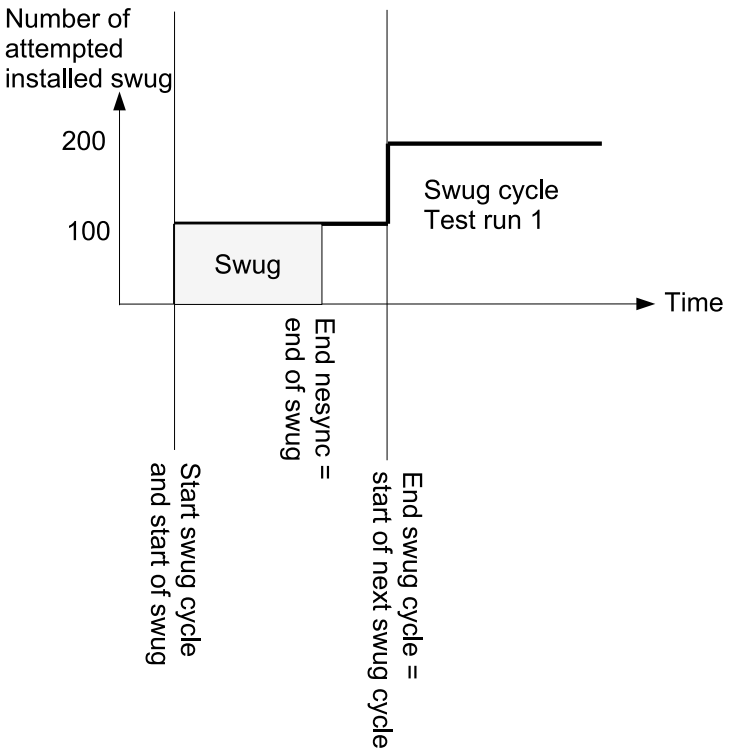

Figure 3.7: Start and end of a SWUG.

The sample times are discrete values and the start of a SWUG can occur between two values and the end of an nesync can occur between two values. To see the time differences the graphs contains the minimum and maximum times for a SWUG from start of SWUG to end of nesync, see figure 3.8.

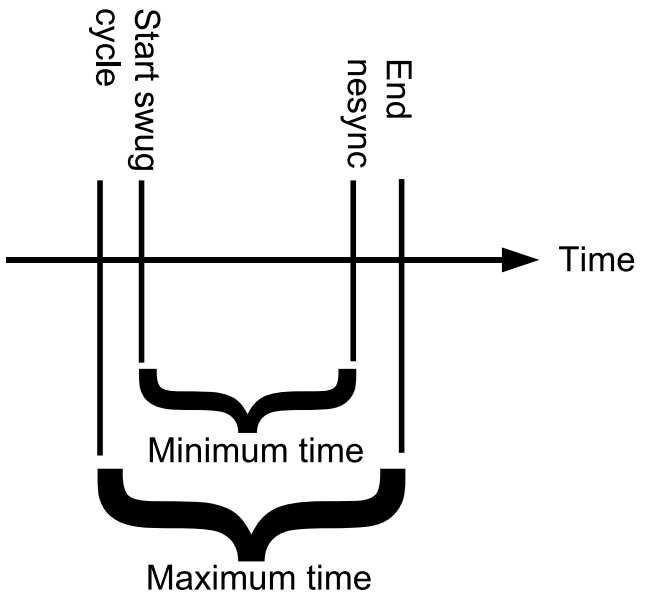

Figure 3.8: Maximum and minimum time difference for SWUG's.

See figure 3.9 for an example of a plot of time difference for a number of SWUG's in a hypothetic test run.

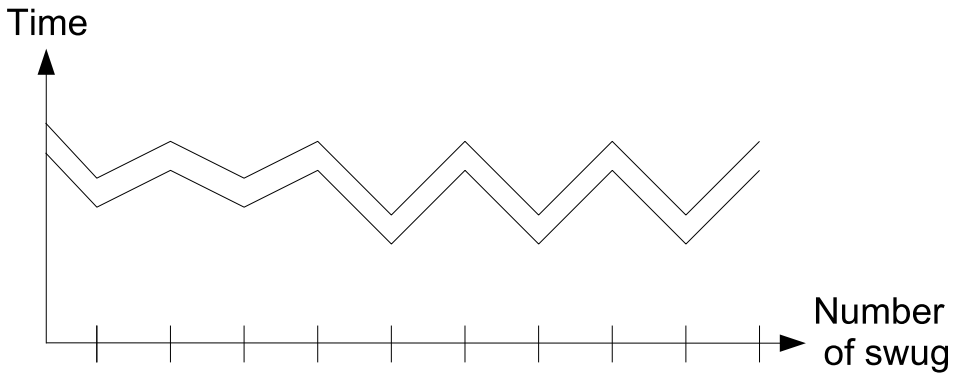

Figure 3.9: Time difference for SWUG's.

Two time difference curves from two test runs were plotted in the same graph. See figure 3.10 for a hypothetic example.

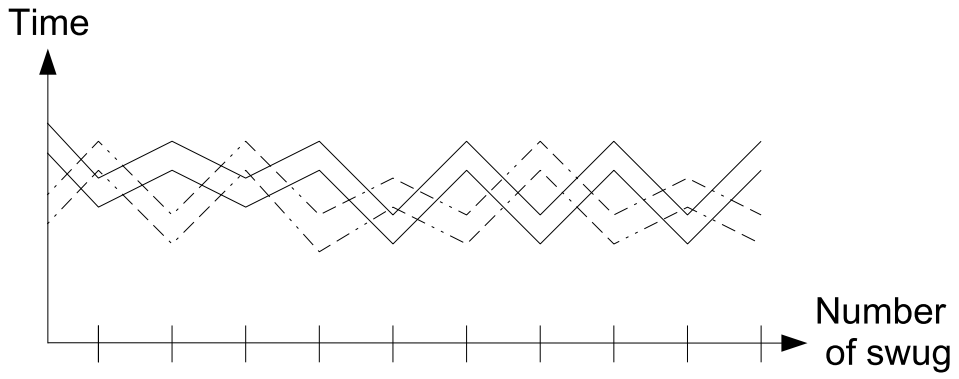

Figure 3.10: Time difference for SWUG's for two test runs.

Disk access load

To measure the disk access load, read and write from and to disk, during a test run we ran a test on a Sun Fire T2000 with Solaris 10 and measured all disk activities with a dtrace script rwsnoop. The hard drives were two SAS interfaced, Serial Attached SCSI, FUJITSU MAY2073RCSUN72G 73.4GB with an approximate read and write speed at 300 Mbyte/s, Interface SAS (3Gbps) [30]. The result was summarized, in Mbyte, with the following awk script.

```
awk 'BEGIN {c = 0} { c = c + $9} END { print c/(1024*1024)}' datafile
```

The results are shown in table 3.1 and peak values in figures 3.11 and 3.12.

PM/SWUG/ Background load	Number of nets	MByte data	Time (minutes)	MByte/s
Background load	1	7.1478	8	0.01
PM	1	121.353	4	0.5
PM	2	241.246	3	1.3
SWUG	1	131.953	15	0.1
SWUG	1	137.172	15	0.2
SWUG	2	272.583	17	0.3

Table 3.1: Disk accesses.

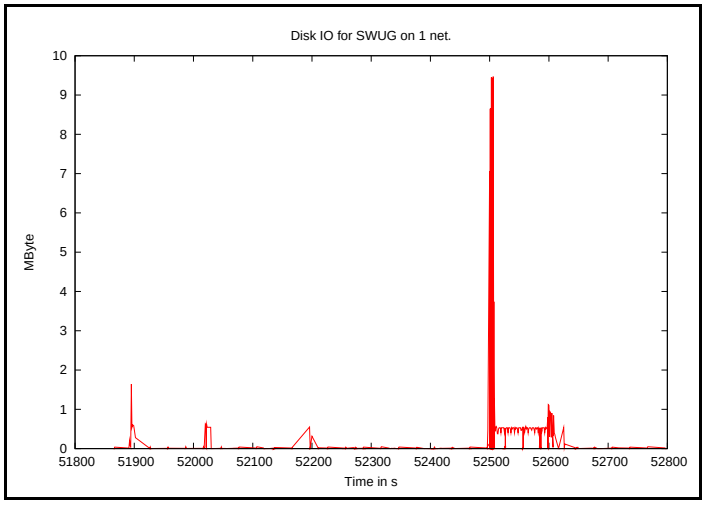

Figure 3.11: SWUG with 1 net.

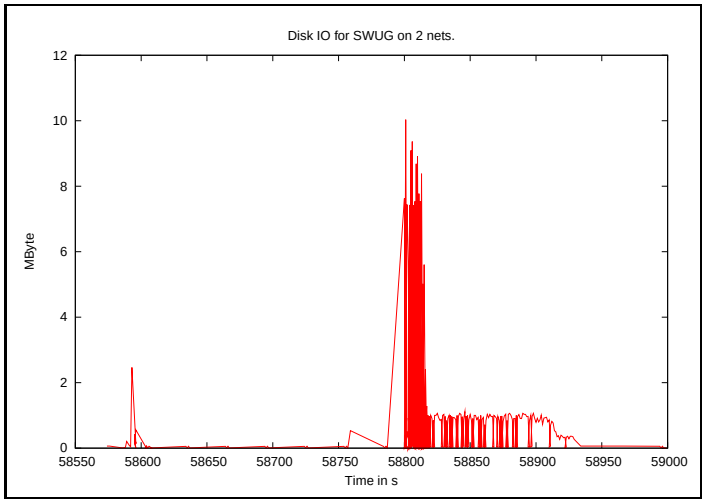

Figure 3.12: SWUG with 2 nets.

We did not run any tests with 10 nets on the Sun Fire T2000 because of a heavy-load bug in the communication between the Erlang process and the SSL module.

The disk load is approximately but not exactly linear regard to number of net and MByte/s. With a linear approximation this gives 5 MByte/s for PM on 10 nets and 1 – 2 MByte/s for SWUG on 10 nets. Compared to the speed for the PCI bus, the SAS interface and the speed for the hard drive disk read and write of 1 – 5 MByte/s is very little. The computer we used for almost all test runs have Ultra SCSI disks at a speed of 320 MByte/s which is also much more than 1 – 5 MByte/s.

Peak disk load is slightly below 10 MB/s for 1 net and slightly above 10 MB/s for 2 nets. That indicates small increments in peak disk load.

With these numbers we can conclude that there are relatively few disk accesses during a simulation run.

3.2 The tests

3.2.1 RAID

We choose RAID-0 because we do not need fault tolerance, and just want improved I/O performance. We implemented Suses built-in software RAID on two physical disks with 4 K bytes chunk size. The root- and swap-partitions were not RAID:ed and each lying on one of the two disks 3.13.

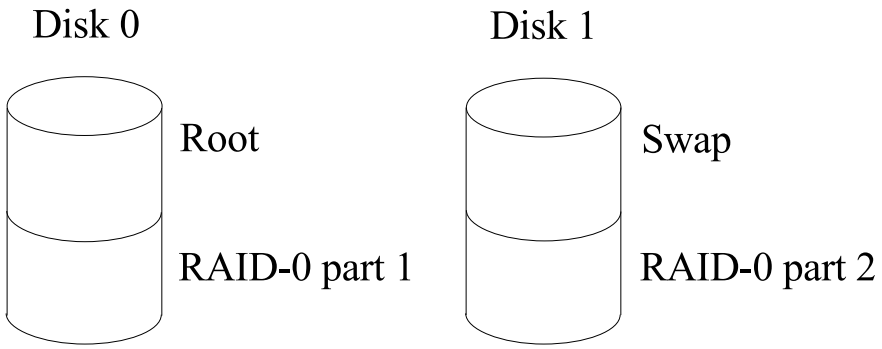

Figure 3.13: RAID-0 with two disks.

3.2.2 Crypto Accelerator Card

Because NETSim use Corba over SSL for comunication, heavy crypto calculations could be eliminated with a crypto accelerator card. We tested Sun Crypto Accelerator 6000 PCI-E Board on a HP ProLiant DL145 G2 with two dualcore AMD Opteron model 285 2.6GHz processors running Suse 9. The crypto accelerator card supports hash functions SHA1 and MD5, block ciphers DES, 3DES and AESSun and RSA/DH public key, with lengths of 512-2,048 bits. Sun Microsystems Inc. does not officially support other platforms than Sun workstations and servers, but supports Suse 9. We use OpenCryptoki as a bridge between OpenSSL and Suns

device driver. OpenCryptoki is a PKCS#11, Public Key Cryptography Standard, wrapper. OpenCryptoki and OpenSSL are open source, while Suns device driver is proprietary. With OpenSSL and Suns crypto accelerator card, we measure double performance in RSA verifying and signing. We could not test NETSim with Suns crypto accelerator card, because Suns device driver did not work faultlessly on the HP platforms.

3.2.3 File systems

The file system used in the reference test was Reiserfs version 3. We tested the other major file systems in Linux, ext3, XFS and JFS and compared them with Reiserfs. For ext3 we did two tests, one with default values and one with index and no update of atime.

In NETSim there are relatively few disk accesses, so changing the file system should only have a small impact on the test run.

3.2.4 I/O Scheduling Algorithms

In Linux there are 4 I/O scheduling algorithms. The two most interesting of them are the Anticipatory elevator and the CFQ elevator. They are interesting because the Anticipatory elevator was the default elevator in the Linux kernel up to version 2.6.17 and the CFQ elevator is the default elevator in the kernel from 2.6.18. Therefore we wanted to test and compare these two algorithms with each other. Both algorithms were tested with a 2.6.8 kernel.

The I/O scheduling algorithm should have little impact on the relatively few disk accesses but it should have more effect on the swap usage when running large simulations with large memory usage with much swapping.

3.2.5 Virtual Memory and Swap

We tested swap on two partitions on two different disks with the same priority for both swap partitions. This should have some effect on large simulations with large memory usage especially when the system starts to swap and needs to swap very much. The effect might be less if the NETSim partition lays on the same disk as one of the partitions. In that case NE disk accesses might occur at the same time as swap accesses so the read/write head needs to move a lot over the disk. If that happens it might be better to have the swapspace on a separate disk.

The virtual memory parameters that we tested was swappiness, vfs_cache_pressure, dirty_ratio, dirty_background_ratio, min_free_kbytes, page-cluster, and dirty_writeback_centisecs.

swappiness

The default value for swappiness is 60. We tested with swappiness 30 and 90. It is overall better to swap less especially when there are time requirements on operations depending on memory and with that depending on swap. The time limit on operations on NEs in NETSim is 2 minutes. So it should be better to have a lower value on swappiness than a higher one.

vfs_cache_pressure

The default value for vfs_cache_pressure is 100. We tested with vfs_cache_pressure 50 and 150.

With the relatively few disk accesses it should be better to reuse the cache memory used for directories and i-nodes and use the memory for NEs. A

vfs_cache_pressure value 100 should be better than 50, and 150 might be even better.

dirty_ratio and dirty_background_ratio

The default value for dirty_background_ratio is 10 and for vm_dirty_ratio it is 40.

In a mail to the Linux Kernel Mailing List Linus Torvalds recommends to set the dirty_background_ratio to 5 and dirty_ratio to 10 [31].

```
Date Fri, 27 Apr 2007 08:18:34 -0700 (PDT)
From Linus Torvalds <>
Subject Re: [ext3][kernels >= 2.6.20.7 at least] KDE going comatose when FS is under
    heavy write load (massive starvation)
Digg This

On Fri, 27 Apr 2007, Mike Galbraith wrote:
>
> As subject states, my GUI is going away for extended periods of time
> when my very full and likely highly fragmented (how to find out)
> filesystem is under heavy write load.  While write is under way, if
> amarok (mp3 player) is running, no song change will occur until write is
> finished, and the GUI can go _entirely_ comatose for very long periods.
> Usually, it will come back to life after write is finished, but
> occasionally, a complete GUI restart is necessary.

One thing to try out (and dammit, I should make it the default now in
2.6.21) is to just make the dirty limits much lower. We've been talking
about this for ages, I think this might be the right time to do it.

Especially with lots of memory, allowing 40% of that memory to be dirty is
just insane (even if we limit it to "just" 40% of the normal memory zone.
That can be gigabytes. And no amount of IO scheduling will make it
pleasant to try to handle the situation where that much memory is dirty.

So I do believe that we could probably do something about the IO
scheduling _too_:

 - break up large write requests (yeah, it will make for worse IO
   throughput, but if make it configurable, and especially with
   controllers that don't have insane overheads per command, the
   difference between 128kB requests and 16MB requests is probably not
   really even noticeable - SCSI things with large per-command overheads
   are just stupid)
```

```
   Generating huge requests will automatically mean that they are
   "unbreakable" from an IO scheduler perspective, so it's bad for latency
   for other reqeusts once they've started.

 - maybe be more aggressive about prioritizing reads over writes.

but in the meantime, what happens if you apply this patch?

Actually, you don't need to apply the patch - just do

echo 5 > /proc/sys/vm/dirty_background_ratio
echo 10 > /proc/sys/vm/dirty_ratio
and say if it seems to improve things. I think those are much saner
defaults especially for a desktop system (and probably for most servers
too, for that matter).

Even 10% of memory dirty can be a whole lot of RAM, but it should
hopefully be _better_ than the insane default we have now.

Historical note: allowing about half of memory to contain dirty pages made
more sense back in the days when people had 16-64MB of memory, and a
single untar of even fairly small projects would otherwise hit the disk.
But memory sizes have grown *much* more quickly than disk speeds (and
latency requirements have gone down, not up), so a default that may
actually have been perfectly fine at some point seems crazy these days..

Linus''
```

Therefore we decided to run a test with $dirty\_background\_ratio = 5$ and $dirty\_ratio = 10$.

min_free_kbytes

Our default value for min_free_kbytes is 282 MBytes RAM. The recommended value for NETSim is at least 2% of the total size of RAM in the computer. With less than 1–2% min_free_kbytes the device drivers could not always allocate memory.

We ran one test with min_free_kbytes 158MB which was 2% of RAM in our case with our computer. As seen above less than 1–2% min_free_kbytes is not good. With a too high value a lot of memory stays unused and the

operating system needs to swap more. Therefore a too high value is not good.

page-cluster

The default value for page-cluster is 3. We tested page-cluster values of 1 and 5. Page-cluster should have little effect on the relatively few disk accesses but it should have more impact on large simulations with large memory usage and much swapping.

dirty_writeback_centisecs

The default value for dirty_writeback_centisecs is 500. We tested with a dirty_writeback_centisecs value of 1000. The dirty_writeback_centisecs should have little effect on the relatively few disk accesses.

Chapter 4

Results

In this chapter we present the test results of our tests. All response times below are in milliseconds.

4.1 Mean with sliding window

Besides the test runs with different values for the different paramaters we run three reference runs and compared them with each other. The results were plotted with mean value calculation with sliding window. Figure 4.1 shows the IDL method get_MO_attributes for two reference runs, ref 1 and ref 2. We compared the IDL methods for two test runs at a time because that was what our program was designed for.

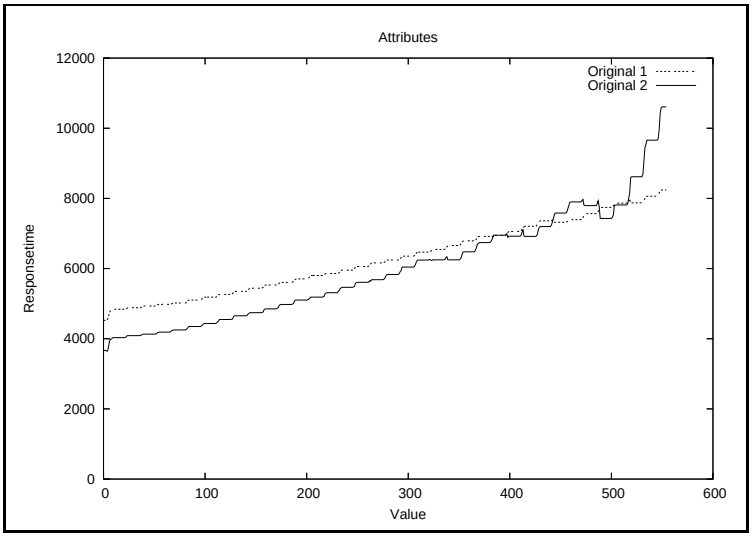

Figure 4.1: Comparing get_MO_attributes for two reference runs.

Figure 4.2 shows IDL method get_MO_attributes for reference run 2 and test run with 2 separate swap partitions.

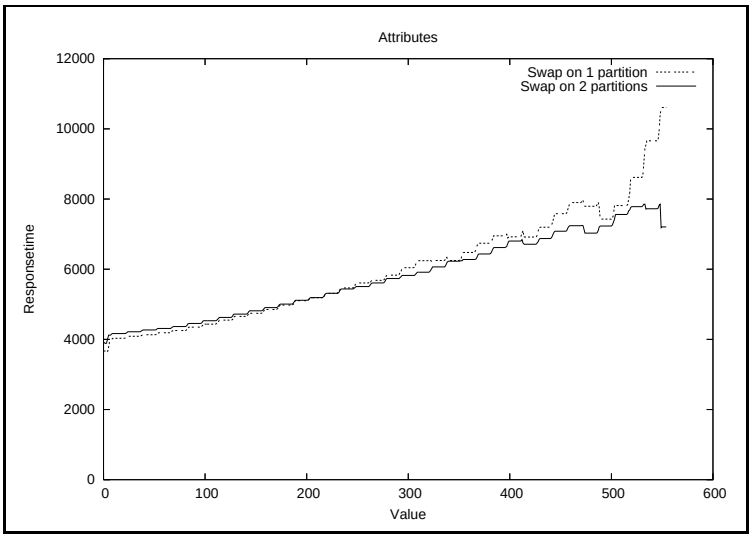

Figure 4.2: Comparing get_MO_attributes for reference run 2 and a swap run.

Figure 4.3 shows the IDL method basic_create_MO for two reference runs, ref 1 and ref 2.

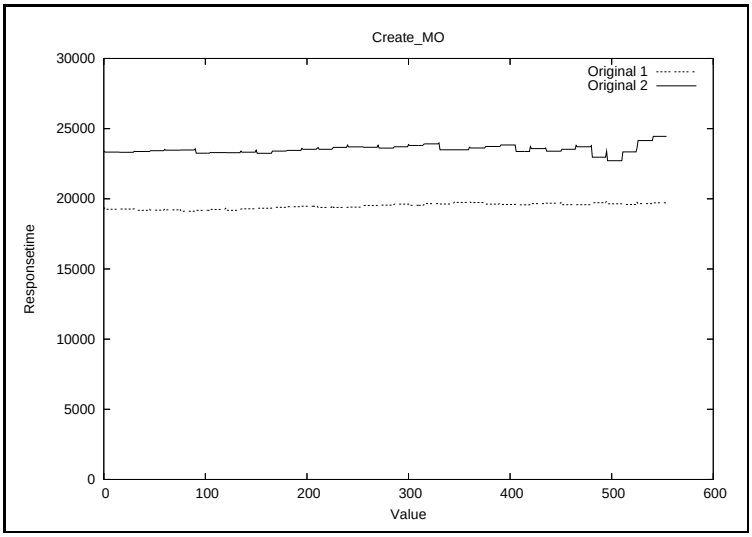

Figure 4.3: Comparing basic_create_MO for two reference runs.

Figure 4.4 shows the IDL method basic_create_MO for reference run 2 and test run with 2 separate swap partitions.

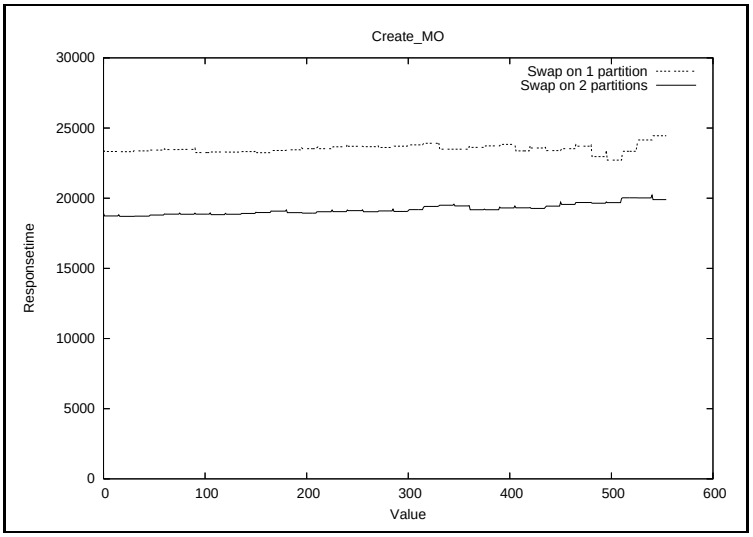

Figure 4.4: Comparing basic_create_MO for reference run 2 and a swap run.

Figure 4.5 shows the IDL method basic_get_MO_containment_short for two reference runs, ref 1 and ref 2.

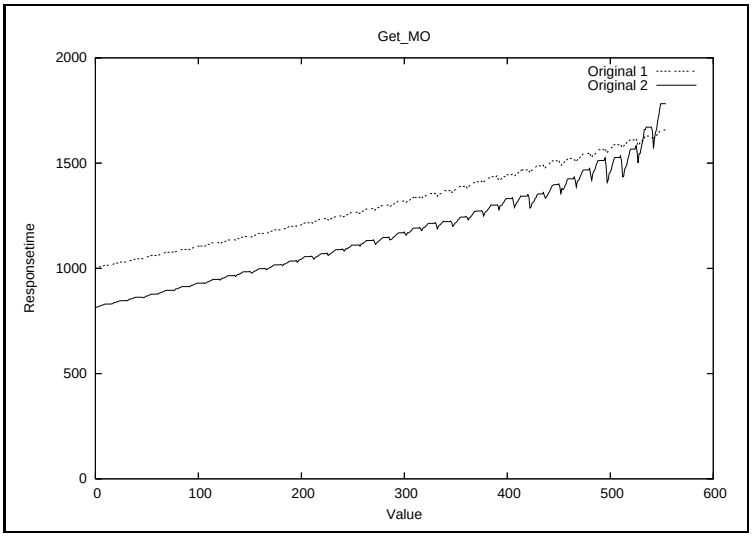

Figure 4.5: Comparing basic_get_MO_containment_short for two reference runs.

Figure 4.6 shows the IDL method basic_get_MO_containment_short for reference run 2 and test run with 2 separate swap partitions.

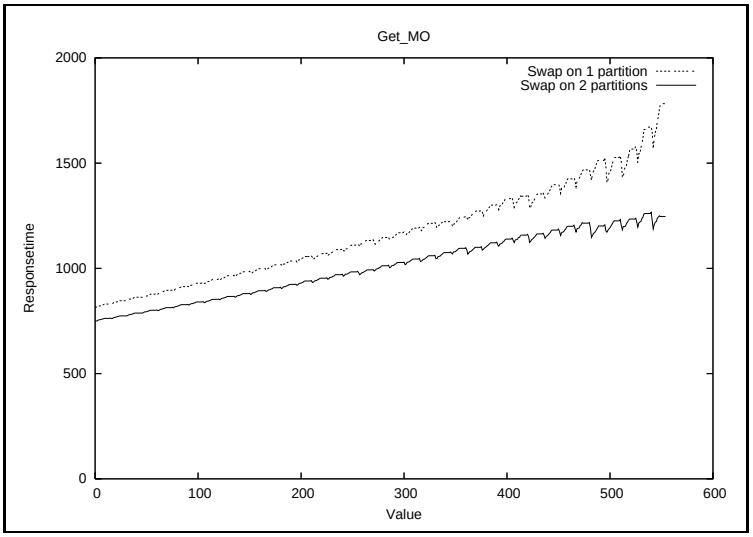

Figure 4.6: Comparing basic_get_MO_containment_short for reference run 2 and a swap run.

4.2 Mean with whisker bars

We plotted graphs with mean values, one for each SWUG, and combined them with least square for a second degree function and whisker bars. Figure 4.7 shows the IDL method get_MO_attributes for two reference runs, ref 1 and ref 2.

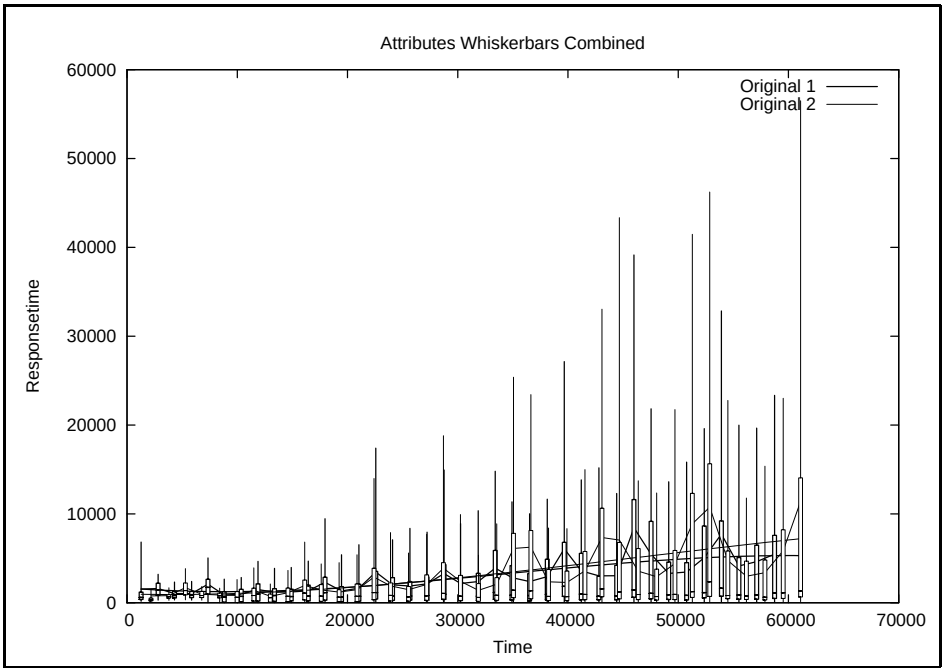

Figure 4.7: Comparing get_MO_attributes for two reference runs with whisker bars.

Figure 4.8 shows the IDL method basic_create_MO for two reference runs, ref 1 and ref 2 with whisker bars.

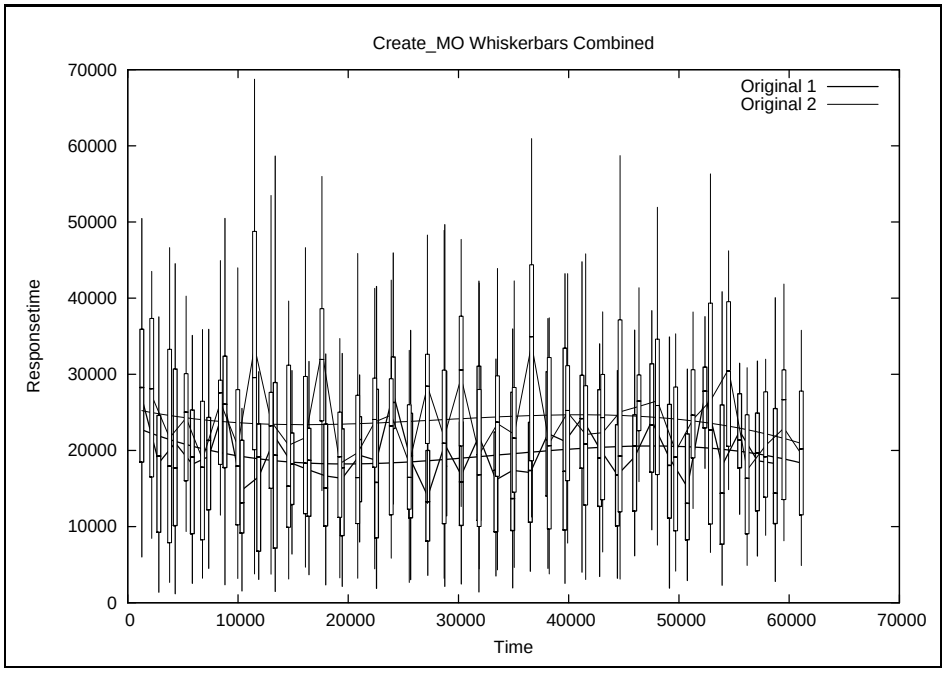

Figure 4.8: Comparing basic_create_MO for two reference runs with whisker bars.

Figure 4.9 shows the IDL method basic_get_MO_containment_short for two reference runs, ref 1 and ref 2 with whisker bars.

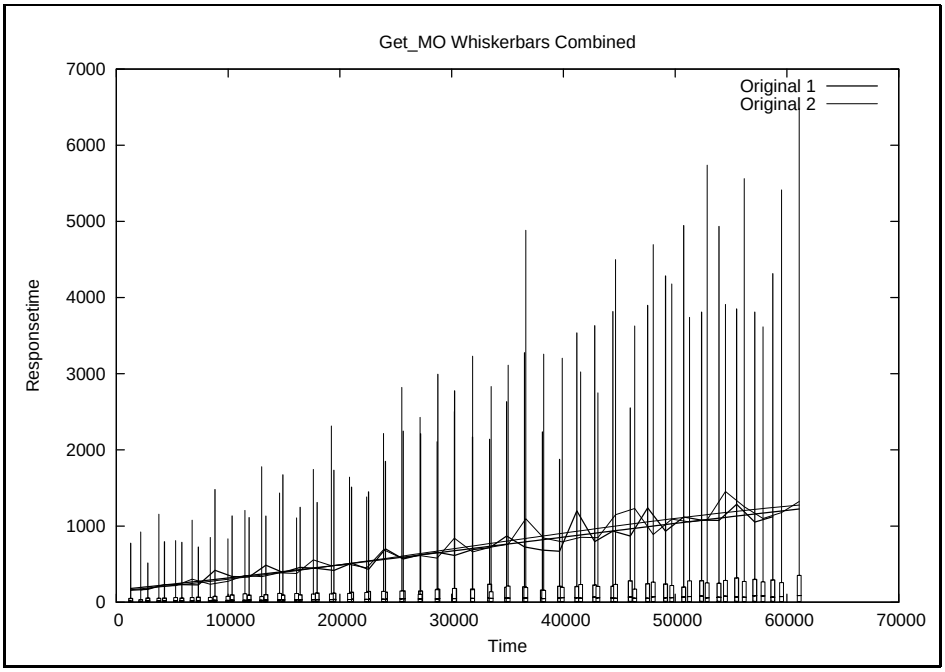

Figure 4.9: Comparing basic_get_MO_containment_short for two reference runs with whisker bars.

Figure 4.10 shows the IDL method get_MO_attributes for reference run 2 and test run with 2 separate swap partitions with whisker bars.

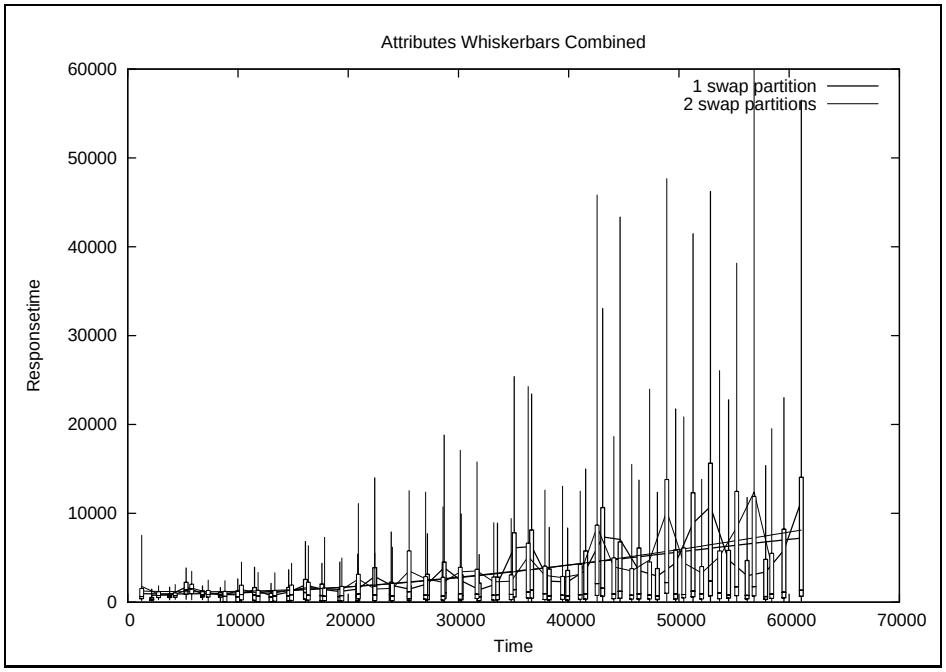

Figure 4.10: Comparing get_MO_attributes for reference run 2 and a swap run with whisker bars.

Figure 4.11 shows the IDL method basic_create_MO for reference run 2 and test run with 2 separate swap partitions with whisker bars.

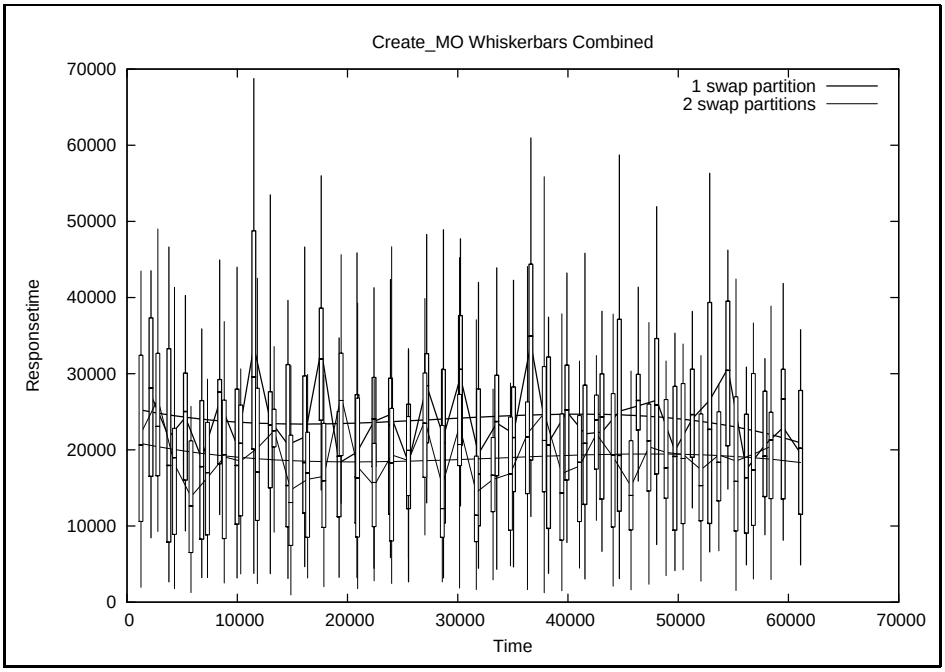

Figure 4.11: Comparing basic_create_MO for reference run 2 and a swap run with whisker bars.

Figure 4.12 shows the IDL method basic_get_MO_containment_short for reference run 2 and test run with 2 separate swap partitions with whisker bars.

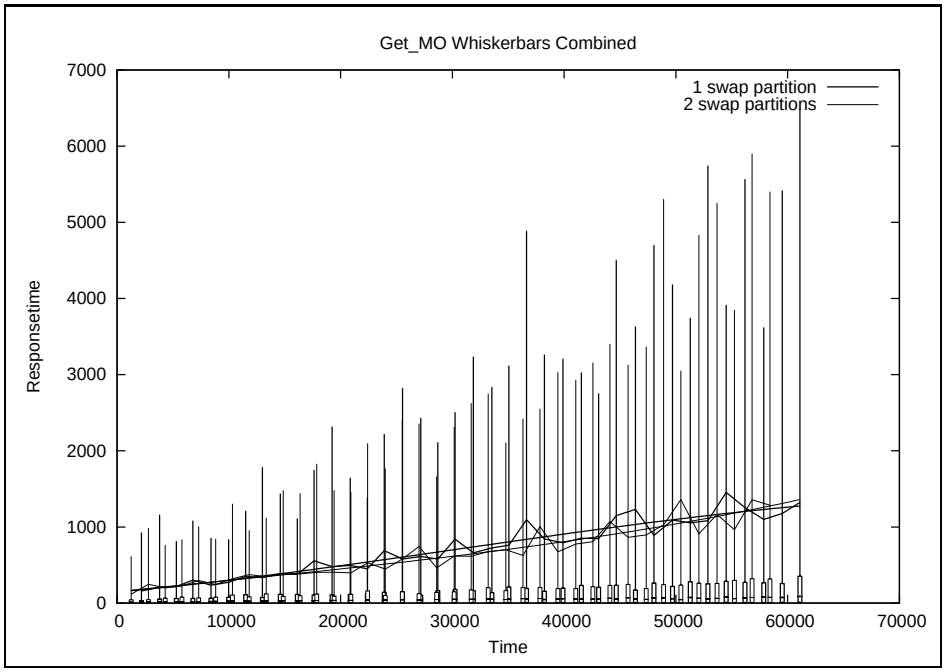

Figure 4.12: Comparing basic_get_MO_containment_short for reference run 2 and a swap run with whisker bars.

4.3 Swug-times

We plotted time differences for SWUG cycles from start of SWUG to end of nesync. Figure 4.13 shows the difference times for SWUGs for reference run 1 and 2.

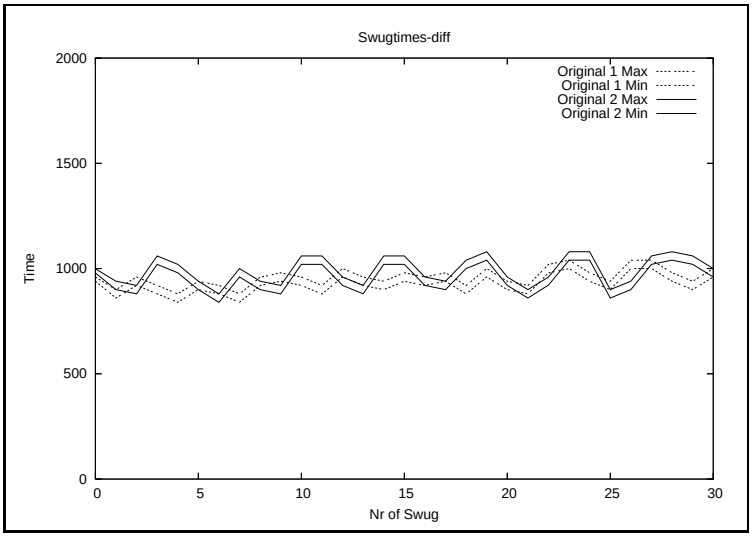

Figure 4.13: Time difference for SWUGs for two reference runs.

Figure 4.14 shows the difference times for SWUGs for reference run 2 and for a test run with swap on 2 separate partitions.

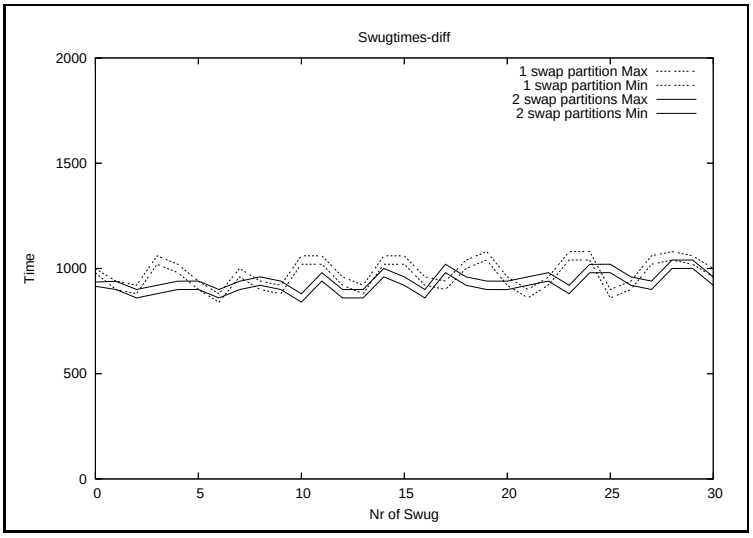

Figure 4.14: Time difference for SWUGs for two test runs.

4.4 Memory usage

When we run a simulation with 10 nets on a computer with 8 GB RAM the memory usage increased during the simulation run and the computer started to swap after 10–12 hours, see figure 4.15.

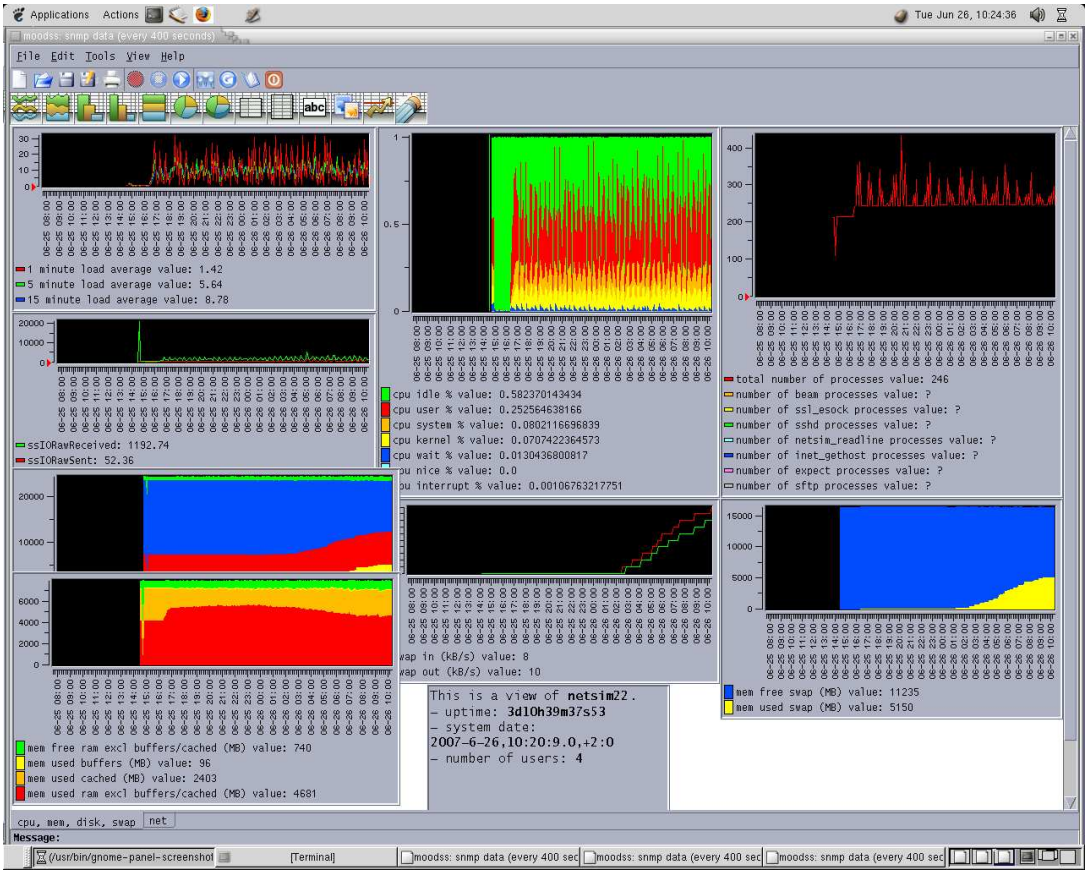

Figure 4.15: Memory and swap usage.

The memory and swap used increased and after 15–17 hours the swap reached around 5GB then the swap started to level out, see figure 4.16.

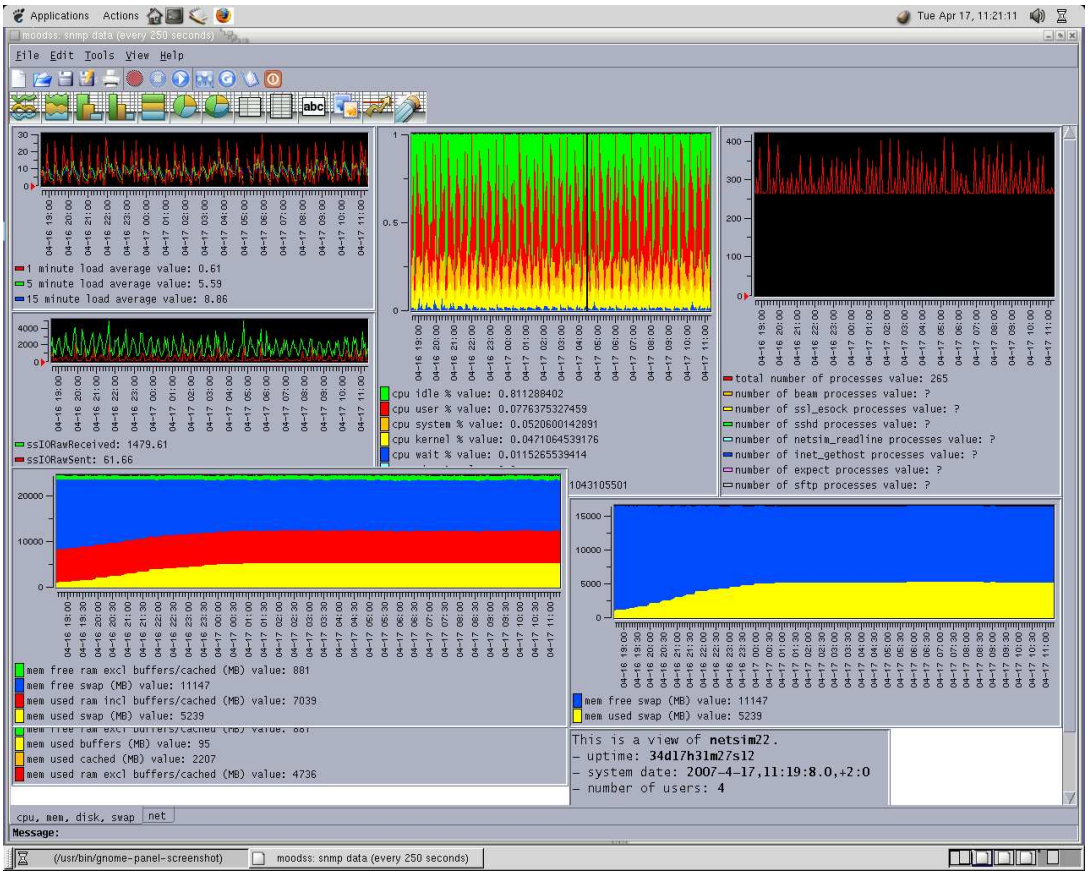

Figure 4.16: Memory and swap usage where the swap levels out.

Figure 4.17 shows CPU and memory usage for 8 nets on a computer with 8GB RAM. The maximum memory used was 4521 MB RAM. This run was little to short for the swap to level out.

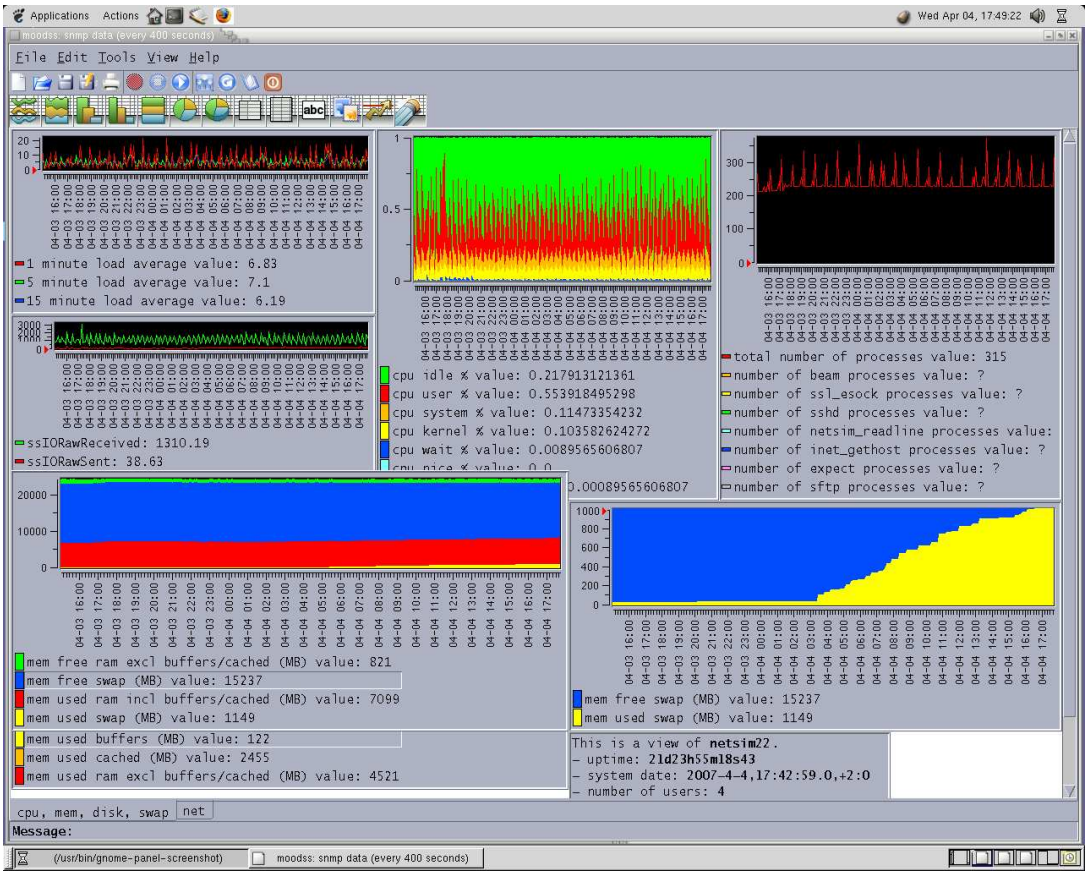

Figure 4.17: CPU, memory and swap usage for 8 nets.

Figure 4.18 shows CPU and memory usage for 10 nets on a computer with 8GB RAM. The maximum memory used was 4736 MB RAM and used swap was 5239 MB.

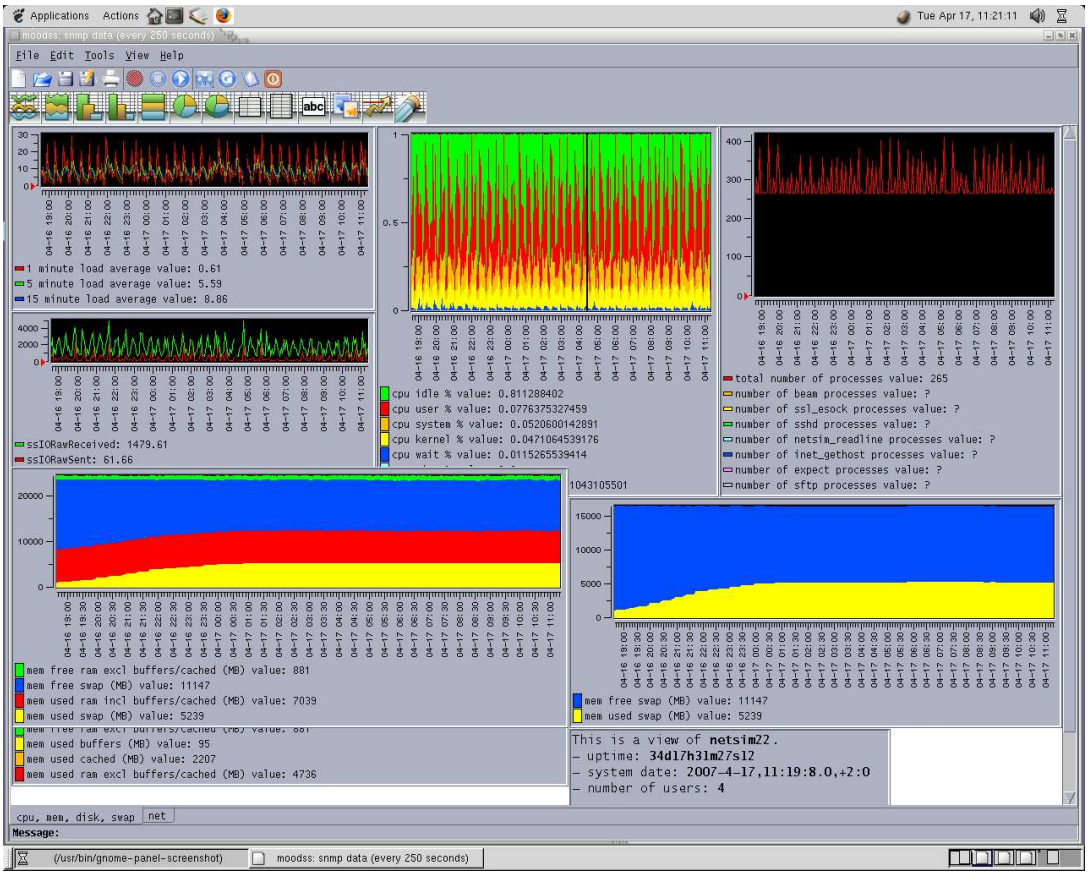

Figure 4.18: CPU, memory and swap usage for 10 nets.

Figure 4.19 shows CPU and memory usage for 11 nets on a computer with 8GB RAM. The maximum memory used was 4922 MB RAM and used swap was 5676 MB.

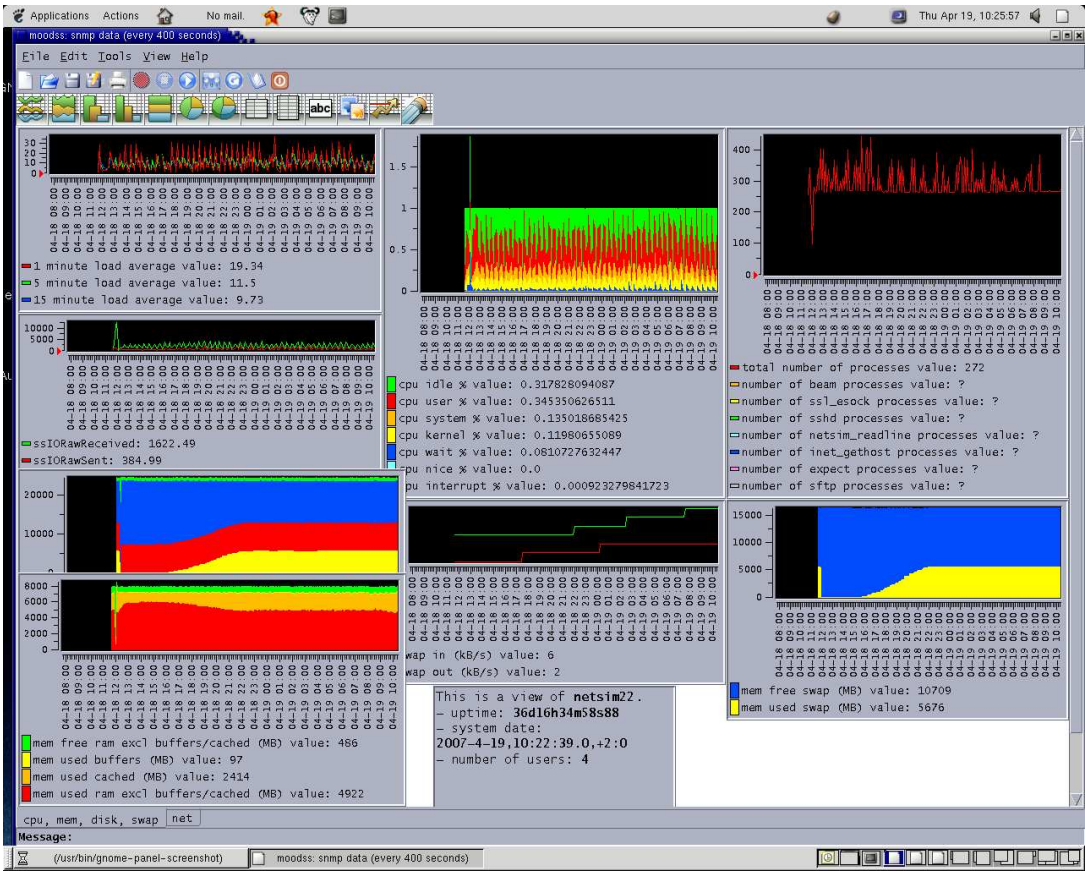

Figure 4.19: CPU, memory and swap usage for 11 nets.

Figure 4.20 shows CPU and memory usage for 12 nets on a computer with 8GB RAM. The maximum memory used was 5294 MB RAM and used swap was 6298 MB.

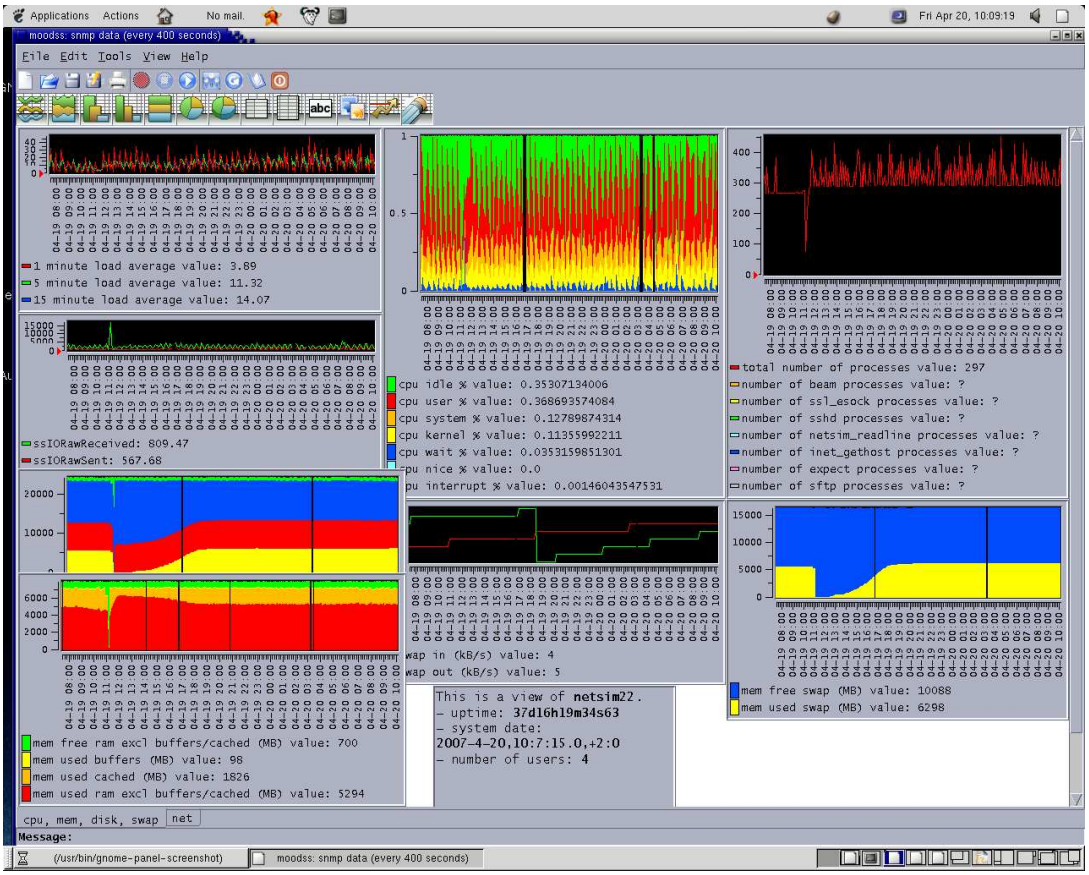

Figure 4.20: CPU, memory and swap usage for 12 nets.

When using 8 GB RAM the limit for the computer was 10 nets. There where a few minor errors with 10 nets and more and more serious errors for 11 nets.

There where 3 lgfmNumRenewalsAttempedTooLate, for 10 nets, see figure 4.21. LgfmNumRenewalsAttempedTooLate bigger than zero indicate that the load generator is overloaded and do not manages to renewals FM, fault management, subscription in time. lgfmNumRenewalsAttempedTooLate is a measurepoint in NETSim to detect error behaviour.

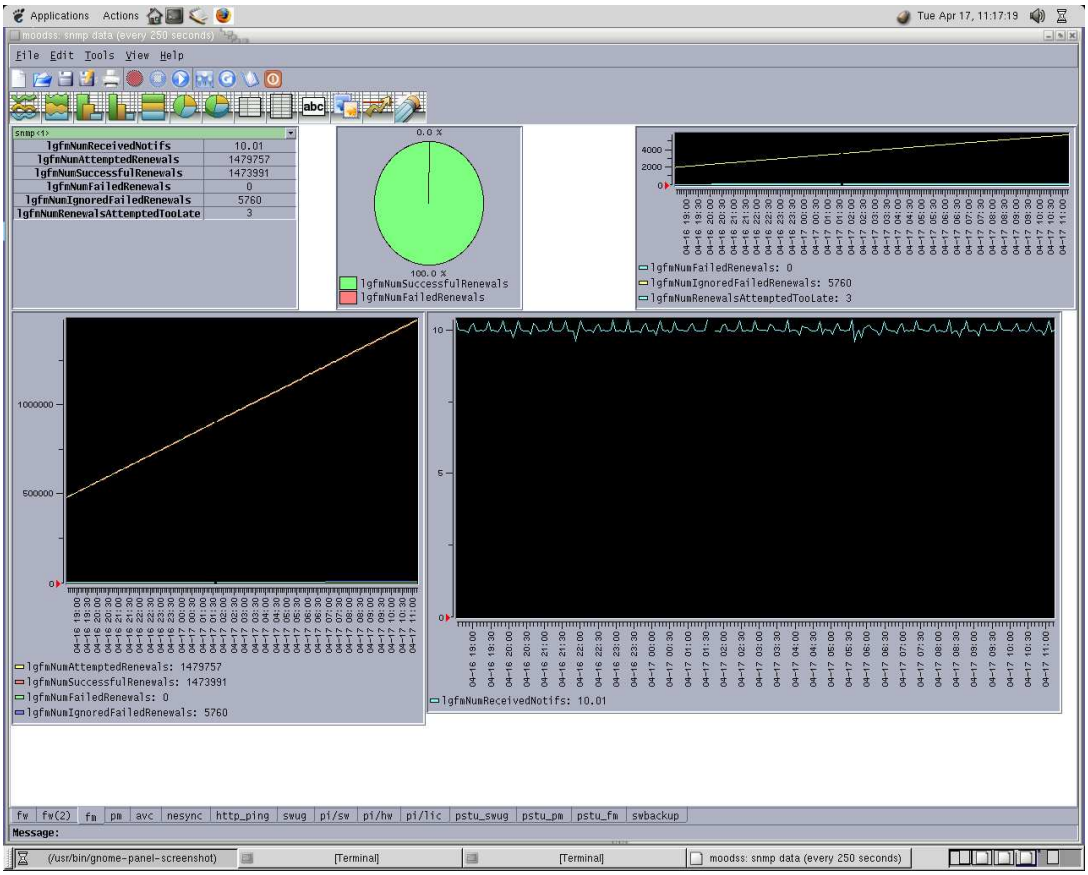

Figure 4.21: FM for 10 nets.

There where no errors for PM for 10 nets, lgpmNumFailedRops are zero, see figure 4.22. LgpmNumFailedRops are a counter for all failures in preformance monitoring, like IDL method invocations errors and downloading data errors.

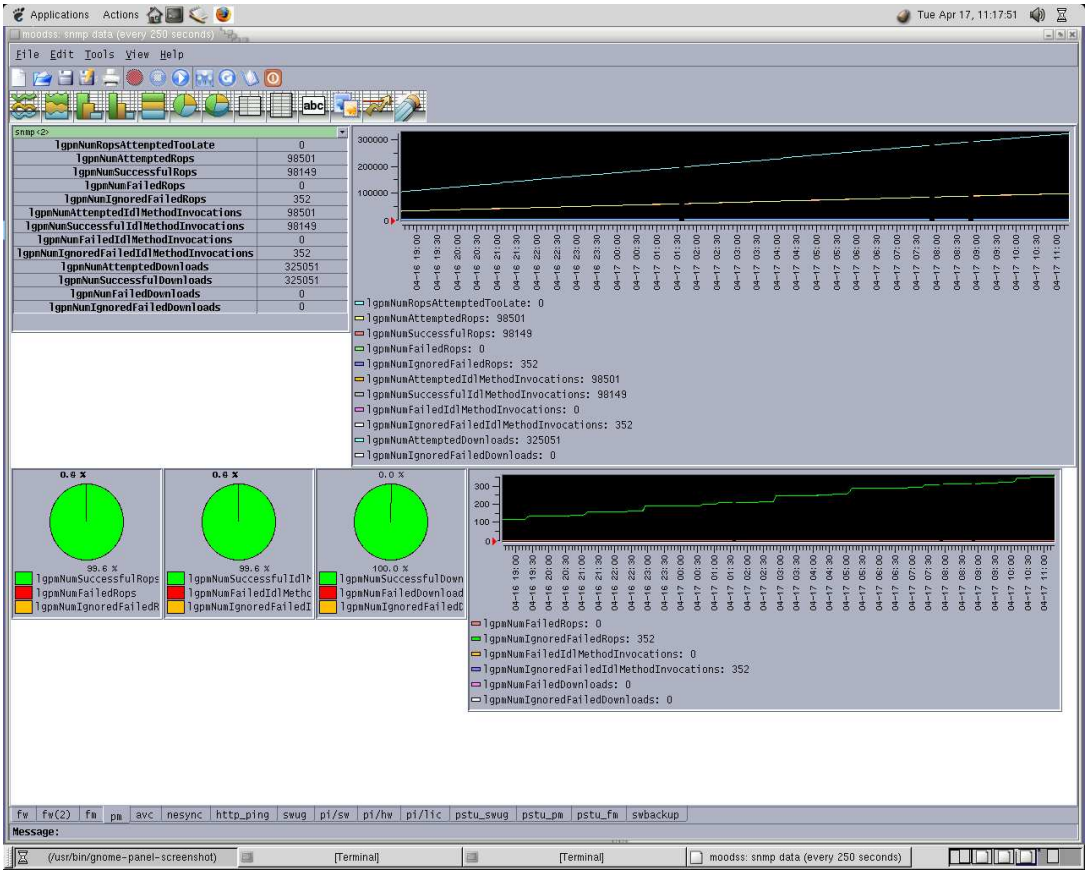

Figure 4.22: PM for 10 nets.

For FM for 11 nets the lgfmNumRenewalsAttemptedTooLate was 8, see figure 4.23.

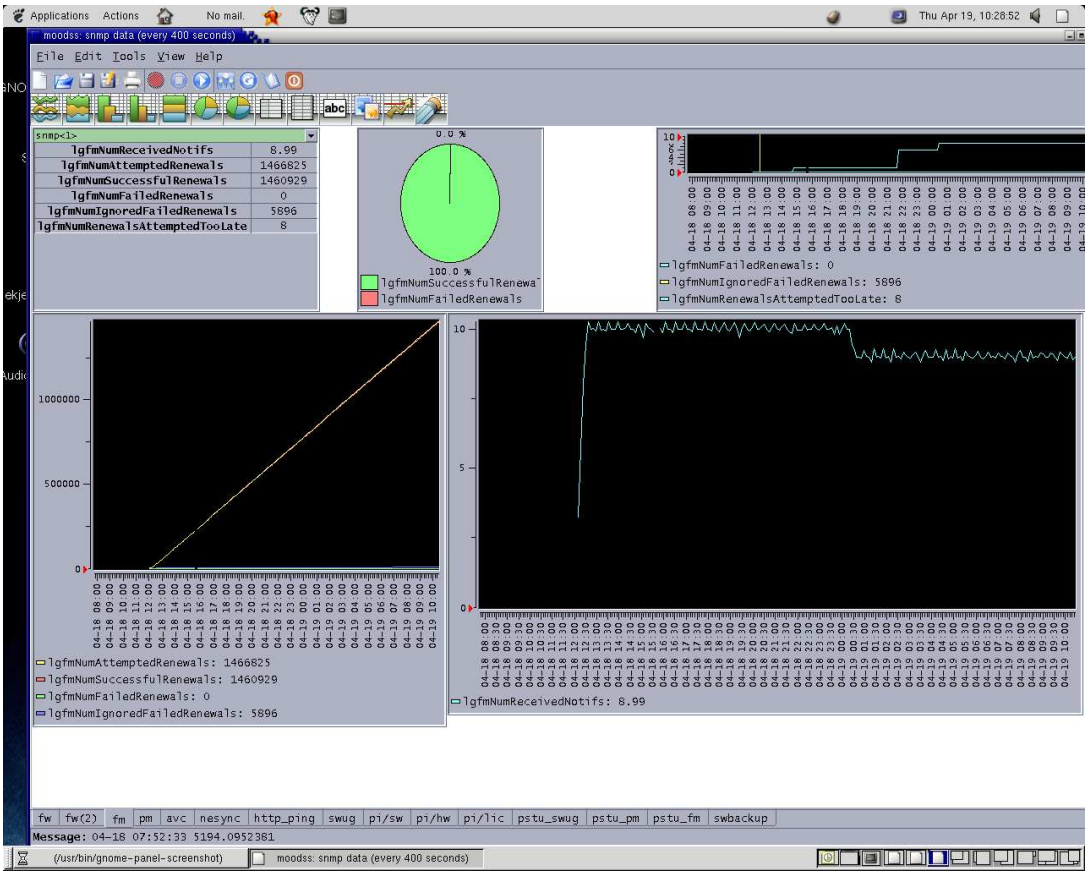

Figure 4.23: FM for 11 nets.

For PM for 11 nets the lgpmNumFailedRops was 1 and the lgpmNumFailedIdlMethodInvocations was 1, see figure 4.24.

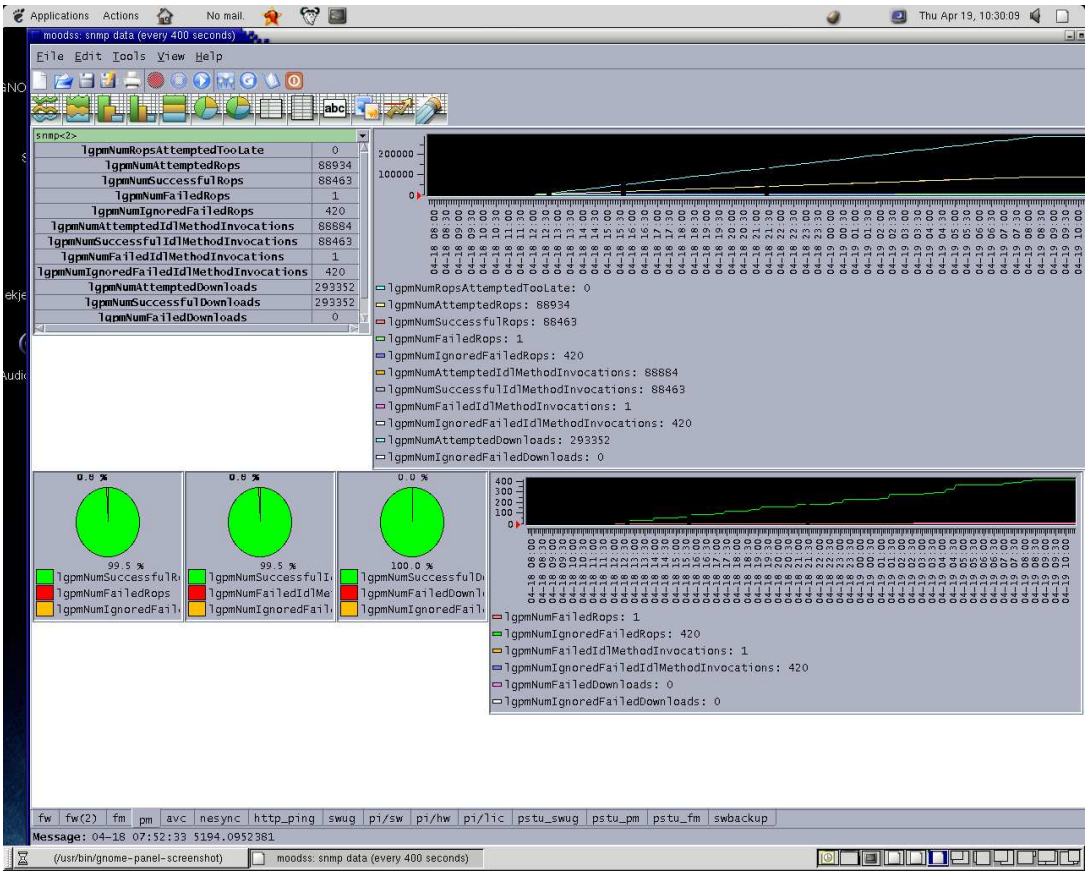

Figure 4.24: PM for 11 nets.

On the same computer we used for most of the tests we halved the amount of RAM from 8 GB to 4 GB. With 8 GB RAM the limit was 10 nets and with 4 GB the limit was 6 nets.

Figure 4.25 shows SWUG's for 6 nets and figure 4.26 shows SWUG's for 7 nets. There were no errors for 6 nets and several errors for 7 nets.

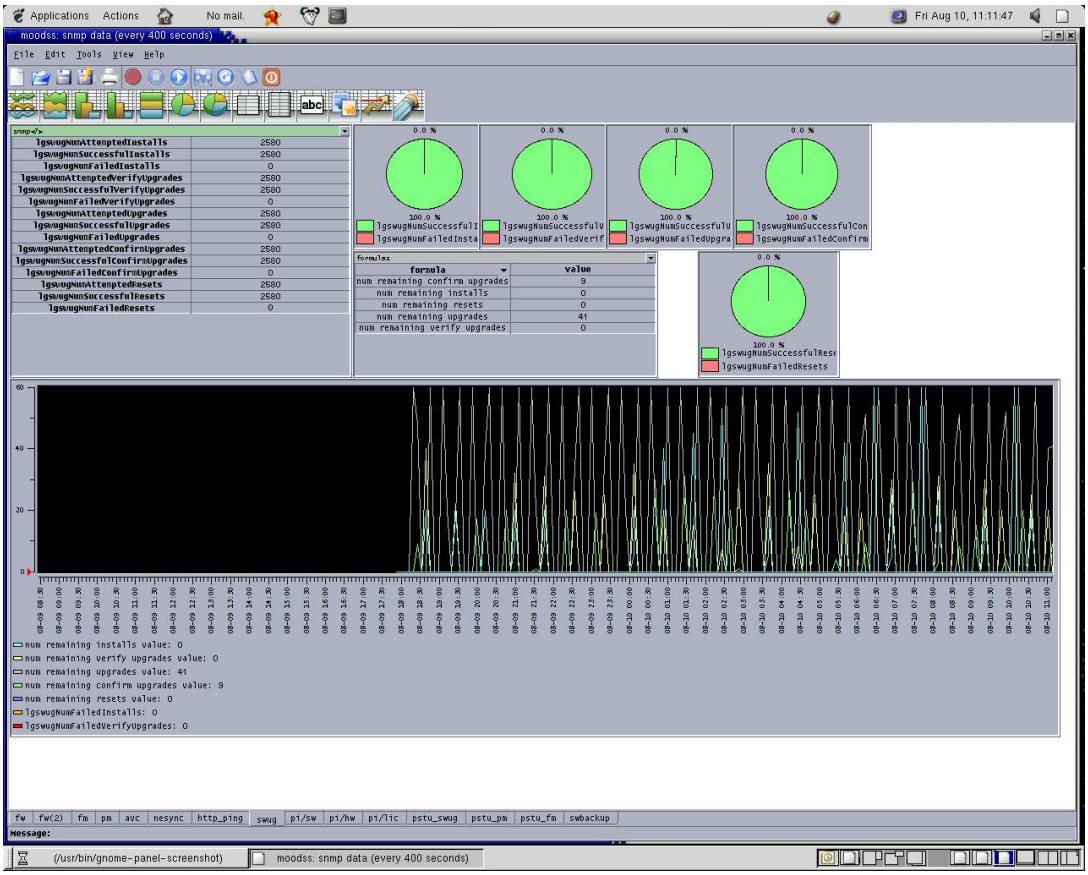

Figure 4.25: SWUG for 6 nets on a computer with 4 GB RAM showing no errors.

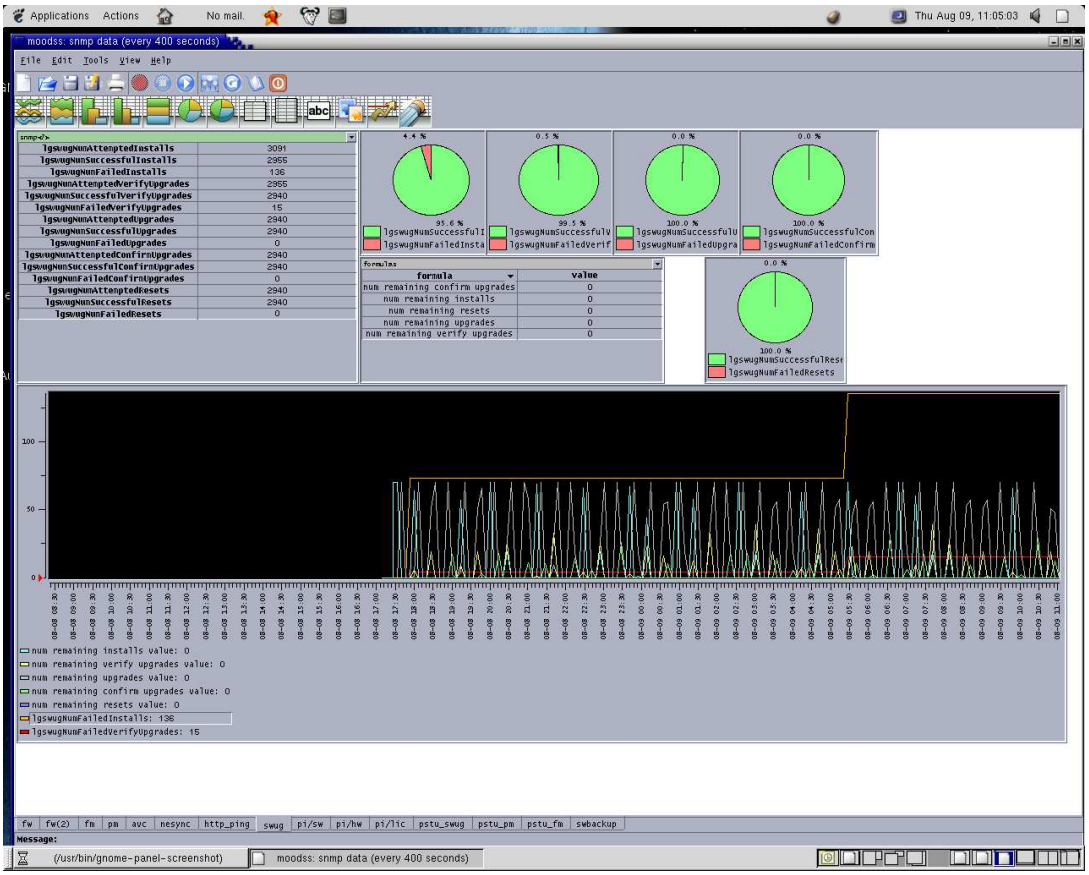

Figure 4.26: SWUG for 7 nets on a computer with 4 GB RAM showing several errors.

Chapter 5

Discussion, conclusions, recommendations and future work

5.1 Discussion and conclusions

In figures 4.1, 4.2 we can see that the difference for get_MO_attributes between two reference runs is bigger than get_MO_attributes when we compare swap on two partitions with a reference run (original 1).

The figures 4.3 and 4.4 shows that basic_create_MO for swap on two partitions is almost the same as for reference run 1 (original 1) and the figure shows that there is a big difference between the two reference runs.

For basic_get_MO_containment_short the response times is little lower than for the two reference runs but the difference between the two reference runs are as big as the difference between swap on two partitions and the reference

run.

The quartiles 1 and 3 and min95 and max95 in figures 4.7, 4.8 and 4.9 for the three reference runs shows that there are big variations. If we study the figures 4.10, 4.11 and 4.12, showing a reference run and a run with two swap partitions, we can see that the variations are overlapping.

A SWUG is a heavy operation so we studied and compared the times it took for each SWUG. Figure 4.13 shows the time difference for two reference runs and figure 4.14 shows the time differences for a reference run and swap on two partitions. As one can see in figure 4.14 the times are overlapping and the max values and min values are almost the same for the two runs. This gives that it is not possible to draw any conclusions that the SWUG times for one test run is better than for another test run.

The memory usage is approximately linear for NETSim. On a computer with 8 GB RAM the limit was 10 nets and on the same computer with 4 GB RAM the limit was 6 nets. The size of the swap partition was 16 GB in both cases.

During a run the swap usage increased and leveled out. With 8 GB RAM and 8 nets the maximum memory used was 4521 MB, for 10 nets it was 4736 MB RAM, for 11 nets it was 4922 MB RAM and for 12 nets 5294 MB RAM. Figure 5.1 shows the memory usage plotted for 8, 10, 11 and 12 nets.

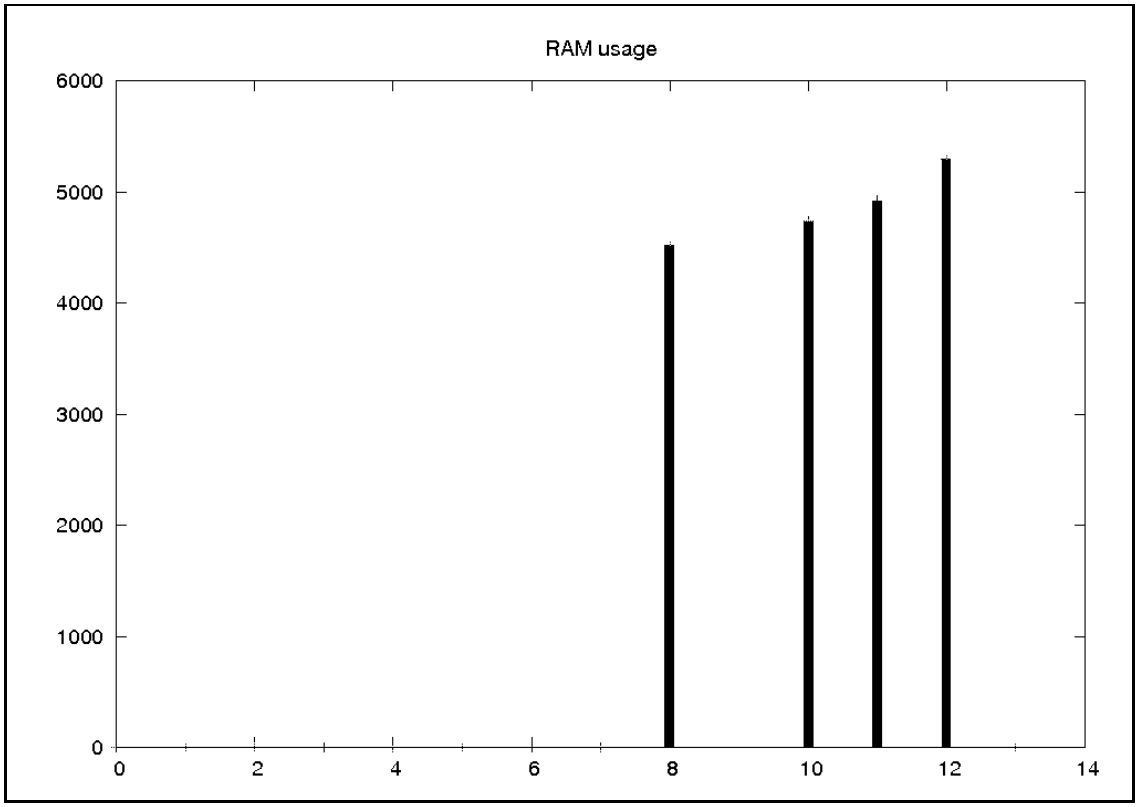

Figure 5.1: Memory usage for 8, 10, 11 and 12 nets.

The swap leveled out at 5239 MB for 10 nets, at 5676 MB for 11 nets and at 6298 MB for 12 nets.

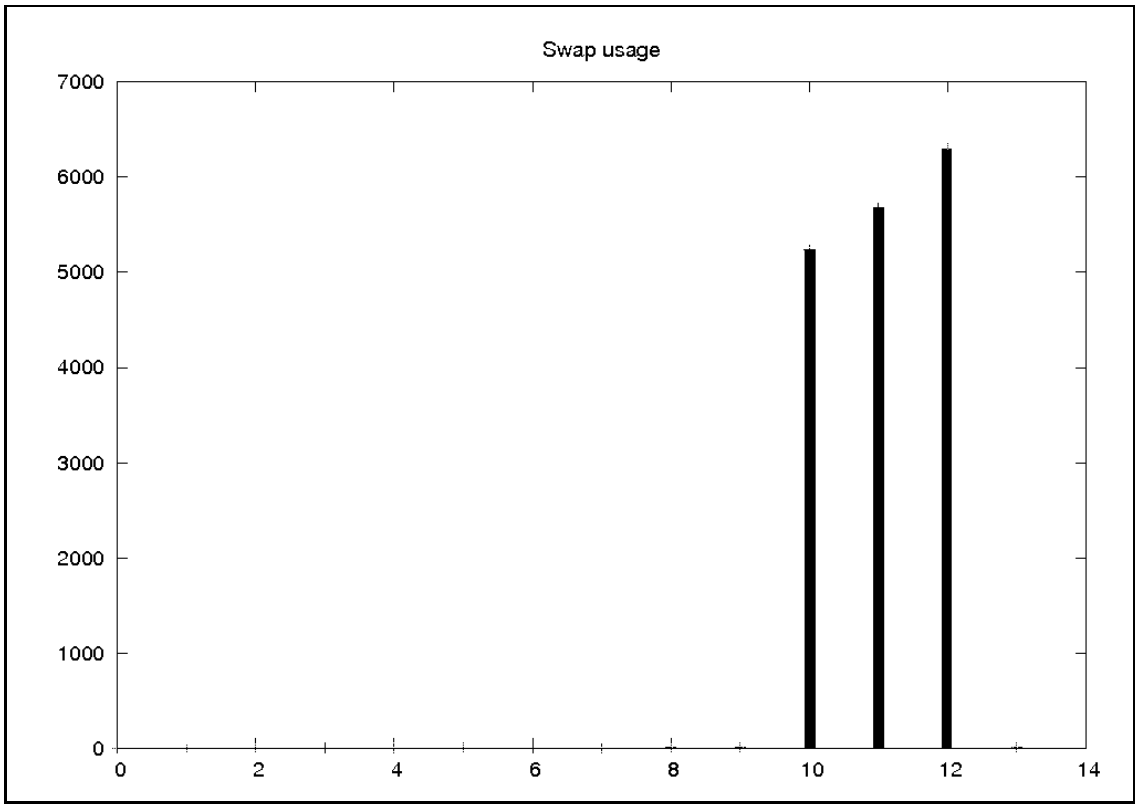

Figure 5.2: Swap usage for 8, 10, 11 and 12 nets.

Unfortunately the simulation runs are nondeterministic. This gives that to be certain one needs to run several simulation runs. This was not practically possible because one simulation run took at least 16 hours which gives little more than a month for all tests. This and the difference between reference runs gives an uncertainty in the test results.

To get statistically significant values from the reference runs we calculate a confidence interval for the three IDL methods. From every test run we get approximately 30 SWUG cycles. Unfortunately we can not use 30 samples per test run, because the individual SWUG cycles are codependent.

We set confidence level to 0.99, and calculate with one mean value per IDL method from each of the three original test run. For the IDL method get_MO_attributes we get the confidence interval [64;77], with arithmetic mean 71, which means that the confidence interval mean ratio is 19%. For the IDL method basic_get_MO_containment_short we get the confidence interval [2.0;2.8],with mean 2.4 and there 35% confidence interval mean ratio. For the IDL method basic_create_MO we get the confidence interval [-1700;5300], with mean 12000 and there 260% confidence interval mean ratio. In an attempt to calculate how many test runs we need to get sufficiently small intervals, maximum 5% of the arithmetic mean of the IDL methods. We use the three test runs in round robin fashion to calculate standard deviation and test for different values of n, see section 2.6. We get n = 8 for get_MO_attributes, n = 15 for basic_get_MO_containment_short and n = 539 for basic_get_MO_containment_short. To run 539 test runs for every parameter is for obvious reasons not possible, but even 8 test runs is outside time limits for this thesis.

5.2 Recommendations

The disk usage is not critical and it is needless to spend time on optimising disk and file system parameters. The amount of memory used by the simulations increased approximately linearly with the size of the simulation. The amount of swap space is not critical as long as the swap size is big enough. This gives that it is possible with more primary memory to increase the size of the simulation on a computer. The size of the swap space is not a limiting factor.

32-bit Erlang uses less memory than 64-bit Erlang because 64-bit Erlang use pointers whith double size. Primary memory are the biggest limiting factor, indicating that it is better to use 32-bit Erlang for NETSim simulations than 64-bit Erlang.

We also recommend to do memory optimisation in NETSim.

Because the CPU load does not seems to be a limiting factor we recommend using CPU intensive algorithms instead of memory intensive algorithms where it is possible.

5.3 Future work

Today SSL is not included in Erlang core. Instead it is running in a separate process. The Erlang core team is working on implementing SSL in Erlang. The current SSL module uses polling and the new implementation supports Kpoll and Epoll. One thing to do in the near future when Erlang with SSL support included is released is to ascertain which is the best to use of Epoll and Kpoll.

Developing a theoretical model of NETSim regarding disk usage, network usage, CPU usage, primary memory usage and swap usage makes it possible to do theoretical studies on different parameters' impact on NETSim. With a theoretical model it is possible to study the parameters' impact on NETSim without test runs and one can determine which parameter changes that have no or little impact and run test runs only for the parameters that have any impact.

Another thing to ascertain is if it is possible to implement trace-driven simulation in NETSim. With trace-driven simulation all activities are recorded and a test run can with all its activities be replayed at a later time. This gives a deterministic system that is easier to test. One advantage is that it is possible to run a single test run and compare it with another single test run based on the same tracking with a changed parameter. In that way it is possible to draw conclusions on a few test runs.

Another advantage with a deterministic system is that it is possible to test several parameters at the same time with fractional factorial design. In fraction factorial design we can test combinations of parameters in different test run and calculate a single parameters effect of the result and synergy effects. [32].

Further advantage with tracking is that it is possible to save specific test runs, for example test runs known to be problematic.

Appendix A

Abbreviations

A.1 Abbreviations

ATM Asynchronous Transfer Mode

AUC Authentication Centre

AVC Attribute Value Change

AXE An Ericsson telecom platform.

BBC Best Balanced Choice

BSC Base Station Controller

BTS Base Transceiver Station

CSH Context Sensitive Help

EIR Equipment Identity Register

FAQ Frequently Asked Questions

FTP File Transfer Protocol

GGSN Gateway GPRS Support Node

GMSC Gateway MSC

GPRS General Packet Radio Service

GSM Global System for Mobile

GTP GPRS Tunnel Protocol

GUI Graphical User Interface

HLR Home Location Register

IDL Interface Description Language

IIOP Internet Inter-ORB Protocol (Corba over TCP/IP)

IP Internet Protocol

L1 Layer 1

LSP Logical Scoring of Preferences

MGw Media Gateway

MIB Management Information Base

MO Managed Objects

MSC Mobile Service Switching Center

MTP Message Transfer Protocol

NE Network Element

NME NETSim Management Extension

NMS Network Managing Subsystem

Node B Base Station in UTRAN networks

O&M Operation & Maintenance

OSS Operations and Support System

PFRA Page Frame Reclaiming Algorith

PLMN Public Land Mobile Networks

PM Performance Monitoring

PSTN Public Switched Telephone Network

PSTU Packet Switched Termination Unit

RANAG Radio Access Network Aggregator

RANOS Radio Access Network Operation Support

RBS Radio Base Station.

RNC Radio Network Controller

RNS Radio Network Subsystem

ROAS Region of Acceptable Solutions

RXI Older version of RANAG

SAS Serial Attached SCSI

SCP Service Control Point

SFTP Secure File Transfer Protocol

SGSN Serving GPRS Support Node

SNIA Storage Networking Industry Association

SNMP Simple Network Management Protocol

SRP System Requirement and Parameter tree

SSL Secure Sockets Layer

SSLIOP Secure Socket Layer Inter-ORB Protocol (IIOP over SSL).

STN Site Transport Node

SWUG SoftWare UpGrade

UE User Equipment

UMTS Universal Mobile Telecommunications System

UTRAN UMTS Terrestrial Radio Access Network

VLR Visitor Location Register

X.25 A standard for packet networks.

Bibliography

[1] NETSim User's Guide. Ericsson AB, 2006. 1553-CRL 121 03 Uen Rev AS 2006-12-14.

[2] Jansson L. Annerberg ULf, Boija Beatrice. OSS-Overview, Operation and Support System Trainee's Guide. Ericsson Radio Systems AB, 1996.

[3] Toskala A. Holma H. *WCDMA for UMTS*. John Wiley & Sons Ltd., 2001.

[4] Gunnarsson Frida Knutsson Björn, Björsson Anders. Radio Access Network (UTRAN) Modeling for Heterogenous Network Simulations, A Brief Description. 2003.

[5] Johansson Christer Hameleers Heino. IP Technology in WCDMA/GSM core networks. Ericsson AB, 2002. Ericsson Review No. 1, 2002.

[6] UMTS Overview. Ericsson AB, 1999. ERA/UX/T-99:108 Rev PA1.

[7] Althoff M. P. Walke B., Seidenberg P. *UMTS The Fundamentals*. John Wiley & Sons Ltd., 2003.

[8] Ericsson Tomas Narup Micael, Helgeson Claes. *Att förstå Telekommunikation 1*. Studentlitteratur, Ericsson Telecom AB, Telia AB, 1997. SV/LZT 101 1402R1B.

[9] NETSim basics course material. Ericsson AB, 2006. From 2006-04-10.

[10] The NETSim programming course material. Ericsson AB, 2007. From 2007-03-16.

[11] NETSim System Administrator's Guide. Ericsson AB, 2007. 1543-CRL 121 03 Uen Rev AF 2007-02-23.

[12] PerfCons How To Test. Ericsson AB, Fetched 2007-03-02. From an Ericsson-internal wiki.

[13] Common RAID Disk Data Format Specification, Revision 1.2 with Errata A Applied. SNIA Storage Networking Industry Associattion, 2007-08-29. http://www.snia.org/tech_activities/standards/curr_standards/ddf/SNIA-DDFv1.2_with_Errata_A_Applied.pdf.

[14] Holmquist Björn Blom Gunnar. *Statistikteori med tillämpningar*. Studentlitteratur, 1998.

[15] Cesati Marco Bovet Daniel P. *Understanding the Linux kernel 3 ed.* O'Reilly, 2006.

[16] Florian Buchholz. The structure of the Reiser file system, Fetched 2007-04-23. http://homes.cerias.purdue.edu/ florian/reiser/reiserfs.php.

[17] Naujok Barry. XFS Filesystem Structure. Silicon Graphics, 2006. 2nd Edition, Revision 2.

[18] Best Steve. JFS overview. IBM, January 2000.

[19] Rivest L. Ronald H. Cormen Thomas, Leiserson E. Charles. *Introductions to algorithms second edition*. McGraw-Hill Companies Inc., Fourth printing 2003.

[20] Hans Reiser. ReiserFS v.3 Whitepaper. namesys.com, Fetched 2007-04-23. http://web.archive.org/web/20030621014013/namesys.com/content_table.html.

[21] Koren Dan Mostek Jim, Earl William. Porting the SGI XFS File System to Linux. Silicon Graphics.

[22] Kleikamp Dave Best Steve. JFS layout. IBM Linux Technology Center, May 2000.

[23] Best Steve. JFS Log, How the Journaled File System performs logging. IBM Linux Technology Center, October 2000.

[24] Kernel changelogs from 2.6.0 to 2.6.21.

[25] Gagne Greg Silberschatz Avi, Galvin Peter. *Applied Operating System Concepts*. John Wiley & Sons Inc., 2000.

[26] Documentation/filesystems/proc.txt from the Linux kernel source tree for kernel 2.6.8.

[27] Documentation/sysctl/vm.txt from the Linux kernel source tree for kernel 2.6.8.

[28] Blom Gunnar. *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 1980.

[29] Wittmeyer-K Linde Eldén Lars. *Numeriska beräkningar - analys och illustrationer med MATLAB*. Studentlitteratur, 2001.

[30] MAY2073RC MAY2036RC HARD DISK DRIVES PRODUCT/MAINTENANCE MANUAL. FUJITSU, 2005. C141-E230-01EN.

[31] Linus Torvalds. Re: [ext3][kernels >= 2.6.20.7 at least] KDE going comatose when FS is under heavy write load (massive starvation), Fetched 2007-04-27. http://lkml.org/lkml/2007/4/27/289.

[32] Runesson Per Berling Tomas. Efficient Evaluation of Multifactor Dependent System Performance Using Fractional Factorial Design, september 2003. IEEE Transactions on Software Engineering, Vol 29, No. 9.

Avdelning, Institution
Division, Department

SaS,
Dept. of Computer and Information Science
581 83 LINKÖPING

LINKÖPINGS UNIVERSITET

Datum
Date

2008-01-18

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-IDA-EX--08/001--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version

Titel Möjligheter för att öka storleken på NETSim-simuleringar genom parameterjusteringar på OS-nivå

Title Potential for increasing the size of NETSim simulations through OS-level optimizations

Författare Kjell Enblom,Martin Jungebro
Author

Sammanfattning
Abstract

English

This master's thesis investigates if it is possible to increase the size of the simulations running on NETSim, Network Element Test Simulator, on a specific hardware and operating system. NETSim is a simulator for operation and maintenance of telecommunication networks.

The conclusions are that the disk usage is not critical and that it is needless to spend time optimizing disk and file system parameters. The amount of memory used by the simulations increased approximately linear with the size of the simulation. The size of the swap disk space is not a limiting factor.

Svenska

Detta exsamensarbete undersöker om det är möjligt att öka storleken på simuleringskörningar av NETSim, Network Element Test Simulator, på en specifik hårdvaru- och operativsystemsplattform. NETSim är en simulator för styr och övervakning av telekomnätverk.

Slutsatserna är att diskanvändandet inte är kritiskt och att det är onödigt att ägna tid åt att optimera disk- och filsystemsparametrar. Minnesutnyttjandet ökar approximativt linjärt med storleken på simuleringarna. Storleken på swapdisken är inte någon begränsande faktor.

Nyckelord
Keywords NETSim, simulating UMTS networks, OS parameters,
Linux, Ericsson AB

Copyright

Svenska

English