OSE emulation on Linux - Performance study

Monthadar Al Jaberi

Lunds Tekniska Högskolan
Department of Computer Science
&
ST-Ericsson

October 14, 2009
to my mother
Abstract This master thesis aims to optimize an emulator for programs written for an operating system called OSE on Linux. The target platform is an ARM 9 processor. A known bottleneck in LOSE\(^1\) is synchronization between tasks. Several profiling tools were evaluated and Oprofile was chosen as the main tool. Oprofile, which is a statistical profiling tool, was used to profile the emulator and pthread mutex was found to dominate a common test case that is found in ST-Ericsson code for OSE. Different synchronization mechanisms were investigated and tested against each other. The fastest synchronization mechanism was futex. A speedup of at least 10\% was achieved.

\(^1\)Legacy OSE.
Acknowledgment  Credit goes to many colleagues, friends and family whom without this master thesis would probably had not existed. Thanks and gratitudes goes to Jonas Åberg who have not just guided me through this project but also been an inspiration to me. Special thanks goes also to my brother Mostafa Al Jaberi for proofreading this thesis. Thanks to Johan Eneberg for giving me this chance and believing in me. Thanks to Jonas Skeppstedt for accepting my master thesis and for being an excellent teacher. I appreciate my colleagues’ help at work and for being friendly during all this time. And lastly but not least i am grateful to my family and friends who have been with me and believed in me all this time. All errors are mine alone.
## CONTENTS

- 5.2.4 Kernel space locking ................................................................. 18
- 5.3 Test case 1: locking ................................................................. 18
- 5.4 Test case 2: synchronization ...................................................... 19

6 LOSE in real life 25
- 6.1 Implementation ................................................................. 25
  - 6.1.1 POSIX Condition Variable ................................................... 25
- 6.2 Test case 1: Message passing one to one ................................... 25
- 6.3 Test case #2: Message passing many to many ............................... 26

7 Inter LOSE Communication 33
- 7.1 Unix domain sockets .............................................................. 33
- 7.2 POSIX message queue ............................................................ 33
- 7.3 The test ..................................................................................... 33

8 Conclusion 35

A LOSE dual vs. single core 37

B Source Code 39
  - B.1 Futex ................................................................. 39
    - B.1.1 atomic.h .......................................................... 39
    - B.1.2 my_futex.h ....................................................... 40
    - B.1.3 my_futex.c ....................................................... 40

Bibliography 42
Chapter 1

Introduction

1.1 Background
LOSE is an emulator developed by ST-Ericsson as a solution to minimize porting activity for already written software from the operating system OSE to Linux. LOSE does not emulate all the features of OSE but only what is used by ST-Ericsson code. Currently LOSE is not used or planned to be used in any ST-Ericsson products.

1.2 Methodology
This master thesis focuses on optimizing a known bottleneck and shed some light on other parts of the emulator, LOSE, that can be optimized. This bottleneck will be optimized by a 'black box' concept, where it is avoided to change the source code’s structure in order to not to break the requirements that are enforced on LOSE. LOSE was profiled and the focus is on the outside functions that it relies on to work properly. LOSE has no synchronization mechanism of it’s own but relies on POSIX mutex\(^1\), called pthread mutex. LOSE also relies on IPC facilities, inter process communications. LOSE needs to be able to communicate with another LOSE instance. The internal memory allocation can also be investigated, but because of lack of time it was not investigated.

To be able to profile LOSE it was necessary to evaluate possible tools and consider their pros and cons on running them on the target platform which is an ST-Ericsson mobile phone.

1.3 Report outline
The report consists of nine chapters with two appendices.

1. Introduction: this chapter.

2. Problem: a description of a known bottleneck found in common ST-Ericsson code for OSE.

\(^1\)http://.opengroup.org/onlinepubs/007908775/xsh/pthread.h.html (2009-07-08 11:30).
CHAPTER 1. INTRODUCTION

3. Target Platform: a brief description of the target platform.

4. OSE vs. Linux: overview of major differences between OSE and Linux that are interesting for this project.

5. Tools: evaluation of different tools that would be used to profile LOSE.

6. Synchronization profiling: investigating different synchronization mechanisms and conducting tests on them.

7. LOSE in real life: LOSE is tested in different test cases with alternative synchronization mechanisms that where chosen from the previous chapter.

8. Inter LOSE communication: a simple test case that sheds lights on other inter process communication layers that are faster than the one used by LOSE, LINX.

9. Conclusion: conclusion summing up this master thesis.

10. LOSE dual vs. single core: an interesting discovery of the impact on LOSE in an SMP system.


1.4 Limitations

Because of time limitations the focus for this master thesis has fallen on optimizing the synchronization and investigate alternatives. While conducting a simple test on testing different IPC layers as a base for future optimizations. LOSE internal memory allocation is also a possible candidate for optimization but was left out from this master thesis.

1.5 Target platform

The target platform consists of two sides. One is called the application side and the other the access side. Each side has it’s own processor which is an ARM 9. Each side can be considered as a separate embedded system with some shared resources. For example each side can run it’s own operating system. For this thesis we are concerned about the application side which we will run Linux on, while the access side runs OSE because it is a realtime operating system.
Chapter 2

Problem

ST-Ericsson has a huge amount of software written for OSE. One way to keep the porting activity to a minimum if switching OS is to develop an emulating layer. ST-Ericsson has developed an emulation layer called LOSE which emulates OSE on Linux. This master thesis is about investigating how to speed up LOSE. LOSE emulates only the sub-set of OSE that ST-Ericsson’s software utilizes. LOSE is currently not used in any active project.

The programs usually use standard library functions, like POSIX. These standard libraries are supported on many operating systems, including Linux. On the other hand different operating systems impose different concepts on how the software should interact with the operating system and/or between other tasks. One major feature that is different between OSE and Linux is signals. In OSE a signal is used to not only indicate an event but also send data, so it acts like inter process communication, while on Linux it is mainly to indicate an event. ST-Ericsson heavily uses signals between hundreds of tasks.

It was previously identified that LOSE is slow compared to OSE, especially during signal allocation and sending/receiving messages. The focus of this master thesis is optimizing this bottleneck. The common usage in ST-Ericsson’s code written for OSE is based on allocating a signal buffer, sending it to another process and then receiving it back with the needed information, see figure 2.1.

Just allocating a message and sending it from task\(^1\) A to B and then sending it back is almost 6 times slower than running the same code on OSE, see left graph 2.2.

Passing messages is not the only problem, signal buffer allocation is slow too. Calculating the speed involves a simple test of allocating a signal buffer and then freeing it. This time LOSE is about 4 times slower compared to OSE, see right graph 2.2.

LOSE was compiled without it’s debugging code and with optimization -O2 on gcc\(^2\) compiler in Linux\(^3\). But these two simple measurements are not enough. The fundamental differences between OSE and Linux will make LOSE even slower at passing messages between two processes than between threads in

\(^{1}\)LOSE is sending the message between pthreads, while OSE between processes, that is because OSE does not have a concept of thread, more on Chapter 3.1.


\(^{3}\)Linux version 2.26.8, see http://www.kernel.org/ (2009-07-06 14:26).
Figure 2.1: Common code in ST-Ericsson’s programs written for OSE.

Figure 2.2: Passing messages between two processes on LOSE and OSE. And allocation time for a signal buffer with size 10 KiB.

Linux, if we desire some memory protection between different processes.
Chapter 3

OSE vs. Linux

3.1 Thread and process definitions

One difference between OSE and Linux is how they structure and execute code. In both, the building block of executing code is called a process. Both see no difference between a process and a thread from a structural point of view [4], [6]. For OSE there is absolutely no difference, in fact the word thread is not used at all except when using POSIX to port Unix programs to OSE.

In Linux a thread is a process that has some resources shared with it’s parent. A thread is created by the system call \texttt{clone()} with the appropriate flags to tell Linux what these two processes have in common. A thread is a process that is lighter on the system because it reuses already existing resources. Sharing the address space is the most noticeably common resource that distinguishes the difference between threads and processes in Linux.

3.2 Signals

In OSE a signal is a message containing some data that is sent between tasks. The signal is saved in a signal pool. Each task has it’s own signal queue and when sending signals between tasks in the same domain only a pointer to the signal buffer is sent to the receiving tasks. On the other hand if the receiving task was in another domain the signal buffer will be copied to the signal pool of the other domain. The receiving task is the new owner of the signal buffer and can either reuse it or free it. \texttt{alloc()} is responsible for allocating a signal buffer large enough to have the required meta data plus the requested size, minimum 4 bytes for the signal number. ST-Ericsson’s code resides in the same domain so only a pointer will be sent between tasks.

While Linux uses a signal to notify an event, it does not send data between tasks. A task needs to register a function, called signal handler, that will be invoked when a signal with the corresponding number arrives. Tasks can ignore or block a specific signal number. Blocked signals will be pending until the process takes a way the block or ignores it. Arriving signals with the same number won’t be stacked, only one will arrive, in contrast to when using real time signals. There are some signal numbers that can’t be ignored nor blocked,
CHAPTER 3. OSE VS. LINUX

such as SIGKILL which will kill the process$^1$.

This is the major difference in the way OSE and Linux uses signals and the philosophy they impose on their usage. OSE signals are heavily used by the ST-Ericsson code.

3.3 Synchronization

The semaphores in OSE are counting semaphores. Normal semaphores are not owned by it’s creating process, but all the processes can wait and signal it.

In OSE each process has one so called fast semaphore, which is faster than a normal semaphore. Only the owner of the fast semaphore that can wait for it, while other processes can signal it. The fast semaphore can be accessed by only knowing the name of the owning process, any process inside or outside the domain can signal a fast semaphore unlike a normal one.

OSE also uses mutex which is a binary flag and it’s faster than a semaphore operation. A mutex is used to protect shared objects and it’s control block resides in user space.

Linux provides different synchronization mechanisms depending on wether they would be used in user or kernel space. Kernel space provides semaphores, mutexes and spin-locks among others. A semaphore can be initialized to behave as a mutex. Linux also supports read-writer semaphores which is to be used in cases where there is a distinct separation between writing (and reading) compared to reading only. So several processes can lock on read only, while only one can lock on writing and reading.

Another synchronization mechanisms is variable completion, which is used between tasks to notify an event. When an event occurs all waiting processes will wake up.

Spin-lock behaves like a mutex but instead of going to sleep it will loop until the lock is released. This is useful if the lock won’t be held for a long time. Spin-locks are most useful in multiprocessor systems, but they are useful even for uniprocessor systems because they disable preemption in kernel code so no other process kernel with higher priority can gain access to the locked data. The spin-lock code is disabled on a uniprocessor but preemption will still be disable until the lock is released. Interrupt handlers can use spin-locks.

While synchronization in user space is provided not only by the Linux kernel itself (as futex), but also by Glibc$^2$ such as pthread mutex. See chapter 5.2, for synchronization mechanisms available in user space.

3.4 Memory model

OSE uses a SAS, same address space, methodology to manage processes. This means that there is one logical address space which does not have to be the same as the physical address. If the CPU has an MMU unit there will be memory protection unless configured otherwise, ST-Ericsson’s code is running in an unprotected mode. SAS is more friendly to real time requirements because


it minimizes page switching, but all the programs will share the same address 
space in contrast to how Linux handles it with each process having it’s own 
logical address space. This is called MAS, multiple address space.

3.4.1 Memory types
OSE has two types of memory buffers: either a pool or a heap. Pools are used 
to allocate signal buffers and process stacks. There is one system pool used by 
the kernel and each program can allocate it’s own pool to increase security, by 
not allowing the mixing of signal buffers from different domains. ST-Ericsson 
code uses only one pool. Heaps are used for large dynamic allocations. Heap 
memory is not for allocation of signal buffers.

3.4.2 Memory allocation
OSE uses two memory allocating functions that are mainly used by programs:

- `alloc()` is a deterministic fast allocation that has very low external frag-
  mentation and allocates from the pool. It is heavily used to allocated 
signal buffers.

- `malloc()` is used to allocate from the heap, it has a small overhead and 
can allocate larger buffers.

In Linux user space applications uses C library functions to allocate in bytes such 
as `malloc()`, kernel space applications uses `kmalloc()`. According to the doc-
umentation in the source code for `malloc()`, it behaves differently for different 
size requests. `malloc()` has a very small overhead. One of the biggest difference 
between kernel’s `kmalloc()` and user space `malloc()` is that `kmalloc()` allo-
cates memory continuous in physical memory, while `malloc()` only guarantees 
that the allocated memory is logically contiguous.
Chapter 4

Tools

4.1 LOSE

LOSE as discussed is meant to run programs written for OSE under Linux. It achieves that by defining structures and methods needed to run OSE programs. One can run more than one LOSE simultaneously called a LOSE instance. This would provide memory protection between tasks that are residing in different LOSE instances. In each LOSE instance there is a processes called a Proxy that listens and adds/removes phantom processes\(^1\) to keep all the LOSE instances in synchronization with each other. LOSE uses an inter process communication layer called LINX\(^2\) to communicate between tasks that should resides on different LOSE instaces. LINX is choosen for it’s simplicty and because it is used to communicate with tasks residing in the access side of the platform. LOSE uses the POSIX threads and mutex for sending signals between process in the same LOSE instance.

Every time a process wants to send a signal it checks the destination process’s structure saved in LOSE’s global structure. If the destination process turns out to be a phantom process it sends the message through LINX, otherwise the signal is added to the process signal queue, see figure 4.1.

Both communication within a LOSE instance and between two LOSE instances are interesting from an optimization point of view. Tasks in the same LOSE instance tend to communicate a lot, while communication through LINX is much slower.

4.2 Lauterbach

One is not always able to choose the desired tool to analyze a program. In this case the desired tool is Lauterbach which gives data down to the instruction level on the platform. Unfortunately without using the correct license, Multi Core, it can only analyze a single MMU context. Linux kernel space uses one MMU context and user space uses another MMU context for each process. The

\(^1\)Phantom processes are empty processes where their only job is to indicate that the requested process is residing in a different LOSE instace.

\(^2\)LINX is developed by ENEA as an independent IPC, see http://sourceforge.net/projects/linx (2009-05-14 15:35).
CHAPTER 4. TOOLS

4.3 Oprofile

4.3.1 Background

Oprofile is a statistical tool that samples the whole system. It can output the result in many different ways, generally as a table listing the symbols and binary samples, annotated source listing or as XML data to be used by other tools. What makes Oprofile a low-overhead profiling tool is that it utilizes special hardware in the processor. If the processor does not have the needed hardware Oprofile will run in interrupt mode, where it will generate software interrupts to sample the system. Oprofile is divided into two parts, one that resides in the kernel from the beginning and one that user space starts/controls/stops profiling and shows the output.

Hardware support is specific to each processor type, where each type has it’s own events that can trigger Oprofile to sample. When Oprofile takes a sample it will gather the necessary information according to how it has been configured. It can for example trace the stack to save information on which symbols are calling the sampled symbol. The given output graph looks similar to gprof’s\(^3\) graph. The GNU C compiler needs to place a special code in the beginning

---

and at the end of every function, during compilation, so that gprof is able to
profile the program. After the termination of a program information about the
call graph will be logged in a predefined file, but only if the program exited
normally. Also gprof will sample the data, i.e. it is also statistical tool. The
problem with gprof is that it can’t work properly with threads which LOSE
uses heavily. To run Oprofile we don’t need to recompile our program and if
hardware support is available it won’t impact performance compared to gprof.
Oprofile can also group samples according to the given separators which are
according to the Oprofile manual [2004]:

- none: no separation.
- lib: separates dynamically linked library samples per application.
- kernel: separates kernel and kernel module samples per application. Hav-
ing kernel separation also imply library separation.
- thread: separates for each thread.
- cpu: separates for each cpu.
- all: all of the separations options.

Oprofile needs to have access to the vmlinux binary with debug information to
be able to show meaningful data about what is happening inside the kernel.

4.3.2 Hardware events

As mentioned earlier for Oprofile to work as intended it needs to run on a
processor that has hardware support for so called performance counters. For
each processor type there are events that the processor can track by hardware
counters. When the counter reaches zero the processor interrupts and Oprofile
will sample the currently running code. For example Intel Core 2 Duo\(^4\) have
events that can count how many cycles the processor have been running, how
many multiplications it has executed, how many rejected L2 cache requests and
how many pipeline flushes. For more events see Appendix A in [1]. Oprofile can
list all the available events for the processors type it is currently running on by
running the binary ‘ophelp’.

When the counter reaches zero an interrupt is issued. For example on a
Intel Core 2 processor the interrupt is a none maskable interrupt called NMI,
[5]. NMI interrupts have the highest priority, the processor cannot be in a state
that is not interruptible by these interrupts.

4.3.3 Running Oprofile example

Before profiling one needs to specify the events that the hardware counters need
to track, for this example the counters should generate an interrupt every 6000
cpu cycles. Afterwards Oprofile is started before executing the program to be
profiled. In the end Oprofile should dump the data and stop profiling:

>> opcontrol −−event=CPU_CLK_UNHALTED:6000
>> opcontrol −−start
>> ./my_program
>> opcontrol −−dump
>> opcontrol −−stop

Then another tool is used to view the stats of the system after dumping the profile data in the file system, with the following output:

<table>
<thead>
<tr>
<th>samples</th>
<th>cum. samples</th>
<th>%</th>
<th>cum. %</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>78989</td>
<td>78989</td>
<td>7.3949</td>
<td>7.3949</td>
<td>vmlinux</td>
<td>cpupri_set</td>
</tr>
<tr>
<td>72763</td>
<td>151752</td>
<td>6.8120</td>
<td>14.2068</td>
<td>vmlinux</td>
<td>schedule</td>
</tr>
<tr>
<td>62929</td>
<td>214681</td>
<td>5.8913</td>
<td>20.0982</td>
<td>vmlinux</td>
<td>mwait_idle</td>
</tr>
<tr>
<td>34681</td>
<td>395576</td>
<td>3.2468</td>
<td>37.0333</td>
<td>vmlinux</td>
<td>sched_clock_cpu</td>
</tr>
<tr>
<td>29070</td>
<td>424646</td>
<td>2.7215</td>
<td>39.7548</td>
<td>oprofile</td>
<td>oprofile</td>
</tr>
<tr>
<td>26931</td>
<td>451577</td>
<td>2.5212</td>
<td>42.2761</td>
<td>vmlinux</td>
<td>futex_wait</td>
</tr>
<tr>
<td>24717</td>
<td>476294</td>
<td>2.3140</td>
<td>44.5901</td>
<td>vmlinux</td>
<td>update_curr_rt</td>
</tr>
<tr>
<td>19823</td>
<td>496117</td>
<td>1.8558</td>
<td>46.4459</td>
<td>vmlinux</td>
<td>futex_wake</td>
</tr>
</tbody>
</table>

This is an example output that is modified to just give a sense for how the output looks like, in this case Oprofile is listing per symbol.

4.3.4 Considerations and limitation

Unfortunately the platform is running the V5 generation of ARM processors which means there are no performance counters, these were added in V6. Thus Oprofile on the platform can only run in timer interrupt mode. In timer interrupt mode Oprofile can’t sample code as often as it would had done with hardware support. Without hardware support code that has interrupts disabled can’t be sampled. Instead the profiling will be conducted on an x86 computer which support many events. Not all events that are available on x86 are interesting because of the difference in architecture compared to ARM 9.

Also on the x86 one cannot rely on the samples to be accurate down to the instruction level, only to the surrounding code. That is because the x86 architecture has atomic instructions that can’t be interrupted by NMI while it is running. Also, because of multiple execution units, out-of-order execution and delays in the interrupt handling the sample can be assigned to the wrong instruction. There are other issues that cause the inaccuracy of instruction level sampling, which can be found in [5].

The longer we run a program the more accurate stats we get, because the longer run increases the chances of more code being sampled which gives a more accurate image of which part the code is dominating.

Another thing we need to keep in mind are buffers. Oprofile reports when there are overflows in the buffer, so when this occurs either we make the buffer bigger or shorten the profiling time. This depends on how Oprofile was configured and how frequent the events happen. Furthermore, it is important to note the depth of the desired call graph. Other such factors will affect how fast a buffer becomes full and how fast Oprofile can flush it to the file system.
4.4 Valgrind

4.4.1 Background

Another tool that would seem interesting is Valgrind. “Valgrind is an award-winning instrumentation framework for building dynamic analysis tools”\(^5\). It uses another approach which checks a program for its correctness with some profiling capabilities. See table 4.1 for a summary of Valgrind’s main tools [2009].

<table>
<thead>
<tr>
<th>Tool name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcheck</td>
<td>Memcheck detects memory-management problems.</td>
</tr>
<tr>
<td>Cachegrind</td>
<td>Performs detailed simulation of the L1, D1 and L2 caches.</td>
</tr>
<tr>
<td>Callgrind</td>
<td>Cachegrind plus extra information about call graphs.</td>
</tr>
<tr>
<td>Massif</td>
<td>Massif is a heap profiler.</td>
</tr>
<tr>
<td>Helgrind</td>
<td>Thread debugger which finds data races in multithreaded programs.</td>
</tr>
</tbody>
</table>

Valgrind can analyze any program without recompiling or relinking the program as it uses dynamic binary instrumentation. One needs to prefix the program with Valgrind followed by the tool that should debug/profile the program. The most interesting tool for this thesis would be Callgrind which builds a call graph and calculates cost for each function. The cost is based on data reads, cache misses etc.. Callgrind will calculate the inclusive cost for each function. The resulting data is written to a file and can be viewed by a graphical application called KCachegrind\(^6\). Callgrind can also perform cache simulation which is based on the Cachegrind tool.

4.4.2 Considerations and limitation

It is not possible to run Valgrind on the platform because it has not been officially ported by the Valgrind team to support arm. It is out of the thesis’s scope to try to port it or attempt to use unofficial patches to Valgrind. For this thesis this means that Valgrind can only be used on a x86 PCs.

4.5 Measuring code

There is still a need to know when a program is running faster, and that is accomplished by taking time measurements while running LOSE.

4.6 Conclusion

4.6.1 Choosing a platform

Valgrind can inform how many calls there have been for a function, but not how much time is spent inside it. Oprofile, on the other hand can provide information

\(^5\)Description of Valgrind is found at Valgrind own website, http://valgrind.org/ (2009-03-25, 15:15).

on which functions are dominating according to how many samples were taken
during execution, which can be approximated as elapsed time. This leads to the
conclusion that Oprofile is suitable for this thesis.

Oprofile is useful most on x86 with hardware support, and this means that all
optimization will be done on the x86. Finally the optimization will be adapted
to the platform to verify the existence of a speedup, see figure 4.2.

The optimization will run on an x86 (Intel Core 2 Duo 6600 @ 2.40 GHz) PC
with Kubuntu 7.10 installed with a recompiled kernel. Main system components:

- Linux kernel 2.6.28.8
- Oprofile 0.9.4
- LINX for Linux 2.1.0

All profiling and verification of LOSE is done in single mode on x86. In single
mode a process like the X graphical system is turned off to minimize disturbance
while measuring time. This helps by making the environment as similar as
possible to the one on the platform.

The platform is running the similar major kernel version 2.6.28. Another
situation worth considering is to disable SMP on the PC, which removes support
for symmetrical multiprocessing, because the platform is running on a single
core from the application point of view. The problem that was faced was that
Oprofile did not support running on none SMP kernels with hardware support
enabled. Luckily there was another solution, to turn off a core even when
running under SMP kernels. The only change needed was to write a zero in
the file /sys/devices/system/cpu/cpu1/online. For information on the effects of
dual or single core see appendix A.

Figure 4.2: Work flow that was used to optimize LOSE.

\[\text{http://sourceforge.net/projects/linx/ (2009-04-02 09:53)}\]
Chapter 5

Synchronization profiling

The goal is to measure the time it takes to synchronize messages passing between two processes within one LOSE instance.

5.1 Oprofile on LOSE

Oprofile shows which code is executed the most. Oprofile is only needed to run once to know which symbol got the most samples. To have a sufficiently good idea of which symbols are executed the most LOSE needs run long enough, for example $4 \cdot 10^6$ sending/receiving messages. Profiling was made with the Oprofile option CPU_CLK_UNHALTED=90000, which means that it samples the system every $9 \cdot 10^4$ cycles. It would be possible to lower the cpu cycle down from $9 \cdot 10^4$ to $6 \cdot 10^3$ to sample more often. The drawback with more frequent samples is that Oprofile’s buffers need to be increased, which in turn will lead to slow-downs. The reason for decrease in speed is because Oprofile will need more time to save the results to the file system, thus risks losing samples during that time. The following output shows samples separated by using the kernel separator, discussed in chapter 4.3:

```
CPU: Core 2, speed 2400.36 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 90000
CPU_CLK_UNHALT | | samples | |
-----------------|-----------------|----------|
254015 | 98.6278 losebinary
      |                  |
179382 | 70.6187 vmlinux
      | 43010 16.9321 libpthread-2.6.1.so
      | 31604 12.4418 losebinary
```

Listing the result per symbol gives (threshold 1%):

```
CPU: Core 2, speed 2400.36 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 90000
samples | % | image name | app name | symbol name
15708 | 6.5223 | libpthread-2.6.1.so | losebinary | pthread_mutex_lock
16365 | 6.3541 | vmlinux | losebinary | sysenter_past_esp
16363 | 6.3534 | libpthread-2.6.1.so | losebinary | __pthread_mutex_unlock_usercnt
15464 | 6.0043 | vmlinux | losebinary | schedule
14735 | 5.7212 | vmlinux | losebinary | cpupri_set
14173 | 5.5030 | vmlinux | losebinary | __switch_to
8225 | 3.1936 | vmlinux | losebinary | update_curr_rtt
```

14
5.2 Different synchronization methods

Different synchronization mechanisms are to be tested to see if there is one that is faster than pthread_mutex, which is what LOSE currently uses.

5.2.1 Futex

Futex is a lightweight method for process synchronization that can also be used directly from user space [3]. Pthread mutex uses futex as their primitive lock, the goal here is to try to use only futex skipping eventual overhead in using pthread mutex. Locking an uncontended futex is all done in user space with the help of atomic operations. The kernel is only involved when a futex is already locked to block the calling task\(^1\) and add it to a waiting queue. When a futex is unlocked the kernel arbitrarily chooses a task from the waiter queue to wake up. The trick to having a fast user space locking/unlocking on an uncontended futex is atomically changing an integer to indicate that the futex is locked.

When a request to sleep is done both a pointer to the futex and an expected value will be provided to the kernel system call. The kernel will put the task to sleep only if the expected value is equal to the value that the futex have. Otherwise the system call returns. The wake up request needs a pointer to the futex and a parameter specifying how many tasks to wake up, usually one or all.

The problem with futex is that it needs to be manually implemented and that the atomic operations are architecture dependent. The thesis author’s implementation was used, and it is based on Ulrich Drepper’s paper “Futexes Are Tricky” [2008] with some small modifications, for a deeper description refer to the mentioned article:

```c
void futex_lock(int *futex) {
    int c;
    if (!atomic_compare_and_exchange_val_acq(futex, 0, 1)) != 0){
```

\(^1\)Task is referring to both a process and a thread in Linux, futex does not have an owner, anyone can unlock it after it has been locked.
CHAPTER 5. SYNCHRONIZATION PROFILING

```c
if (c != 2) {
    c = atomic_exchange_acq(futex, 2);
} while (c != 0) {
    futex_wait(futex, 2);
    c = atomic_exchange_acq(futex, 2);
}
```

```c
inline void futex_unlock(int *futex) {
    if ((c = atomic_exchange_acq(futex, 0)) == 2) {
        futex_wake(futex, 1);
    }
}
```

There are three valid values for this futex implementation, 0 meaning the futex is unlocked, 1 meaning locked and 2 that there are also waiters on this futex. It would have been sufficient with just two states locked and unlocked, but that would have required that every time the futex is unlocked there will be a system call to wake up no one if there where no waiters.

It was not an easy task to port the atomic operations directly to the platform because they used specific x86 instructions that had no counterpart in the ARM V5 architecture. Basically how one would implement atomic operation on the platform is to make a system call to disable interrupts. By disabling the interrupts one is sure that the code is not interrupted while working on the futex.

A solution was found for the ARM architecture implementation in the Linux kernel. The solution is having a read only code in the kernel memory space that does the atomic operation surrounded by two labels. Then in the IRQ and data abort handlers there is a check to see if the program counter of the processor was equal or in between one of these two labels. If that is the case it would restart the operation and afterwards continue the interrupt handling. The atomic code won’t be interrupted another time because interrupts are disabled now.

There is only one atomic operation implemented in this way, it is called compare and exchange. The code resides in `/arch/arm/kernel/entry-armv.S` and looks basically like this:

```asm
1:  ldr  r3, [r2]  ; load current val
   subs r3, r3, r0  ; compare with old val
2:  streq r1, [r2]  ; store newval if eq
   rsbs r0, r3, #0  ; set return val and C flag
   usr_ret lr
```

What this does is load the current value, compare with the old value (or expected value) and if they are equal it saves the new value. The code returns 0 if the new value is stored in memory otherwise a non-zero value.

The return value of the operation on the ARM architecture was another problem. Drepper’s algorithm was based on that the code should return the current value in all cases. So the algorithm was adapted accordingly to this atomic operation:

```c
void futex_lock(int *futex) {
    if ((atomic_compare_and_exchange_val_acq(futex, 0, 1) != 0) {
        while ((atomic_compare_and_exchange_val_acq(futex, 0, 2) != 0) {
            if (*futex == 1) {
                atomic_compare_and_exchange_val_acq(futex, 1, 2);
            }
        }
    }
}
```

The code is from kernel 2.6.28.8
CHAPTER 5. SYNCHRONIZATION PROFILING

```c
inline void futex_unlock(int *futex)
{
    if (atomic_compare_and_exchange_val_acq(futex, 2, 0) == 0)
    {
        futex_wake(futex, 1);
    }
    else{
        *futex = 0;
    }
}
```

The main idea is the same: we have a futex with three states. A big change from the previous implementation is that the lock function checks the futex if it is currently having the value 1 and only then calls the atomic operation to change it into 2. This is not an atomic operation which could fail unexpectedly but it is an optimization and it does not break the atomicity of futex. Complete source code is found in appendix B.1.

5.2.2 Semaphore

Semaphore is referring to the semaphore that is defined by the Opengroup. Basically a semaphore is initialized to 1 to function like a mutex. Using `sem_post()` will increase the count. While `sem_wait()` will decrease the semaphore if the count is greater than 0, otherwise the function will block until the count is increased again or the call is interrupted. During initialization of a semaphore it can be configured to being either private or public, if public then any process can call `sem_wait()`.

5.2.3 System V semaphore

This semaphore comes from the old System V IPC semaphores. The biggest difference is that these are persistent semaphores which means that they are still in the system even when the programs exit, until they are closed explicitly. So you can view all the semaphores in the system by using the `ipcs` which will show something like this:

```
--------- Shared Memory Segments ---------
key shmid owner perms bytes nattch status
--------- Semaphore Arrays ---------
key semid owner perms nsems
0x6d01914a 0 monthadar 660 1
0x6d019102 32769 monthadar 660 1
--------- Message Queues ---------
key msgid owner perms used-bytes messages
```

The `ipcs` tool shows not even the semaphore but also shared memory and message queues for the System V IPC. In this case there are two semaphores arrays each having one semaphore. Each semaphore is a structure containing 25 sub semaphores but for this test only one is used. For the interested reader there is a very informative description about how to use them, http://www.cs.cf.ac.uk/Dave/C/node26.html.

4Last checked address validity 2009-05-15 17:53.
5.2.4 Kernel space locking

Linux has its own locking mechanism, but these are only accessible from kernel space. So for a user space program to use for example the kernel mutex it needs to communicate with kernel space one way or another. Two apparent solutions exist which are either adding a new system call to the kernel or writing a new module that can be loaded dynamically. Communication with the module is through the IOCTL mechanism, which controls streaming devices.

Two kernel modules were implemented, one for mutex and the other for semaphore, both behave as a mutex and thus both have the following methods:

- void create_mutex(int file_desc, unsigned int *mutex)
- void free_mutex(int file_desc, unsigned int *mutex)
- void lock_mutex(int file_desc, unsigned int mutex)
- void unlock_mutex(int file_desc, unsigned int mutex)

The main idea is that when a user space program opens a predetermined device node the program uses ioctl to issue commands to the module along side a parameter. The parameter is of type unsigned int and has different meaning depending on if it is creating/freeing a mutex or locking/unlocking a mutex. The idea is not to maintain a list of created mutexes but instead using the SLAB allocator to allocate and free memory. When a user space program wants to create a new mutex it passes along the user space address to where the module should save a pointer of the newly created mutex, and the same goes for freeing. When a lock/unlock operation is requested user space passes the stored address to the module and the module interprets it as a pointer to the mutex.

5.3 Test case 1: locking

The first test on x86 is one task to lock/unlock a lock seeing how much latency each synchronization type has for an uncontended lock. This will be tested by the different synchronization mechanisms with a different number of locks, verifying the linearity of the elapsed time. The different runs are $10^5$, $10^6$ and $10^7$ locks/unlocks. For each number of locking the test will run 100 times and the maximum, minimum, arithmetic mean and standard deviation of the elapsed time will be noted. Matlab is used to calculate the results. Standard deviation in Matlab is calculated for 100 elements as:

$$\text{std}([x_1, x_2, \ldots, x_{100}]) = \sqrt{\frac{\sum_{i=1}^{100}(x_i - \bar{x})^2}{99}}$$

$\bar{x}$ is the calculated arithmetic mean value of all the 100 elements. One can assume that the distribution of the elapsed time is a normal distribution.

\footnote{It is out of this thesis’s scope to describe how ioctl works, see http://www.opengroup.org/onlinepubs/009695399/functions/ioctl.html for more information. (2009-05-29 11:06)}

\footnote{This is a security risk, because a segmentation fault will happen if the address is incorrect. But for this thesis this implementation just shows how fast kernel locking can be at best. (2009-07-07 15:07)}
and thus the standard deviation will act as an indication of how spread the measurements are. The less the standard deviation value is the better.

From the graphs, in figure 5.1, it is clear that mutex and futex are the winners here. Calculating a speedup value between mutex and futex shows how much futex is faster.

<table>
<thead>
<tr>
<th>#lock/unlock</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^9$</td>
<td>0.1713</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.1697</td>
</tr>
<tr>
<td>$10^7$</td>
<td>0.1710</td>
</tr>
</tbody>
</table>

Table 5.1: Speedup futex vs mutex locking time.

The speedup is calculated as:

$$1 - \frac{\text{futex mean}}{\text{mutex mean}}$$

As shown in table 5.1 there is about 17% speedup.

### 5.4 Test case 2: synchronization

The next test shows how much time it takes when two threads try to synchronize between each other. Assume we have two threads $A$ and $B$ and two locks $L_1$ and $L_2$. This is how the test will run:

Thread $A$ would try to lock $L_1$, do something then unlock $L_2$ to tell $B$ that it is done. $B$ starts by locking $L_2$, do it’s job and then unlock $L_1$. Initially $L_2$ is locked to be sure that process $A$ starts first. This way we always have contention on the locks, both processes will always go to sleep and be awakened. Of course in this test both processes are not doing anything while synchronizing. Results from graph 5.2 shows that there are big variations between maximum and minimum elapsed time, especially for user space synchronizations.

It is important to note that this test will not work when pthread_mutex or Linux kernel space mutex has error checking enabled. That is because in this scenario thread $A$ locks $L_1$ while thread $B$ unlocks it and this is a violation of how a mutex was intended to be used. For a pthread mutex the Open Group dictates “Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior”\textsuperscript{8}. While Linux kernel documentation\textsuperscript{9} says that when debugging is turned on the mutex will check that only the thread that have locked it can unlock it. Despite that, the goal is to see which one is faster when it is always contended, so the violations are ignored.

From the results it turns out that it is not guaranteed to be 100% contention on the locks, and that is reflected in the large standard deviation result and that large difference between maximum and minimum time elapsed. The main reason for this is because there is no guarantee of atomicity while unlocking one lock and locking the other so that the calling thread would block. The idea was for the code to be executed as:

\textsuperscript{8}http://opengroup.org/onlinepubs/007908775/xsh/pthread_mutex_lock.html (2009-06-20 15:56)

\textsuperscript{9}Found in the documentation folder associated with the Linux vanilla source code. Path /Documentation/mutex-design.txt.
CHAPTER 5. SYNCHRONIZATION PROFILING

1: Initially lock L2 is locked.
2: Thread A
3: lock(L1)
4: do_something()
5: unlock(L2)
6: lock(L1)
7: lock(L2)
8: do_something()
9: unlock(L1)
10: lock(L2)
11: do_something()
12: unlock(L2)
13: lock(L1)
14: do_something()
15: unlock(L1)
16: lock(L2)
17: goto line 11

The start is arbitrary but the assumption was that each thread would work until it blocks itself which would result in 100% contention. The problem is that for example line 9 and 10 are not executed atomically.

To verify the explanation of the variance in elapsed time an extra variable was added to the futex implementation that is incremented every time the calling thread tries to take the futex and finds it unlocked. The total number of times lock() is called from both threads is $2 \times \#syncs + 1$, 1 being for the first lock to block one thread from start, see table 5.2 for a run of $10^6$ synchronizations. The gathered data will show that the maximum values of elapsed time from the graphs (see 5.2) is when there is almost 100% contention. The data also shows that the minimum elapsed time is when there is about 50% contention, everything thing else is in between. The minimum contention is 50% otherwise the test is incorrect, because from a higher level the two threads will still alternate execution of do_something() as described above.

<table>
<thead>
<tr>
<th>#unlocked</th>
<th>elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.3</td>
</tr>
<tr>
<td>1,000,002</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 5.2: Number of times futex was unlocked when a thread tried to lock it and the elapsed time (grossly approximated) of the synchronization test.

Running in a FIFO, first in first out, scheduling policy gives a much more deterministic results, as can be seen in graphs 5.3. This is confirmed by that the locks where only found to have always been 3 times unlocked during a lock request. The speed is slower but it is something one has to be prepared to give up when seeking more deterministic values.

A pthread configured with the SCHED_FIFO attribute will always run over all none real-time and lower priority tasks. If there is already a running task with the same priority the new task will be put in a first in first out queue.

10Best case: number of unlocked times should have been 2 and not 3. But because thread A starts before B, B finds L2 unlocked, but that is just arbitrarily.
With this policy the task will run until completion, it yields or a higher priority SCHED_FIFO/SCHED_RR, round robin, task wants to run. SCHED_RR behaves like SCHED_FIFO but with a time slice. When a SCHED_FIFO task yields it is put in the end of the queue of its priority level [6] pages 59-60. In this test both threads have the lowest realtime priority 1, the highest is 99.

It is relatively easy to change the scheduling policy for a pthread to SCHED_FIFO as shown in the program listing below:

```c
pthread_t th;
struct sched_param param;
pthread_attr_t attr;
param.sched_priority = 1;
pthread_attr_init(&attr);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
pthread_attr_setschedparam(&attr, &param);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_create(&th, &attr, do_something, NULL);
```

Note in the example code above that if inheritance scheduling is not set to PTHREAD_EXPLICIT_SCHED before creating a new thread, then the created thread would inherit the scheduling attributes from the creating thread. For a task to have the right to change the scheduling policy it needs to be running with root\textsuperscript{11} privileges.

\textsuperscript{11}Root is the user in Linux that has total control and can change anything in the system.
Figure 5.1: Graphs showing elapsed time for locking/unlocking in different using different mechanism.
Figure 5.2: Graphs showing elapsed time for synchronization between two task for different mechanisms.
CHAPTER 5. SYNCHRONIZATION PROFILING

Figure 5.3: Graphs showing elapsed time for synchronization between two threads for different mechanism with FIFO real-time scheduling.
Chapter 6

LOSE in real life

6.1 Implementation

The decision was made to implement futex because it was the fastest in an uncontended scenario and kernel mutex because it’s the fastest in a contended scenario. Both were easy to implement. It was sufficient to change all pthread mutex declaration, initialization and lock/unlock method invocations.

6.1.1 POSIX Condition Variable

LOSE uses pthread condition variables to notify a process that there are incoming signals. This is only done if the condition variable flag is enabled while compiling LOSE otherwise a pthread mutex is used as a condition variable. The reason is performance, as condition variables are slower than mutexes. Another reason is that there will always be a maximum of one waiter on the pthread mutex. To use a mutex in this way means that the structure for the mutex needs to be probed to know how many times to call \texttt{lock}().

Using a futex as a condition variable by probing the sturcture is straight forward. On the other hand kernel mutex hides the structure from user space so there is no easy way of probing it. Implementing kernel mutex in LOSE means that all the pthread mutexes have been changed except for the one that is used as a condition variable.

6.2 Test case 1: Message passing one to one

This test is the same as the synchronization test, but using the send and receive function in LOSE to send messages between two tasks. LOSE uses first in first out realtime scheduling policy and thus needs to have the correct permissions to change priority. The messages sent between the tasks are empty.

The same test ran twice, once where both tasks have the same priority, figure 6.1, the other both having different priorities, figure 6.2.

It is clear that when having high contention on the locks the speedup of futex, compared to mutex, is less than having low contention.

Then the test was conducted on the platform to verify the speedup. This time the values where more deterministic than what was on the x86, one reason
can be that the kernel on the platform does not have SMP enabled which would make locking easier. For example spin locks will be compiled away when not using SMP, just disabling interrupts is enough to guarantee not being interrupted in a critical section [6]. Again running the test twice with different priority settings. Futex is the fastest in both tests, for same priority see figure 6.3, for different priority tasks see figure 6.4. Only the arithmetic mean elapsed time has been taken. A speedup of about 12% for same priority tasks and almost 14% for the other test. The kernel locking method is the slowest in both cases.

An important aspect to note here is that despite the effort to have a similar organization between the PC and the platform, the difference in cache organization made an impact on the speedup. The structure that is used the most in this test case is the specific structure for each process in LOSE. The platform has only one level of cache, commonly called L1, to speed up data/instruction access latency from the main memory. The L1 is divided into two separate caches, one for instructions (32KiB) and the other for data (16KiB). While the PC has both L1 and L2 caches on the core. L1 is also divided into two caches, both are 32KiB, while L2 is a unified cache for both instructions and data, it’s size is 4MiB.

If the structures size is not modulo the line length of the cache there will be a performance penalty. It is apparent only on the platform because there is no L2 cache that can work as a buffer every time the L1 data cache is full or is flushed. Because of this the structure was padded to the size that would make it module line length. Without this padding there is a small speedup penalty.

6.3 Test case #2: Message passing many to many

A more interesting test is to have many tasks send and receive from/to each other. This is to test a scenario that is close to how the ST-Ericsson software is written. For this test there are twenty processes that all send signals to all other tasks and awaits a signal from them all. The signals are allocated before measuring the time. And this iteration is repeated several times with different priority level scenarios. Initially all tasks have the same priority, secondly, half the processes have one priority and the other half have the another. Finally, all tasks have different priorities. In this test all tasks execute the same function so there is no master/slave relationship between them.

Figure 6.5 shows the results for running $10^3$, $10^4$ and $10^5$ times, futex is still faster than the other two methods in all scenarios.

Because this test runs very slowly on ARM, the $10^5$ test was not conducted. It would take ioctl 50 minutes to complete one iteration. Because the standard deviation is small on the platform it was meaningless for it to run that long. See figure for the results 6.6.

Again futex is the fastest with the following speedups compared to mutex, most notable speedup was when all the tasks had the same priority, see table 6.1.

The third scenario gave a noticeably large speedup compared to the other two scenarios. Running Oprofile on all the scenarios and comparing pthread mutex against futex reveals an interesting fact. In the first two scenarios both LOSE implementation of pthread mutex and futex spend about the same time in the kernel, about 70% and 65% for different priorities and two priority classes
respectively. The difference is when all tasks have the same priority, then LOSE with pthread mutex spends about 40% in the kernel, while the futex implementation spends 18%. We have not had time to investigate the reason for this behaviour.

<table>
<thead>
<tr>
<th>Priority test</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different</td>
<td>13</td>
</tr>
<tr>
<td>Two priorities</td>
<td>16</td>
</tr>
<tr>
<td>Same priority</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 6.1: Different speedups depending on the priority scenarios.
Figure 6.1: Graphs for LOSE using different synchronization mechanisms. Both tasks have the same priority.
Figure 6.2: Graphs for LOSE using different synchronization mechanisms. The tasks have different priority.
CHAPTER 6. LOSE IN REAL LIFE

Figure 6.3: Graph showing mean time for LOSE on the platform, all tasks running with same priority.

Figure 6.4: Graph showing mean time for LOSE on the platform, all tasks running with different priority.
Figure 6.5: Graph showing elapsed time for LOSE on the x86 between many tasks.
Figure 6.6: Graph showing elapsed time for LOSE on the platform between many tasks.
Chapter 7

Inter LOSE Communication

LOSE uses LINX as a way to communicate with other LOSE instances. One could run a simple test of sending/receiving signals between two tasks that reside on different LOSE instances. To find if there is another IPC layer that is faster than LINX.

7.1 Unix domain sockets

Unix domain sockets is a POSIX standard which means it is widely available. It uses files to open connections between the different tasks but use the main memory to transfer data.

7.2 POSIX message queue

POSIX message queue is also available on many operating systems, also like the other two IPCs uses the file system space to establish a connection between different sides. The biggest difference is that POSIX message queue are kernel persistent. A sent message stays in the message queue if the program has terminated without reading the message or until a system reboot is done.

7.3 The test

Because of the scope of this master thesis only the IPCs themselves will be tested.

The test will send/receive messages between two different tasks each in it’s own process. For this test it won’t make a difference what priority each task has, because they use the functions send and receive and not the lower level function of lock and unlock and thus there are no race conditions. See figure 7.1 for the results on x86 and figure 7.2 for the platform values. Both tests were sending/receiving 10^6 messages. On both tests POSIX message queue was the fastest, most notably on the platform.

On both architectures POSIX message queue was the fastest with a 45% speedup compared to LINX on the platform and 19% on x86. One important difference is that the x86 and the platform are not running the same LINX
CHAPTER 7. INTER LOSE COMMUNICATION

Figure 7.1: Graph showing elapsed time for different IPCs on the PC.

Figure 7.2: Graph showing elapsed time for different IPCs on the platform.

version. The platform is running 2.0.1 while x86 is 2.2.1. There were difficulties to run the same version on both architectures. The conclusion is that there is a noticeable speed up on both architectures and thus concludes that POSIX message queue is the fastest for this test.

1LINX version 2.1.0 on x86 gave about the same results as 2.2.1.
Chapter 8

Conclusion

Futex addressed some issues with how the majority of ST-Ericsson’s code was written, see figure 2.1, and delivered a noticeable speedup of around 10% or more depending on the the priority of tasks in LOSE. Even though on a lower level futex didn’t deliver any noticeable speedup when the locks where always contended. Despite that there still is a speedup on LOSE while sending and receiving messages. That is probably because the locks are not fully contended all the time in real life applications where tasks do a lot more than just lock and unlock. Futex’s advantage is when the lock is unlocked and thus skipping some overhead that pthread mutex has to deal with.

The drawback of using futex is that there is no standard library with functions to use. What this thesis also demonstrated is that even for old projects it is easy to gain extra speedup by just exchanging all pthread mutex into futexes.

The current implementation of futex is a simple one, that lacks a lock function with time out, which is needed by LOSE to emulate a receive function with time out. It should be relatively easy to implement such a function because the kernel provides a wait operation with time out.

Interestingly the kernel mutex was noticeably faster than all the user space mechanisms explored in this thesis in a fully contended lock. One way to investigate an eventual speedup is to move either only the send()/receive() functions of LOSE into kernel space or maybe move the whole LOSE. Still one is not guaranteed to see any speedup because the lock are not always contended and maybe the overhead imposed by always doing a system call would make it slower.

The synchronization is not the only thing one can optimize, as seen in chapter 7, LINX can be exchanged with some other IPC facilities. Also LOSE’s internal memory allocation for signal buffers was noted to be slow.

The investigated tools were an important part of finding the source of the bottleneck. Sadly due to the limitations imposed by the ARM 9 architecture a PC with an x86 processor was used for profiling. Almost all speedups noted on the PC were reflected on the platform. An exception was that the caches are not the same, and thus implied some small difference in the speedup. These issues where fixed by changing the size of the effected structure so that they would be aligned by cache lines which gave a speedup.

Oprofile has been an easy tool to use and only a fraction of it was evaluated in this thesis, which works on most of todays home PCs. Valgrind was not investigated much deeply because Oprofile gave satisfying results. The author lacked
the correct Lauterbach license and thus it wasn’t used for more than debugging purposes. Also Lautebach can’t give an overview of a complete programs life cycle, but only a current execution window with great details. Using an execution window would have meant that it is known where to look to optimize the code on a low level. Instead this thesis was focusing on optimizing on a high level.
Appendix A

LOSE dual vs. single core

To measure the impact on LOSE when running with either both cores enabled or just one a simple test was conducted. The test was sending/receiving $10^6$ message between two tasks, without allocating new messages but instead reusing message. Results are shown in table A.1. It is apparent how much LOSE is slowed down by running on two cores, it is more than 60% slower! The slowdown can depend on many factors, but probably it is because of that LOSE was not designed to run on multi core systems.

<table>
<thead>
<tr>
<th></th>
<th>time elapsed (seconds)</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Core</td>
<td>2.67</td>
<td>0.03</td>
</tr>
<tr>
<td>2 Core</td>
<td>7.12</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table A.1: Comparing elapsed time on dual vs single core.

Digging a little bit more before deciding on which configuration will be used to run the tests Oprofile is used to profile both cases. The following two listings shows samples for each binary.

Dual core:

```
CPU: Core 2, speed 2400.04 MHz (estimated)
Counted CPU CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 90000

CPU_CLK_UNHALTED... | samples | %|
181292  55.0402 losebinary
141463  42.9481 vmlinux
```

I wrote a script that resets the profiling buffer, start profiling, run the test program, dump data and then stop profiling.

\[\text{\footnote{I wrote a script that resets the profiling buffer, start profiling, run the test program, dump data and then stop profiling.}}\]
Single core:

CPU: Core 2, speed 2400.12 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 90000

<table>
<thead>
<tr>
<th>CPU_CLK_UNHALTED...</th>
<th>samples</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>losebinary</td>
<td>62183</td>
<td>97.1533</td>
</tr>
<tr>
<td>vmlinux</td>
<td>45611</td>
<td>73.3496</td>
</tr>
<tr>
<td>libpthread -2.6.1.so</td>
<td>9744</td>
<td>15.6699</td>
</tr>
<tr>
<td>losebinary</td>
<td>6819</td>
<td>10.9660</td>
</tr>
</tbody>
</table>

Clearly that when running in dual core the application losebinary takes less than 60% of the samples, while when in single core it takes almost 100% of the samples. Using both cores results in the kernel taking more than 40% doing other tasks outside the losebinary application. So to be able to compare with data from the target platform the tests needs to run on a single core.
Appendix B

Source Code

Source code is modified for readability purposes and all debug printouts are removed.

B.1 Futex

B.1.1 atomic.h

/* Internal macros for atomic operations for GNU C Library. 
This file is part of the GNU C Library. 
Contributed by Ulrich Drepper <drepper@redhat.com>, 2002. 

The GNU C Library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 2.1 of the License, or (at your option) any later version. 
The GNU C Library is distributed in the hope that it will be useful, 
but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. 
You should have received a copy of the GNU Lesser General Public License along with the GNU C Library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. */

#if !defined_ATOMIC_H
#define _Atomic_H 1
#endif __i386__
#define atomic_compare_and_exchange_val_acq(mem, oldval, newval) __sync_val_compare_and_swap(mem, oldval, newval)

/* Atomically store NEWVAL in MEM if MEM is equal to OLDVAL. Return zero if MEM was changed or non-zero if no exchange happened. */
#define atomic_compare_and_exchange_bool_acq(mem, newval, oldval) (!__sync_bool_compare_and_swap(mem, oldval, newval))

/* Add VALUE to MEM and return the old value of MEM. */
#define atomic_exchange_and_add(mem, value) 
{ __typeof (*mem) __atg6_oldval; 
  __typeof (mem) __atg6_memp = (mem); 
  __typeof (*mem) __atg6_value = (value); 
  do 
    __atg6_oldval = *__atg6_memp; 
  while (!__builtin_expect 
    (atomic_compare_and_exchange_bool_acq (__atg6_memp, 
      __atg6_oldval + __atg6_value, 
      ___atg6_value), 
      ...) 
  *__atg6_memp += __atg6_value; 
  return __atg6_oldval; 
} 

39
APPENDIX B. SOURCE CODE

40

/* Note that we need no lock prefix. */
#define atomic_exchange_acq(mem, newvalue) 
{
    __typeof__(*mem) result;
    if (sizeof(*mem) == 1)
        __asm__ __volatile__ ( "xchgb %b0, %1" :
            "=q" (result), "=m" (*mem) :
            "0" (newvalue), "m" (*mem) ) ;
    else if (sizeof(*mem) == 2)
        __asm__ __volatile__ ( "xchgw %w0, %1" :
            "=r" (result), "=m" (*mem) :
            "0" (newvalue), "m" (*mem) ) ;
    else if (sizeof(*mem) == 4)
        __asm__ __volatile__ ( "xchgl %0, %1" :
            "=r" (result), "=m" (*mem) :
            "0" (newvalue), "m" (*mem) ) ;
    else
    {
        result = 0;
        abort();
    } 
    result ;
}
#define __kernel_cmpxchg(t) (int oldval, int newval, int *ptr)
#define atomic_compare_and_exchange_val_acq(mem, oldval, newval)
    __kernel_cmpxchg(oldval, newval, mem)

B.1.2 my_futex.h

#define _MY_FUTEX_H
#define _MY_FUTEX_H
void futex_lock(int *futex);    
void futex_unlock(int *futex);    
#endif //_MY_FUTEX_H

B.1.3 my_futex.c

#include <linux/futex.h>
#include <syscall.h>
#include <errno.h>
#include "my_futex.h"
#include "atomic.h"
#define C(x)
#define NULL ((void*)0)
#endif

static void futex_wait(int *futex, int curr_val){
    int err;
    if((err = syscall(240, futex, FUTEX_WAIT, curr_val, NULL, NULL, 0, 0)) != 0){
APPENDIX B. SOURCE CODE

```c
err = errno;
if (err != EWOULDBLOCK) {
    perror("FUTEX_WAIT: ");
    return;
}
errno = 0;
}

static int futex_wake(int *futex, int num) {
    int num_waked = syscall(240, futex, FUTEX_WAKE, num, NULL, NULL, 0, 0);
    return num_waked;
}

void futex_lock(int *futex) {
    int c;
    #ifdef __i386
        if ((c = atomic_compare_and_exchange_val_acq(futex, 0, 1)) != 0) {
            if (c != 2) {
                c = atomic_exchange_acq(futex, 2);
            }
            while (c != 0) {
                futex_wait(futex, 2);
                c = atomic_exchange_acq(futex, 2);
            }
        }
    #elif defined(__arm__)
        if ((c = atomic_compare_and_exchange_val_acq(futex, 0, 1)) != 0) {
            while ((c = atomic_compare_and_exchange_val_acq(futex, 0, 2)) != 0) {
                if (*futex == 1) {
                    atomic_compare_and_exchange_val_acq(futex, 1, 2);
                }
                futex_wait(futex, 2);
            }
        }
    #else
        #error BAD ARCHITECTURE
    #endif
}

inline void futex_unlock(int *futex) {
    int c;
    #ifdef __i386
        if ((c = atomic_exchange_acq(futex, 0)) == 2) {
            int num = futex_wake(futex, 1);
        }
    #elif defined(__arm__)
        if ((c = atomic_compare_and_exchange_val_acq(futex, 2, 0)) == 0) {
            int num = futex_wake(futex, 1);
        } else {
            *futex = 0;
        }
    #else
        #error BAD ARCHITECTURE
    #endif
}
```
Bibliography


