OSEck core-to-core Communication Characteristics in a Multicore Environment

ERIK JOHANSSON

ROBERT ANDERSSON

Master of Science Thesis
Stockholm, Sweden 2007

KTH/ICT/ECS-2007-132

Abstract

Multicore processors are making their way into embedded systems. Effective inter-core communication mechanisms are needed to utilize the full power of these processors. To implement and optimize such mechanisms requires knowledge of how a system behaves under different configurations. This thesis investigates characteristics of various message passing implementations in multicore environments, with focus on shared memory and DMA. The research is conducted with a set of Nios II/f processors, synthesized in an FPGA, running the real-time operating system OSEck. The inter-process communication architecture LINX is used.

The first part of the thesis describes the porting of OSEck to the Nios II/f environment. This port is later used during the second part, where the performance of LINX is investigated under various premises. Test cases are constructed based on real-world use cases describing traffic patterns and environment characteristics of telecom infrastructure. Drivers are implemented for shared memory and DMA to be used in conjunction with LINX for measuring the performance in the test cases. Measurements include latency, throughput, jitter and system load.

The results indicate that both shared memory and DMA can be put to good use in conjunction with LINX. To generalize, shared memory is better for small messages, while DMA is faster and generates less processor load for large messages. Several interesting phenomena introduced by the processor caches have been encountered. General discussion regarding the impact of cache configuration in a multicore system is included.

Sammanfattning

Processorer med flera processorkärnor blir allt vanligare i inbyggda system. För att till fullo utnyttja beräkningskraften i dessa processorer krävs effektiva kommunikationslösningar. Implementering och optimering av sådana lösningar kräver kunskap om hur ett system beter sig vid olika typer av konfigurationer. Detta examensarbete undersöker olika egenskaper vid utbyte av meddelanden mellan processer i ett system med flera processorkärnor. Under arbetet syntetiseras den mjuka processorn Nios II/f i en FPGA. Realtidsoperativsystemet OSEck tillsammans med kommunikationsprotokollet LINX används.

Första delen av examensarbetet beskriver arbetet med att portera OSEck till Nios II. Porteringen används i den andra delen av arbetet, där prestandan i LINX utvärderas under olika premisser. Testfall utarbetas baserat på verkliga användarfall från telekomindustrin. Drivrutiner för kommunikation med LINX i delat minne och med DMA implementeras och utvärderas. Mätresultat som presenteras inkluderar latens, kapacitet och processorlast.

Resultaten visar att både delat minne och DMA är bra sätt att kommunicera i multiprocessormiljö med LINX. Generellt är delat minne snabbare för små datamängder, medan DMA lämpar sig bättre för större meddelanden. Flera intressanta fenomen orsakade av processorns cache-minnen observeras. En diskussion kring cache-minnen och deras betydelse för multiprocessorsystem tillhandahålles.

Acronyms and abbreviations

ACK	Acknowledgement
AMP	Asymmetric Multiprocessing
ANSI C	American National Standards Institute, C language standard
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
BSP	Board Support Package
CDA	Core Dump Analyser
CM	Connection Manager
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
D-cache	Data cache
DM	Device Memory
DMA	Direct Memory Access
DSP	Digital Signal Processor
ELF	Executable and Linkable Format
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FRAG	A header for Fragments
GCC	GNU Compiler Collection
GSM	Global System for Mobile communications
HAL	Hardware Abstraction Layer
I-cache	Instruction cache
IDE	Integrated Development Environment
IP	Intellectual Property
IPC	Inter-Process Communication
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group (Standard Test Architecture)
KTH	Kungliga Tekniska Högskolan (Royal Institute of Technology)
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LINX	Portable message passing architecture from Enea
µC/OS II	Real-Time Operating System from Micrium
MMU	Memory Management Unit
MPU	Memory Protection Unit
MTU	Maximum Transmission Unit
OpenMPI	Open Message Passing Interface
OSAL	Operating System Adaption Layer
OSE	A real-time operating system from Enea
OSEck	OSE compact kernel
OSI	Open Systems Interconnection
OSI model	Open Systems Interconnection model
PC	Personal Computer
PCI-Express	Peripheral Component Interconnect – Express
PID	Process Identifier
PTP-CM	Point-to-Point Connection Manager
RLNH	Rapid Link-Handler
RTOS	Real-Time Operating System
RX	Receive
SD-Card	Secure Digital Card
SDRAM	Synchronous Dynamic Random Access Memory
SG-DMA	Scatter-Gather Direct Memory Access
SHM	Shared Memory
SMP	Symmetric Multiprocessing

SOPC	System-on-Programmable Chip
SRAM	Static Random Access Memory
TCP/IP	Transmission Control Protocol on Internet
TI	Texas Instruments
TS	Timeout Server
TX	Transmit out
UART	Universal Asynchronous Receiver/Transmitter
UDATA	A header for User Data
UDP	User Datagram Protocol
USB	Universal Serial Bus
VGA	Video Graphics Array

Acknowledgements

This thesis would not have been possible without the contribution and support of several key figures. We would like to wholeheartedly thank:

Ola Redell, our supervisor at Enea, for his guidance and dedication to our thesis

Henrik Wallin at Enea for brilliant technical advice and insights into the world of OSEck

Graham McKenzie at Altera for great hardware support and good pointers

Dan Lilliehorn at Enea, for getting us started and setting us up at Enea

The entire OSEck team at Enea, with hangarounds, for all the good coffee breaks

We would like to extend a special thank you to Lena Engdahl at Altera for supplying us with the development boards and licenses.

Thank you!

Image rights

All images in this thesis are copyright © 2007 Robert Andersson and Erik Johansson, except the following:

Figure 2.3.2-1 and Figure 2.3.2-2, Copyright © 2007 Altera Corporation.
Used with permission.

Table of Contents

1 INTRODUCTION **1**
1.1 BACKGROUND 1
1.2 HISTORY BEHIND THE THESIS 1
1.3 THESIS OVERVIEW 2
1.4 PROBLEM DESCRIPTION 2
PHASE ONE **7**
2 OSECK ON NIOS II **9**
2.1 INTRODUCTION 9
2.2 OSECK 9
2.3 ALTERA NIOS II 13
2.4 SYSTEM DEVELOPMENT FLOW 18
2.5 DELIMITATIONS 19
2.6 IMPLEMENTATION 19
2.7 TESTING, VALIDATION AND PERFORMANCE 21
2.8 CONCLUSIONS AND FUTURE WORK 22
PHASE TWO **25**
3 MULTICORE COMMUNICATION CHARACTERISTICS **27**
3.1 INTRODUCTION 27
3.2 CACHES 27
3.3 LINX 29
3.4 USE CASES 33
3.5 TEST CASES 34
3.6 MULTICORE DRIVER IMPLEMENTATIONS 37
3.7 METHOD 43
4 RESULTS **49**
4.1 OVERVIEW 49
4.2 T0 - LATENCY 49
4.3 T1 - INTERFERENCE 53
4.4 T2 – BURSTY TRAFFIC 55
4.5 T3 - THROUGHPUT 60
4.6 T4 - ARBITRATION 62
4.7 T5, T6 – MULTIPLE UNITS 63
5 DISCUSSION **65**
5.1 HARDWARE IMPACT 65
5.2 CONNECTION MANAGERS (CM) 66
5.3 DRIVERS 66
5.4 COMPARISON OF SHARED MEMORY AND DMA 67
6 CONCLUSIONS AND FUTURE WORK **71**
6.1 CONCLUSIONS – PHASE TWO 71
6.2 FUTURE WORK – PHASE TWO 71
7 REFERENCES **73**
APPENDIX A **76**
APPENDIX B **78**
APPENDIX C **85**

1 INTRODUCTION

1.1 Background

Multicore processors, with two or more computing units, are becoming increasingly popular in a wide range of systems stretching all the way from home desktop computers to small embedded devices. There are several factors explaining this development. For a long time the focus in silicon industry lay on raising clock speeds on single core chips to gain computation power resulting in massively increased power consumption and heat generation. This can in part be solved by making the processor physically smaller using new gate technology or by reducing functionality, but a more practical solution is multicore processors where two or more processor cores are placed closely together in the same integrated circuit die. Going from one core to two intuitively doubles the computation power while preserving clock speed.

Multicore processors are not without disadvantages, however. Software developers must adapt to the new situation and take advantage of the parallelism provided in order to utilize the processor to its full potential, typically by assigning tasks to different cores. A growing demand for fast and reliable inter-processor communication mechanisms for applications is a natural consequence. Two processes (tasks) wishing to communicate with each other could reside on the same core, on different cores in the same chip or in completely different chips, perhaps significantly separated. Transferring messages efficiently between processes in these systems is a challenge.

Ultimately, this thesis is about investigating, evaluating and implementing different approaches to communication between cores in a multicore system and analysing how various parameters, in both software and hardware, affect the performance of these approaches.

1.2 History behind the Thesis

It has been concluded at Enea that the OSEck real-time operating system would benefit from a better understanding of performance characteristics and environment requirements of different core-to-core communication and synchronization mechanisms. Focus should be on use cases relevant for OSEck (mainly used in telecom industry). To be able to investigate these mechanisms under different hardware settings a soft microprocessor solution with an FPGA is ideal. This allows the hardware to be modified and redesigned with the help of software tools and then rather easily downloaded onto the FPGA chip. One such FPGA solution is the Nios II microprocessor from Altera. It is in the interest of Altera to widen the number of operating systems supporting the Nios II microprocessor, and with customers specifically asking for OSE support there was a win-win situation at hand. Both companies would gain from having OSE supporting the Nios II architecture.

At the Royal Institute of Technology (KTH) in Stockholm the Nios II architecture is used in several courses teaching embedded systems engineering. Large parts of these courses centre on a real-time operating system (RTOS) running on the Nios II. With OSE being used to a high degree by the telecom industry, having the students learning OSE would fit well with wishes from the industry, not least from Enea where a widened OSE knowledge by newly examined engineers is desired.

As the circle closes it is clear that all three parties, Enea, Altera and KTH, stand to gain from the cooperation that this project and thesis represents.

1.3 Thesis Overview

The thesis is written by Erik Johansson and Robert Andersson and consists of two parts. The first part was done in collaboration and describes the work of porting the real-time operating system OSEck to the soft microprocessor Nios II/f later used in our research. The second part has been written both in collaboration and separately, where Erik has written the sections about message passing using DMA and Robert the sections about shared memory message passing.

1.4 Problem Description

As mentioned earlier, this thesis is about investigating and implementing different approaches to communication between cores in a multicore processor, but communication can mean a lot of things. For example, synchronization can be viewed as a form of communication where a core communicates that it has reached a certain point in the code. Information can be obtained without actually sharing any data.

Data sharing is another form and can be achieved in lots of different ways. The research in this thesis will be based on OSEck, an operating system that uses message passing as its primary communication mechanism. We therefore present the challenges of that particular mechanism on different media in this problem description. The number of ways to implement message passing are numerous and involves both hardware and software solutions.

Message passing is a widespread approach to inter-process communication in distributed systems. One common open source implementation is OpenMPI [16]. In a large context, looking from an application programmer's point of view, message passing typically involves transferring packets of data from one process' address space to that of another, using primitives like *send* and *receive*. Lower level implementation details are usually not of concern.

A message passing implementation is not necessarily a part of the OS, but could be a third party library that implements message passing on top of existing technologies and protocols, such as TCP/IP. Architecturally, a message passing system is usually designed as a protocol stack with similarities to the OSI reference model [1] shown in table 1.4.1.

Data Unit	OSI Layers
Data	Application
Data	Presentation
Data	Session
Segments	Transport
Packets	Network
Frames	Link
Bits	Physical

Table 1.4.1: *The OSI reference model*

At the bottom of the stack is the physical layer, which is the actual transfer media. Common media in telecom industry are RapidIO, Ethernet and PCI-Express. For example, a GSM basestation often consists of racks with mounted boards interconnected with fast connections utilizing either of these technologies.

Higher layers implement protocols controlling addressing, reliability, etc. and can be arbitrarily complex. How to design protocol stacks is not an objective of this thesis, even though this will be reflected upon in the second part of the report. We will focus on lower layers and drivers controlling them.

A multicore system has the advantage of close coupling and should try to limit the overhead of data processing to avoid performance penalties. Having two processes on different cores in a multicore processor communicate via e.g. Ethernet is not a good idea, regardless of link capacity. Instead, the message passing system should consider using shared memory, DMA-transfers or custom hardware specifically optimized for such data transfers. One example of such hardware is a core-to-core FIFO buffer. Some of the challenges one may encounter with each of these transfer methods are presented below.

1.4.1 Shared Memory Message Passing

In the best case scenario, a shared memory is a memory to which all cores have read and write access, but other variants exist. For example, in some quad-core systems, three cores are only allowed to read/write their own private memories, while the fourth core can read/write all memories. The discussion below assumes no specific configuration.

In the shared memory case, the core wishing to send a message allocates a buffer in a memory and fills it with data. The core then notifies the receiving core, e.g. by issuing an interrupt, that a message is available and possibly where it can be located. The receiving core fetches the data, interprets it and deallocates the buffer or reuses the buffer to send a response if possible. A multitude of parameters, soon to be described, affect the performance of shared memory message passing.

Only the mechanism to transfer data between potential shared memory drivers will be considered and not how the data is actually transferred between kernel processes. The information needed to find recipient processes is embedded within the data and is managed by higher layers of the protocol stack.

The first problem that needs to be addressed is how each driver allocates memory that can be shared among cores. Assuming that each core's software is linked to different memory regions, an application programmer doing *alloc*[1] will most likely allocate a buffer in private memory. The first solution that comes to mind is having a shared memory pool from which all cores allocate memory for individual messages. If the programmer knows in advance that the message will be transmitted to a remote core, different *alloc*:s can be used. The shared memory pool could of course be used for local messages as well, but that might be ineffective due to critical sections (see below). Instead, keeping private allocations and letting the driver allocate shared memory and copy the data to the shared buffer and back to private memory is one possibility. Memory copies, however, could be slow if the messages are large. There is no simple solution.

Accesses to shared data might require a synchronization mechanism that prevents cores from altering data in administrative memory regions simultaneously. This can be done with hardware mutexes or semaphores, but spinlocks, non-blocking and lock-free mechanisms are also possible. If the critical section is too large, performance will suffer in heavy loaded systems and cores could potentially be starved. Critical sections should be avoided or at least kept as small as possible. Splitting them into smaller pieces is another option.

It is also important that newly made changes to shared data is immediately visible to other cores. In the case where all cores lack a memory data cache this is not a problem. In systems with caches that lack cache coherence protocols however, the caches need to be explicitly flushed before a core releases its lock of the pool. If the entire cache needs to be

---

[1] A system call used to allocate memory buffers.

flushed a significant burst of data on the bus would occur delaying other bus activity for a short period of time. This phenomenon is commonly known as bus congestion and should be avoided. Some processors implement instructions to flush individual cache lines, which could reduce this effect. Another approach that might be better, if supported by hardware, is to bypass the cache and read/write data directly to the memory, thus getting a limited impact on the bus.

If the system implements a cache coherence protocol then no special attention is needed by the programmer to guarantee data correctness. Though, the protocol might influence bus performance in ways difficult to predict and could be interesting to investigate.

Once a buffer is allocated, data needs to be written into it by the software. Again, caches play an important role in ways similar to what is mentioned above, but this time no locking is needed since the sender is the only owner of the buffer and no one else should touch it.

The transmission is done either by copying all data into a buffer owned by the receiver, which would take some time, or by just passing the receiver a pointer to the buffer. Either way, the sender and receiver must determine how to share pointers to buffers with each other in advance.

The sender needs to inform the receiving end that a message is waiting. Typically this would be done by triggering an interrupt request. Considering different hardware configurations and the availability of IRQ lines, that might not be possible. In such a case the drivers must use another approach, for instance polling a byte in memory. When notified, the receiver collects the message from memory and returns the memory buffer to the shared pool by locking administrative fields and placing the buffer in a free list similar to allocation but reverse.

Frequency and size of data messages is also an important aspect influencing bus performance. Different sizes should be evaluated against different number of packets.

1.4.2 Message Passing using DMA

DMA stands for Direct Memory Access, implying that the memory is somehow accessed directly by a device other than the central processing unit. One typical application of DMA is a storage device connected to the bus. Instead of having the processor constantly reading data from the device into memory, the device stores the data directly into memory, leaving the processor free to focus on other tasks.

In the case of message passing, DMA could be used to transfer the message from one part of memory (belonging to the sending core) to another part (belonging to the receiver). To accommodate this, the system needs to be equipped with certain hardware connected to the memory, namely a DMA controller. A typical DMA controller allows continuous copying of one address range to another. A single controller can, depending on the hardware implementation, have any number of channels allowing simultaneous copying of different address ranges. Some DMA controllers only have a single channel.

A core wishing to send a message would just have to set up the DMA controller to copy the message into memory belonging to the receiving core. When the DMA controller is finished it will assert this by issuing an IRQ to its associated core(s). We will need to look into how many DMA channels should be used, and if the controller should be able to interrupt more than one core. It would be preferable if the DMA controller could interrupt the receiving core, as this would allow for easy synchronization and would probably also have a positive impact on performance. The receiving core would then put the message into the message queue of the receiving process.

If the memory used for passing messages is cached by the data caches then both processors need to flush the caches, or at least the concerned cache lines. The receiver might only have to invalidate the concerned lines, presuming the cache hardware has support for this. If

there is a cache coherence protocol the flushing can be skipped, as the hardware will automatically handle data integrity. It could prove critical or at least beneficial to align the messages in memory with the cache lines. The same problem with bus congestion as for the shared memory model applies when flushing caches.

The sender will need to get hold of a pointer to an appropriate part in the receiver's memory before initiating the transfer. How this is to be done remains to be investigated and could prove to be tough implementing in an uncomplicated way. Possible solutions include some sort of handshake procedure with the receiver placing the appropriate address in the destination register of the controller, or always using the same memory range at the receiving end. The last approach would surely lead to extra copying having a negative impact on performance. A third possibility would be to use dedicated communication hardware (mailboxes) for exchange of small, pointer-sized messages.

In the case of having just a single DMA controller it might, depending on the hardware design, have to interrupt all cores. Each time a core is interrupted the context switch routine is initiated, resulting in a lot of memory traffic due to the pushing of registers on the stack. When many cores are sharing the bus this would lead to a lot of simultaneous bus traffic, resulting in bus congestion. This might be avoided with a clever interrupt service routine able to dismiss IRQs not directed to the current kernel. Cache configurations will also affect the amount of bus traffic and is a good topic for analysis. As a shared resource the DMA controller would need to be protected by a hardware mutex. With clever assignments of channels to particular cores, the need for a hardware mutex could be avoided.

An alternative approach is having one DMA controller per processor core. The controller could be either at the sending or at the receiving end. If the controller is at the receiving end, a core wishing to initiate a send would wait on a hardware mutex guarding the receivers DMA controller. When the controller is free, the sender sets up the memory transfer and starts it. When the transfer is finished the DMA controller would then just interrupt its own core. Whether this introduction of several locks (mutexes) could lead to a deadlock remains to be investigated properly, as well as race conditions, starvation and any unwanted behavior resulting from the parallelism. A good design is a simple design, and this approach might prove too complicated, especially for systems with more than two cores. Ease of hardware design is an issue, as well as the chip surface. With the controller being "owned" by the sending side the need for a hardware mutex might be unnecessary, and the resource could be protected by a semaphore in software. One negative aspect to this is that the sender needs to notify the receiver when the transfer is finished, and this would then have to be done through some other mechanism than an interrupt from the controller. Sending a regular interrupt from the sender to the receiver would be one solution.

DMA Chaining and Scatter-Gather

Advanced DMA controllers have support for carrying out a number of transfers consecutively. A list with all the settings describing each transfer is set up in memory by the processor and then the controller is kicked into action. The controller handles one transfer after the other, copying memory from and to several locations. When all the transfers are finished, the controller interrupts the processor announcing the completion of the job. This is called DMA chaining, as several transfers are chained together. One special case of chaining is when a large continuous block of memory is scattered into several smaller blocks. The reverse operation is also possible, gathering several smaller blocks to form a larger. A controller capable of chaining is sometimes referred to as a scatter-gather DMA controller.

1.4.3 Shared Memory and DMA Summary

To sum up, a lot of challenges have been briefly presented and there is no simple path through all the permutations of possible solutions. The key areas needed to be further investigated are:

- Memory allocation
- Inter-core synchronization
- Interrupt handling
- Caches and data correctness
- Bus impact
- DMA chaining
- Message sizes, frequencies and traffic patterns

1.4.4 Custom Hardware

Some suggestions have been made on how to implement message passing in hardware.

A possible implementation is a system in which each core gets a custom FIFO-queue and two hardware semaphores. Decreasing a semaphore with a value of zero will stall the core trying the operation. The first semaphore is initialized to the number of buffers available in the FIFO-queue and the second semaphore should indicate the number of buffers used. The idea is for senders to decrease, and wait if necessary on, the semaphore indicating free buffers. When a buffer is available the message is placed in the receiver's FIFO-queue. The receiver is notified by the sender who increments the hardware semaphore indicating used number of buffers. The receiver wakes up, reads the message and when it is ready to receive more messages it decreases the same semaphore, while increasing the semaphore for free buffers. A sender waiting for a buffer wakes up and can proceed with the delivery. This hardware implementation should only be used in one-way communication since deadlocks are otherwise possible.

In the course of adding more processor cores to chips, network-on-chip researchers are investigating different network topologies. Inter-core connections can be arranged as rings, such as the *Synergistic Processing Elements* [14] in the *CELL*-processor, as meshes, as a torus, etc. These topologies require some kind of routing if cores that are not neighbours wish to communicate. A neighbouring core, or several, would need to act as a relay, forwarding packets in the right direction. Network-on-chips is a large research area, including many problems. Except for routing, there is also the issue of flow control and deadlock avoidance among other things. Crossbar switches on the other hand do not have these problems, but are expensive to integrate on chips.

For a good view on different interconnection network topologies, see [15].

PHASE ONE

2 OSEck on Nios II

2.1 Introduction

The first phase of this master thesis begins by presenting the real-time operating system OSEck and its components as well as the target architecture, the Nios II microprocessor. We later describe the work of porting OSEck to Nios II, the delimitations we made and how that affects future work on the platform and in our thesis.

2.2 OSEck

OSEck (OSE compact kernel) is a fixed priority preemptive real-time operating system mainly used in telecom industry. It is optimized for high performance digital signal processors (DSP), with low latency demands, therefore lacking some of the features of OSE 5, its heavyweight sibling.

2.2.1 Signals

The key concept of the OSE-family is called signals and signalling [8, section 2.2.1]. A signal is a message sent from one process to another. Unlike UNIX-signals, OSE signals carry a data payload. See figure 2.2.1 for an illustration of a signal. Before sending a signal, the user must allocate a buffer for the signal from a memory pool. It is also possible to reuse the buffer from a received signal if the buffer size is large enough. The first location of the buffer contains the signal number followed immediately by any user defined data. The sender issues the system call *send* to send the signal, providing the address of the signal and the process identity of its receiver as parameters.

Send example

```
union SIGNAL *sig;
sig = alloc(sizeof(struct start_motor_t), START_MOTOR);
send(&sig, motor_pid);
```

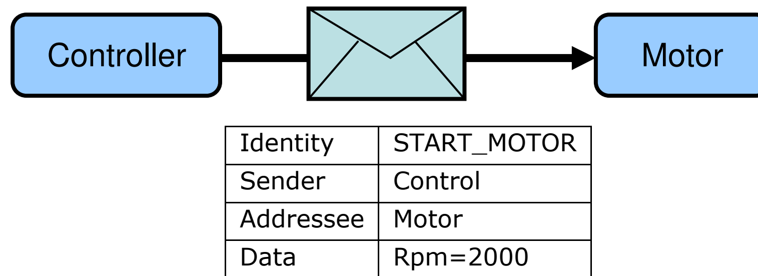
The receiver in turn waits for and receives signals by using the system call *receive*. The receiver can selectively choose what signal numbers it is interested in by declaring an array of the datatype *SIGSELECT*. If the signal queue does not hold a wanted signal, *receive* will block turning CPU-time over to a ready process. The receiving process remains blocked until a signal with the specified signal number(s) becomes available (unless *receive-with-timeout* is used).

Receive example

```
union SIGNAL *sig;
SIGSELECT any_sig[] = {0};
SIGSELECT motor_sig = {2, START_MOTOR, STOP_MOTOR};

sig = receive(motor_sig);
```

Signal queues cannot be explicitly created and each process only has one.



Identity	START_MOTOR
Sender	Control
Addressee	Motor
Data	Rpm=2000

Figure 2.2.1: *An illustration of a signal sent between two processes.*

Two types of processes exist, priority processes and interrupt processes. There are 32 priority levels per process type. It is possible to have several processes with equal priority, in which case the processes are executed in the order they get ready. Priorities are preconfigured, but the priority of priority processes can be explicitly changed at runtime if needed.

2.2.2.1 Priority Processes

Priority processes are regular processes and are used to implement application tasks. They execute as long as no higher priority process is ready to execute or they get blocked. They can use all system calls.

2.2.2.2 Interrupt Processes

Interrupt processes are used to handle external interrupt requests and can only be preempted by higher priority interrupt processes (interrupt nesting). They can only issue non-blocking system calls, such as sending signals to other processes. Interrupt processes may be created without the existence of an external interrupt source, which is beneficial when testing hardware drivers or when no hardware exists. An interrupt process becomes ready and is executed immediately if a signal is sent to it or if another process triggers its *fast semaphore*, described below.

2.2.3 Semaphores

Enea encourages the use of signals rather than semaphores for synchronization. However, there are two types of semaphores, both behaving as semaphores commonly used in other operating systems.

2.2.3.1 Fast Semaphore

Each process, both priority processes and interrupt processes, has a fast semaphore attached to itself. The fast semaphores are usually used for process synchronization since they are faster than signals. They are however not as powerful as signals since they lack the ability to carry data and can not be distributed.

An interrupt process is not allowed to wait for semaphores. Instead, signalling a fast semaphore owned by an interrupt process will cause a software event and always change the process' state to ready whereby the current running process will be preempted. This feature is useful for debugging interrupt processes. Triggering a fast semaphore owned by a higher priority priority process in waiting state will also preempt the current process.

2.2.3.2 Regular Semaphore

A regular semaphore has no connection to a specific process and can be explicitly created.

2.2.4 Memory management

2.2.4.1 Protected and Virtual Memory

OSEck does not support protected or virtual memory as OSE does, which means that all processes share the same global address space and could potentially corrupt each-others data. To facilitate these features there is a need of special hardware in the form of a memory protection or management unit (MPU, MMU). This hardware has historically been rare on DSPs and microcontrollers that OSEck is designed for and customer demand has been low.

2.2.4.2 Memory Allocation

The standard way of allocating memory in OSEck is by using the system call *alloc*, which reserves a signal buffer for the calling process. To free previously allocated memory *free_buf* is used. These functions are part of the kernel and make it easy to handle memory allocation for signals, but they can also be used to allocate memory for other purposes. Besides a size parameter, the *alloc*-function also expects a signal number, which is then automatically put in the start of the buffer. This can be ignored if the allocated memory is to be used for something else than a signal.

OSEck uses memory pools to handle allocations. Every system has at least one memory pool, known as the system pool. The *alloc* system call always allocates memory from the system pool. To allocate memory from another pool, the *s_alloc* is used, passing the chosen pool ID as a parameter. Each pool has its individual settings concerning the sizes of allocatable units (the buffers), with 8 different sizes allowed. These settings are fixed and defined when the pool is created. When allocating memory from a pool, the process receives a block of memory matching one of these predefined sizes. If the requested size doesn't exactly match one of the predefined sizes a larger block is reserved. For example, *alloc(7)* will allocate a buffer with size 16 if a block size of 16 has been configured and is the smallest block size that would fit a buffer of seven bytes. This can lead to unwanted internal fragmentation, as more memory than necessary is reserved. To avoid this it is important to assign buffer sizes carefully, trying to match the memory requirements of the application. One advantage of the approach with fixed buffer sizes is that external fragmentation is avoided. Another benefit is that the need for advanced algorithms, to choose where in the list of free memory to place the block, is eliminated — the system calls are fast and deterministic.

2.2.4.3 Heap Component

The Heap component implements an extra set of functions to handle a complementary model for memory allocation. The interface has a number of functions to handle memory on a heap, just like the ones originally defined by Kernighan and Ritchie for the original C programming language [17]. Using these familiar functions, like *malloc* and *free*, it is easy for programmers new to OSE to get started writing application programs, as well as porting code from other systems. For applications that allocate memory in a large number of different sizes the heap may also result in reduced fragmentation, thereby conserving memory resources. As the heap splits and merges blocks as necessary to limit external fragmentation, it is consequently somewhat slower and less deterministic than memory pool allocations. The heap cannot be used to allocate signal buffers, since signals carry additional

information added by *alloc*. Also, the standard *free_buf*-function would not know how to free a block of memory allocated on the heap.

2.2.4.4 Device Driver Memory

Device driver memory is allocated using an allocator specifically designed for device drivers and their clients. It only allocates one specific size, typically the size of the maximum transmission unit (MTU). The MTU is the size of the largest packet that can be sent in a single transmission. Larger messages need to be split into fragments.

The allocator is configured with additional information not needed by a regular memory pool, such as cache line size and alignment requirements. It is used by message passing drivers developed for the second part of this thesis. Buffers allocated from the device memory pool are referred to as DM-buffers.

2.2.5 Other Components

2.2.5.1 Timeout Server

The Timeout Server provides functionality to register timeouts (alarms) at specific points in time [9]. When a timeout occurs, a user defined callback function is called. This is a good way to set up periodic tasks in the system, but there is also the possibility of having one-shot timeouts. To be able to use the TS component in an application the programmer must have a driver for a hardware timer. The driver then needs to be registered with the Timeout Server using a special API defined in the TS component.

2.2.6 OSEck and Multicore Systems

OSEck supports asymmetric multiprocessing (AMP), i.e. one instance of the kernel runs on each core without co-operation and process migration. This means that the system designer needs to assign processes to cores in advance. This is typical in a real-time operating system where deterministic execution is of great importance. Migration of a process from one core to another could possibly cause the process and others to miss their deadlines. A feasible schedule may no longer be feasible when the set of processes scheduled on a core changes. Figure 2.2.6 shows an example of an asymmetric system, with three regular OSEck kernels running on separate CPUs.

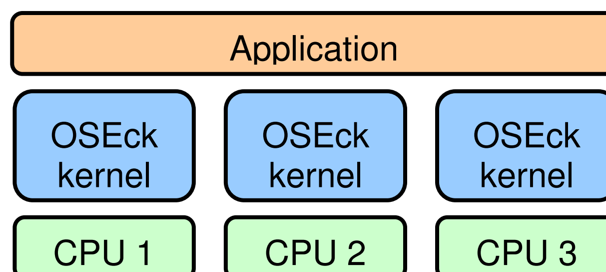


Figure 2.2.6: *An asymmetric multiprocessor system.*

2.2.7 Libraries

Other developers of RTOS:es have chosen to deliver source code to customers. OSEck is distributed as precompiled libraries with binary files. When the end user builds his application the linker uses the libraries to create the final program file. To allow different levels of debugging the kernel is distributed in three versions [7, section 4.3]:

- libos
- libosh
- libosbpsh

The first type, libos, is the standard kernel with no extra debugging features. The next type of kernel, libosh, has the added feature of hooks. A hook is a routine that is called by the kernel in conjunction with certain system calls or events. A swap hook, for example, is called every time a context switch occurs. Finally, libosbpsh is a kernel with buffer checks, precondition checks, stack checks and hooks. Buffer checks and stack checks watch the buffers and the stack for overflow. Precondition checks looks at input parameters to system calls and check if they are within certain bounds. The addition of hooks and checks makes the system somewhat slower.

A developer could start developing an application using libosbpsh and use the added features to simplify debugging. Nearing the end of development when performance may be an issue, it could be logical to make a switch to libos.

2.3 Altera Nios II

Altera Nios II is a 32-bit microprocessor for embedded systems. The processor is a soft processor meaning that it is not fixed in silicon. It is normally implemented in field programmable gate arrays (FPGA) allowing it to be customized in various ways, described later, and then downloaded to the FPGA. Using vendor-supplied tools, the system designer can implement single- and multicore Nios II hardware that interfaces any peripherals of choice.

2.3.1 Nios II/f

There are three different variants of the processor, Nios II/e (Economy), Nios II/s (Standard) and Nios II/f (Fast) [3, chapter 5]. We have chosen to use Nios II/f for our research since it allows the data cache to be bypassed in software.

Nios II/f has a 32-bit instruction set, data path and address space, 32 general purpose registers and 32 external interrupt sources. The pipeline consists of six stages — fetch, decode, execute, memory, align and writeback.

Nios II/f can address 2 GB of physical memory, although pointers are 32-bit. Bit 31 is used as *cache bypass*-bit allowing software to read and write memory without cache interference. There is no cache coherence protocol, which means that cores in a multicore system may get an inconsistent view of memory. Using the bit-31 cache bypass feature is therefore convenient when reading and writing limited amounts of cache-sensitive data.

There are two separate L1-caches for instructions and data. They can be configured in sizes stretching from 0 to 64 kB. Cache line sizes may be configured as well, 4 to 32 bytes. No managed tightly coupled L2-cache is available.

The user can add single-cycle and multi-cycle custom instructions to a Nios II core through an opcode extension mechanism. The instructions are available to assembler code as well as C source code using macros generated by special tools.

2.3.2 The System Interconnect Fabric

A Nios II system doesn't have a conventional bus. The *System Interconnect Fabric* [6] is a generated set of connections based on what peripheral hardware different masters, such as Nios II-cores, can control. The construction allows multiple masters to talk to different slaves simultaneously without interference from each other.

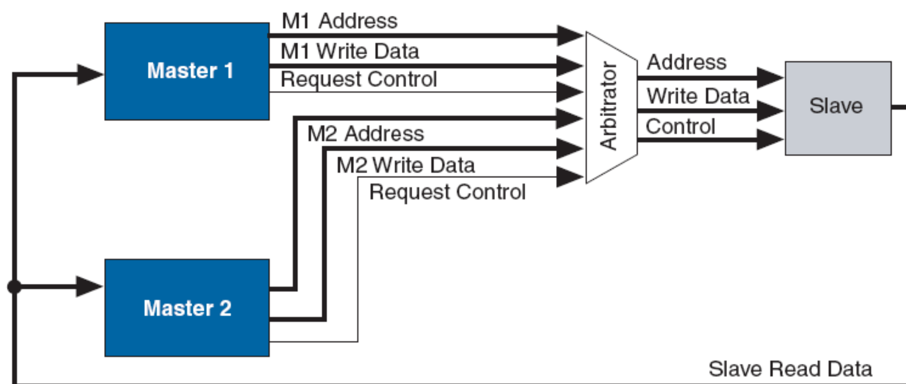


Figure 2.3.2-1: *Two masters and one slave connected by the System Interconnect Fabric.*

Figure 2.3.2-1 shows how two masters, possibly Nios II-cores, are connected to a single slave. When the masters issue a request to write data to the slave it is up to the arbitrator to select which one that can proceed. This is similar to how a conventional bus works. An arbitrator is generated for each slave port in the system and they can be configured to favor a specific master by applying weights. The arbitration algorithm is guaranteed to be fair and no master can be starved due to contention.

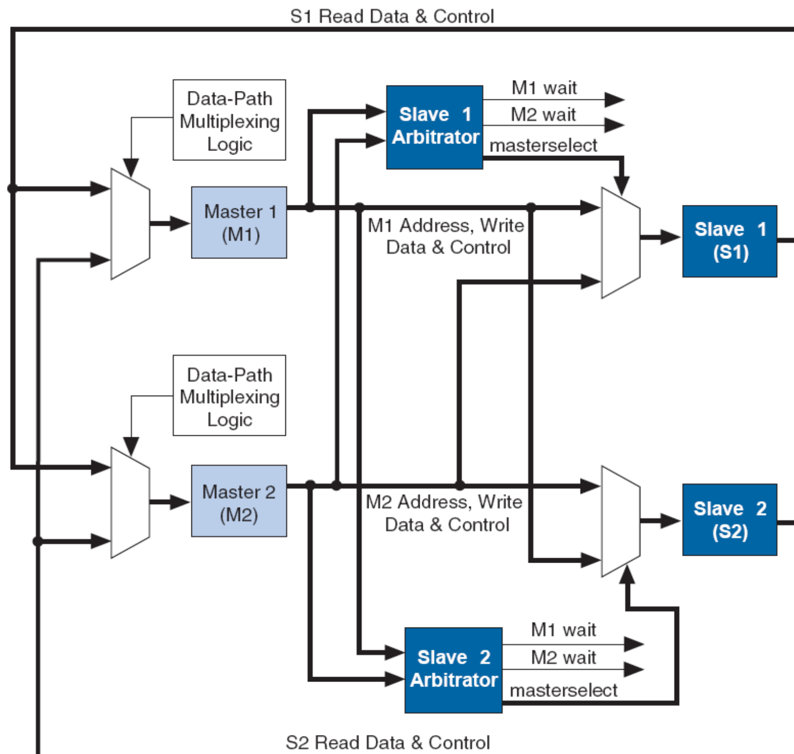


Figure 2.3.2-2: *Two masters and two slaves connected by the System Interconnect Fabric.*

Figure 2.3.2-2 shows how this slave-side arbitration works in a system with multiple masters and slaves - two masters are connected to two slaves. In this case *Master 1* can access *Slave 1* and *Master 2* can access *Slave 2* simultaneously without contention. However, if both masters try to access the same slave, an arbitration will take place and one of the masters will be stalled until it is granted access when the other master has finished the transfer. A master who initiates a burst transfer is granted access for the duration of the burst and again blocks access to the slave.

The System Interconnect Fabric also controls interrupt requests and maps slave IRQs to different master IRQ-numbers. For example, the *Slave 1*-peripheral may generate a *data ready* interrupt request with number 0 to *Master 1* and number 7 to *Master 2*.

The flexible nature of a Nios II system makes it particularly suitable for this master thesis where testing and evaluating different aspects of hardware is one of the objectives.

2.3.3 Altera Software

2.3.3.1 Quartus II

Altera Quartus II is a hardware synthesis tool for FPGAs and other customizable integrated circuits like CPLDs. It supports a complete design flow, including: writing code in a hardware description language (HDL), simulation, analysis & synthesis, place & route, timing & power analysis and finally outputting files that can be used to program the device. For normal use you do not have to care about all these steps, you can make your top level design and then click on the compile button. Using existing IP-blocks (Intellectual Property) from various vendors, a very complicated system can be fairly easily put together. With help from the *MegaWizard Plug-In Manager* these building blocks can be customized and instantiated in the design, resulting in a number of components, or "little black boxes", which are connected together at the top level. One tool for creating such components is *SOPC Builder*.

2.3.3.2 SOPC Builder

The System On Programmable Chip (SOPC) Builder allows straightforward implementation of an embedded system within a Quartus design. A complete system with CPU, memory, timers and interfaces to peripherals can be created with a few clicks. When setting up a system with a Nios II core, a memory and some peripherals, SOPC builder will automatically generate a System Interconnect Fabric that connects all the components. In figure 2.3.3.2, such a system is shown. This particular system has two Nios II cores, one timer per core, an SDRAM controller, a serial port, interfaces for buttons and LEDs, and finally a JTAG UART connection for each core. A system like this can be set up in 5 minutes, without the designer having to worry about the bus interconnections. Details about arbitration, wait states or even address ranges are effectively handled by the tool. When adding a new component, SOPC builder tries to automatically connect it to other bus masters and slaves. The designer can then modify this interconnection by clicking on a connection grid, adding or removing connections. Components that can trigger an interrupt are, in a similar way, connected to one or more masters, for example a CPU. The tool will also help assigning appropriate IRQ numbers, thereby avoiding conflicts.

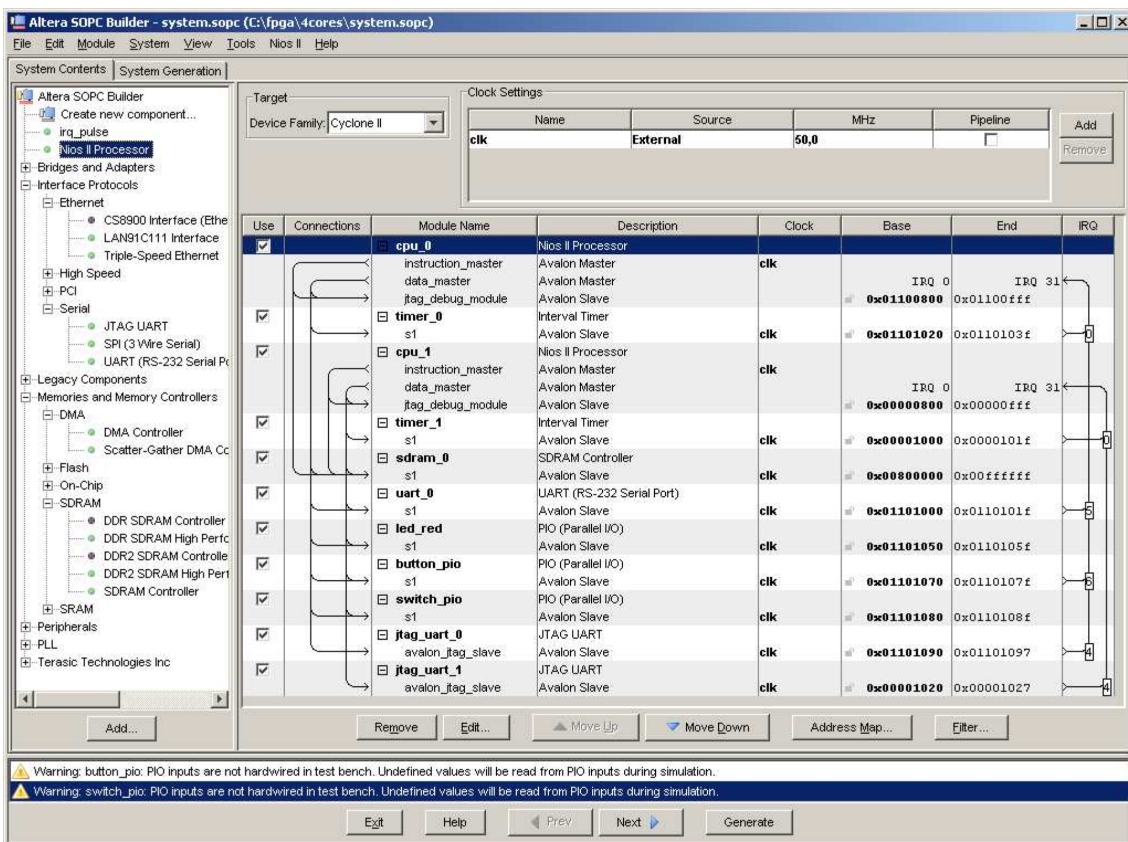


Figure 2.3.3.2*: A screen dump of an SOPC builder system containing two Nios II CPU cores*

SOPC builder can handle a wide range of components. Besides Nios II, there is support for other microprocessors like ARM or older 8 and 16 bit cores like 6811 or 8051[1]. On-chip memory can be instantiated in the tool and connected directly to the interconnect fabric. External memories can be accessed through the use of an interface that appears as a slave to the interconnect fabric. This is also true for other external components that, thanks to a number of ready-made interfaces, can be connected to the system.

[1] 6811 and 8051 are old 8-bit microcontrollers from Motorola and Intel, respectively.

Adding custom components that have been written in an HDL is also possible. With the help of a guide you start by specifying the HDL files. The next step is to map inputs and outputs to the appropriate signals, thereby creating a slave interface. Timing details about the interface can then be entered, defining setup time, wait states etc. Finally the component is given a name and is then available for use in future designs.

When a system design is finished in SOPC builder the system is generated and automatically imported into Quartus II. SOPC Builder creates a large number of HDL files describing the components in the system, including testbenches for simulation. The result is displayed as a system symbol in the top layer of the Quartus design, ready to be connected to other components within the FPGA. Input and output ports from the system's interfaces need to be mapped to the appropriate pins on the FPGA.

2.2.3.3 Other Tools

Altera ships an easy-to-use development tool called the *Nios II Integrated Development Environment (IDE).* This tool is based on the Eclipse IDE [18]. The IDE allows for easy application development, with a graphical user interface that makes it easy to get started programming using point-and-click. The IDE uses GCC [21] and related binary utilities, with a Nios II backend.

Several other command line tools are provided, such as tools for downloading software to a specific core, a terminal program for reading and writing data to the JTAG UART connection, etc.

2.3.4 Altera Nios II HAL

Hardware details are hidden from the application through an automatically generated Hardware Abstraction Layer (HAL). It provides drivers for almost all Nios II peripherals instantiated in the system and they are tightly coupled to the Newlib C-library, which allows the software developer to use standard C library functions like *fopen, fwrite, fread,* etc, to access hardware in a uniform manner.

The HAL is also responsible for initializing the system at reset. It invalidates caches and initializes drivers, and finally passes the control to *main*.

The HAL is explicitly generated using scripts or implicitly by using the Eclipse-based IDE. s*ystem.h* is one of the autogenerated files. It defines a large set of macros describing peripheral hardware. Generated macros help the user find a peripheral's memory base address, IRQ-number, virtual filesystem name, among other things. Some of the macros define global system settings, such as which timer to use as the system clock timer, standard input/output files, etc.

2.3.5 DE2 Development Board from Terasic

The development board we are using is called DE2 and is manufactured by Terasic[1]. It has one FPGA, a Cyclone II from Altera [4]. The board is also equipped with an, for our research, abundant amount of peripheral hardware devices:

[1] Two boards were kindly supplied by Altera.

- 16x2 character LCD
- Buttons and switches
- LEDs
- JTAG-connection
- RS-232 UART
- 100Mbit Ethernet
- VGA
- Composit video
- Audio
- USB
- 512 KiByte SRAM
- 8 MiByte SDRAM
- 4 MiByte Flash Memory
- SD Card Reader

2.4 System development flow

It can be difficult, with the many software and hardware tools mentioned above, to get a good overview of how system development is generally carried out. To help the reader, here is a short description explaining all steps from start to finish.

2.4.1 Hardware

The hardware project is created in and organized through Quartus, the synthesis tool. From Quartus the user launches SOPC-builder which is an HDL-generation tool. The user creates the system, adds processors, memories and peripherals, and customizes memory addresses and IRQ-numbers. When the system is satisfactory it is generated. The generated HDL-code is implicitly imported into Quartus. The user will have to compile the code and assign FPGA pins to HDL-ports so that external hardware, such as memories, can be accessed by the hardware implemented in the FPGA. An SRAM object file is created and can be downloaded to the FPGA using Quartus once the compilation has succeeded.

2.4.2 Software

Since the hardware is of a dynamic nature, a lot of information is needed by software to make it execute properly. This information is obtained by running Altera BSP-tools [2, chapter 3]. The tools need the SOPC hardware description file and generate standard C source code and headers for all the hardware in the system, also known as the Altera HAL. The HAL is compiled into a static library which can be linked with the developer's application, either using a custom Makefile or one generated with Altera tools.

2.4.3 Execution

The last step is to run the application on the hardware. Each Nios II core has a JTAG chain which can be used to halt the core, download an ELF-file to memory, and restart the core at its reset address. Altera provides an application that performs these tasks, nios2-download. The system is complete.

2.5 Delimitations

It has been established that the project would use OSEck running on a Nios II. There was no previous support for OSEck on Nios II and we had to start with making a port. Porting an operating system to a new hardware platform is non-trivial and can be quite a lot of work. What can be even more time consuming is the need for drivers and a board support package. To make it feasible to finish the port within the first half of this project the following scope and delimitations were defined.

To be included

- A complete and fully functional kernel running on the Nios II/f with hardware division
- A basic board support package with drivers for the most critical hardware
  - Kernel timer driver
  - Timeout Server-driver
  - JTAG UART-driver
- Rudimentary build tools to allow rapid application development

To be left out

- Drivers for complicated hardware like Ethernet
- Support for all Nios II variants

2.6 Implementation

2.6.1 The kernel

Most of the OSEck-kernel is written in generic C code, which is independent of the target architecture. The parts that had to be rewritten were target specific, such as interrupt handling and context switching. This was mostly written in assembler, since exact control over register usage is required.

The key challenge was to design the context saving and restoring in such a way that it allowed for fast yet reliable switching of processes. The interrupt handler needed support for external interrupts, software interrupts and interrupt nesting, allowing interrupt processes with higher priority to interrupt those with lower priority.

The standard configuration of the Nios II does not have an interrupt vector in hardware, relying on the software to calculate which interrupt service routine should be called. To calculate the IRQ number, the interrupt handler needs to do a number of bit operations. The current algorithm uses a divide and conquer approach to find the lowest bit that is set, using up to 70 cycles[1]. An alternative and faster algorithm would be a loop-less testing of bits, which could cut the number of cycles in about half, reducing the interrupt latency by 35 cycles. Calculating the IRQ number in software is slow, but it's the only way to do it on the Nios II without requiring interrupt vectors in hardware, which would narrow the number of Nios II systems able to run OSEck and thereby limiting the usefulness of the port. One possible future improvement would be to supply a special solution that utilizes (and requires) a hardware exception vector [3, chapter 2]. This should reduce interrupt latency by yet another 30 cycles.

---

[1] A quick theoretical calculation shows that the routine takes about 60 cycles on average, plus a few cycles for an eventual call instruction. Branch delays, control register stalls and shift stalls have all been considered. Depending on the success of branch prediction, the worst case scenario could add about 10 cycles to this.

For testing of the kernel, see section 2.7 where the testing procedures are explained in detail.

2.6.2 Heap and Timeout Server (TS)

There is no or little target specific code in these two components, and without much work we managed to get them to pass all the tests in their respective testsystems. The timeout server demanded most of our time, as it requires a driver for the timer hardware and therefore at least a limited form of BSP (see below).

2.6.3 Board Support Package (BSP)

A board support package is a software environment designed for a particular development board, allowing application programmers to create software without having to worry about hardware details and writing drivers. A fully fledged BSP should naturally support all hardware peripherals on the development board. Altera supplies a Hardware Abstraction Layer (HAL) that includes drivers for all the standard components that can be instantiated in a Nios II system [2, section II]. The HAL is a runtime environment for single-threaded applications. Through integration of the *Newlib ANSI C standard library* the HAL supports easy access to hardware with functions like *open* and *printf*. Using the software build tools from Altera it is rather easy to create a BSP with automatically generated files. The BSP will have full support for all hardware components, as well as initializing the runtime environment, setting up the linker and creating a Makefile for easy building of applications.
To maximise the usefulness of OSEck for Nios II, it would have been nice with similar support. However, developing such a BSP from scratch is entirely out of the scope of this thesis project. A few alternative solutions presented themselves:

1. Skip the HAL and build a clean BSP with newly written drivers for the most important hardware components using OSEck system calls
2. Build OSEck on top of the HAL, using the HAL interface to create interrupts. (This is how it is done in the µC/OS-II RTOS for Nios II.) [2, chapter 9]
3. Take over the hardware and interrupt handling, but override some of the functions in the HAL and replace them with OSEck system calls. This could potentially lead to easy integration of existing drivers.

It was clear that alternative 1 would not only be more work than the others, but also result in limited support for different hardware configurations. Alternative 2 was unattractive as it would result in less control and probably worse performance with the operating system not owning the interrupts. We decided to look into alternative 3 to see if it was feasible overriding some of the HAL functions. By rewriting the HAL's interrupt handling and patch into the runtime initialization we managed to get OSEck and the HAL to coexist.

2.6.4 Drivers

As a result of the successful integration of the HAL there are functioning drivers for almost all hardware. There are two Altera drivers that won't work with OSEck: the UART and the JTAG-UART. The UART driver is for the serial port and the JTAG-UART is a virtual serial port over a JTAG-connection between the Nios II system and the host PC. The reason these two particular drivers won't work is that they use event flags, a feature of the µC/OS-II operating system. Event flags allow a process to wait for a combination of events, somewhat similar to a set of binary semaphores. This functionality does not exist in OSEck, forcing us to either implement this functionality, writing new drivers or choosing not to support this particular hardware. It was decided that the JTAG UART should be prioritized as it allows us an unlimited number of connections compared to a regular UART of which we only have one (on

the board). We wrote a new driver from scratch, but integrated it in the device hierarchy of the HAL. This way the JTAG UART still appears as a device in the HAL, making it possible to map standard input and output to it and using functions like *printf* to display text in the terminal.

The Timeout Server component of OSEck needs a timer driver. This driver needs to conform to an OSEck TS-specific API and was implemented from scratch. An OSEck driver for the LEDs is also in place, as some of the example applications use these.

2.6.5 Building and Linking Applications[1]

To make it easy for the application programmer three shell scripts have been developed that help with the creation of a BSP library as well as an application Makefile. The first script creates the BSP using the Altera tools with appropriate settings. A file defining the hardware (.sopc) is fed as input to this script. The second script patches the files in the newly created BSP directory, replacing some files as well as updating the BSP Makefile with the modifications needed for the HAL to comply with OSEck standards. The third and final script creates an application Makefile, which makes it easy to build and link an application with the OSEck and BSP libraries, producing a binary that can be downloaded and run on the Nios II processor.

There is also a superscript that runs all the scripts mentioned above. This script creates OSEck configuration files and a template application for each core in a multicore system, as well as a top-level Makefile used to build all applications and download them. All scripts can run on any host platform supported by the Altera tools.

Unfortunately it is currently impossible to integrate the OSEck port into the Eclipse-based IDE, one of the key reasons being that the IDE doesn't use Altera's BSP-scripts, which is a rather new feature.

2.7 Testing, Validation and Performance

While implementing the kernel, initial testing was done with a simple application consisting of two processes sending signals to each other. To test external interrupts we set up an interrupt process that was connected to a button IRQ. With all the functionality in place additional testing was performed. To validate functionality, correctness and robustness, the testsystems developed at Enea were used.

2.7.1 Validation using OSEck Testsystems

All OSEck components have their own testsystems and each component has been tested individually. A testsystem is divided into functionality tests and performance tests, each containing several test cases. The test cases are run one after another in sequential order until one of them fails. Status codes are written to memory so that the developer can determine if and where there is a problem.

Most of the test cases in the kernel testsystem are run on generic parts of the kernel, such as different system calls, in an effort to identify compiler bugs. Tests of architecture specific code involve testing context switches, that no register is trashed at a critical point in the code, for example during interrupt nesting.

[1] Enea: See [20] for directions to internal information.

2.7.2 Performance

The performance of the kernel has been tested with Enea's generic performance testsystem [7, chapter 7], which tests a large number of system calls and measure execution time in cycles. It has the disadvantage that different aspects of system calls are tested, but the functions called are not pre-cached, causing the first execution of a system call to show particularly bad performance compared to consecutive runs. For example, the *s_alloc*-function is tested first once where it will return a completely new buffer from the pool. It is later run again, this time allocating a buffer from the free-list. The caches will be cold during the first execution while the second greatly benefits from the fact that the function already has been run and cached. In this particular case the uncached execution is several hundred percent slower than the cached execution.

Because of this limitation in the performance test the prime use for the test results are to compare different Nios II systems running OSEck, for example to determine if a code optimization was successful or to see if a change in hardware has a clear performance effect.

Test result data is available in Appendix A. It is clear that the size of the caches in our Nios II-systems implemented on the DE2 board greatly affects performance (depending on the application footprint). Quick comparisons have shown that the performance is on par with ports to similar targets, like the MicroBlaze port. [13]

2.8 Conclusions and Future Work

2.8.1 Conclusions

We have managed to port the OSEck operating system to the Nios II architecture according to the specifications and requirements of this thesis. Beside the kernel, the heap and TS components have been successfully ported. Drivers for the important hardware are in place. There is a functioning work flow where OSEck and Altera tools have been integrated to allow application development.

2.8.1.1 The Challenge of Porting an OS to a Soft Core Processor

Using a soft microprocessor solution like the Nios II will let us vary the number of cores. There is also the possibility of adjusting other parameters, like cache sizes. Almost any type of peripheral can be instantiated in the FPGA. Custom components can be created for specific purposes, which could prove important in phase two, e.g. for inter-core synchronization. This flexibility is a two-edged sword though, increasing the amount of work when porting the OS during phase one. It's been a challenge maintaining as much as possible of this flexibility, trying to allow the system to be built in any number of ways. For a closely related take on this, please see the master thesis by Daniel Staf describing his work of porting OSEck to the soft microprocessor MicroBlaze from Xilinx [13].

2.8.2 Future work

2.8.2.1 Better development environment

A functioning work flow has been implemented, allowing an application programmer to write programs for OSEck on a Nios II/f. An end user trying to use this work flow might find it a bit rough and unpolished, which is only natural considering the goal of this thesis never was to produce a fully fledged development environment. Nonetheless, improving the work flow and easing the application development is an important feature for any product, be it commercial or academic. We would like to integrate the OSEck distribution better with the Altera tools, delivering a solution that works out-of-the-box. It is also important that our solution remains compatible with future releases from Altera.

2.8.2.2 Support for other flavours of Nios II

As it is now, we only support the most advanced type of Nios II, the Nios II/f, with the additional requirement of hardware division being enabled. Other variants could easily be supported by recompiling libraries. Support for all combinations of hardware parameters with different OS configurations would result in 12 different libraries. It would cause a lot of work as well as confusion regarding which library to use. Instead, using the unimplemented instruction exception to emulate required instructions in software could be a better approach. This is not supported in the current exception handler.

Switching the algorithm used to calculate IRQ numbers, as described in section 2.6.1, would lead to lower interrupt latency. Another possible performance improvement would be to add support for the (optional) hardware exception vector, thereby reducing interrupt latency even further.

2.8.2.3 More drivers

To support all hardware on the DE2 board, as well as all the components that can be instantiated in SOPC builder, would require additional work. Most of these are still supported thanks to the integration of the Altera HAL, but a few are still missing. Top of this list is an Ethernet driver, followed by a UART driver for serial communication.

2.8.2.4 Additional OSEck components

Two additional components that would be good for debugging purposes are the Core Dump Analyser (CDA) and Illuminator. With CDA a dump of the system memory can be analysed on the host (PC). Illuminator is a system analyser/debugger, giving real-time feedback from a running system. With an Ethernet driver in place a TCP/IP-stack could also be considered.

PHASE TWO

3 Multicore Communication Characteristics

3.1 Introduction

The second phase of this thesis is about investigating, evaluating and implementing different approaches to communication between cores in a multicore system. It has already been established that the operating system will be OSEck and that the hardware will be based on Nios II from Altera. Multicore IPC under OSEck is preferably done with LINX, therefore all evaluation has been done using LINX. For an explanation of LINX, see section 3.3.

Since this is a joint thesis, we would like to once again point out that the parts about shared memory have been developed and written by Robert Andersson, and the parts about DMA by Erik Johansson. Remaining parts have been produced in collaboration.

3.1.1 Delimitations

Having too many test cases, hardware parameters, measurement points and driver implementations would lead to a gigantic test matrix. Deciding what is relevant, interesting and within scope is not easy. The delimitations within each area are described in detail under their respective section.

3.2 Caches

Caches will soon prove to be very influential during the evaluation. A short overview of cache theory is therefore provided below. Those already familiar with the concept of different cache implementations can skim this section and continue to section 3.3.

3.2.1 Directly Mapped

Directly mapped caches are a simple form of cache where a block of external memory is mapped to one distinct entry, or line, in the cache. For example: A cache of 128 bytes with 16 byte wide lines would have 8 lines. Memory addresses 0, 128, 256 etc will be mapped to the same line, line 0, and addresses 16, 144, 272 etc to line 1. In case a memory read access to address 0 occurs followed by a read access to address 128 a cache collision will occur, line 0 will be flushed out to the external memory address 0 and the line will be filled with 16 new bytes of data from address 128. This will result in performance degradation.

Directly mapped caches are still used in many processors, despite the problem with cache collisions. They are easier to implement and require less combinatorial logic than associative caches. Directly mapped caches are more common as instruction caches than data caches since instruction data is less likely to collide due to linear access by CPUs.

3.2.2 Fully Associative

A fully associative cache has no limitations to where a particular block of memory is mapped in the cache. A 16 byte range in external memory can occupy any location in the cache. This

is optimal in terms of performance, but can be very expensive to implement in hardware and the cost can rarely be motivated.

3.2.3 K-way Set-Associative

K-way set-associative caches are a compromise between directly mapped caches and fully associative caches. They are organised into sets rather than individual lines. A 2-way set-associative cache allows an entry in external memory to be cached into two different locations, or ways, in the cache. Two external memory ranges, which would have collided in a directly mapped cache, can be in a 2-way set-associative cache at the same time. For example: A 2-way set-associative cache of 128 bytes with 16 byte large ways would have 4 sets. Memory addresses 0, 64, 128 etc would be mapped to set 0 (way 0 or way 1), addresses 16, 80, 144, etc to set 1.

K-way set-associative caches are commonly used as data caches, where data often is accessed randomly and are more likely to cause collisions. They are however not as common in small and cost efficient microcontrollers, where a simple design is important.

3.2.4 Caches on Nios II/f

Caches are optional on the Nios II microcontroller. Both instruction and data caches are directly mapped. Only level 1 instruction and data caches can be configured, higher level caches are not supported. An effective workaround is to place critical code and data in tightly coupled on-chip memories which have constant one cycle access times. If the instruction cache is disabled, all code *must* be placed in on-chip memory, with the result that large programs that won't fit in on-chip memory must be run on a system with instruction cache enabled. This is the case for all tests in this thesis, as the code size is fairly large.

A few configuration parameters are available with regards to caches, such as their size and line size. The user can choose if data transfers between cache and external memory should use burst transfers or not.

The caches use *allocate-on-write-miss, write-back* and synchronous flushing. This means that write misses are costly, as the CPU must wait for the currently stored cache line to be flushed and then wait for the entire new cache line to be read before writing to the cache. Write hits are always fast thanks to the *write-back* principle, but may have a negative impact later when the dirty cache line will have to be flushed.

The Nios II/f core implements a virtual addressing mode called *bit-31 cache-bypass*, as we have already mentioned (see section 2.3.1).

3.2.5 Cache Collisions

To determine the cost of cache collisions during a *memcpy* a test was made where 1 kbyte data is copied. The source and destination addresses are aligned in such a way that they always are mapped to the same cache line. A line is 32 bytes. By adjusting one of the addresses further away from the other, but still keeping the first byte of both data blocks on the same line, we can determine the cost of a single line collision to different degrees. For example, the source address is kept fixed at byte 0 on line 0, while the destination address is adjusted and mapped to line 0, byte 0 followed by byte 1, byte 2 etc.

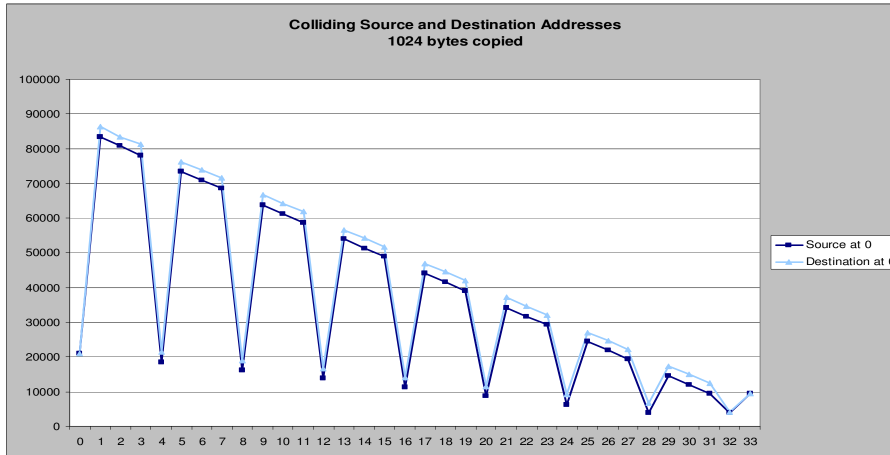


Diagram 3.2.5: *Shows the cost of cache collisions with decreasing degree of collision during memcpy. At X-axis value 0 a total of 32 bytes in source and destination data block collide, while at X-axis value 28 only 4 bytes collide.*

Diagram 3.2.5 shows the result of the test. It can be determined that cache collisions in our system are very expensive, especially when the *memcpy*-function needs to do unaligned byte and halfword copies at offsets 1, 2, 3, 5, 6, 7, etc. Normally the offsets are a multiple of 4, meaning that the lower half of the diagram above is the interesting part. The difference between no collision (offset 32) and maximum collision (offset 0) is a slow down of about 5 times. This is a massive speed reduction.

3.3 LINX

LINX [10] is a distributed inter-process communications (IPC) protocol developed by Enea. It has been designed with portability in mind and is independent of operating system and communication media. It is commonly used by OSE, OSEck and Linux systems to exchange signals between processes located on different processor cores, often by using media such as Ethernet and RapidIO. Figure 3.3.1 shows a block diagram of the LINX stack.

3.3.1 Application using LINX

Assume the development of an application for a dual-core system. The application requires signals to be exchanged between processes running on different cores. In order to do so, a driver is needed for the transfer media as well as a connection manager.

Assuming a connection is already available, what is left to do is to create and establish a link over the connection. The link is used to identify the remote side in case a shared media is used, such as Ethernet, where there is only one connection. The exact details of the procedure are not relevant to understand the concept of LINX and are therefore not mentioned here.

A connection and a link have been established between our two nodes. The OSE-family usually uses *send* and *receive* to communicate signals between processes. The same is true when using LINX, but *send* needs a process identifier (PID) for a remote process instead of a local. The PID is obtained using the system call *hunt*. Hunt takes a parameter specifying the name of the remote process as well as which link to use to locate the process. For example, to find the process *consumer* found through the link *node1, hunt("node1/consumer")* is used. Irrelevant parameters have been omitted. The call is blocking and returns a PID to use once the process has been found. This is a special type of PID and is *not* the actual PID of the process on the remote core, it is only valid on the local node. The actual PID cannot be used as it is possible that it would collide with the PID of a local process. A signal can now be sent to the remote process just as if it was running on the local core. The kernel will identify the

PID as belonging to a remote process and let LINX deliver the signal. Remote processes have a predefined range of PID numbers.

LINX also provides both link and process supervision. It is possible, through the system call *attach, to* subscribe to event notifications, i.e. a specific signal will be sent to the attacher if the link goes down or a remote process is killed.

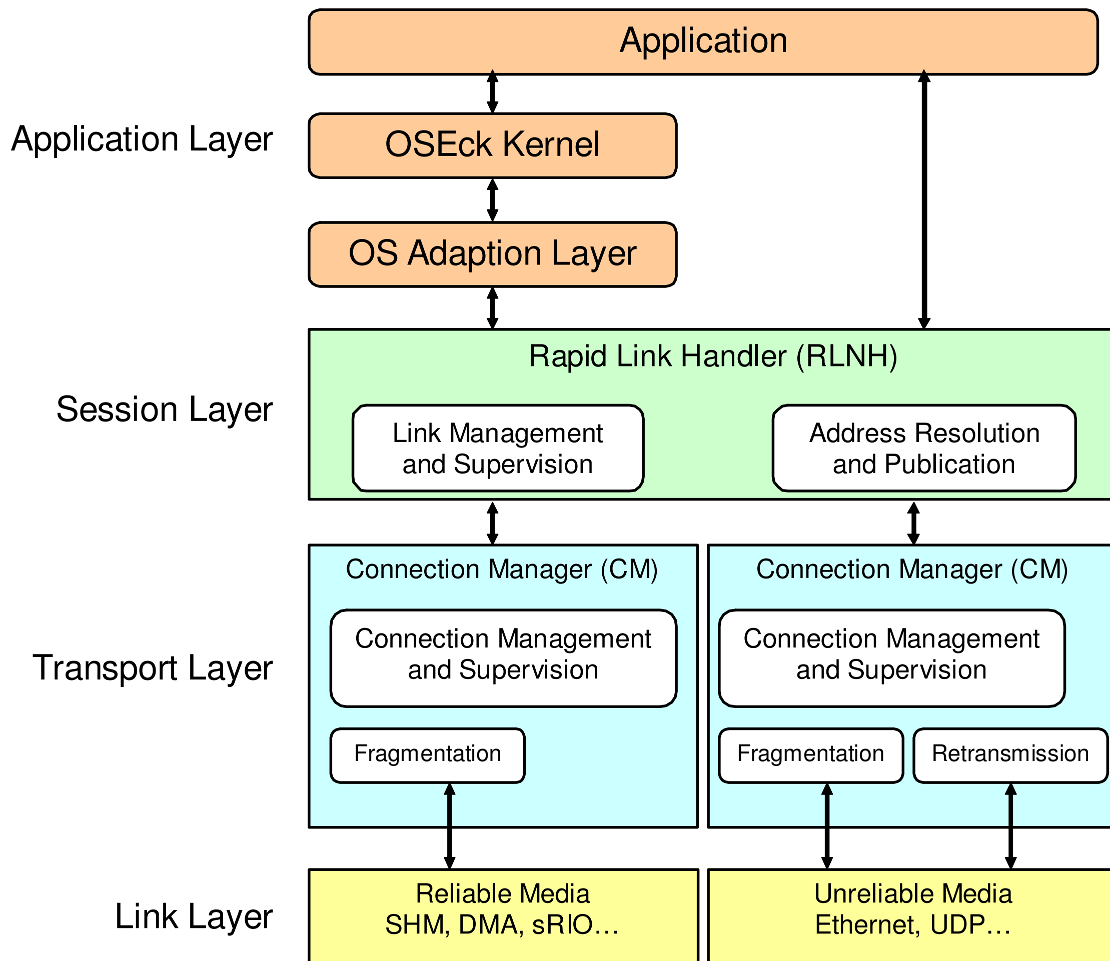


Figure 3.3.1: *LINX Protocol Stack.*

3.3.2 Rapid Link Handler (RLNH)

The Rapid Link Handler is a generic component of LINX, comparable to an OSI session layer. It manages all links in the system and keeps track of the processes using a link as well as the connection manager and driver in use.

RLNH is the LINX component responsible for doing process lookups, *hunt*s. When doing *hunt* in OSEck a process identifier is returned. Because of the operating system independent nature of LINX, the identifier can't be assumed to identify a remote process. It could very well be a Linux file descriptor or something else. Therefore an operating system independent identifier is created every time either side of a link issues a *hunt*. This identifier is used by the protocol throughout the session lifetime.

The mapping of the protocol's independent identifier and OS dependent identifier is managed by an OS Adaption Layer (OSAL). The OSAL is also needed for LINX to be able to

communicate events and signals to the application. The OSAL makes the integration with the guest OS and provides RLNH with functions to send and receive signals, allocate and free memory etc, through callbacks.

RLNH is implemented as a prioritized process in OSEck. Signals sent by application processes destined for remote processes are forwarded to the RLNH process by the OSEck kernel. RLNH distinguishes these signals from regular signals sent to it by looking at the signal's addressee PID. If the signal is not intended for RLNH itself then the PIDs are translated to OS independent IDs and the signal is forwarded to and transmitted by the connection manager.

3.3.3 Point-to-Point Connection Manager

A LINX connection manager (OSI transport layer protocol) is responsible for maintaining the media connection, such as an Ethernet connection. Since different media have different properties, the connection manager must take special care to handle unreliability, sequencing and retransmission of lost packets, in-order delivery, fragmentation etc.

The CM notifies RLNH about changes to the connection, when a connection is established, is lost or has timed out. Regular heartbeat packets are sent across the media. In the case that the remote side doesn't respond RLNH is notified. Local processes attached to remote processes will then receive signals from RLNH telling them that the PIDs of remote processes no longer are valid.

The Point-to-Point connection manager (PTP-CM) is designed to be used with media that are reliable and non-broadcast. It is suitable for use with multicore shared memory- and DMA-drivers, where reliability is provided in hardware and low protocol overhead is desired.

A CM interfaces with RLNH above and the media driver below. Important callbacks in the interface between RLNH and a CM are *transmit* and *deliver*. *Transmit* is a CM-provided function called by RLNH when it has a signal ready for transmission and is run in the context of the RLNH-process. *Deliver* is an RLNH-provided function called by a CM from the context of a prioritized CM receive-process. Once *deliver* returns, the processed signal has been sent to its receiver.

The interface between CM and driver consists of driver callbacks to start and stop the driver, read and write data, etc. The CM is responsible for providing the driver with callbacks upwards in the stack to allow the driver to inform the CM about transmitted and received data, *handleTX* and *handleRX*. These functions make it possible for the CM to implement flow control in the driver.

The PTP-CM pads the signal data obtained from RLNH above with different types of headers. In the case where the entire signal can be transmitted without fragmentation, the header named UDATA (user data) is used. UDATA is also used to indicate the first frame of a fragmented signal, in which case the indicated total size of the data exceeds the driver's maximum transfer unit (MTU). When fragmentation occurs UDATA is followed by FRAG headers and data for each transmitted frame.

The PTP-CM provides its underlying driver with a list of data buffers to be transmitted as a contiguous block of data, thus when a shared memory driver is used no extra copying is done. Each buffer in the provided list can be copied directly into a shared memory area with different offsets, as is the case with certain DMA-controllers.

The PTP-CM functions are called from varying process contexts during different operations. For example, during a transmission both the CM's and the driver's functions are called from within the context of the RLNH process. When a frame is received the driver ISR is started by the OS, and when the driver is done processing the frame it calls the CM *handleRX* function, which will queue the frame in the CM and signal the CM receive-process that a frame is

waiting. The CM RX-process will then decode the headers and call the necessary RLNH-functions needed in order to deliver the packet to the destined application process.

The above statements are true as long as the CM doesn't need to implement flow control on the driver during transmissions. When that happens queuing of frames occurs within the CM and the driver must acknowledge that it is ready by calling the CM *handleTX* function. The next frame queued in the CM will be transmitted entirely from within the context of a driver ISR.

3.3.4 Driver

Drivers control the media and act as logical OSI link layers. The drivers follow a well defined interface adopted by the entire OSE-family; hence it should be possible for other clients but LINX to use them, for example the IP-stack.

The interface is based upon callbacks such as *start, stop, read, write* and *writev.* They are mostly self explanatory. The difference between *write* and *writev* is that the latter takes a list of data buffers instead of a contiguous block of data. The PTP-CM only uses *writev*.

A driver typically needs to install interrupt service routines (ISRs) and it is done through the *start* routine. An interrupt process has the highest priority class of all processes in an OSE-application and its execution time should be limited as much as possible to not disturb the responsiveness of the system. The functionality executed by LINX driver ISRs implemented at Enea is narrowed down to receiving frames and passing them along to the CM RX-process. In some cases frames queued by the CM due to flow control are also handled and transmitted directly from the context of the ISR to shorten latency.

We will return to discuss the actual driver implementations used within the thesis.

3.3.5 Process Priorities

Given the conditions stated above, there are at least three processes in any common LINX configuration. The priorities of these processes greatly affect the behavior of a system. In order to clearly understand the sequence of events at a later stage we list the processes with their respective priority in the table below, with the highest priority process listed first. The design of the LINX architecture requires the priority of the RLNH process to be higher than that of the CM RX process. To make RLNH transmit signals immediately it must also have a higher priority than the test application process.

Name	Type	Priority
Driver ISR	Interrupt Process	0-31
RLNH	Priority Process	9
CM RX Process	Priority Process	10
Test application	Priority Process	12-31

3.4 Use Cases

3.4.1 Background

Enea has a long history of embedded system projects. We decided early on to take advantage of the knowledge available and construct our use cases as well as tests around real world application characteristics. The research ended up in five different use cases, of which three are presented below. They are all related to base station platforms used in mobile network infrastructure.

3.4.2 Management of a DSP Platform

Background
The platform used as basis for the first use case is a middleware platform for telecom base stations. The middleware is running on-top of OSEck and its core components, providing the customer application with various services to control, monitor and distribute workloads across a cluster of DSPs.

Traffic
Three major traffic types exist within the middleware. All traffic is exchanged between a generic microcontroller and several DSPs using LINX and signals.

1. Control traffic. Small signals of less than 200 bytes carrying a request are sent from controller to DSP. The DSP responds to the order given by sending a similar signal back to the controller. This request and reply-based sequence takes place less than ten times per second.
2. Text log retrieval. Less than once every second the microcontroller instructs a DSP to provide its system log by sending a small request signal, less than 200 bytes, to the DSP. The DSP sends back the system log which rarely exceeds 1500 bytes.
3. At system startup all DSPs are fed with configuration signals of the same type as described in traffic number 1, i.e. no more than 200 bytes per signal. The signals are sent at a frequency of 100-1000Hz for the duration of second.
4. File downloads. Very rarely the DSPs download large files from the controller. The files are sent with signals forcing LINX to do fragmentation of them.

3.4.3 HSPA, UDP Processing

Background
The introduction of third generation mobile networks has resulted in an increased data traffic load on the networks. This use case is based upon a platform used as a gateway between the Internet and radio networks. UDP-packets are received at high speed and are processed and prepared for transmission to mobile network customers.

Traffic
UDP-packets are encapsulated within signals and are forwarded from one DSP with access to an Ethernet interface to another DSP for processing. The UDP-packets are received by the first DSP at a high and even frequency. The faster the DSP can forward them to the second DSP the better. The signals are smaller than the media’s maximum transmission unit and no fragmentation is necessary.

3.4.4 Media Gateway

Background
A media gateway is another type of platform. It provides the mobile network with services such as TV, radio etc. The platform is run by several DSPs controlled by a microcontroller, similar to the previous use cases. UDP-packets are received, decoded and forwarded to the DSP. LINX is used when exchanging control traffic between the microcontroller and the DSPs.

Traffic
Small signals with configuration data sent from controller to DSPs, less than 200 bytes at low frequency, typically 10 Hz. The DSPs acknowledge configuration signals by sending reply signals. In the background the UDP-packets are transmitted, resulting in increased bus traffic.

3.5 Test Cases

3.5.1 Overview and Delimitations

The use cases identified for LINX on OSEck, described earlier, are used as a starting-point for the creation of test cases. As can be clearly seen by just looking briefly at the use cases, most of the traffic described is request and reply-based with relatively small signal sizes where fast delivery is the most important aspect. Throughput can be set aside to allow specific latency aimed optimizations. There is a throughput use case though, which can't be completely ignored, but since the small signal based traffic usually have higher priority throughput is seen as a secondary objective.

Four test cases have been constructed intended for dual-core systems and three similar test cases for quad-cores.

3.5.2 Dual-Core Test Cases

The dual-core test cases use two Nios II processors, where one simulates a controller and the other a DSP, as described by the use cases. Figure 3.5.2 gives a conceptual overview of the system and the signal exchange.

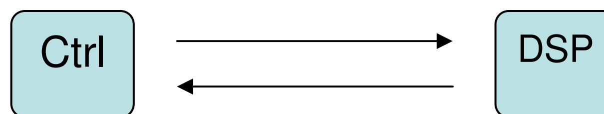


Figure 3.5.2: *Illustration of the signal exchange in the dual-core test cases. The system simulates a controller and a DSP.*

3.5.2.1 Test Case 0 (T0) – Latency

Test case 0 simulates single request and reply-based traffic, based on use cases where small control signals are exchanged.

Signal sizes are varied from 16 bytes to 1024 bytes, measured during different executions.

1. Core 0 send a request signal to Core 1 every 0.1 s.
2. Core 1 receives the request signal from Core 0
3. Core 1 sends a reply signal of equal size to Core 0
4. Core 0 receives the reply

3.5.2.2 Test Case 1 (T1) - Interference

Simulated single request and reply-based traffic as in test case 0, but in addition a text log retrieval request is sent from Core 0 to Core 1 once per second. The text log request signal is sent just before a regular 10 Hz request signal is sent, thus forcing the normal high priority request to be delayed. The test case is based on traffic type 2 in use case 1, DSP management (see 3.4.2).

The sequence of actions is the same as in T0 except once every second when the following happens:

1. Core 0 sends a system log retrieval request signal to Core 1.
2. Core 0 sends a regular request signal to Core 1 (every 0.1s)
3. Core 1 receives the system log retrieval request from Core 0.
4. Core 1 send the system log (1500 bytes) to Core 0.
5. Core 1 receives the regular request signal from Core 0.
6. Core 1 send a regular reply signal to Core 0.
7. Core 0 receives the system log from Core 1.
8. Core 0 receives the regular reply signal from Core 1.

Due to the parallelism of the system, the order of events listed above can be slightly different from a global perspective. However, the sequence of events is always the same from a core's local perspective.

3.5.2.3 Test Case 2 (T2) – Bursty Traffic

The purpose of test case 2 is to simulate and analyze bursty control traffic, for example at the startup of a platform or a regular forwarding of UDP-packets. This can be achieved in two different ways. A large number of signals could simply be sent the normal way, or the RLNH-process could be stopped, i.e. disallow it from executing, send the burst of signals and then start RLNH again.

If RLNH is stopped the signals will not be processed by LINX on the sender while the application process executes *send.* Normally the application process would be swapped out after it has done the first *send* and RLNH would be swapped in to handle and transmit the signal. With RLNH stopped the signals are queued in RLNH's signal queue and all of them will be immediately available without any delay when RLNH is started. The same effect could be accomplished by adjusting the priority of RLNH so that it has lower priority than the application process, but we have chosen to use the stop method. Actions 1 and 3 are only performed in tests where RLNH is stopped.

1. (Core 0 stops RLNH)
2. Core 0 sends 100 signals directly after each other, 32 bytes large, to Core 1
3. (Core 0 starts RLNH, LINX begins transmitting the signals)
4. Core 1 receives 100 signals from Core 0
5. Core 1 sends a confirmation signal to Core 0
6. Core 0 receives the confirmation from Core 1

3.5.2.4 Test Case 3 (T3) - Throughput

Test case 3 simulates a file download and is used to measure throughput. A number of signals with the largest possible signal size, 65000 bytes, are transmitted. The throughput is later calculated as the received data during a time span divided by that same time. Since the normal priority of processes cause a natural flow control the RLNH-process is stopped as in the previous test case and a number of signals are sent and queued in its signal queue.

1. Core 0 stops RLNH
2. Core 0 sends a number of 65000 byte signals to Core 1.
3. Core 0 starts RLNH, LINX begins transmitting the signals
4. Core 1 receives the large signals from Core 0

3.5.3 Quad-Core Test Cases

3.5.3.1 Test Case 4 (T4) – Bus arbitration

Test case 4 is exactly the same test case as T0 except that two pairs of cores exchange signals, i.e. Core 0 send requests to Core1 and Core 2 sends requests to Core 3, Core 1 replies to Core 0 and Core 3 replies to core 2. The topology is shown in figure 3.5.3.1.

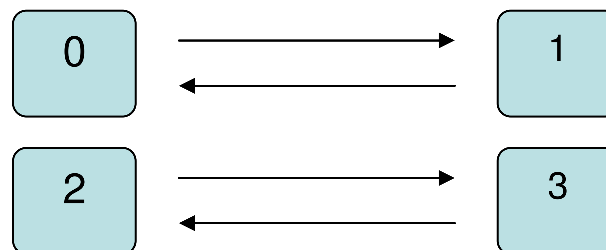


Figure 3.5.3.1: *Illustration of the signal exchange in the quad-core test case T4.*

Given the memory layout we are using, with code and data in a single SDRAM, cache misses and increased bus arbitration should become visible, resulting in increased latency compared to T0.

3.5.3.2 Test Case 5 (T5) – Multiple Units

Test case 5 is a further development of T0, where only one of the cores, Core 0, sends requests to the three other cores which reply back to Core 0. The test will show how good (or bad) the driver handles multiple units and connections. The topology is show in figure 3.5.3.2.

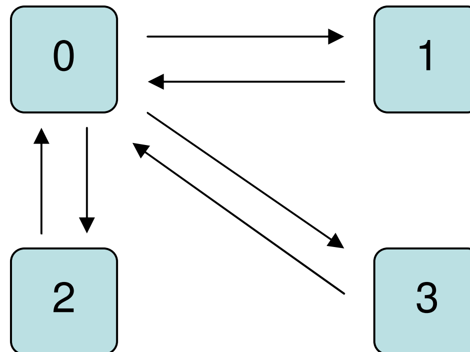


Figure 3.5.3.2: *Illustration of the signal exchange in the quad-core test cases T5 and T6.*

3.5.3.3 Test Case 6 (T6) – Multiple Units

Test case 6 is a mirrored T5, where three simulated DSPs send requests to a single core, the controller, which will send replies to each of the DSPs. See figure 3.5.3.2.

3.6 Multicore Driver Implementations

3.6.1 Shared Memory Drivers

3.6.1.1 Shared Memory Implementation Delimitations

Of the possible implementation alternatives discussed in section 1.4.1 the following were chosen:

- Shared memory buffers as intermediary storage or single copy between memory pools
- Shared memory buffers can be either cached or uncached
- The sender notifies the receiver by issuing an interrupt
- The receiver acknowledges the reception of a signal with an interrupt, *or*
  The receiver acknowledges the reception of a signal by changing a status flag

3.6.1.2 Overview

Three different shared memory drivers have been developed. All of them are roughly based on an existing OSEck LINX shared memory driver with modifications to how the interrupt hardware is designed.

All of the drivers make use of a shared memory area. The area is divided into smaller regions, one for each *unit* the driver controls. A unit can be considered as a hardware device, such as an Ethernet interface. There is one unit per core-to-core connection, i.e. there will be a total of three units on each core in a quad-core system.

The memory region allocated to each unit contains an administrative block and a data buffer. The data buffer is used as a storage area for data to be transmitted. It is divided into two pieces with each piece given to each side of the connection. Each piece thereby acts as both RX- and TX-area. A core sending data writes its data into the TX-area and a core receiving reads the data from the RX-area (see figure 3.6.1.2).

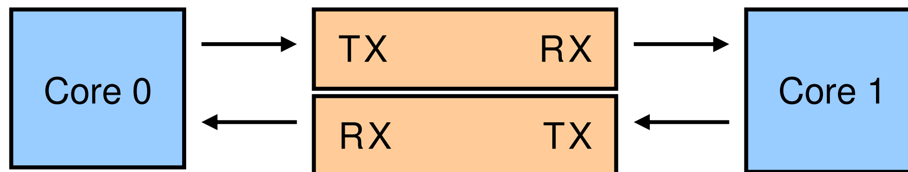


Figure 3.6.1.2: *Two data buffers used for signal exchange.*

The administrative block contains special flags used to inform each side of the connection about the current status of the RX and TX-area. Legal states are:

1. EMPTY - no data available
2. WRITING - a core is writing data into its TX-area buffer
3. WRITTEN - a core has written its data into its TX-area buffer
4. READING - a core is reading data from its RX-area buffer
5. READ - a core has read the available data from its RX-area buffer

Other administrative information is for example the size of the user data written into an area.

3.6.1.3 Shared Memory with Acknowledge Interrupt

See figure 3.6.1.3 for an illustration of the data flow. When the CM calls the driver's *writev* function, the driver will check the status of its TX-area. If the current state is either EMPTY or READ the sending driver may proceed and copy the data passed down from the CM into the TX-area data buffer. It changes the state of the TX-area to WRITING. Since *writev* takes a list of buffers, the driver will copy each of these buffers into the shared TX-area data buffer so that they are stored together as a continuous block of data.

Once data has been copied into shared memory the sender changes the state of its TX-area to WRITTEN and an interrupt request is made to force the receiver's driver to take care of the data. The receiver ISR is started by the kernel and it takes a look at the current state of its RX-buffer, which should be WRITTEN. The driver will change the state to READING and starts copying data from the shared memory into a private device memory (DM) buffer (see section 2.2.4.4). When it is done, the state is changed to READ and the driver notifies the CM that new data has arrived by calling *handleRX* with the DM-buffer as parameter. It will also trigger an interrupt back to the sender acknowledging that data has been taken care of. The sender is then able to continue with the next pending signal to be transmitted over the connection.

Before returning from the interrupt process, the receiver driver will check to see that no new interrupt has occurred while it was occupied with data copying by checking the bitmask of the interrupt hardware. If new data is available on another unit it will be taken care of

immediately, thus saving the overhead of switching context out of the interrupt process and back in again.

After the interrupt process has ended its execution it is likely that the CM will be swapped in to handle the received signal. It will parse CM-protocol headers and copy any signal data available in the frame to a newly allocated memory pool buffer. It finally calls an RLNH callback to get the signal delivered to the receiving process.

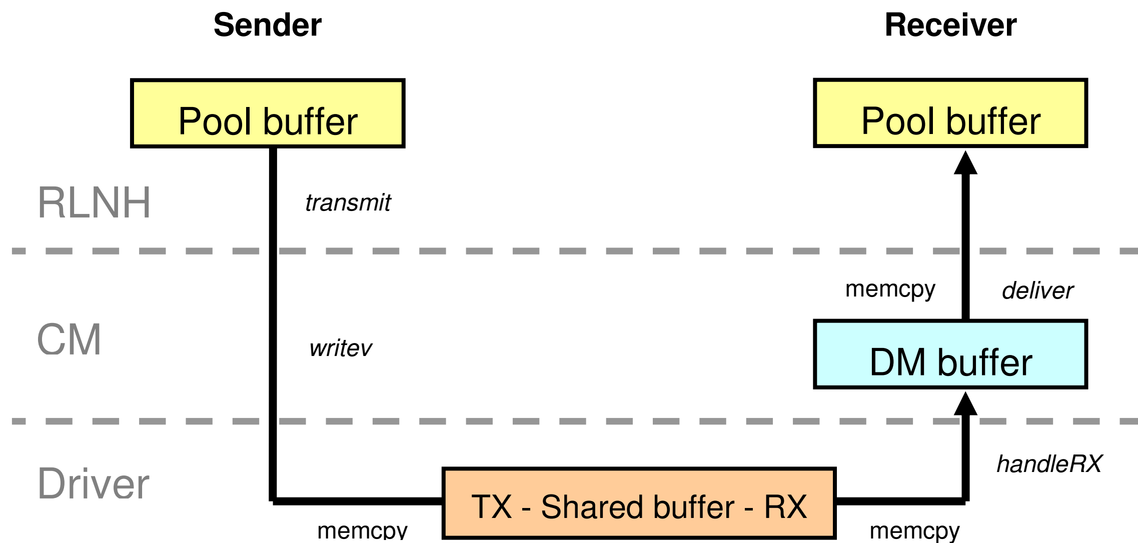


Figure 3.6.1.3: *Block diagram of a shared memory signal transfer.*

3.6.1.4 Shared Memory without Acknowledge Interrupt

The second driver implementation is very similar to the one mentioned above. A reasonable theory is that some time could be saved by not triggering the acknowledge interrupt as is done in the previous driver implementation, which would shorten the signal latency.

Without the acknowledge interrupt the sender's driver repeatedly polls the state of the TX-area to know when it can proceed and write its data into the shared memory. The receiver makes the area available rather quickly after a transmission and the polling time should be limited.

The entire transmission procedure is usually executed from the context of the RLNH-process. When interrupt acknowledgement is used and the TX-area is occupied, data passed down to the driver is queued while waiting for the interrupt. The queued signal data would later be directly written into the shared memory by the driver ISR when it is triggered by the acknowledge interrupt. Queuing is impossible when polling is used, since we never get the interrupt - all polling will always occur within the context of RLNH and any process with lower priority will be blocked indefinitely while RLNH and the driver is spinning on the TX-area state flag.

3.6.1.5 Shared Memory with Single Copy

In both the implementations mentioned above a total of three memory copy operations occur, one on the sending core from a memory pool buffer into the TX-area. The receiver driver in turn copies data from its RX-area to a DM-buffer. The CM finally copies the signal data from the DM-buffer to a memory pool buffer. This is far from optimal.

The third and final driver implementation is designed to only do one data copy - from sending application's memory pool directly to the receiver's memory pool. A requirement for this to work is that pool memory is accessible for any core involved, i.e. the memory must be fully shared. This is the case with our Nios II environment.

In order to achieve a single copy a few modifications to the CM are necessary. The CM expects the driver to pass it data with CM headers and user data in the same buffer (a DM-buffer). We can't simply pass a pointer to the signal buffer from sender to receiver and provide the CM with it.

The solution chosen was to add a word of data to the CM's UDATA header where the signal pointer is stored. The CM receiving a UDATA header then allocates a memory pool buffer, extracts the pointer to the signal buffer from the UDATA header and copies the data directly from buffer to buffer. That way the first driver implementation could be reused unaltered. Only small CM headers are passed through the shared memory area and the signal data is copied once by the CM. The effects of removing two copy operations of signal data will soon prove to be of great significance.

Since pool buffers are cached in first level data cache, data correctness is guaranteed by letting the sending side of a link flush the buffer. The receiver in turn enables *bit 31 cache bypass* on the address range it intends to copy. The Nios II processor doesn't support cache line invalidation, which could otherwise have been used.

3.6.2 DMA Drivers

3.4.2.1 DMA Hardware Variants and Delimitations

Altera offers two types of DMA controllers: a normal controller, that can only be set up to make one transfer, and a scatter-gather controller that can do several consecutive transfers (see chapter 1.4.2). Both types only have a single channel. To get more than one channel several controllers need to be implemented when designing the hardware. We chose to use only one of the controllers, namely the scatter-gather controller, for the following reasons:

- The hardware interface of the two types of controllers is totally different, meaning that the DMA-driver would, to a large extent, have to be rewritten if both types were to be tested.
- The LINX down-call from CM to driver, *writev*, sends a list of data buffers that should be gathered and sent one after the other. This type of scenario would clearly favor a scatter-gather controller, which could simply set up all the transfers in one go.
- The controllers in use on the major hardware targets for OSEck have scatter-gather capabilities.

When setting up the Altera SG-DMA controller in hardware there are a few configuration parameters [5]. The most interesting of these being the data width, deciding how many bits should be transferred to and from the memory each time. Values between 8 and 128 are all possible, but as the memories on the DE2 board are only 16 bits wide only two alternatives remained: 8 bits or 16 bits. A reasonable assumption would be that using 8 bit transfers would take twice as long as 16 bit transfers. Considering that other targets use at least 32 bit transfers, 8 bit transfers were deemed uninteresting.

Of the possible implementation alternatives described in section 1.4.2 the following were chosen:

- One SG-DMA controller per core at the receiving side
- One hardware mutex for each controller
- Caches flushed at both ends
- Transmission of pointers in conjunction with acknowledgement of previous transfer
- With or without acknowledgement interrupt (two different drivers)

3.6.2.2 Overview

The driver is based on an existing OSEck LINX driver for the 320c64x DSP-architecture from Texas Instruments (TI). The driver had to be rewritten to a large extent as the scatter-gather DMA controller used by this DSP is totally different from the one available from Altera. For starters, the 320c64x has a controller with 64 channels compared to the single channel in the Altera controller. Further on the controller from TI have hardware support for managing descriptors, while the Altera controller relies on descriptors stored in normal memory managed by the CPU.

To make a transfer with the Altera SG-DMA controller a chain of descriptors is set up. Each descriptor specifies one segment of data to be copied — source address, destination address and the number of bytes to be copied (among other things). A pointer points to the next descriptor in the chain. To start the transfer the CPU gives the first descriptor to the controller and starts it. The controller processes the descriptors one after the other, and when the last segment has been copied an interrupt is issued, letting the associated CPU know a transfer is finished.

Step by step
See figure 3.6.2.2 for an illustration of the data flow. When the driver receives the *writev* down-call from the CM it sets up one descriptor for each header in the list. The destination address is an RX buffer in the receiver's memory space. An additional descriptor is used for a driver-specific header with some details about the transfer like total length and identity of the sender. Finally, an empty descriptor is set up to indicate the end of the chain. The driver then waits on a mutex guarding the receiving controller. When the controller is free the first descriptor is handed over and the transfer is started.

The receiving core receives an interrupt indicating that "it has mail". The receiving driver hands over the message to the CM using the *handleRX* up-call. The CM processes the message and then queues the signal in the CM RX-process. When *handleRX* returns the driver checks which core sent the message and then acknowledges the transfer and gives the sending core a pointer to a new DM-buffer (see section 2.2.4.4). This buffer is then used by the sender for the next transfer. What about the first transfer? When the unit is set up the drivers on both sides go through a handshake procedure to interchange the first DM-buffers.

The CM RX-process allocates a signal buffer and copies the signal from the DM-buffer to the newly allocated signal. The signal is then delivered to RLNH for further delivery to the receiving application process. This means that a signal is copied two times when using the DMA driver: once with DMA and once with a normal *memcpy* in the CM.

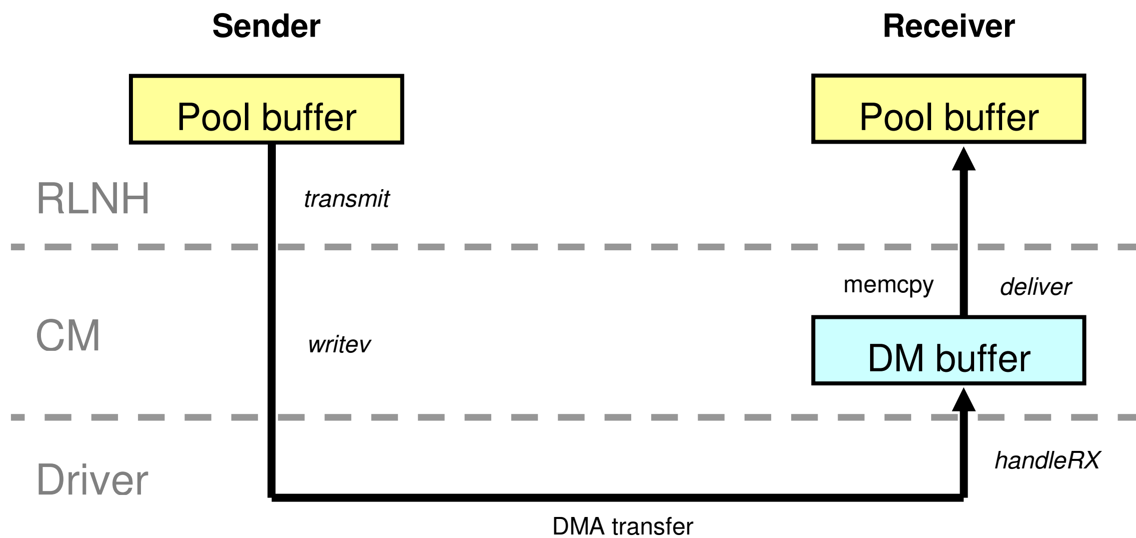


Figure 3.6.2.2: *Block diagram of a DMA signal transfer.*

3.6.2.3 DMA with Acknowledge Interrupt

If an ACK-interrupt is used, the receiver writes an ACK-message into the header and then interrupts the sending core. The ISR on the sender checks which core sent the ACK and then up-calls *handleTX* for the corresponding unit. If a new message is waiting it is immediately sent, otherwise the ISR just exits.

3.6.2.4 DMA without Acknowledge Interrupt

When no ACK-interrupt is used the acknowledgement of a received signal is checked before next transmission. When the sender wants to initiate a transfer it checks for the ACK from the receiver. If it is still not there the sender spins until the previous transfer is confirmed. Then *handleTX* is up-called. This means that part of the execution time of a certain signal is moved to the next signal. Compared to using an ACK-interrupt there is no implicit flow control, but the number of interrupts will be just one instead of two per signal, resulting in less context switches and hopefully faster transfers.

3.6.2.5 Implementation Details and Challenges

The existing driver for the TI DSP used one channel for each unit. If every core has a link to all the others there will be $n*(n - 1)$ units in a system with n cores. An additional channel was used to send acknowledgement interrupts, resulting in a total channel use of $2*n*(n - 1)$ channels. For a quad-core system this would result in 24 channels, which was impossible to implement with the single channel controller from Altera. Instead, one controller per receiver was used resulting in as many channels as cores.

The descriptors were setup in cache-bypassed SDRAM. Each core has 10 descriptors that can be used regardless of receiver. Descriptor management was implemented in a simple way to allow allocating and freeing of descriptors. Before a message is sent descriptors are allocated, later to be freed when the transfer has been acknowledged. In the case of all descriptors being taken the sender waits until enough descriptors have been freed.

3.6.2.6 Known issues

At the time of writing there is a confirmed bug in the hardware generation of SOPC-builder preventing transfers to byte-aligned addresses. As long as all buffers are halfword-aligned and all headers are of even length this is no problem. In an earlier version of the CM some headers were of odd length, resulting in broken transfers. This is no longer the case in the current version, but should the headers be redefined in a future CM-version the problem may resurface. When Altera fixes the hardware bug it is recommended to turn on the "allow unaligned transfers"-feature, which should make the system immune to this particular type of alignment problem.

In a system with no ACK-interrupt, and many cores, a sending core might run out of descriptors, leading to the driver being blocked. For a system like this the number of descriptors could easily be increased to avoid such a situation.

3.7 Method

3.7.1 Measurements

3.7.1.1 Definitions

- **Timestamp**
  The value of a timer at a certain time. This is sometimes called time of day. The timestamps that we have used in this thesis are all 32 bit values representing the number of CPU cycles since the timer was reset. This allows measurements within a span of a little more than 85 seconds. All of the test cases take about 1 second to complete, so there is no risk of the timer wrapping around. The resolution of the timestamp is 1 cycle, which at 50 MHz is equal to 0.020 microseconds. Due to the nature of the pipeline in Nios II, two timestamps read directly after each other will be 2 cycles apart, as there is a late result penalty on all load instructions.

- **Latency**
  Latency is defined as the signal travel time, from *send* is called until *receive* returns, both at application level. Normally we measure latency in CPU-cycles. All tests have been done with the processors running at 50 MHz. To calculate a latency value in microseconds, just divide the cycle count with 50.

- **Deliver difference**
  Deliver difference is the time between the returns from two consecutive *receive* calls. Measured in CPU cycles.

- **Throughput**
  One way to calculate throughput would be signal size divided by latency. The system should be saturated with signals so that peak transfer speed is achieved. The problem is when signals are sent faster than they can be transferred (and therefore must be queued). When this happens latency is higher for the signals that have been queued compared to the first signal, which is transferred straight away. To get a fairer indication of speed, the rate at which signals are received can be used. For test case 3 we calculate throughput as amount of data received during a certain time span divided by that same time.

- **Blocking time**
  The time spent during execution of the *send* function, measured in CPU cycles. For the drivers without acknowledge interrupts, this is equal to the total CPU time needed on the sending core to send the signal.

- **Jitter**
  The absolute value of the difference in latency of two consecutive signals, measured in CPU cycles. Some define jitter as the *maximum* difference in latency for a set of signals, but we've chosen to apply the term to consecutive signals.

- **Total CPU time**
  The number of cycles needed to send and receive the signal on both cores, added together. Any time spent in the idle process is disregarded.

- **Round trip time**
  The time from *send* of a request signal until *receive* returns for the corresponding reply signal. In an unloaded system this is approximately the latency of the two signals, added together.

3.7.1.2 Timestamps

To be able to calculate latency and the other types of measurements a log file is created with a large number of timestamps. Each time a measurement point is passed in the code, a timestamp is stored in an uncached data structure. When the test case is finished, all the timestamps are collected from the data structure and output to the log file. One log file is generated for each run of every test case. By importing the log file into a spreadsheet program (like Excel), post analysis of the raw data can be performed.

3.7.1.3 Delimitations

Among the types of measurements considered were:

- Application latency - *send -> receive*
- Application round-trip time – *send request –> receive reply*
- Application blocking time due to *send*
- Application deliver difference - *receive -> receive*
- Driver round-trip time – *writev request – handleRX reply*
- Driver send time – *writev*
- Driver receive time – *ISR –> handleRX*
- Number of interrupts
- Overall system load
- Bytes received per second
- Peak signal queue size in driver and CM
- *memcpy* execution time

Most of the above measurements could be calculated by simple arithmetic on carefully chosen timestamps saved during execution. The last one, *memcpy* execution time, was omitted as it would add up to eight extra timestamps per signal. This was deemed to be unnecessary overhead for a marginally interesting measurement type. As the DMA driver does not make as many *memcpy* calls as the shared memory driver, there would be an unequal amount of overhead, resulting in unfair comparisons of the two drivers. All the other measurements can be calculated from the timestamps in the log files, including number of interrupts and signal queue sizes.

3.7.1.4 Points of Measurements

Three timestamps are recorded for each signal: one just before send, one when send returns and finally one just after receive returns. With these timestamps it is possible to calculate latency, throughput, blocking time, deliver difference and jitter. To be able to calculate the time spent in the various processes in the system a swap hook is used. A swap hook is a piece of code that is run by the kernel every time a new process is swapped in. The swap hook records the PID (Process ID) of the processes being swapped in and out together with a timestamp. Finally, to allow calculation of the time spent in various layers of the LINX stack, a number of up- and down-calls are timestamped on entry and exit:

- CM-transmit - RLNH makes this down-call to have the CM layer transmit a signal
- DRV-write - down-call used by the CM to make the driver transmit a fragment
- CM-handleRX - up-call from the driver to the CM to notify it of a newly arrived fragment (receiving side)
- CM-handleTX - up-call from the driver to let the CM know that a fragment has been successfully received on the other side (sending side)
- DRV-read - down-call from the CM to the driver, handing over a buffer that the driver can use to receive fragments from the other side
- RLNH-deliver - up-call from the CM to RLNH, notifying it of a newly arrived signal

In Table 3.7.1.4 an excerpt from an actual log file can be seen. The first three lines are associated with one signal, since they have the same number. The first line says that signal number 0 from core 0 was sent at time 5000534 and the signal was 128 bytes large. Following that are two lines declaring that the *send* function returned at time 5016356 and that the signal was received at time 5037502. Then we have three lines detailing context switches on core 0. For each context switch we see which process is swapped out and which one is swapped in. Finally, the last two lines declare that function number 5 (CM-handleTX) was called at 14292 and returned 19518.

Timestamp	Core	Type	Number	Info
5000534	0	SIG	0	OFFER SIZE(128)
5016356	0	SIG	0	SEND RET
5037502	0	SIG	0	DELIVER
10871	0	CS	7	FROM(2) TO(8)
22038	0	CS	8	FROM(8) TO(2)
23168	0	CS	9	FROM(2) TO(3)
14292	0	CALL	12	FUNC(5)
19518	0	RET	12	FUNC(5)

Table 3.7.1.4: *Part of a log file with timestamps*

3.7.1.5 Overhead and Interference

Due to the *observer effect* [19] it is impossible to observe (measure) a phenomenon without affecting it in some way. The introduction of the above mentioned timestamps will, quite naturally, add a certain overhead and slow down the programs. That is, without the timestamps the system would be a little bit faster. Measure overhead, for a single signal, is approximately 175 cycles for the three timestamps of the signal itself, plus overhead for the swap hook (137 cycles/context switch, 8 switches normally) and the calls (143 cycles/call, 8 calls normally). All in all this adds up to 2415 cycles. This is in the region of 1% to 10% of the total latency for most signals that we have measured. These numbers apply to warm instruction cache, that is, the code for the measurements is ready in the cache.

As the purpose of this thesis is to compare various test runs, the actual values doesn't matter that much, it's their relative sizes that are interesting. As all signals will have the same overhead we've chosen to ignore the observer effect and report the figures as they are, including overhead. The overhead is constant (with warm caches) and fairly small — the simplification made is considered reasonable.

3.7.1.6 Test Case Execution

The test cases have been compiled into isolated ELF-files and then downloaded to the board. A test case may consist of several executable files where the only difference is the size of signals sent between cores. We never alter any parameters during the execution of a test. The explanation is that we don't want any interference at all between consecutive tests, such as how the memory pool has been altered, how caches are initialized, etc.

When a test case is downloaded to the board caches are invalidated and memories are cleared to assure that the test case is running in a clean environment. Once the initialization phase has completed the test case is started and is run uninterruptedly until it finishes.

3.7.1.7 The Analyser Tool

To get a good view of the actual events during the transmission of signals in a multicore system it is nice to get a graphical display. The Analyser Tool was developed specifically for this thesis, to show context switches, signals and function calls along a timeline. It gives a good view of all the events that transpire during the signalling. Events that occur simultaneously on the different cores can quickly be identified. It also gives fast and easy access to selected data from the log file. The Analyser Tool was developed by Robert Andersson in Java.

In figure 3.7.1.7 a screen dump of The Analyser Tool can be seen. The red bars represent the processes in the system. The upper red bars are the processes on core 0, the lower are the processes on core 1 in a dual core system. The blue bars are signals, with the upper blue bar being a signal sent from core 0 and the lower one being sent from core 1. The green bars are the up- and down-calls within the LINX-stack done on core 0. On the left are the process names for the red bars, the signal sequence number for the blue bars and finally the name of the green bars representing the up- and down-calls.

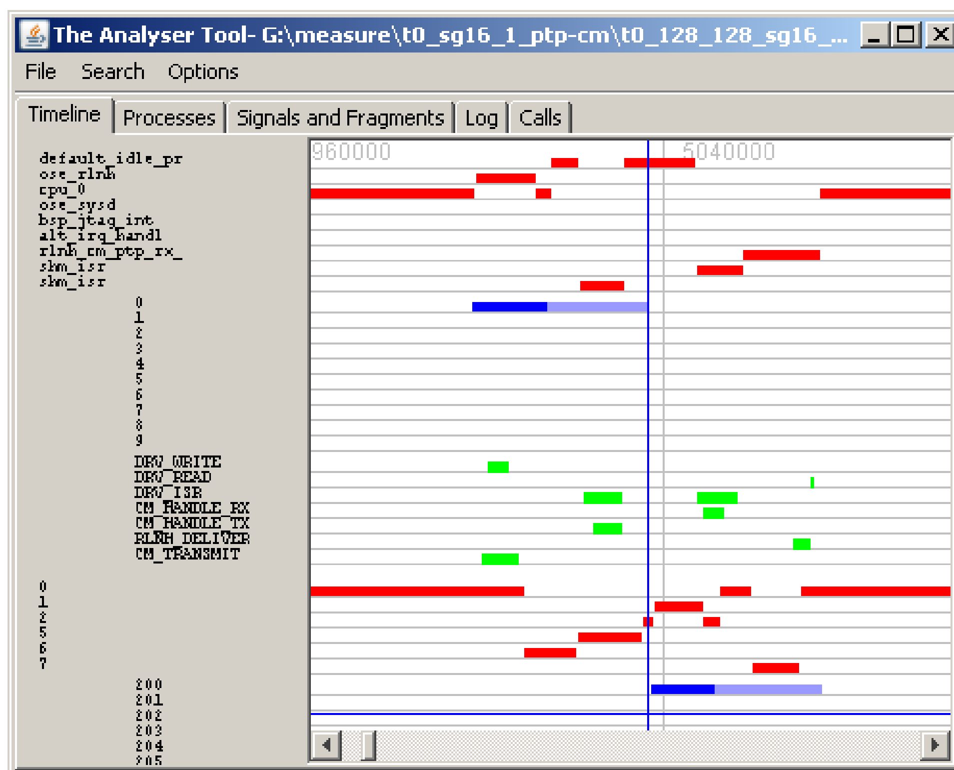


Figure 3.7.1.7: *The Analyser Tool showing the events during a single request and reply.*

3.7.2 System Description

3.7.2.1 Hardware Parameter Delimitations

A number of hardware parameters may have an impact on the performance in this type of message passing, type of CPU, memory bandwidth and cache configuration being among the usual suspects. Varying the type of CPU or decreasing memory bandwidth would just lower the general performance, which was deemed uninteresting, while increasing the number of test runs. It was clear from performance tests on the kernel that also cache configuration affected general system performance very much. Still, different driver implementations

might suffer more or less from gimped caches. It was therefore decided that the only variation in hardware should be cache configuration.

3.7.2.2 The Pulse IRQ Hardware

To send messages from one core to another in a multicore system, with the driver implementations we have chosen, it is necessary for the cores to be able to interrupt each other. The Nios II CPU has no such feature built-in. To facilitate these interrupts we developed and implemented new custom hardware written in VHDL. This hardware component was given the name *Pulse IRQ*. One instance of the peripheral is created for and connected to each core in the SOPC system.

Two 32-bit registers exist, *ipending* and *iclear*. The component triggers an interrupt on its associated core when another core writes a '1' to any of the 32bits in the *ipending*-register. A '1' in the *ipending*-register can be cleared by writing a '1' to the corresponding bit in the *iclear*-register. This allows up to 32 different interrupt sources to be identified by the interrupted core and any number of them can be pending at the same time.

3.7.2.3 Hardware configurations

To cover the various test cases two reference hardware systems were created, one for dual core measurements and one for quad core tests. Due to the limited memory resources in the FPGA, the quad core had to be created with smaller caches than the dual core system. It would also have been interesting to run all code from on-chip memories and only observe the effects of the data cache, but that proved to be impossible due to the same reason: limited memory.

Dual Core

- Terasic DE2 board with a Cyclone II FPGA from Altera (see 2.3.5 for details)
- Two Nios II/f CPU cores with h/w multiplication and division
- Two Scatter-Gather DMA controllers with 16-bit transfers
- Two hardware mutexes for synchronization
- CPU, memories and peripherals clocked synchronously at 50 MHz clock frequency
- 8 kbyte I-cache and 4 kbyte D-cache per core, line size 32 bytes
- 8 Mbyte 16-bit SDRAM
- 512 Mbyte 16-bit SRAM

Quad Core

- Four Nios II/f CPU cores with h/w multiplication and division
- Four Scatter-Gather DMA controllers with 16-bit transfers
- Four hardware mutexes for synchronization
- 4 kbyte I-cache and 2 kbyte D-cache per core, line size 32 bytes
- Otherwise the same configuration as dual core

3.7.2.4 Software configurations

The port of OSEck 3.3 for Nios II/f, which was implemented specifically for this thesis (see chapter 2 for details), has been used exclusively. To allow the use of a swap hook, the libosh.a library was selected (see 2.2.7). The latest versions of the LINX components RLNH and PTP-CM[1] were used (see 3.3). To avoid interference by the heartbeat functionality of the CM the kernel timer was disabled. Throughout the thesis the GCC compiler shipped with the Altera tools v7.1 has been used: nios2-elf-gcc 3.4.1. All test applications, all drivers, RLNH and the CM have been compiled with optimization level zero (-O0).

---

[1] Enea: From OSEck Baseline 41

The reason for not choosing a higher optimization level is that the compiler makes bad optimizations, removing static functions and causing link time errors, thereby forcing the use of level zero. All tests have been run with the same level of optimization, ensuring that the values can be compared. The general performance of the entire measurement system is far lower than what is common in other systems. We are more interested in the relative difference between different implementations, rather than absolute figures.

4 Results

4.1 Overview

The results of our evaluation are presented below. Different traffic properties, such as latency and throughput, have been grouped together with the test case best showing the differences between driver implementations and hardware configurations. Diagrams show the median value of the traffic property when not otherwise stated.

The results are presented and briefly described. For analysis and discussion regarding the findings, see chapter 5. For in-depth descriptions of the various test cases, see section 3.5.

4.2 T0 - Latency

In test case 0 the latency of varying signal sizes are measured. The signals are sent at a frequency of 10 Hz, giving the system ample time to "cool down" between each transmission. The MTU is larger than the signals, so no fragmentation is performed.

The following signal sizes are presented: 16, 32, 64, 128, 256 and 512 bytes. The test has also been run for a signal size of 1024 with consistent results, showing that all drivers scale linearly for even greater signal sizes. The 1024 byte signal has been left out of the diagrams on purpose as it doesn't really fit the use cases. Emphasis should be on the smaller, more common, signal sizes.

4.2.1 Shared Memory Latency

The results are presented as *Typical latency,* where signal latency is close to the average latency, and *Worst case latency*, where signals with maximum latency are looked closer at.

4.2.1.1 Typical Latency

Diagram 4.2.1.1 shows the median latency, measured in cycles, for the request signal as signal sizes grow. Three different shared memory driver implementations have been tested. Two of the tests shown in the diagram have been done with the same driver, but with data caches enabled in the first and with them disabled in the second. Notable in the diagram is the latency trend when using different implementations; the *Single Copy* driver clearly distinguishes itself from other drivers with its much slower increase in latency for larger signals.

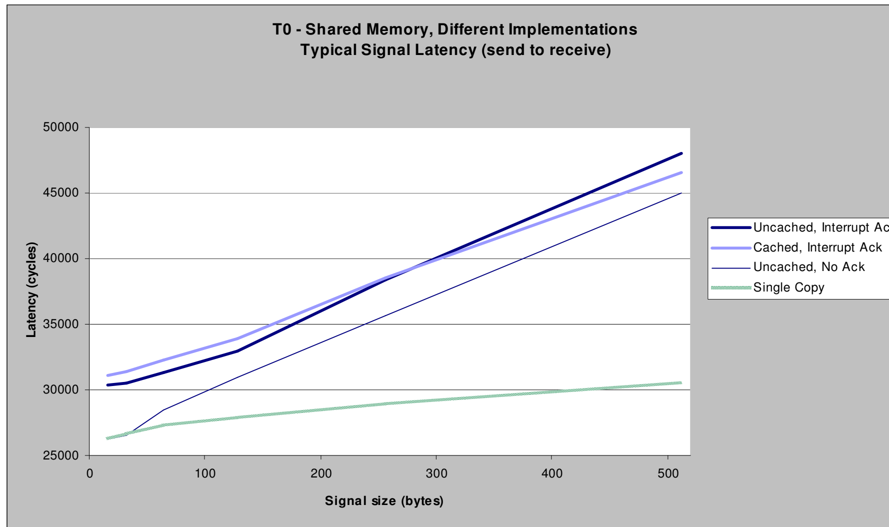


Diagram 4.2.1.1*: Typical signal latency for three implementations of shared memory drivers, with and without cached buffers*

The differences between implementations become even more visible when looking at the distribution of time across different layers of LINX. Diagrams B1 and B2 in Appendix B show this distribution for the regular *interrupt acknowledge*-driver with single signals of increasing size. On the sender we see that the time in the driver increases with the signal size, while other layer times remain nearly constant. On the receiver, however, both the time in the driver and in CM increases. This outcome is expected, the driver is doing *memcpy* at these three locations in the LINX stack.

In the *Single Copy*-implementation, shown in diagrams B3 and B4, only the time spent in the CM on the receiver is affected by increasing signal sizes due to the fact that this implementation only does its signal *memcpy* in one place, the CM.

The rather large difference in latency for small signals, about 4000 cycles, between the *single copy*-driver and the *interrupt acknowledge*-driver, can however not be explained by the number of *memcpy*:s. Even when the *memcpy* code is not cached in the CPU instruction cache, the time required to for the CPU to fetch the necessary cache lines from memory would be far less then 4000 / 2 cycles. The phenomenon could be explained by code linkage and cost of memory accesses, i.e. critical code executed often has been linked to addresses that collide in the cache, which is expensive with a directly mapped instruction cache. This introduces a large insecurity factor to our measurements not previously mentioned. A closer look at the CM layer times for the *interrupt acknowledge*-driver and the *single copy*-driver can further illustrate this. The execution time on the sender is 6000 and 8000 cycles respectively, a large difference although the CM code is close to equal for the sender in both implementations. Having small instruction caches, 8 KiByte in our case, with a code footprint several times larger is not optimal when doing these kinds of evaluations. Unfortunately, there is nothing that can be done to prevent this, since the FPGA has limited resources.

Returning to diagram 4.2.1.1 and the driver without acknowledge interrupt; we see that the difference in signal latency between a driver with ACK and without ACK is rather constant, where the driver without ACK is coming out on top. The difference is in the span between 3000 and 5000 cycles. This is less time than a sender, with *interrupt-ACK,* usually spends in the ISR, which is close to 10000 cycles.

With data caches enabled the *interrupt acknowledge*-driver is slightly slower for small signals but overtake the same driver with uncached buffers for large signals. When large signals are transmitted, having cached DM-buffers is helping to increase speed during *memcpy* between DM-buffer and pool, the memory doesn’t need to be involved as much and more data is

moved directly in the cache. Cached buffers have a huge disadvantage, though, presented in the next section.

4.2.1.2 Worst Case Latency

Data presented above regarded typical latency. The measured worst case latency, as well as best case latency, differs very little from the median latency, often just a single percent more or less, but there is one exception.

Signals in a system using drivers with cached shared memory buffers can typically be an order of magnitude slower at random occasions. The latency for the slowest signal of different sizes is shown in diagram 4.2.1.2.

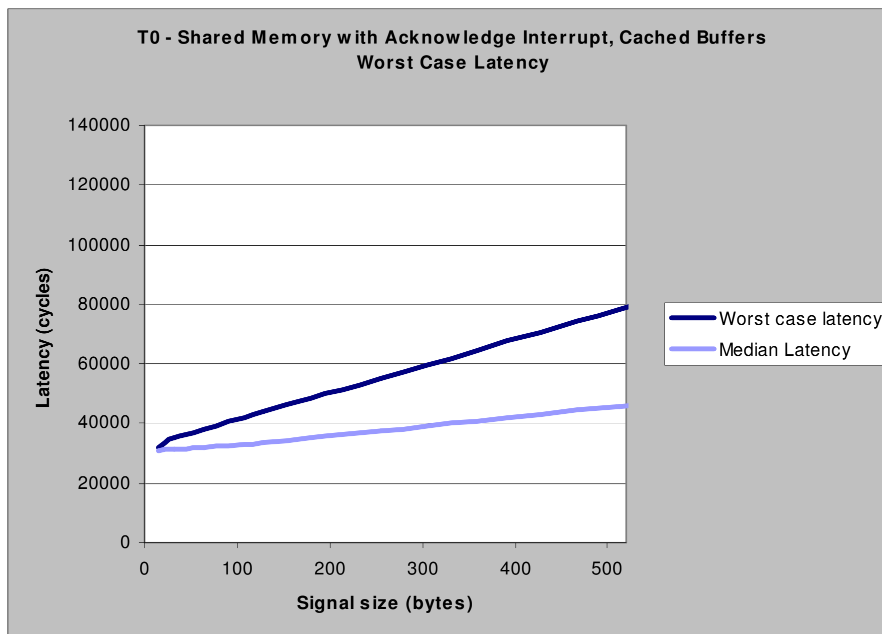


Diagram 4.2.1.2: *Worst case vs typical signal latency for signals in systems with cached shared memory buffers. The difference in latency increases with signal size.*

Interestingly, the difference between median latency and worst case latency in the case with cached shared memory buffers is linearly increasing with signal size. A layer time analysis reveals that the extra time for worst case signals is spent during *memcpy* in the CM on the receiver side. Further investigation shows that the source and destination addresses collide in the data cache, causing extensive invalidation and flushing of data to memory. This behaviour has been analysed in section 3.2.5.

Since a total of three *memcpy* occurs in all drivers but *Single Copy*, it is obvious that although the maximum latency is poor with caches enabled, it could have been much worse if we had been unlucky. The difference between median and maximum latency is roughly 40000 cycles for a signal of 512 bytes, caused by a single *memcpy* cache collision. It could as well have been 120 000 cycles if all three *memcpy* operations had been unlucky with cache collisions.

4.2.2 DMA Latency

4.2.2.1 Latency results

In diagram 4.2.2.1 the latency of both DMA drivers for varying signal sizes can be seen. The plotted lines are the median value from 10 measurements. The bars indicate min and max value. The gap between min and max is larger for the driver with ACK-interrupt. The latency scales linearly with signal size, except for a small "bump" at signal size 32. The distance between the two lines is quite constant - the driver without ACK-interrupt is about 3000 cycles faster for all signal sizes.

Analysis
The "bump" mentioned above for signal size 32 is hard to explain. It is present, at least to some extent, in all measurements made. It could be the result of this particular buffer size being unlucky when it comes to cache line alignment (see section 3.2.5). For this type of traffic, with low load, the driver *without* ACK-interrupt is not only faster, it also shows a more deterministic behaviour.

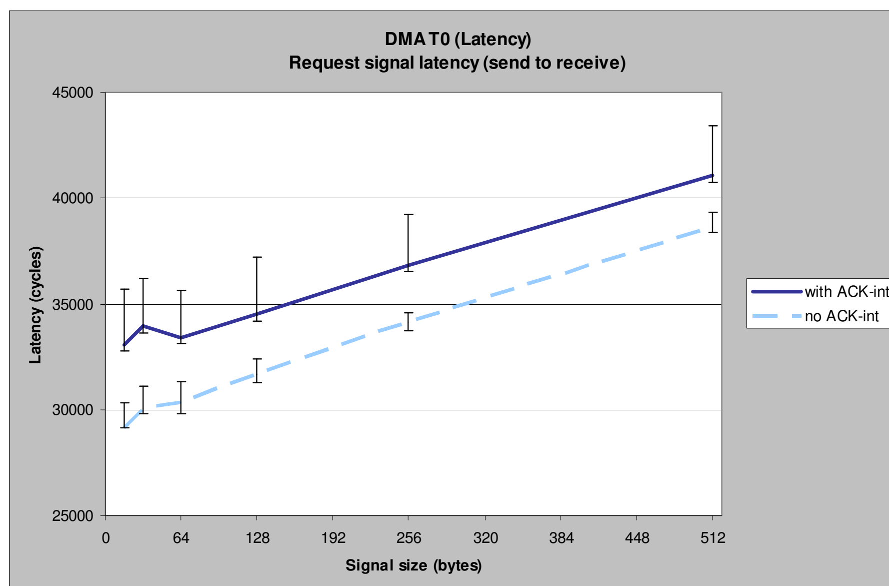


***Diagram 4.2.2.1**: DMA - Latency for varying signal size (test T0)*

4.2.2.2 Layer time results

In diagram 4.2.2.2 the time in each of the layers of the LINX stack is shown. The total time is exactly two times the latency of the signal, that is, all time during the transmission is counted on both cores. A lot of time is spent in the idle process, which is only natural considering that the receiver is idle before the signal arrives, and the sender is idle after the DMA has been started (except for the ACK-interrupt). Most of the active time is spent in the CM, with one of the tasks being to copy the newly arrived message. The idle process and the CM show a clear increase in time for the larger signal, while the other three layers are fairly constant. The time in the application is small, with the system calls *send* and *receive* being the only cycle-eaters.

Analysis
Any effort to optimize the stack should probably be put into the CM and the driver.

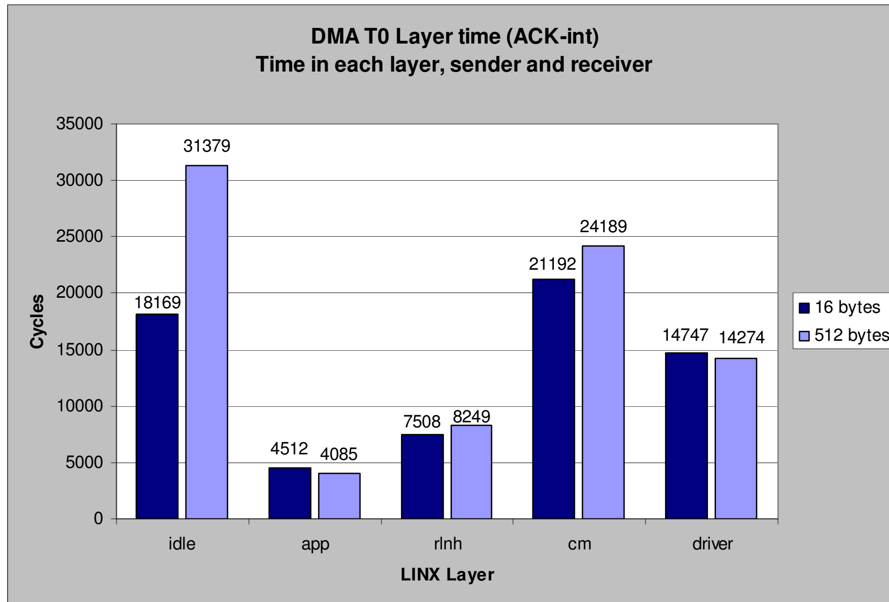


Diagram 4.2.2.2*: CPU time in each LINX layer (test T0).*
DMA driver with ACK-interrupt

4.3 T1 - Interference

In test case 1, the main purpose is to see what happens when signals are sent back and forth at the same time. How do they interfere with each other, and how much is normal request/reply traffic affected by an occasional extra (large) signal squeezed in within the normal flow?

4.3.1 Shared Memory

Tests have been performed with results being almost identical to the ones for the DMA drivers. Interesting phenomena are difficult to isolate in this test case and was therefore saved for later discussion of other test cases.

See the next section about T1 with DMA-drivers, where all results apply to the shared memory drivers as well.

4.3.2 DMA

4.3.2.1 Interference Latency

Diagram 4.3.2-1 shows the round trip time for three 32 byte signals. The only difference is the amount of interference. The first is a normal request/reply signal with no interference at all. For the other two, an extra request signal is interposed just before the normal request. The interfering request signal is 32 bytes, with the reply being 32 bytes in one case and 1500 bytes in the other. The first DMA-driver with ACK-interrupt is used.

Not surprisingly, the round trip time increases with the amount of interference. With the replier being busy receiving and replying to the first request, the second request is delayed.

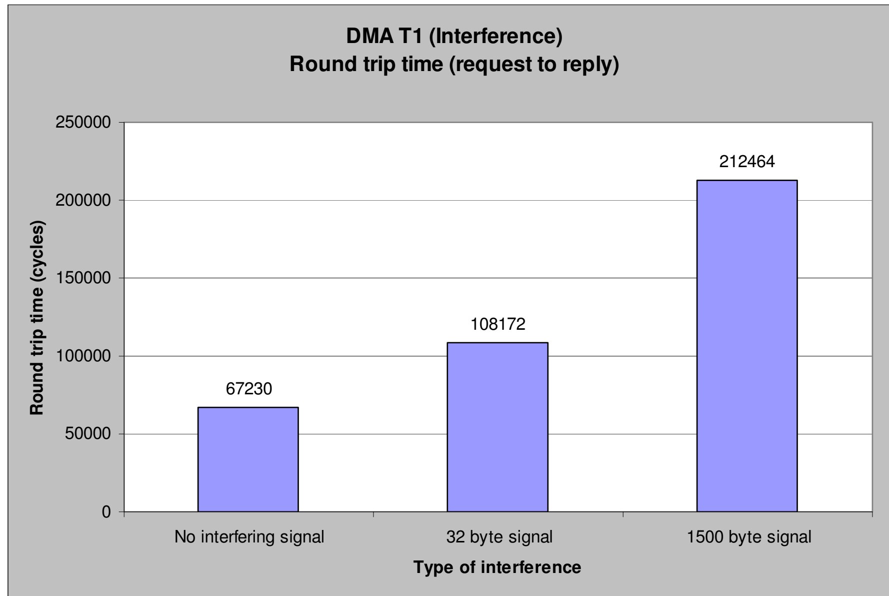


Diagram 4.3.2-1: *DMA - Interference between consecutive signals (test T1)*

Analysis
To see the flow of signals the Analyser Tool is used. Figure 4.3.2-2 shows the events transpiring during the transmission with the 1500 byte interference. The blue bars represent the signals, with the upper two being the request signals from core 0 and the lower two the reply signals from core 1. Both request signals are 32 bytes, but the latency of the second one is much higher. While the second request is being sent, core 1 is busy sending the first reply, which in this case is a 1500 byte signal. Not only the latency is higher for the second request, the blocking time is also much higher (from 13 kcycles to 38 kcycles, about 200% increase). This is surprising, as all the work during the blocking time is done on the sending core. A part of the explanation is the ACK-interrupt of the first signal arriving during this time, costing about 10 kcycles. Still, it is clear that there is inter-core interference. The remaining 15 kcycles of extra time is the result of increased bus load, as the other core receives the prior signal. With larger caches this phenomenon would likely be less dominant and therefore not so visible.

The latency of the first request and the last reply are both somewhat higher than what is normal for a 32 byte signal in an unloaded system. The first request is slowed down by the second request, and in the same way the last reply is hindered by the previous reply. Again, the increased bus load results in inter-core interference.

Note that both reply signals arrive virtually simultaneously. Copying the large reply signal in the CM takes so long that the smaller signal arrives during the copying, resulting in both being handled before delivery to the application. That is, both signals are queued in the application process' signal queue before the process is swapped in.

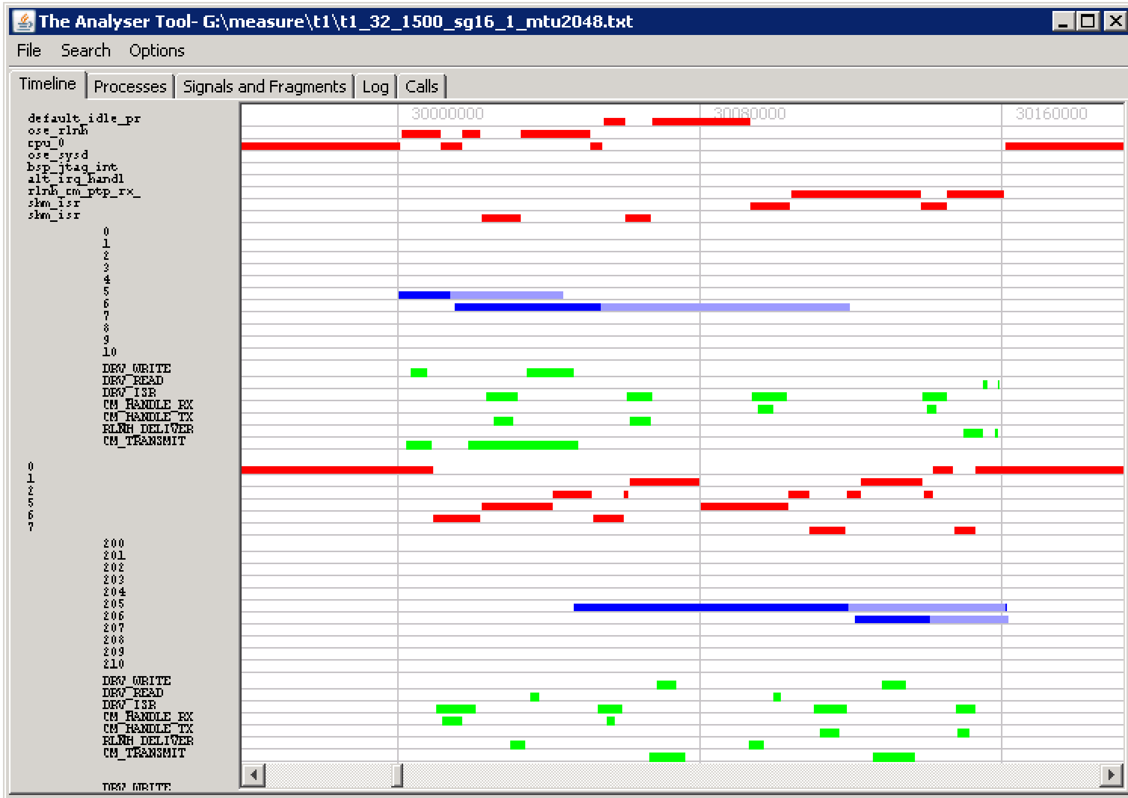


Figure 4.3.2-2: *DMA - Interference between consecutive signals (test T1) in The Analyser Tool, 1500 bytes interfering signal. Diagram repeated in full size in Appendix C (C5) for clarity*

4.4 T2 – Bursty Traffic

Test case 2 is a burst of 100 signals that are sent back-to-back. The sender does not wait for the previous signal to be received — it sends signals as fast as possible. All signals in the burst are 32 bytes and the MTU is larger to guarantee no fragmentation of signals.

4.4.1 Shared Memory

4.4.1.1 RLNH Running

When RLNH is allowed to run during consecutive *send*s the signals will be handled and transmitted immediately. With the process priorities we have chosen, the application process is swapped out after a *send* and it is therefore prevented from trying to send more than one signal per scheduled slice. When RLNH is swapped in it will either transmit the signal or queue it in the driver if the media is occupied.

Diagram 4.4.1 shows the total duration of the burst in terms of cycles for our three different implementations of drivers. Unexpectedly the *Single Copy* implementation shows to be the slowest one and the driver without acknowledgement is still fairing well.

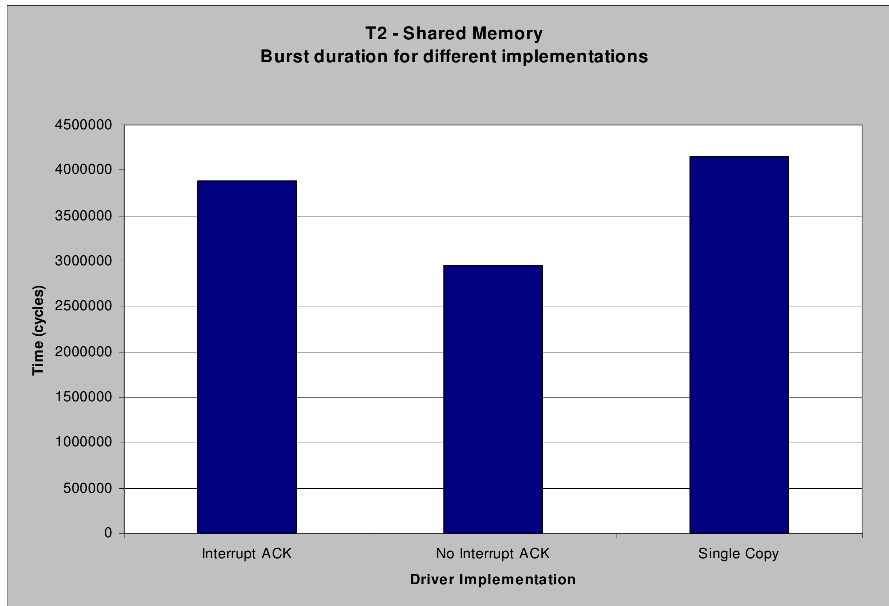


Diagram 4.4.1.1*: Duration of burst of 100 signals for three implementations of shared memory drivers, without cached buffers.*

Additional diagrams, B7 and B8, showing different properties of individual signals in the burst are provided in Appendix B. We see that while signals sent with the *acknowledge interrupt*-driver are highly deterministic, latency for signals sent without acknowledgement quickly climb to a very high maximum at the 10th signal. The latency then falls down linearly to an expected low level. Almost all signals are received by the application process at the end of the burst when all signals have been transmitted. Although the application process is allocated CPU-time regularly it only manages to finish *receive* every 10-20 time slices, each 5000 cycles long, i.e. according to our measurement system, *receive* takes 100 000+ cycles to finish. Each process time slice, as recorded by our system, includes the cost of the context switch as well as the overhead of running the swap hook. Assuming also that both data and instruction caches are dirty and must be initialized to the correct data, it is possible to reach 5000 cycles per slice. As soon as the context switch finishes the application process should be running, but interrupts are enabled again and the application process is swapped out immediately since the sender core has triggered an interrupt telling the receiver that a new signal is waiting. The application process is preempted having executed only a few cycles, if any. The sender is too fast for the receiver in order to achieve a periodic signal rate at the application level.

The diagram of signals using the *Single Copy*-implementation (see diagram B9) shows that the latency increases slightly as the burst progresses, with the difference between delivery-timestamps being kept at a constant low level. Once again the sender is notably faster than the receiver, but because of the ACK-interrupt the sender can't proceed. Instead signals are prepared and queued in the driver for transmission when it is possible. The sender manages to put several signals in the source queue, thus causing an increasing application level latency.

A short decrease in delivery difference times as well as latency can be observed close to the end of the burst for the *Single Copy*-driver. These faster signals are the remaining signals in the queue when the application process has sent the entire burst. Their headers are sent out on the media by the driver ISR, without any interference and cache collisions caused by RLNH and CM code, which is otherwise the case. The phenomenon becomes clearer when RLNH is stopped and we will return to it later.

The reason *Single Copy* is slower processing bursts of signals is that the acknowledge interrupt is delayed until the CM has copied the signal from pool to pool. Usually the interrupt

is triggered by the driver ISR. This is not possible since the sender deallocates the pool buffer when it gets an ACK-interrupt. If the buffer is deallocated it is once again available and could possibly be reallocated and overwritten before the receiver CM has had a chance to copy it. Therefore, a signal sent with *Single Copy* is usually delivered before the sender ISR has been run to handle the acknowledgement; consecutive signals will be delayed.

4.4.1.2 RLNH Stopped

T2 has also been run with RLNH stopped. As a consequence all signals will be available to RLNH once it is started and an increase in latency occurs for each signal since RLNH must process them one at a time.

Diagram 4.4.1.2 shows the burst duration times for different driver implementations when RLNH has been stopped. All three implementations are faster when fed with all signals directly. *Single Copy* is still slower than the other two and the one without acknowledgement is still the fastest.

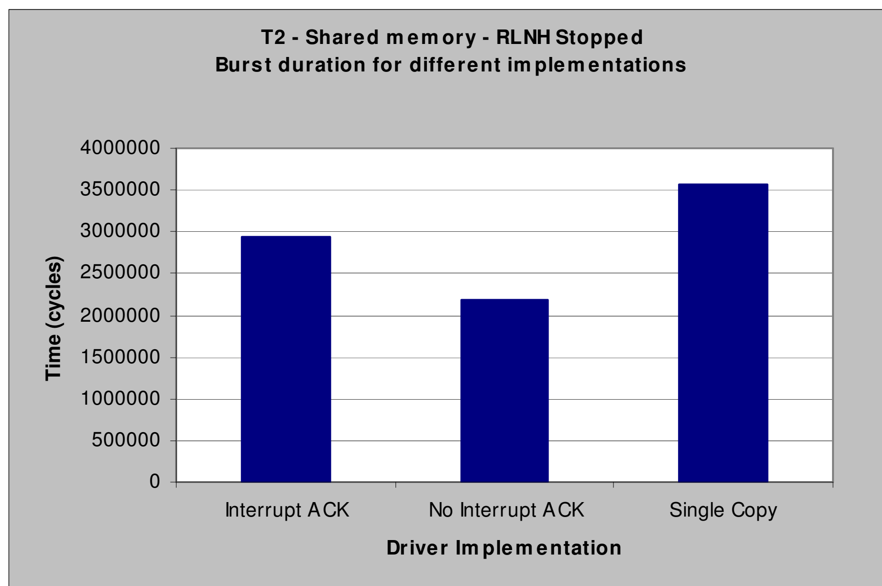


Diagram 4.4.1.2*: Duration of burst of 100 signals for three implementations of shared memory drivers with RLNH first stopped. Uncached buffers.*

Diagrams B10 - B12 are provided in Appendix B, showing latency for individual signals with RLNH stopped. Since all signals are offered to LINX almost at the same time, the latency corresponds to the time of delivery.

In the ACK-implementation the application process starts to suffer from the same symptoms as the NO ACK-driver did with RLNH running, i.e. it it scheduled for execution but immediately preempted before it is able to do anything. The effect is not as drastic though, and all signals are delivered sporadically during the burst.

When acknowledgement is omitted, the signals are all received by the application process at the end of the burst, i.e. all signals are waiting in the process' signal queue until the burst has finished. Similarly to when RLNH was running, the interrupt prevents the application process from executing, but this time the effect is even worse and not a single signal is delivered until the end of the burst.

The *Single Copy*-driver produces an entirely different delivery pattern. The application process is scheduled regularly and signals are deterministically received one at a time. A noticeable decrease in the delivery difference time, as was seen with RLNH running, occurs this time too, but earlier around signal number 70. Figure B13 shows allocation of time to processes at the time immediately before and after the cut in latency (signal 70), indicated by the blue line. Core 0 is sending signals and Core 1 receiving them.

Before the line RLNH still has signals to transmit in its signal queue and is regularly scheduled until after the blue line when the queue has been emptied. The driver queue, however, has been filled with signals that could not be transmitted; the media was occupied. After the blue line, RLNH has nothing to do and is not scheduled. The driver queue is processed by the driver ISR. Less code is running on Core 0 and suddenly the reception of signals on Core 1 accelerates.

There is a clear connection between co-scheduled processes and execution time on the two cores. When RLNH is scheduled on Core 0, the driver ISR and CM are scheduled on Core 1. The execution time of the ISR and CM is much longer than when the idle process is scheduled on core 0 after the blue line. Due to the increased bus traffic caused by RLNH, in combination with small caches, the latency of signals is pushed up 20 000 cycles. The cost of high memory bus load is even clearer in test case 4 where a quad-core is used.

4.4.2 DMA

4.4.2.1 Burst with ACK-interrupt

In diagram 4.4.2.1 the latency and blocking time for each signal in the burst are shown. The first DMA driver has been used, *with* ACK-interrupt. The latency is fairly constant at about 68 kcycles on average, except for the first signal which is considerably faster. Both latency and blocking time show a repetitive pattern every ten signals. Worth noting is that the highest blocking time within each group of ten signals is the signal *after* the one with the highest latency. For example, signal no 7 has high latency and signal no 8 has high blocking time. Total delivery time for all 100 signals is 4 million cycles.

Analysis
Although not completely deterministic, the latency and blocking times are rather constant. The ACK-interrupt acts as a natural flow control, giving the receiver time to actually handle each signal.

The small spikes with higher latency and blocking time are the results of small cache collisions when copying from the receivers DM-buffer to the local pool. The same buffer in the pool is reused, and there are 10 DM-buffers in use, hence the repeating pattern. One of the DM-buffers is mapped to the same cache line as the pool buffer, resulting in ineffective cache usage and increased latency for every tenth signal. This ineffective cache handling results in a high bus load, which slows down the next signal being sent so that this signal will also have a somewhat higher than normal latency. This time the extra cycles are spent on the sending side, manifesting itself as an increase in blocking time.

The spikes in this particular measurement are, although distinct, quite small. The cache collision is likely only partial, with the colliding lines not overlapping entirely. For an extreme case of colliding cache lines, see diagram C4 (appendix C). For an in-depth analysis of this effect, see section 3.2.5.

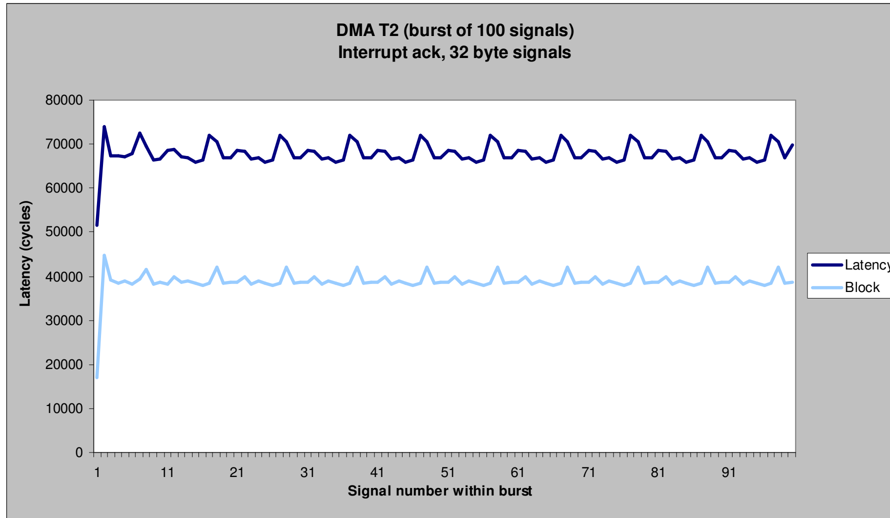


***Diagram 4.4.2.1**: DMA - Latency and blocking time of signals in a burst of 100 signals. Driver with ACK-interrupt*

4.4.2.2 Burst without ACK-interrupt

Latency and blocking time for test T2 with the second DMA-driver, *without* ACK-interrupt, is shown in diagram 4.4.2.2 below. It shows a totally different picture compared to the previous driver, where the latency was fairly constant. Without the ACK-interrupt the receiving core is swamped with messages and the application won't be given time to receive the signals, resulting in all signals being delivered at the same time from the lower layers. The first signal in the burst will therefore have the highest latency (according to the definition of latency in 3.7.1.1). The latency decreases linearly from 3 million cycles down to 30 000 cycles. Average latency is 1.5 million cycles. The blocking time is fairly constant averaging 28 000 cycles. Total delivery time for all 100 signals is 3 million cycles, which is 1 million cycles faster than the other driver.

Analysis
Although the driver without ACK-interrupt showed good performance in test T0, it has clear disadvantages for a large burst like this. The lack of flow control leads to the receiving core being drowned with messages that it can't handle fast enough. The total delivery time of the entire burst is lower though, which can be more important than average latency for some applications.

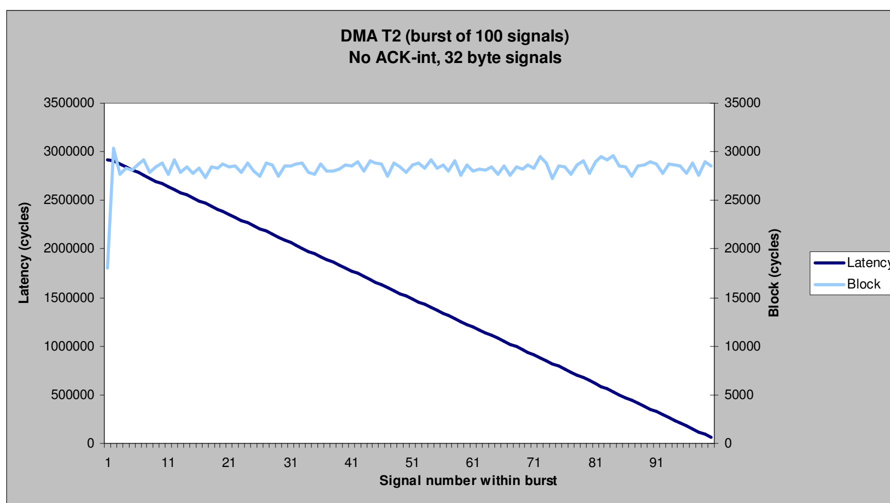


***Diagram 4.4.4.2**: DMA - Latency and blocking time of signals in a burst of 100 signals Driver without ACK-interrupt*

4.5 T3 - Throughput

The throughput test case, T3, was conducted with varying MTU sizes, 512, 1024, 2048, 4096 and 8192 bytes. RLNH was stopped and several signals were queued for transmission. The rate of delivery is used to calculate the throughput. The signals are 65000 bytes large, thus requiring fragmentation for all tested MTU sizes.

The results of the throughput tests should be used with caution. With the receiver being put under heavy load, the application process only gets limited execution time. This not only leads to aperiodic signal reception, but also to greater error bounds in the measurements.

4.5.1 Shared Memory

The measured throughput for shared memory drivers, *Single Copy* excluded, is presented in diagram 4.5.1.1.

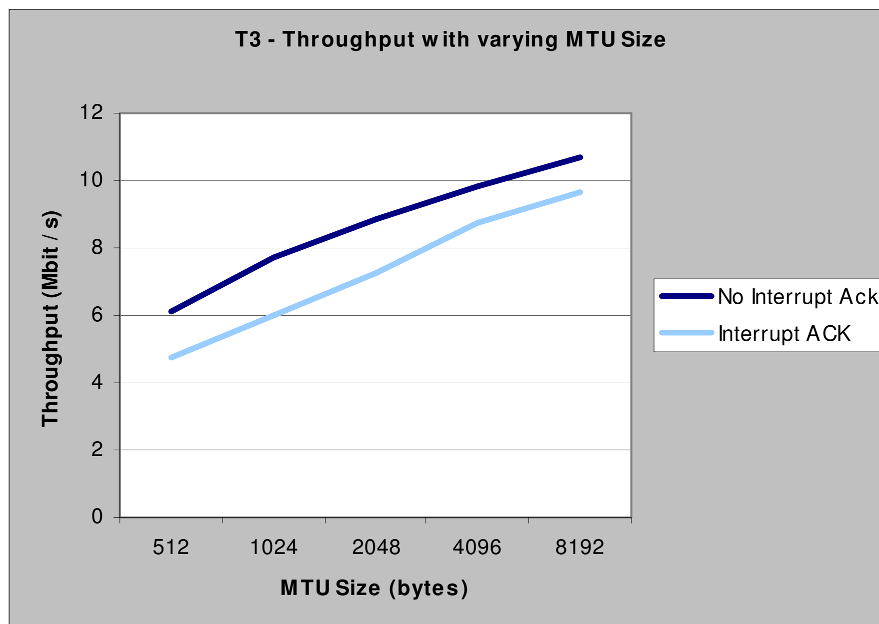


***Diagram 4.5.1.1**: Measured throughput for shared memory drivers with varying MTU sizes.*

The driver without acknowledge interrupt keeps showing good results and the gap down to the acknowledgement driver is close to constant as expected. An important property not seen in the diagram is the stability of the drivers, i.e. if the media and the drivers can keep the throughput once the media becomes saturated. All three driver implementations have this property with very little time difference between deliveries.

Single Copy has no natural MTU and is not plotted in the diagram. Only CM-headers could be affected by the MTU, but are not. When the signal data is copied from one pool to the other, the size of the signal payload is well known, hence there is no need to allocate a temporary buffer with a fixed size. The throughput measured using *Single Copy* is close to 60 Mbits / s, a huge improvement over all other tested implementations.

4.5.2 DMA

4.5.2.1 Throughput when Varying MTU Size

In diagram 4.5.2.1 we see the throughput of the two DMA-drivers for varying MTU sizes. It is clear that the throughput increases with MTU size. When sending signals larger than the MTU they must be fragmented. The overhead associated with setting up a DMA transfer is taken for every fragment, as well as the penalty for the ACK-interrupt. A larger MTU reduces the number of fragments. This leads to less overhead and, hence, faster transfers for larger MTUs.

Comparing the two drivers a very consistent pattern emerges, with both being almost identical in speed across the line. It's only for MTU size 2048 that the driver *without* ACK-interrupt is clearly faster. This is unexpected, as one would think the missing interrupt should give a speed advantage, especially for small MTUs where the number of interrupts is higher.

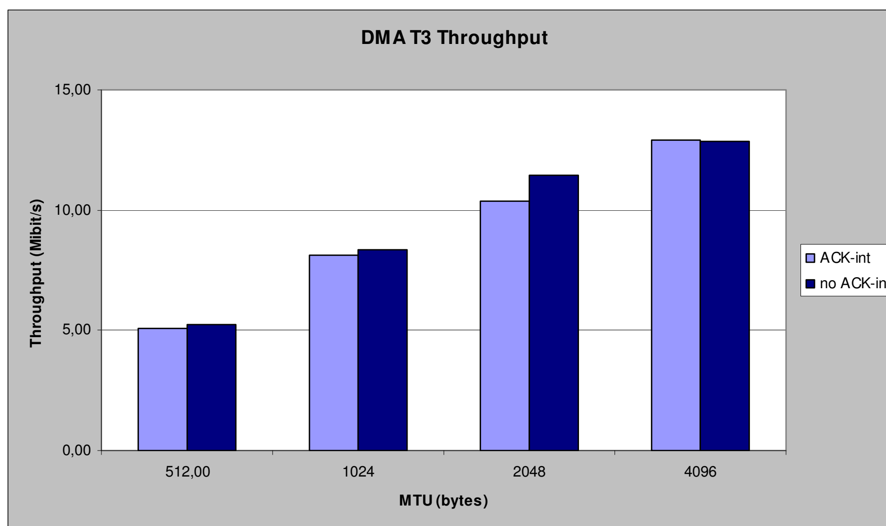


Diagram 4.5.2.1: *DMA - Throughput measured on large (65000 byte) signals for varying MTU sizes (test T3)*

4.5.2.2 Deliver Difference Variance

Diagram 4.5.2.2 shows a disturbing picture. With an MTU size of 2048 bytes the deliver difference is constant, meaning that the signals are delivered very periodically. For the other MTU size this is not at all the case, with a lot of jitter on the delivery. This type of erratic behavior is present for other MTU sizes as well, smaller as well as larger.

Where is this jitter introduced? Examining the delivery from LINX to application shows that signals are actually delivered periodically, i.e. without jitter, regardless of MTU size. The jitter is a result from the application process not managing to *receive* the signal before it is swapped out (because a new fragment has arrived). For some MTU sizes the scheduling and execution times fit nicely and the application manages to finish *receive*. For others this is not the case and *receive* is delayed until the next time the application is scheduled.

Analysis
For a high throughput application that is sensitive to jitter on the receiving side, an MTU size of 2048 would be a good choice for this particular driver. One alternative could be to experiment with a higher priority for the application.

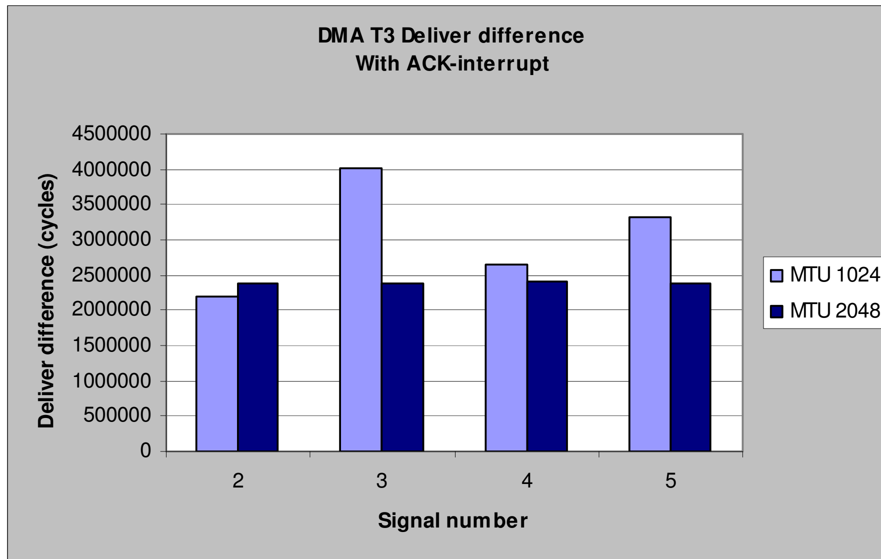


Diagram 4.5.2.2: *DMA – Deliver difference on large (65000 byte) signals for varying signals in the burst (test T3)*

4.6 T4 - Arbitration

Test case 4 takes a step further into the world of quad-cores. Two pairs of cores execute test case 0 independently of each other, using the same memory and bus. The signals are 32 bytes large, with an MTU big enough to prevent fragmentation. A shared memory driver with acknowledge interrupt and uncached buffers has been used.

4.6.1 Shared Memory

The quad-core has smaller caches to fit in the FPGA. Quad-core results are therefore not directly comparable to the dual-core system and as a consequence T0 was rerun on the quad-core using only two cores and a single active link. The new latency figures are plotted in Diagram 4.5.1.1, using *acknowledge interrupt* and uncached buffers. The narrow line is the old values from T0 on the dual-core.

Next, another link was added between the two additional cores in the quad-core system, forming two pairs of cores communicating. With the added bus traffic, contention starts to occur and the requests and replies are transmitted very slowly.

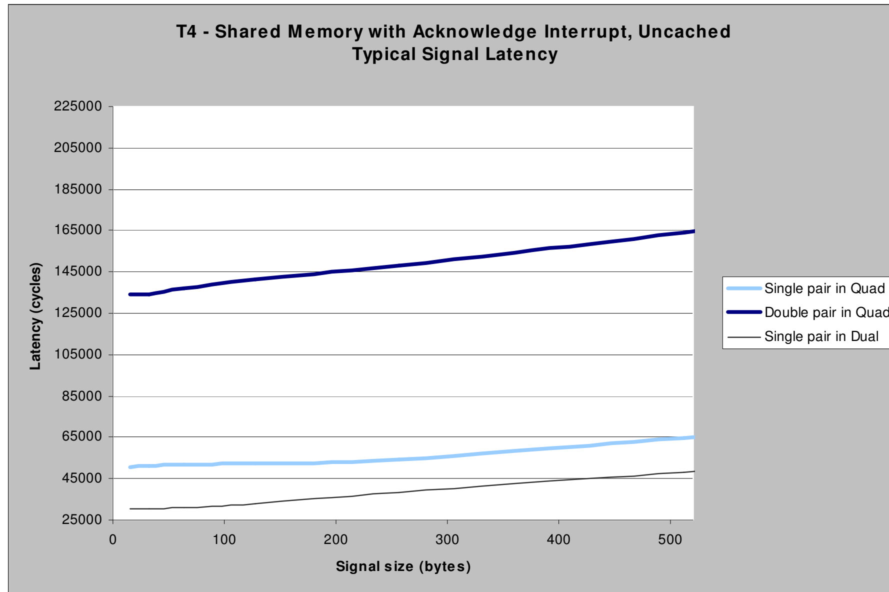


Diagram 4.5.1.1*: Signal latency in a quad-core and dual-core. Two active links result in a more than doubled latency in the quad-core due to increased bus contention. Old dual-core results shown as reference.*

The latency has almost tripled when the second link was added and all layers execute much slower. At first glance, one might think the latency only should have doubled with two active links, since the transmitted data has doubled. With very small caches in the quad-core, contention on the bus increases drastically.

When two cores communicate they individually only compete for the bus with one other core. In the quad-core case a single core competes with three others. Both the risk that arbitration will take place as well as the cost of the arbitration increase as the number of cores grows.

The *System Interconnect Fabric* uses round-robin arbitration, i.e. when four cores request bus access at the same time, the average waiting time for a single core becomes (0+1+2+3)/4 = 1.5 memory access units. In the dual core case this time is (0+1)/2 = 0.5 memory access units. This calculation is greatly simplified and should not be considered to be the full explanation, but rather a hint as to why the latency increases more than expected.

4.7 T5, T6 – Multiple Units

In T5 and T6 a request and reply based traffic pattern is executed on a quad-core, simulating a controller and three DSPs. The tests are intended to expose the driver to a situation in which it should handle transmission and reception of signals on multiple units simultaneously.

T5 sends three request signals, one to each other core in the system, expecting a reply back. In T6, three cores send a request signal each to one core which should reply back. The procedure is repeated at a frequency of 10 Hz for both patterns.

The signals are 32 bytes large, with an MTU large enough to prevent fragmentation. A shared memory driver with acknowledge interrupt and uncached buffers has been used.

4.7.1 T5

The latency of the first request signal sent by Core 0 is low, similar to a signal sent with one active link in T4. The second signal is somewhat slower, and the third even slower. It is clear that when cores 1, 2 and 3 start to wake up and execute to receive the signal, the bus traffic increases and the entire system slows down. The latency of the last request signal, sent to Core 2, is close the three times as high as the first signal sent.

A new, earlier unseen, sequence of events appears. During the acknowledge interrupt of the second request, the driver's ISR on Core 0 first notifies the CM about the successful transmission of a signal and secondly receives and notifies the CM about a received signal – the reply of the first request. Core 1 has, during the execution of the ISR on Core 0, transmitted its reply signal and triggered an interrupt which is handled by Core 0 immediately without any context switch. The same procedure occurs once again for the acknowledgement of the third request signal, with the second reply being delivered to the CM.

Because of the slow transmission of the third request the CM, with lower priority than RLNH, is not scheduled to process the replies until the third request signal has left Core 0. The first and second reply signal is delayed and all reply signals are delivered to the application at the same time when the CM is finally allowed to run.

An interesting conclusion can be drawn. The entire request and reply sequence would have been faster if Core 0 had sent a request and waited for its reply before proceeding with the next request - at least as long as the cores not involved remain idle or execute code from a separate memory.

4.7.2 T6

The outcome of T6 is very similar to T5. Cores 1, 2 and 3 transmit request signals simultaneously to Core 0, whose driver ISR will handle all three units and deliver the signals to the CM in a single execution slice.

Due to the implementation of the driver, the request signals are delivered in the same order as the ID of their respective originator core, i.e. the request from Core 1 is delivered before the request from Core 2, etc.

The first reply has a lower latency than the following due to bus contention as already discussed. When the requests have been sent the requester cores go idle, waiting for the reply. It would not have been faster to send the requests serially and take advantage of lower system load, as was the case with T5.

Taking care of several interrupts during a single ISR execution round has both advantages and disadvantages. The overhead of context switching is reduced, while prioritized processes are blocked and prevented from executing.

5 Discussion

5.1 Hardware Impact

5.1.1 Cache Configurations

The impact of instruction and data caches are large in any system, and within a multicore system it can be huge. Throughout the measurement work of this thesis, whenever something is slowed down or behaves in a strange manner, the first impulse has been to “blame the caches”. Many times, after some analysis, this impulse has been proved true. The hardware used resulted in fairly small caches. Some of the phenomena found and reflected upon may not be noticeable on a system with larger and better caches. Before trying to apply the results presented here on a different hardware, careful analysis must be done to see whether it is applicable at all.

The cache behavior of the Nios II/f is described in section 3.2.4. The following parameters can greatly affect the performance in the types of measurements that have been done.

- Size
- Line size
- Associativity
- Write miss and write hit policy
- Extra levels of cache

The first two parameters can be configured on the Nios II/f, but the others can’t. Modern processors will probably have caches that are at least two-way set-associative. The policy for write misses and hits may also be different. A L2 cache is probably present, reducing the cost of misses in the L1 cache.

Here is a quick comparison of the data cache on the Nios II/f and a modern DSP, the TMS320c64x+ from Texas Instruments (TI). This DSP is a good example of a typical DSP in use in some of the OSEck-systems actually developed. The TI DSP can be configured in many ways, this is just one possibility.

Processor	Size	Associativity	Policy
Nios II	0.5-64 kB L1	1-way	Write allocate
TI c64x+	32 kB L1/2 MB L2	2-way L1/4-way L2	No write allocate

Table 5.1.1: *Comparison of caches on Nios II and TI c64x+.* [3, section 2.12] [22]

As can be seen in the table, the two cache systems don’t have much in common. When a cache line collision occurs in a Nios II/f system (see 3.2.5), resulting in massive performance loss, it will probably pass unnoticed on the TI DSP. The lack of write-allocate, the added associativity and the bigger sizes all help to avoid the problem with cache collision.

5.1.2 Memory Configurations

All tests have been performed with code and data in SDRAM. The DE2-board uses an 8 Mbyte SDRAM with a 16-bit data width. Both the Nios II/f processors and the memory are clocked at 50 MHz. The synchronous clock reduces the effect of a cache miss, but misses in the instruction cache still cripple performance, especially when several cores are active on the memory bus.

Having the critical parts of the code in an on-chip memory would be beneficial in a multicore system like this, eliminating the negative impact of misses in the instruction cache. It is

impossible, however, to determine if the execution patterns would have been different if the measurements had been conducted with on-chip memory. Some of the results in this thesis can only be expected in an environment with shared external memory.

5.2 Connection Managers (CM)

The point-to-point connection manager has been used throughout this thesis. It is a generic "plug and play" design for reliable media, quickly providing developers with a working solution. It can be questioned whether PTP-CM should be used in time critical multicore systems. For small request and reply signals, most of the latency is caused by execution in different layers of LINX rather than doing the actual data transmission. The signal size scalability is also an issue, due to extra *memcpy*s.

A lot of the functionality provided by the CM could either be moved upwards to RLNH, such as connection heartbeats, and other functionality moved downwards into the drivers, for example payload fragmentation. Private headers within the CM protocol force the CM to do an extra *memcpy* when user data is received, since it must remove the headers from the user data. If the headers used to set up and maintain the connection were to be changed to regular signals sent between virtual CM-processes on each side of the connection, similarly to how RLNH is implemented, the *memcpy* could be removed. All data transmitted by drivers would be signals and they could be delivered directly from ISR context on the receiver, without any need for a CM RX-process. The time in interrupt context would increase negligibly, while saving overhead for context switching to a CM RX process that does memcpy to remove headers from user data.

If the UDATA-header is removed, the entire signal must be transmitted, including the hidden administration block at the beginning of the signal containing information about source and destination address, which is otherwise embedded in the CM-header. Passing knowledge of the transmitted data down through the layers, possible down to the driver, is usually not considered politically correct. Flattening a protocol stack also removes a lot of the reusability as well as portability. It proves however, that further refinement of the CM and driver concept is possible to increase performance. Shortening the delivery time on the receiver should be considered desirable to alleviate one of the problems discussed next — receiver livelock.

5.3 Drivers

It has already been concluded that systems suffer from receiver livelock [11] at high signaling frequencies causing the application process to starve, especially when acknowledge interrupts are omitted. A sender trying to transmit data too quickly will cause an amount of interrupts on the receiver high enough to prevent the execution of application code. As a consequence, the application level latency is severely degraded.

Receiver livelock is a common problem with high network loads in interrupt-driven operating systems. Many gigabit Ethernet adapters and drivers implement *interrupt coalescing* [12] as a countermeasure to prevent starvation. Interrupt coalescing itself only shifts the point where livelock occurs and there is no guarantee against livelock. Coalescing will increase the low load signal latency, while lowering the high load traffic latency. There is no simple way to achieve low latency in both cases.

Other common techniques to counteract receiver livelock are to [11]:
- Drop packets.
- Temporarily disable the interrupt when the receiver input queue becomes full.
- Let the kernel disable the interrupt when the driver ISR has reached a certain time quota.

The only feasible technique today, for a LINX driver in OSEck, is to drop packets, which

would require a CM for unreliable media. PTP-CM is designed for reliable media and can not be used if packets are to be dropped.

The chosen application process priority is also to blame for the livelock encountered in the thesis. If the priority had been higher than the priority of the CM RX-process, the application would have been allowed to run between interrupts as signals were delivered. This does not, however, rule out the possibility of livelock in multicore systems, where multiple units can trigger interrupts.

In the use cases looked at in this thesis, the signaling frequency is mostly low except for the occasional short burst. Receiver application starvation will not be a big problem, regardless of driver implementation, as long as the signal traffic conforms to the investigated use cases. However, in the development of future use cases the problem should be considered.

One alternative is to implement flow control at the application level. By using a sliding window approach the sender will have to give the receiver time to receive the signals and acknowledge them [8, section 3.1.2].

5.4 Comparison of Shared Memory and DMA

When designing a multicore system that will use IPC between the cores, the designer will have to make a choice between traditional shared memory and DMA. To compare the two types, knowledge of the traffic patterns is important. This assumes that the system has been designed with DMA capabilities. If the DMA controller has to be added specifically for the IPC, the cost of the added hardware must also be considered. In a Nios II system, an SG-DMA controller can, depending on configuration, use almost as many FPGA resources as a CPU. Adding several controllers can thus be costly and may require migrating the hardware design to a larger FPGA.

5.4.1 Low Frequency Traffic

In a system with low signaling frequency test case 0 is most relevant. Diagram 5.4.1-1 shows a comparison of the various drivers for increasing signal size. For small signals the shared memory drivers show lower latency, while the DMA drivers are better for larger signals. The breakpoint is somewhere in the region of 160 bytes. The drivers *without* ACK-interrupt show a generally better performance for this traffic type, as discussed in section 4.2.

The shared memory driver with single copy has the best performance of all drivers – for all signal sizes. To use this driver an alternative CM must be used, as explained in 4.2.1, the standard PTP-CM cannot be used.

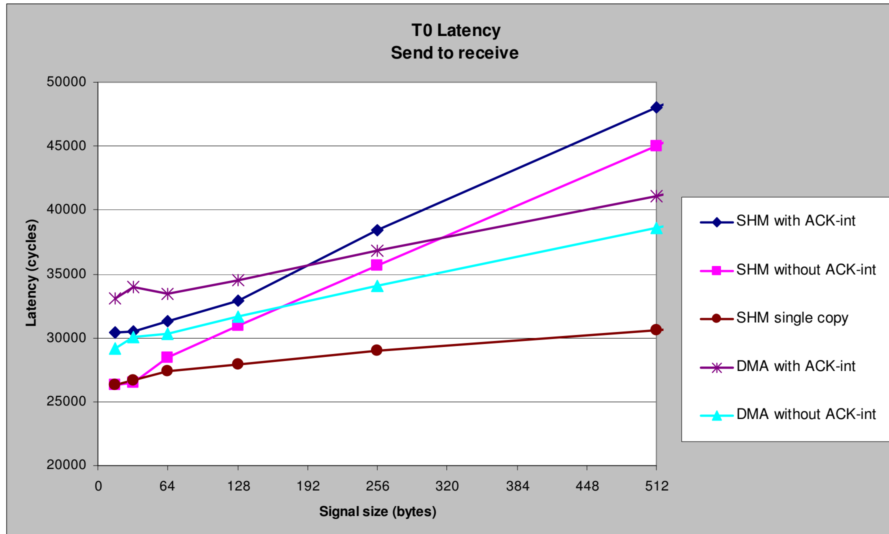


Diagram 5.4.1-1: *Latency comparison of all the drivers (test T0)*

The amount of CPU time (load) required to send and receive the signals is also interesting. Table 5.4.1-2 shows the amount of cycles required to send and receive a 128 byte signal, as well as the total load on both cores. The drivers without ACK-interrupt need less CPU resources. Shared memory and DMA are pretty equal, with both needing about the same amount of cycles in total. The single copy driver uses ACK-interrupt.

Driver	Load sender	Load receiver	Load total
SHM ACK-int	22413	24509	46922
SHM no ACK-int	16439	18411	34850
SHM single copy	22840	20337	43177
DMA ACK-int	23981	24044	48025
DMA no ACK-int	18615	15446	34061

Table 5.4.1-2: *CPU load in T0 for a 128 byte signal*

5.4.2 High Frequency Traffic

In a system where signaling occurs more frequently, either at a constant high rate or as bursts, performance under heavy load is more interesting than average latency in an unloaded system. Test case 2, with its burst of 100 signals, does exactly this.

Diagram 5.4.2.1 is a comparison of the different drivers, showing the total delivery time of the entire burst of 100 signals. This time is measured from *send* of the first signal to *receive* of the 100th signal. The drivers with ACK-interrupt need about 4 million cycles to deliver the entire burst, while the ones without ACK-interrupt do the job in just 3 million cycles. The single copy-driver, which was the fastest implementation for low frequency traffic, ends up at last place in this scenario. For an explanation of this, see 4.4.1.1.

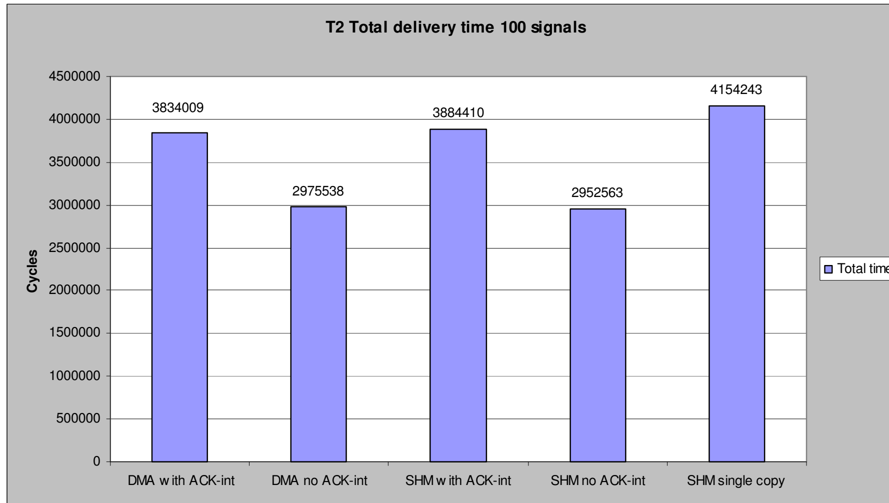


Diagram 5.4.2.1: *Total delivery time of a burst of 100 signals (test T2)*

The average latency of each signal in the burst, shown in diagram 5.4.2.2, is very high for drivers without acknowledge interrupt. This is in sharp contrast to the previous diagram where these drivers were the fastest. The main reason for the high latencies is receiver livelock, previously discussed.

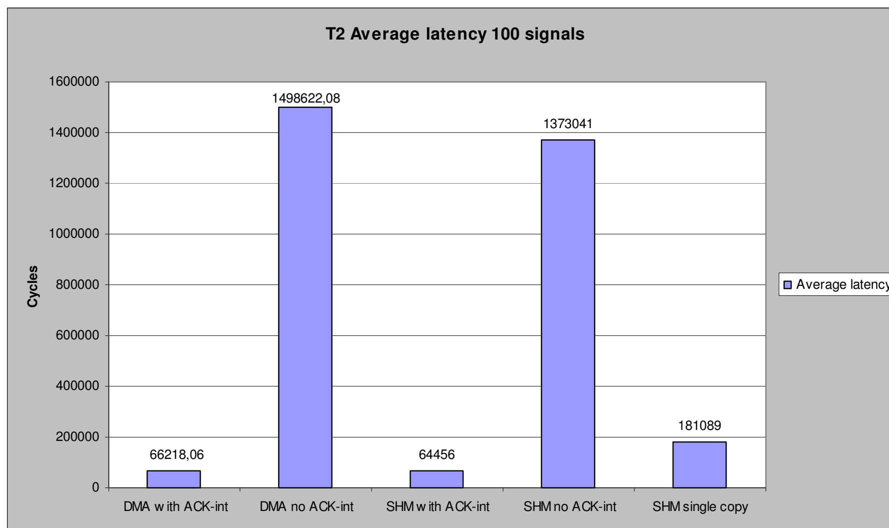


Diagram 5.4.2.2: *Average latency per signal in a burst of 100 signals (test T2)*

5.4.3 High Throughput Traffic

The measured throughput for the different drivers is shown in diagram 5.4.3.1. A straightforward conclusion is that the number of times data is copied is directly related to the measured throughput. The *Single Copy*-driver does one *memcpy* and offers the best throughput, followed by DMA-drivers doing three, and finally generic shared memory drivers doing four. It should be mentioned that the PTP-CM has been optimized by Enea during the course of this thesis. The *memcpy* previously used when fragmentation is needed has been removed. Systems using the latest version should offer better throughput than what is presented here.

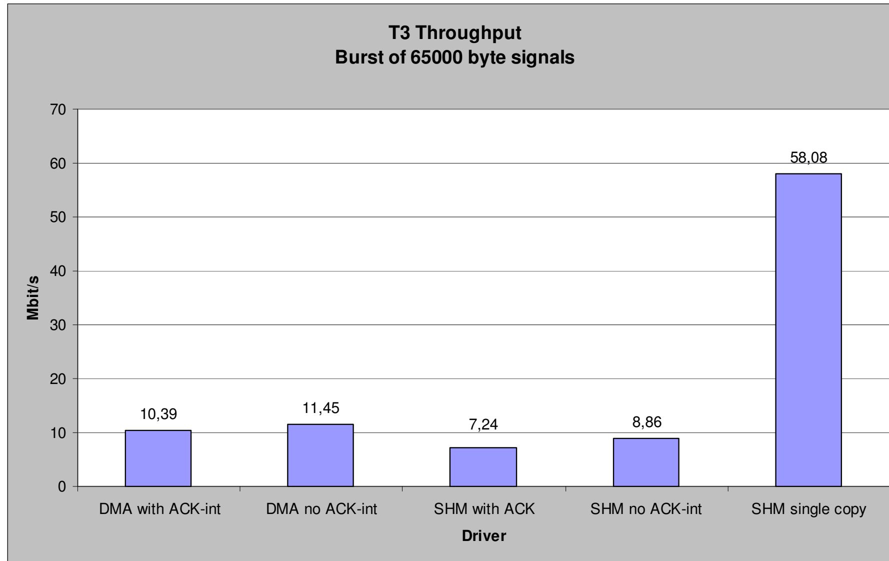


Diagram 5.4.3.1: *Throughput with MTU=2048 bytes*

The DMA-drivers, as well as shared memory without ACK-interrupt, once again suffer from receiver livelock. The signals are delivered irregularly, leading to varying latency and therefore inconsistent throughput during the burst for these three drivers, resulting in a higher error margin than usual.

6 Conclusions and Future Work

For conclusions and future work discussion regarding phase one, see section 2.8.

6.1 Conclusions – Phase Two

This thesis has been about investigating and evaluating LINX characteristics and performance in a multicore environment. While LINX and the drivers evaluated have shown generally good qualities, room for further improvement exists. The use cases relevant for LINX and OSEck today don't expose any bad habits in need of immediate attention.

Several different variants of drivers have been implemented and evaluated. The measured performance turned out as expected. Shared memory provides the best latency for small signals, while DMA performs better as signals grow. Using interrupts to acknowledge received signals introduces a slight overhead. The natural flow control induced by the interrupt reduces the risk of receiver livelock at application level. This has a very positive effect on average latency for bursty traffic.

The single most important factor affecting performance is, without doubt, memory cache configuration. Instruction and data caches in the Nios II soft processor are directly mapped. This, in combination with limited FPGA memory, has had a huge impact on the results.

With minor modifications to the LINX architecture a large performance improvement in terms of latency, scalability and throughput has been observed. The point-to-point connection manager used was designed to be as generic as possible, thus sacrificing performance to gain reusability. In a multicore system with high performance requirements it is recommended using a connection manager optimized for the specific shared memory or DMA driver. Optimizations reducing the code footprint as well as memory accesses are highly beneficial as the number of cores grows and caches are small.

It is difficult to conclude whether the characteristics identified during this thesis can be directly applied to systems used in final production. The simple nature of the Nios II processor has brought aspects to light that, with a more advanced system, may have passed unnoticed. It's far from certain, though, that more advanced systems won't experience similar characteristics once they are pushed to their limits in a similar way. For example, a receiver livelock situation might surface in any system with extreme load. Increasing the number of cores in a system with more LINX connections ultimately causes a higher rate of interrupts. It is possible that the phenomena we have encountered with two cores could appear in a high-end system going from, say, four cores to eight.

6.2 Future Work – Phase Two

Further investigation with multiple cores and units should be conducted. A larger set of traffic patterns in these environments could reveal new phenomena that need to be addressed. Running the tests on more advanced hardware would be interesting, where the large impact of small, directly mapped caches could be avoided. Another possible topic of analysis is to try different types of memory configurations. Performance of a DMA driver for a more advanced controller could be looked at.

The suggested connection manager mentioned in the discussion could be implemented and evaluated, as well as different techniques to counteract receiver livelock.

The Analyser Tool developed during the thesis has been a great help in understanding the

behavior of the system. This tool could be expanded upon and used in several other areas where concurrent events are difficult to visualize. The evaluation platform itself, the DE2 board, is a good basis for many other future multicore investigations.

7 References

[1] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection, 1980. In *IEEE Transactions on Communications*, 28(4):425–432.

[2] Altera. Nios II Software Developer's Handbook, Oct 2007. Document Id: NII5v2-7.2

[3] Altera. Nios II Processor Reference Handbook, Oct 2007. Document Id: NII5v1-7.2

[4] Altera. Cyclone II Device Family Overview, Apr 2007. Document Id: CII5v1-3.2

[5] Altera. Scatter-Gather DMA Controller Core, Oct 2007. Document Id: QII55003-7.2.0

[6] Altera. System Interconnect Fabric for Memory-Mapped Interfaces, Oct 2007. Document Id: QII54003-7.2.0.

[7] Enea. OSEck Kernel Reference Manual, 2007. R3.3.

[8] Enea. OSEck Kernel User's Guide, 2007. R3.3.

[9] Enea. OSEck Timeout Server User's Guide, 2007. R3.3.

[10] Enea. LINX for OSEck User's Guide and Reference Manual, 2007.

[11] Jeffrey Mogul, K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel, January 1996. In *Proceedings of the USENIX 1996 Annual Technical Conference.*

[12] Chang, Xiaolin Muppala, Jogesh K. Zou, Pengcheng Li, Xiangkai. A Robust Device Hybrid Scheme to Improve System Performance in Gigabit Ethernet Networks, 2007. In *32nd IEEE Conference on Local Computer Networks.* ISBN: 978-0-7695-3000-0.

[13] Daniel Staf. Adaptation of OSEck for an FPGA-Based Soft Processor Platform, 2007. In *Linköping University Press*. ISRN: LITH-ISY-EX--07/3962—SE

[14] Gschwind, M. Hofstee, H.P. Flachs, B. Hopkins, M. Watanabe, Y. Yamazaki, T. Synergistic Processing in Cell's Multicore Architecture, April 2006. In *IEEE Micro.* ISSN: 0272-1732.

[15] W.J. Dally, B. Towles. Principles and Practices of Interconnection Networks. *Morgan Kaufmann Publishers*, 2004. ISBN: 0-12-200751-4.

[16] OpenMPI website. http://www.open-mpi.org/ , November 2007.

[17] Brian W. Kernighan, Dennis M. Ritchie, March 1988. The C Programming Language, 2nd ed., *Prentice Hall*. ISBN 0-13-110362-8.

[18] Eclipse Foundation. http://www.eclipse.org/ , November 2007.

[19] Observer Effect on Wikipedia. November 15, 2007. Revision ID: 171106556 http://en.wikipedia.org/w/index.php?title=Observer_effect&oldid=171106556

[20] Enea internal. http://twiki.enea.se/twiki/bin/view/EETeng/MasterThesisMulticore

[21] GNU Compiler Collection. http://gcc.gnu.org/ , November 2007.

[22] Texas Instruments. TMS320C64x+ DSP Cache User's Guide,
http://focus.ti.com/lit/ug/spru862a/spru862a.pdf, October 2006. Document ID: SPRU862A

APPENDIX A

```
# Nios II (icache=16k line=32b, dcache=16k line=32b) libosh.a
1.1  alloc from pool                    : 154 cycles
1.2  alloc from free-list               : 109 cycles
1.3  alloc_nil from pool                : 279 cycles
1.4  alloc_nil from free-list           : 106 cycles
1.5  s_alloc from pool                  : 654 cycles
1.6  s_alloc from free-list             : 118 cycles
1.7  s_alloc_nil from pool              : 497 cycles
1.8  s_alloc_nil from free-list         : 123 cycles
1.9  free_buf                           : 61 cycles
1.10 send without swap                  : 372 cycles
1.11 send_w_s without swap              : 256 cycles
1.12 receive on existing signal         : 281 cycles
1.13 receive one selected existing signal: 137 cycles
1.14 receive_w_tmo on existing signal   : 333 cycles
1.15 sender                             : 32 cycles
1.16 restore                            : 96 cycles
1.17 addressee                          : 30 cycles
1.18 get_ticks                          : 8 cycles
1.19 current_process                    : 11 cycles
1.20 sigsize                            : 54 cycles
1.21 set_fsem                           : 17 cycles
1.22 wait_fsem on signaled fsem         : 34 cycles
1.23 signal_fsem                        : 142 cycles
1.24 get_fsem                           : 62 cycles
1.25 create_sem                         : 152 cycles
1.26 wait_sem on signaled sem           : 26 cycles
1.27 signal_sem                         : 29 cycles
1.28 get_sem                            : 21 cycles
1.29 kill_sem                           : 181 cycles
1.30 get_pri                            : 20 cycles
1.31 set_pri                            : 316 cycles
1.32 wake_up                            : 56 cycles
1.33 reset_pool                         : 289 cycles
1.34 stop non-blocked process           : 280 cycles
1.35 stop a blocked process             : 105 cycles
1.36 start a blocked process            : 41 cycles
2.1  send - swap - receive              : 510 cycles
2.2  signal_fsem - swap - wait_fsem     : 314 cycles
2.3  signal_sem - swap - wait_sem       : 217 cycles
2.4  start - swap - stop                : 302 cycles
2.5  set_pri - swap                     : 359 cycles
3.1  os interrupt                       : 488 cycles
3.3  send - wakeup                      : 289 cycles
3.4  signal_fsem - wakeup               : 162 cycles
3.5  interrup overhead                  : 337 cycles
3.6  interrupt - small context          : 292 cycles
3.7  interrupt - big context            : 152 cycles
```

Table A1: *Execution time in cycles for various OSEck system calls, using a kernel with hooks. Both instruction and data caches are 16kB.*

```
# Nios II (icache=16k line=32, dcache=0k) libosh.a
1.1  alloc from pool                      : 471 cycles
1.2  alloc from free-list                 : 376 cycles
1.3  alloc_nil from pool                  : 684 cycles
1.4  alloc_nil from free-list             : 371 cycles
1.5  s_alloc from pool                    : 771 cycles
1.6  s_alloc from free-list               : 379 cycles
1.7  s_alloc_nil from pool                : 791 cycles
1.8  s_alloc_nil from free-list           : 406 cycles
1.9  free_buf                             : 262 cycles
1.10 send without swap                    : 647 cycles
1.11 send_w_s without swap                : 586 cycles
1.12 receive on existing signal           : 657 cycles
1.13 receive one selected existing signal: 503 cycles
1.14 receive_w_tmo on existing signal     : 1171 cycles
1.15 sender                               : 81 cycles
1.16 restore                              : 277 cycles
1.17 addressee                            : 79 cycles
1.18 get_ticks                            : 31 cycles
1.19 current_process                      : 58 cycles
1.20 sigsize                              : 101 cycles
1.21 set_fsem                             : 43 cycles
1.22 wait_fsem on signaled fsem           : 111 cycles
1.23 signal_fsem                          : 264 cycles
1.24 get_fsem                             : 108 cycles
1.25 create_sem                           : 507 cycles
1.26 wait_sem on signaled sem             : 80 cycles
1.27 signal_sem                           : 83 cycles
1.28 get_sem                              : 50 cycles
1.29 kill_sem                             : 498 cycles
1.30 get_pri                              : 67 cycles
1.31 set_pri                              : 987 cycles
1.32 wake_up                              : 106 cycles
1.33 reset_pool                           : 507 cycles
1.34 stop non-blocked process             : 618 cycles
1.35 stop a blocked process               : 252 cycles
1.36 start a blocked process              : 171 cycles
2.1  send - swap - receive                : 1927 cycles
2.2  signal_fsem - swap - wait_fsem       : 1461 cycles
2.3  signal_sem - swap - wait_sem         : 1449 cycles
2.4  start - swap - stop                  : 1442 cycles
2.5  set_pri - swap                       : 1531 cycles
3.1  os interrupt                         : 1094 cycles
3.3  send - wakeup                        : 1000 cycles
3.4  signal_fsem - wakeup                 : 685 cycles
3.5  interrup overhead                    : 1516 cycles
3.6  interrupt - small context            : 1214 cycles
3.7  interrupt - big context              : 1093 cycles
```

Table A2: *Execution time in cycles for various OSEck system calls, using a kernel with hooks. No data cache, instruction cache is 16kB.*

APPENDIX B

Various measurement results for shared memory from Phase Two.

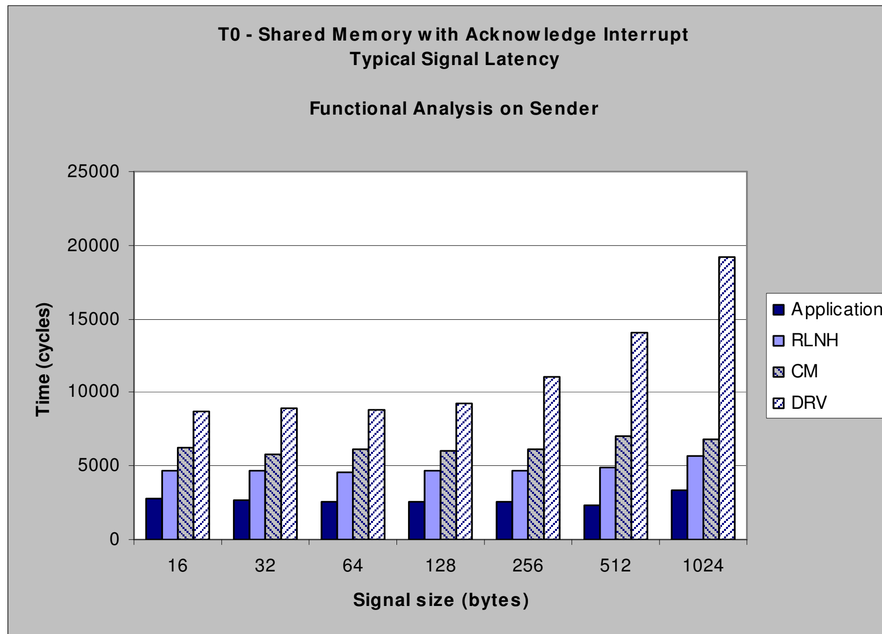


Diagram B1: *The time spent in different functional layers of LINX on the sender during the transmission of a signal. The execution time in the driver increases with signal size due to a memcpy.*

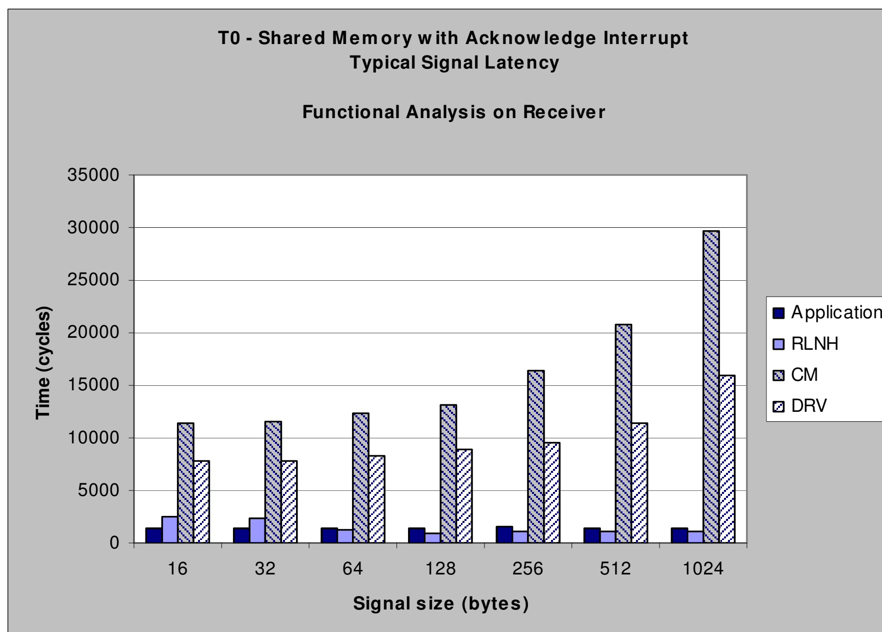


Diagram B2: *The time spent in different functional layers of LINX on the receiver during the reception of a signal. The execution time in CM as well as driver increases with signal size due to memcpy.*

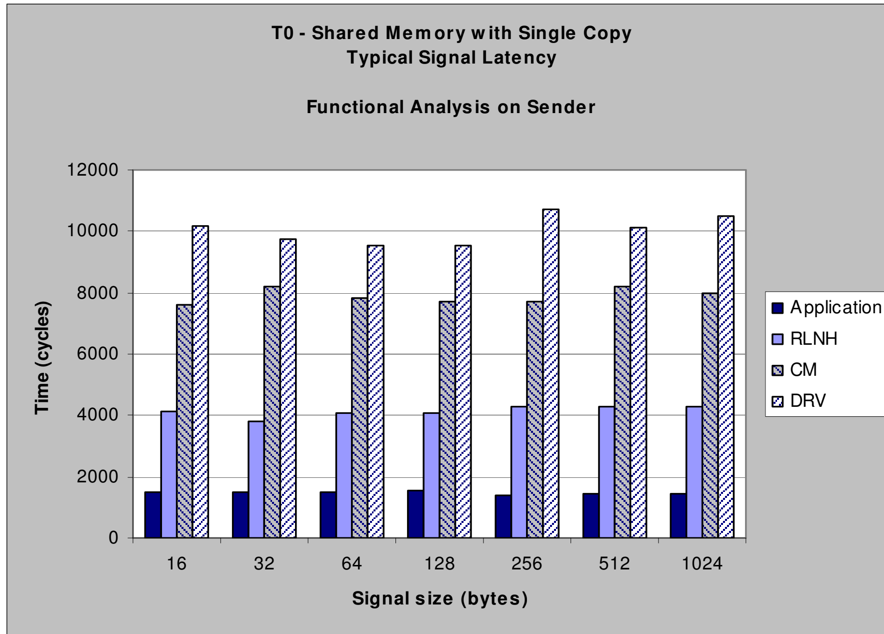


***Diagram B3**: The time spent in different functional layers of LINX on the sender during transmission of a signal. All execution times are constant when using Single Copy.*

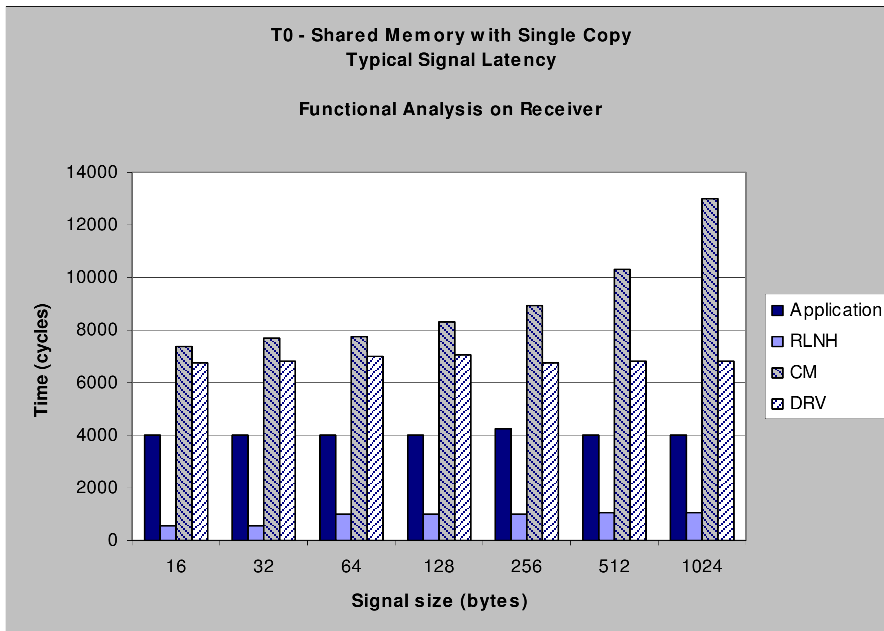


***Diagram B4**: The time spent in different functional layers of LINX on the receiver during reception of a signal. The CM exectution time increases with signal size due to the only memcpy performed.*

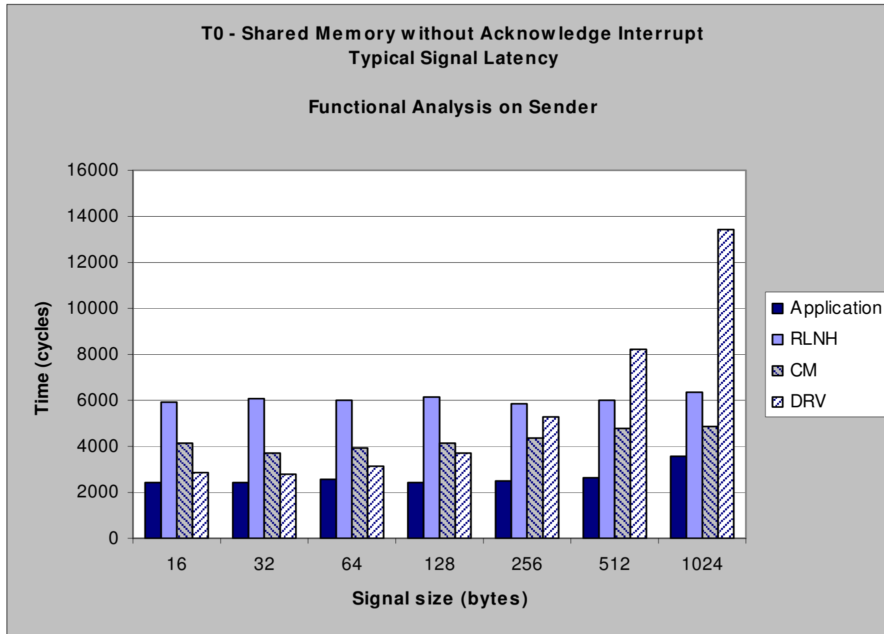


Diagram B5: *Time spent in different functional layers of LINX on the sender during transmission of a signal.*

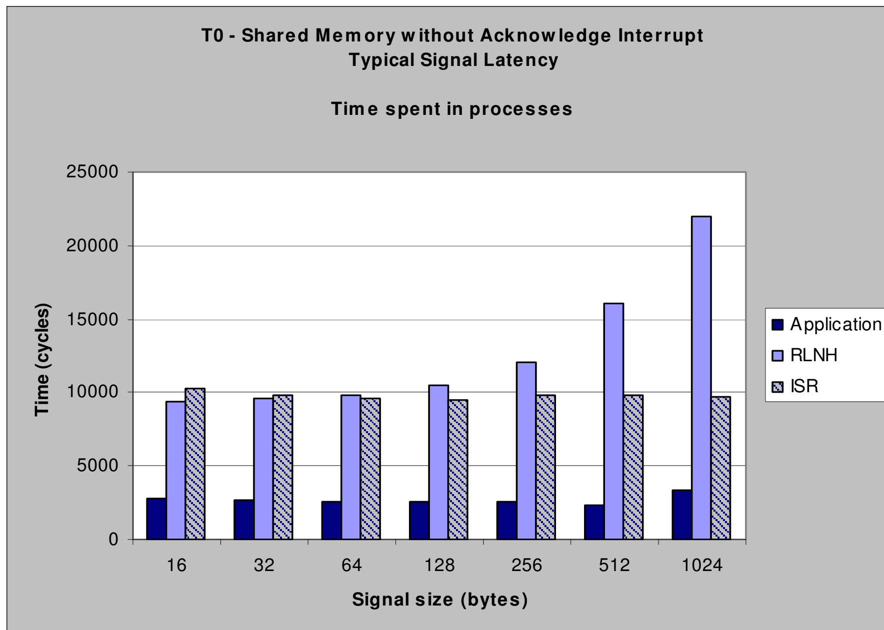


Diagram B6: *The time spent in different processes on the sender during transmission of a signal. Visualizes, together with B5, that signals are transmitted from the context of RLNH.*

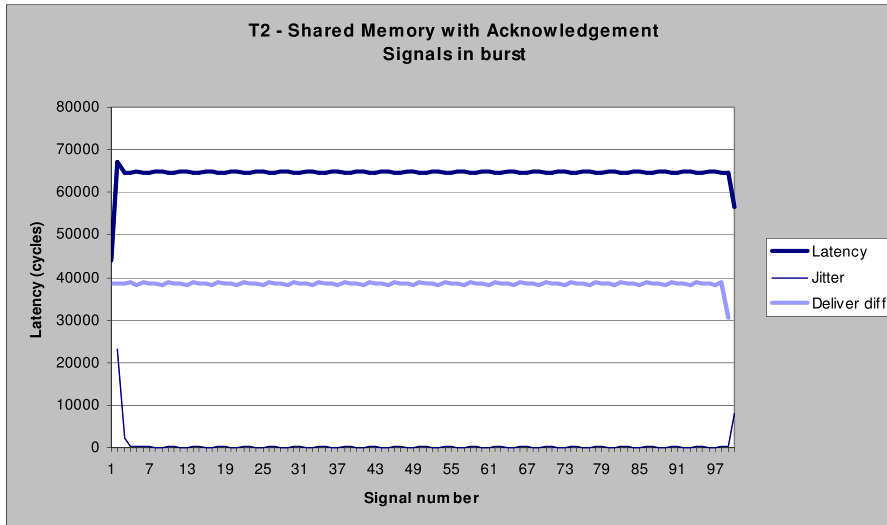


Diagram B7: *Latency, jitter and difference between delivery timestamps for individual signals in a burst of 100. This driver implementation shows a highly deterministic signal latency during burst.*

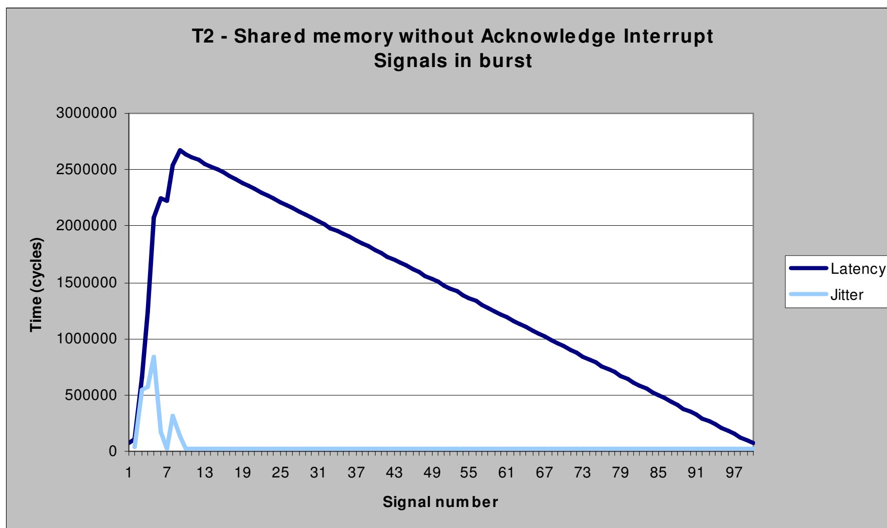


Diagram B8: *Latency and jitter for individual signals in a burst of 100. Almost all signals are received after the last signal has been transmitted. The application process is livelocked from signal nr 10.*

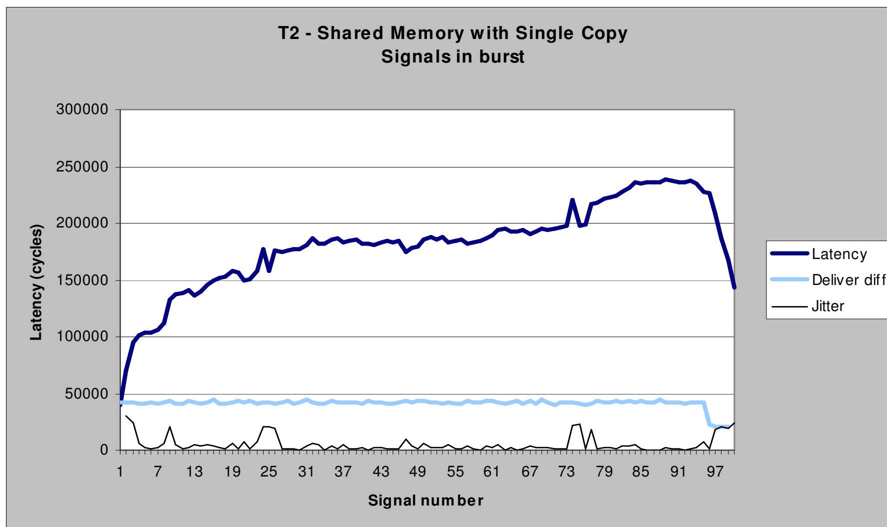


Diagram B9: *Latency, jitter and difference between delivery timestamps for individual signals in a burst of 100. The sender is faster than the receiver and signals are queued, waiting for transmission. The latency increases as the queue grows.*

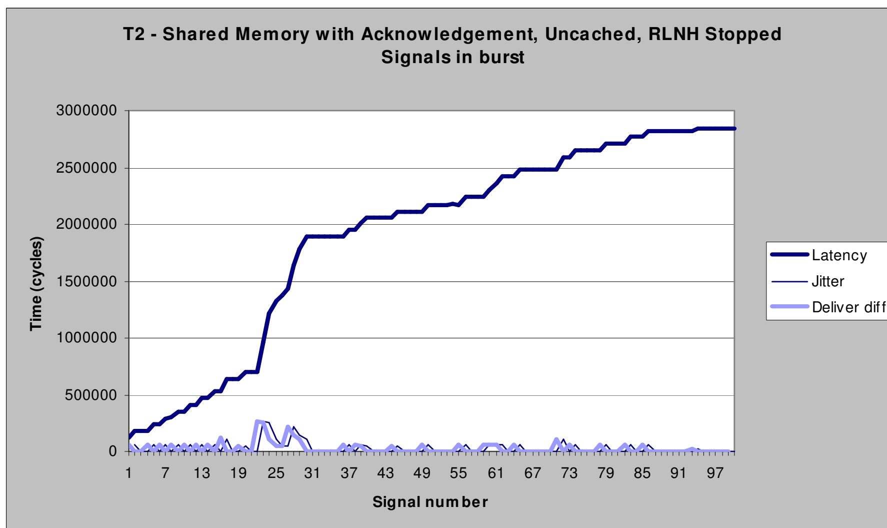


Diagram B10: *Latency, jitter and difference between delivery timestamps for individual signals in a burst of 100 when RLNH is stopped. The application process has difficulty executing long enough to handle received signals, a livelock.*

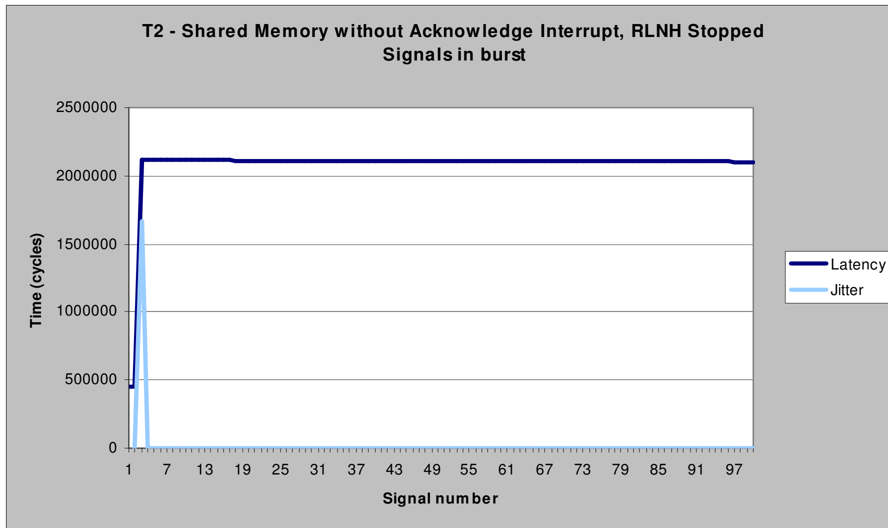


Diagram B11: *Latency and jitter for individual signals in a burst of 100 when RLNH is stopped. Receiver livelock causes all signals to be delivered to the application when the entire burst has been received.*

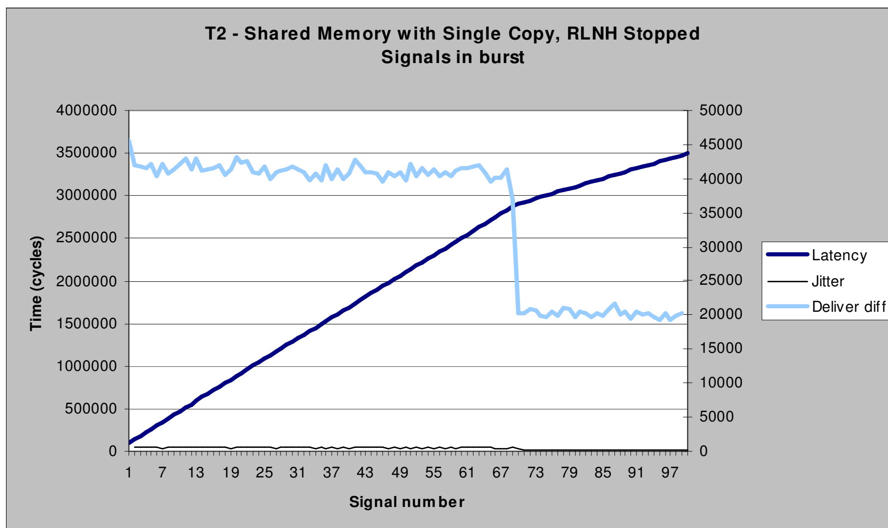


Diagram B12: *Latency, jitter and difference between delivery timestamps for individual signals in a burst of 100 when RLNH is stopped. The increase in latency as the burst progresses is highly deterministic. The deliver frequency doubles around signal 70 when all signals in the burst have been processed by RLNH and are queued in the driver.*

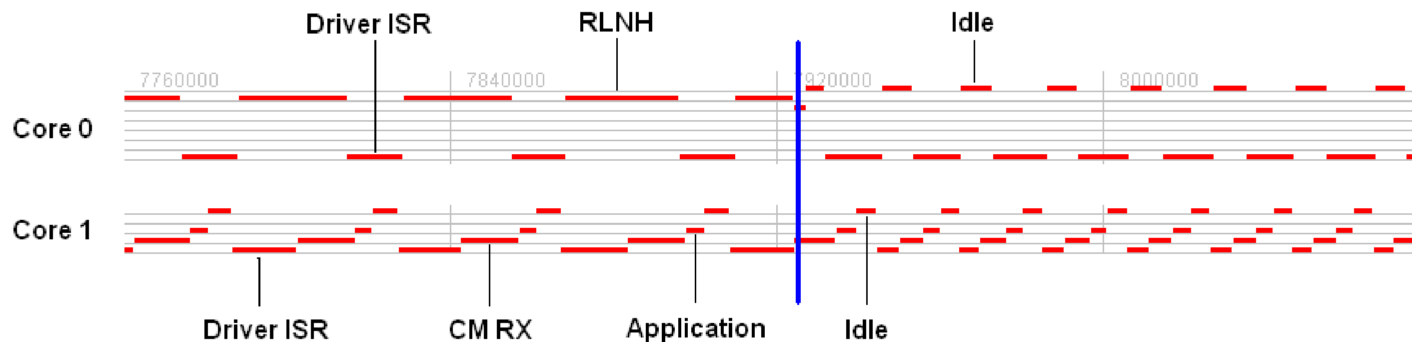


Diagram B13: *Process scheduling in a dual core system during bursty traffic using the Single Copy shared memory driver. Signals are processed and transmitted directly from the driver source queue beyond the blue line. With RLNH not executing and accessing memory, bus contention disappears and the system performs better with signals being delivered faster.*

APPENDIX C

Various measurement results for DMA from Phase Two.

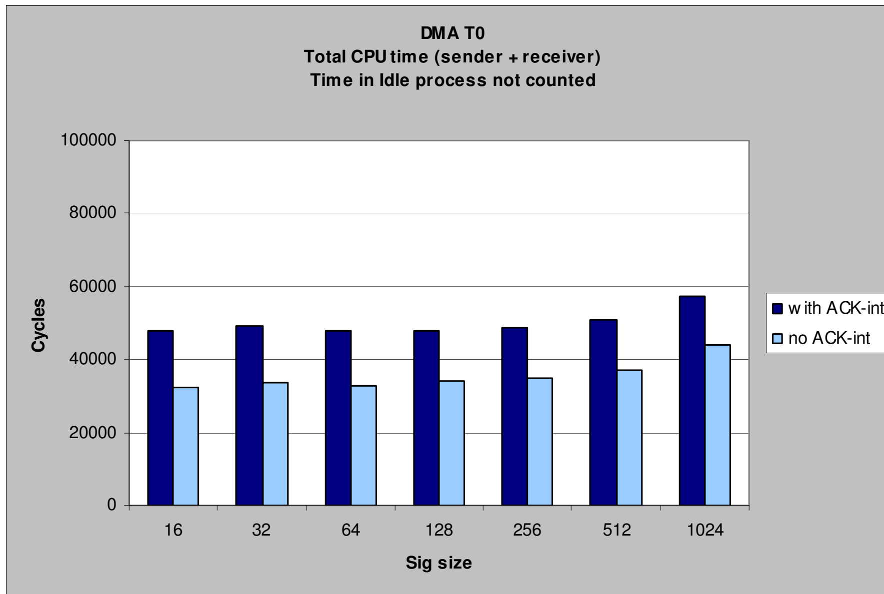


Diagram C1: *DMA – Total CPU time of sender and receiver for test T0. We see that the total CPU-time increases slightly with larger signal sizes for both drivers. The main reason for this is not the DMA-transfer, but rather the memcpy performed by the CM on the receiving side. The driver without ACK-interrupt consumes considerably less CPU cycles for all sizes. Any time spent in the idle process is not included.*

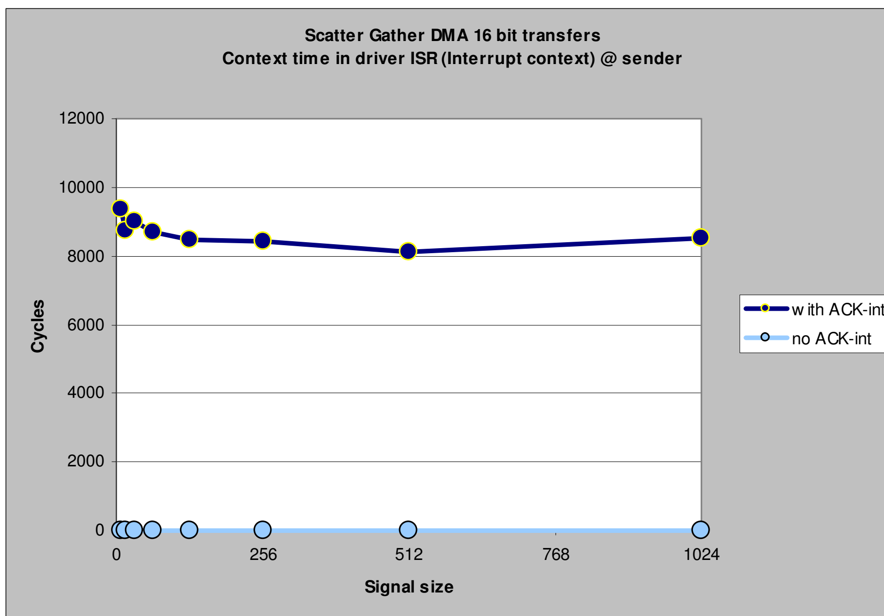


Diagram C2: *DMA – Senders time in interrupt context for test T0. The driver without ACK-interrupt does not use any interrupts on the sending side, hence the zero values. Interrupt time is quite constant for the other driver.*

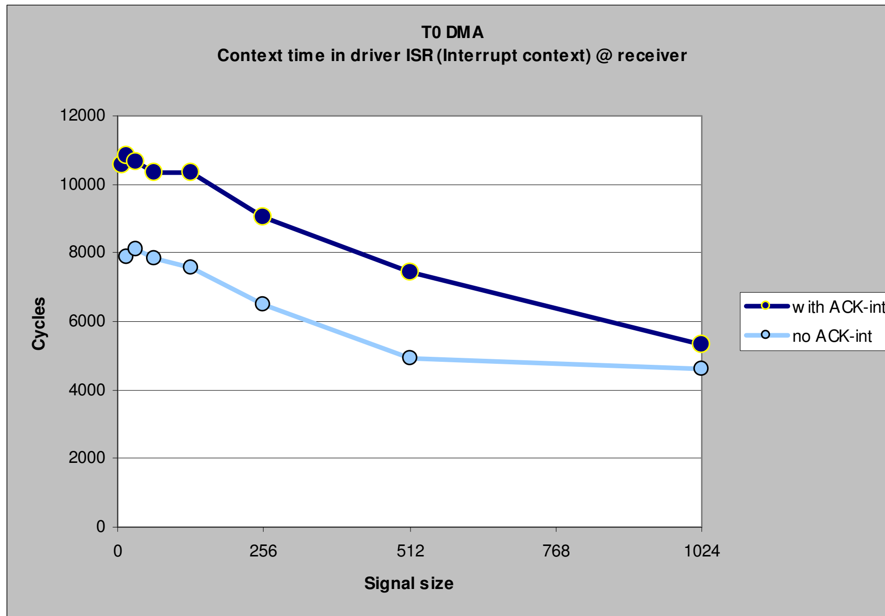


Diagram C3: *DMA – Receivers time in interrupt context for test T0. Surprisingly, the time decreases with larger signals. The likely explanation is that the increased time to finish the DMA-transfer has allowed the sending core to "cool down", leaving the receiver alone on the memory bus. With no bus contention the receiver executes considerably faster.*

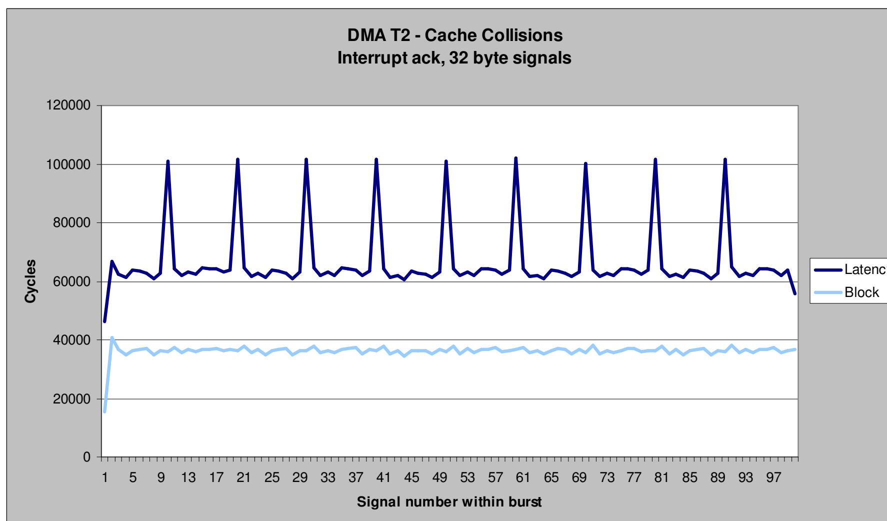


Diagram C4: *DMA - Latency and blocking time in a burst of 100 signals (test T1). A similar diagram is shown in section 4.4.2.1. In this case the cache collision is more serious, resulting in very high spikes in the latency. One of the ten DM-buffers collides with the signal buffer in the pool. The amount of collisions depends on where the linker places the buffers in memory. The driver with ACK-interrupt has been used.*

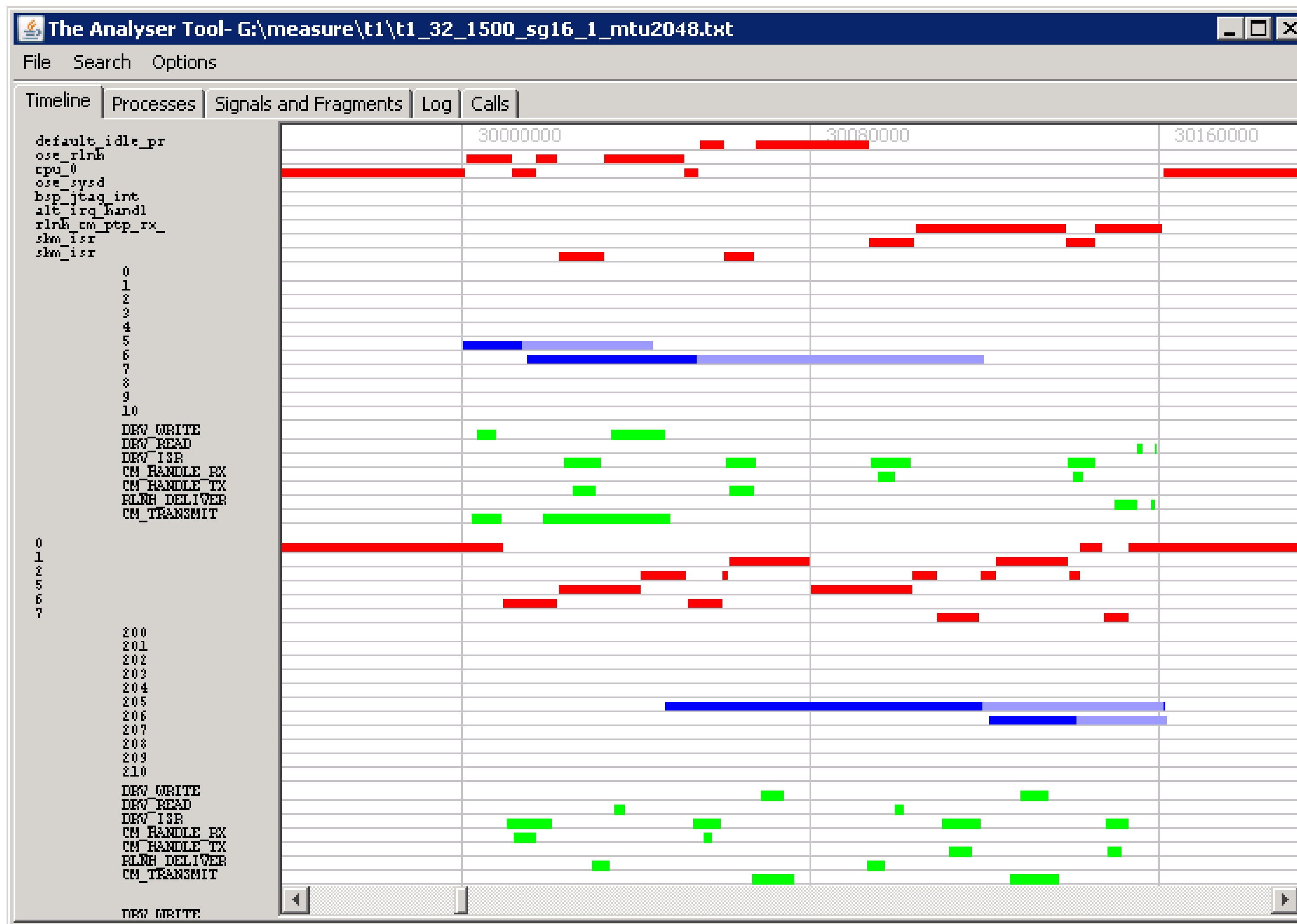


Diagram C5: *DMA - Interference between consecutive signals (test T1).in The Analyser Tool, 1500 bytes interfering signal. Repeated from section 4.3.2.1 in better scale*