FOR REFERENCE ONLY

The Nottingham Trent University
Library & Information Services

SHORT LOAN COLLECTION

Date	Time	Date	Time
~~- 3 NOV 2003~~	~~Ref~~		
- 9 MAY 2007	9pm	Ref.	

Please return this item to the Issuing Library.
Fines are payable for late return.

THIS ITEM MAY NOT BE RENEWED

Short Loan Coll May 1996

40 0725084 2

ProQuest Number: 10290282

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10290282

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A Distributed Real-Time Operating System for a Multi-Processor StrongARM Network

Kenny Eng Wee LIEW B.Eng (Hons)

A thesis submitted in partial fulfilment of the requirements of
The Nottingham Trent University for the degree of
Doctor of Philosophy

School of Engineering

The Nottingham Trent University
Nottingham, U.K.

February 2002

This copy of thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with the author and that no information derived from it may be published without the author's prior written consent.

Abstract

This thesis presents a novel Operating System (OS) that supports inter-processor communications. The OS developed addresses some real time control requirements and is optimised for embedded, distributed parallel processing system applications.

A review gives an overview of the key features of existing OS's and various message-passing mechanisms. Following the review, a novel OS was designed for a parallel processor StrongARM system. To further enhance the high level of abstraction for ease of parallel programming, two tools were developed to complement the OS. One of these tools is the Network Specifier, which allows the network mapping to be captured graphically, using GUI. A second tool, the NTU-Configurer, will generate routing headers automatically using Artificial Intelligence.

The OS design was captured in C, and minimal StrongARM assembly language using a top-down design methodology. The modules designed were tested, before being integrated to form the objective model. In the final form, various test codes were written to verify the functionality of the OS. Benchmarking and various measurement programs were performed to gauge the system overhead.

At a later stage, a project was undertaken in collaboration with Quantel to port an end user application. This was previously developed based on a multi-processor transputer system, using C and 3L libraries. It was adopted for a similar multi-processor StrongARM platform to demonstrate the contributions of this thesis. The project was deemed successful after the porting of code to the new hardware, with only minimal modifications.

The thesis concludes by discussing key features of the system and comparisons are made to the state-of-the-art in parallel processing communication systems.

Acknowledgements

Special thanks and in-depth gratitude are extended to the following people who have been helping me, directly or indirectly, through their actions, words or advice, that have guided me throughout the thick and thin to achieve my childhood dream.

My supervisor, Prof. Brian O'Neill, who was also my supervisor for my final year's degree project, to whom I owe the utmost gratitude and respect.

My second supervisor, Dr. Steve Clarke, who has spent hours to correct my documents throughout my doctoral degree, including conference papers, transfer report, annual monitoring forms, and this thesis.

Alec Cawley, who has given me a chance to prove my project's worth in an industrial project, and who has been my source of inspiration during the initial phase of my project.

My Mom, who has sacrificed her career to become a housewife to ensure her kids are brought up properly, and my Dad, who has given me a chance to study abroad.

My sisters, Peggy, Nancy and Jenny, who prayed for my well being while I am away from home.

My friends in the U.K including Kok Hoe, Wing, Piao, Jer Wang, and Kar Leong who have spent hours discussing about our project's difficulties, and ways to overcome hurdles.

My friends in Malaysia, including Chin, Raymond, Yap TL and Yap WH, who have ensured that I got proper vacation during my holidays in Malaysia, which has helped me to release the pent-up stress.

Last but not least, to my supportive girlfriend, Sharon for her tolerance and understanding during my bad days, and for taking the pain to proof read my thesis.

To anyone whom I unintentionally missed out, thank you.

Table of Contents

List of acronyms

AMD	Advanced Micro Device
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DMA	Direct Memory Access
DNA	Deoxyribo Nucleic Acid
DRAM	Dynamic Random Access Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Arrays
FPU	Floating Point Unit
GNU	GNU Not Unix
I/O	Input Output
IC	Integrated Circuit
IRQ	Interrupt Request
ISA	Industry Standard Architecture
KROC	Kent Retargetable Occam Compiler
MIMD	Multiple Instruction Multiple Data
MPI	Message Passing Interface
MPP	Massively Parallel Processing
OS	Operating System
OSLINK	Over sampling Link
PLD	Programmable Logic Device
PN	Processing Node
POSIX	Portable Operating System Interface
PVM	Parallel Virtual Machine
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SARNET	StrongARM Network
SARNIC	StrongARM Network Interface
SARNUX	StrongARM Operating System
SETI	Search for Extra-Terrestrial Intelligence
SPOC	Southampton Portable Occam Compiler
UART	Universal Asynchronous Receiver/Transmitter

List of Figures

List of Charts

List of Tables

1. INTRODUCTION

The man who removes a mountain begins by carrying away small stones.
An Ancient Chinese Proverb

There exist many types of problems that benefit from parallel computing, primarily due to reasons of geographical or computational intensity. In the former case, these include data acquisition, distributed control and monitoring systems. The latter case includes matrix operations, database operations, simulations, signal processing, ray tracing, data compression and decompression, encryption and decryption. Among the research applications[1] that use parallel computing are computer simulation to build safer cars, design of commercial aircraft that resulted in better fuel efficiency and safety, mapping of DNA genes with the hope to find cures for fatal diseases, and signal processing for weather forecasts. One ambitious project that attempts to harness the parallel computation power from all over the world is the Search for Extra-Terrestrial Intelligence (SETI) [2] project. The SETI project analyses the cosmic rays from the universe captured from the Arecibo Radio Observatory in Puerto Rico by distributing the signal captured to millions of idling home user computers via Internet in search for extra-terrestrial origin. The SETI project is an example of a problem that cannot be easily implemented using sequential computing.

The 1980's were an important decade in the development of parallel computers, with the advent of the CRAY[1], the ICL DAP[1] and the Goodyear Aerospace MPP [1]. These are among the viable systems used to solve hard 'number-crunching' tasks.

A new trend in parallel computing started with the availability of cheap micro-processors and memories to form low-cost Multiple Instruction Multiple Data (MIMD) computers, connected via a common bus to a shared memory[3]. Data was communicated between these processors in the shared memory, with the support of software. The performance of these low-cost computers could approach the performance of super-computers at a fraction of the cost. However, as more microprocessors were added into the parallel network, bus contention increased and therefore degraded the overall performance.

A compromise solution was later made available, where the addition of microprocessors into parallel networks that did not increase bus contention. This solution was the Transputers family. Transputers were developed by Inmos Limited [4], and had many attractive features. These features include four on-chip dedicated communication links and microcoded software support for scheduling and communications that does not require an OS. Transputers were used as a simple building block, to construct complex networks that could perform multi-processing tasks, either as a distributed control system or as 'number crunching' operations. Using a large number of these components together, powerful supercomputers could be built.

A transputer could essentially be run without requiring additional hardware support logic; the code developed does not need software support. Therefore developers could achieve faster prototyping, reduce the design cycle, and lower development costs. As a low cost and yet efficient alternative to other parallel solutions such as supercomputers and shared memory computers, the transputer attracted many companies[5], especially those specialising in embedded systems.

The Transputer family could be divided into two generations[48], the first generation, which includes the T1, T2 and T4 series, and the later second generation, which was the T9000 series. The first generation uses the OSLinks[17] (over-sample link), and the second generation uses DSLinks, which were incompatible with OSLinks. OCCAM[19] was the main language for the transputer systems, but applications were also written in Parallel C[21].

A more recent parallel computing solution is available from the desktop PC environment. With the cost of desktop PCs declining every year, and the advancement of data communication technology (for instance Ethernet[6] which offers 10Mbits to 100Mbits per second), many researchers have enhanced the computing power of the desktop environment with the use of clusters. A cluster is a group of independent computer systems, grouped together in a network to form a parallel computer, which can, at its limit, rival supercomputers. Inter-processor communications are performed via hubs or switches, and applications are parallelised using built-in libraries like Parallel Virtual Machine (PVM)[7] or Message Passing Interface (MPI)[8] for message-passing mechanisms.

In this thesis, a distinction is drawn between the transputer solution and the desktop PC solution, i.e. the transputer is termed as the embedded solution, and the PC is termed as the cluster solution. The project described in the thesis will concentrate primarily on parallel embedded systems solutions.

1.1 Research History

The Nottingham Trent University Parallel Processing Research Group [9] has been carrying out research and design in inter-processor communications[10] since 1989. Initial research interest was in Transputer systems. Transputers were designed as a basic building block for parallel systems. Each transputer chip is equipped with a Floating Point Unit (FPU), memory (RAM), a communications system for inter-Transputer communications, timers and an external bus for external Input Output (I/O) devices. Transputers can be connected point-to-point to form a parallel system. The research group first developed a hardware message router[11], to improve the scalability of first generation transputer networks. A router improves the inter-connectivity of a transputer network, as each processing unit no longer requires a point-to-point connection, and thus is able to scale better. The router developed is known as the ICR C416 packet routing switch[12]. The ICR C416 device is a 16-channel packet routing switch, designed for the first generation of transputers that used OSLinks[17]. The use of this device, in some real-time control systems, has shown the efficiency of routing as a solution to medium scale, low-cost, high performance inter-processor communications[13]. Figure 1-2 shows a basic parallel network enhanced by the router for scalability. Figure 1-2 shows a cascaded parallel network using two routers. For both networks shown, the processing node (PN) represents a transputer.

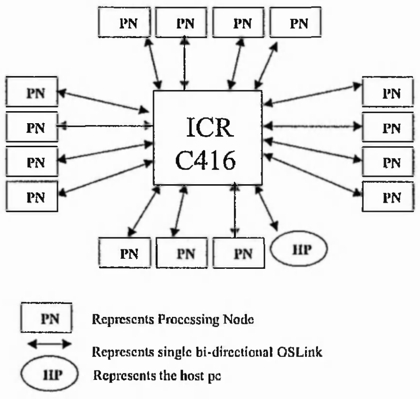
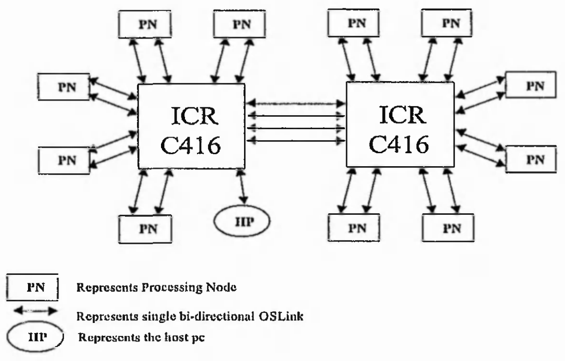

PN
ICR C416
HP
Represents Processing Node
Represents single bi-directional OSLink
Represents the host pc

Figure 1-1 a simple parallel system.


PN
ICR
C416
HP
Represents Processing Node
Represents single bi-directional OSLink
Represents the host pc

PN : Processing nodes.

Figure 1-2 a cascaded parallel system.

1.1.1 The New Distributed Processing System.

With the demise of the transputer, a replacement system [13] similar to the transputer was required for embedded parallel applications. Therefore, the research group focused on this aspect to build a low-cost system, similar to a first generation transputer system. The target was to build a powerful platform for embedded applications. There was a choice of either implementing a system on chip or an on-board solution. An on-board solution was preferable since each individual component could be upgraded without major changes to other on-board devices. In addition, an on-board solution could incorporate state-of-the-art Reduced Instruction Set Computing (RISC)[14] microprocessor into the parallel systems. Among the goals was the aim of maintaining easy upgradability for each component to follow market availability. The modern RISC microprocessors chosen for the new system was the StrongARM device [15] because it met the requirements for an embedded system processor, with high performance but low power consumption.

While adapting RISC microprocessors into the distributed parallel network, the system design retained many features of the transputer, in particular the efficient message passing and the operation of the scheduler. Utilising these features, an IC device was designed for the interface between the StrongARM microprocessor and the ICR C416 packet router. This device was called the SARNIC[16]. The SARNIC interface offered an efficient link to the inter-communication network, provided memory bus management, and had other features to support the RISC processor in the parallel processing environment. The main features implemented in the SARNIC are listed below:

- DMA controller for message transfer,
- OS Link controller for message manipulation and communication,
- either 2 KByte internal booting ROM or booting from OS Link,
- 32-bit 1 MHz real-time timer,
- interrupt controller with dual priority,
- UART peripheral interface.

In the basic system building block, a node for a multi-processing system contained a StrongARM processor, dynamic memory (DRAM) and a custom link device. The custom device, implemented in a Field Programmable Gate Array (FPGA), provided inter-processor communications and other functionalities. The Research Group developed this processor node, shown in Figure 1-3.

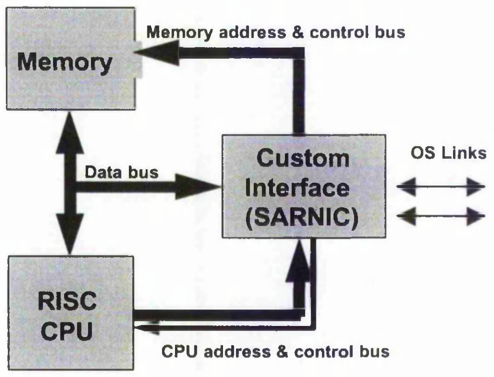

Memory
Memory address & control bus
Data bus
Custom Interface (SARNIC)
OS Links
RISC CPU
CPU address & control bus

Figure 1-3 components of a processing node.

These processing nodes were connected using ICR C416 message routers [9]. Communication was performed via the OSLink [17], which complied to the router's protocol. Direct Memory Access (DMA) engines were used to 'steal cycles' from the bus to perform inter-processor communications, allowing the processor to perform its computation while the communication took place in the background.

The inter-networking of the StrongARM processing nodes using the ICR C416 router in a network is referred to as the StrongARM Network (SARNET) throughout this thesis.

With the development of this hardware platform, there was a need to provide the software for the system. The software layer should provide the functionalities previously available on the transputer's microcoded firmware while taking the full advantage of the additional features on this new platform. Another advantageous feature would be to allow porting of applications previously developed for the transputer to the SARNET. The software should provide the level of abstraction needed, to make the benefits of the transputer available on modern systems and to achieve backward compatibility with systems previously developed based on the transputer. The software will be an OS, and is the focus of this thesis.

The next section will outline various software research conducted by other groups to port applications from transputers to other parallel hardware.

1.1.2 Brief summary of other research in this area.

In the absence of the transputer, many research groups attempted to port the transputer's software features into modern RISC architectures. The key discussion in this section concentrates on the software implementation of those projects, which result in porting previous applications targeted at transputers, written in Occam [19] or Parallel C [20][21][22]. These projects include the Kent Retargetable Occam Compiler (KROC) [23] and Southampton Portable Occam Compiler (SPOC) [24].

The KROC system compiles an Occam source code into transputer assembly language before being translated into the targeted native RISC processor's assembly language. The native assembler would then assemble and link to a kernel to produce an image file. KROC retargets Occam, traditionally used to develop transputer applications, to modern processors, including the Sparc[112], Alpha[113] and PowerPC[70][114].

Similar to the KROC system, the SPOC system generates portable ANSI C from Occam, to be retargeted at modern processors [24].

While the development of retargetable compilers may seem attractive, there is still a need for software, such as the transputer's microcoded functionalities, to enable applications to run on hardware. The system would still need an underlying layer, or kernel, to provide scheduling and other software runtime facilities to execute.

In this work, a novel OS is developed for the StrongARM system. As the StrongARM node communication link uses the OSLink protocol, the software will need to emulate the channel communications, albeit with slight differences such as syntax in procedural calls. Other research groups have shown that it is possible to emulate the behaviour of external channel communications, even via Ethernet, using Occam [25][26].

1.2 The New Operating System for the Parallel StrongARM network.

The key aim of this project is to design and implement a new OS, with real time functionality for the new StrongARM hardware platform. Essentially, this project will provide a software environment similar to virtual transputers[18] for applications on the new StrongARM platform. Communication has to be performed using serial channels, for backwards compatibility.

The new OS is designed based on the following fundamentals:

- Provide the necessary interface to maintain existing software applications written in the transputer's Parallel C language.
- Provide the layer of abstraction for the ease of programming parallel applications.
- Have real-time functionality to meet the requirements for most existing parallel applications.
- Portability across different microprocessors, maintaining the ease of subsequently changing microprocessors in the hardware design.

Parallel C is targeted since it is believed to have wider appeal in the industry, as compared to other languages developed for transputers like Occam[27]. Many developers in the embedded market typically use C/C++ as their programming language, and there are large numbers of applications written in Parallel C based on the transputer system. Consequently, new users do not need to invest in learning Occam, and they will only need to learn the Communicating Sequential Processes (CSP) parallel programming technique, which uses channels for communication.

Backward compatibility is an important issue, since for the industry, hardware upgrades are a major task, particularly when maintaining the existing application software, written and developed over many years (in this case based on the transputer). Consequently, an embedded OS with real-time response and functionality is required to substitute for the transputer micro-coded functions and to provide a layer of abstraction to hide the new hardware platform.

To demonstrate feasibility, a simulator is typically built as a prudent strategy in determining the potential success of a design project. A simulator is a confined software environment, which allows certain languages to run on a typical desktop machine, to simulate the behaviour of the program. Although a simulator is not strictly necessary, nevertheless it is able to show that a specific language can be ported to any hardware if necessary underlying hardware and software features are provided. Another research group[28] has already developed a small Parallel C Simulator[28], which uses a sequential compiler to allow programs written in INMOS Parallel C to be executed in a desktop computer after minor changes. Using this simulator allows only the simulation of the language on a single processor, and not the inherent parallel architecture of the hardware system. Although this simulator is not feasible for the StrongARM system, due to the sequential nature of the simulation and the high overhead involved in simulation, it nevertheless indicates a higher possibility for successful porting of applications to the StrongARM system.

In the development of the system, the general aim is to address the independence of the OS from the underlying hardware platform. The work attempts to include an ease of migration of the OS to other modern processors with the concept that if the targeted StrongARM processor is being discontinued, the OS should not. Since an application is generally built on top of an OS, and an OS is built on top of specific hardware, the structure will ultimately allow the porting of all the layers including the application previously developed on the OS to another hardware platform. Theoretically, the benefit of portability propagates from one layer to another above, allowing the reuse of code.

With the fundamentals required for the new OS discussed, the new OS is developed [29][30][31] to fulfil these. The following section will detail the structure of this thesis, which presents the newly developed OS and associated tools to enhance a high-level of abstraction. In this thesis, the OS developed by the author will be named as SARNUX.

1.3 Structure of the thesis.

The thesis comprises of 7 chapters.

Chapter 2 gives a literature review: surveying existing OS's and message passing mechanisms, including the strengths and weaknesses of each.

Chapter 3 describes the system design study and gives the hardware overview of the facilities available. This study is needed as any hardware design normally influences the design of an

OS. In addition, routing and mapping algorithms are discussed extensively, focussing on the communications aspect of the system.

Chapter 4 gives a breakdown of the OS developed, and the details of each module involved. It explains the communications behaviour, and some other issues that need to be taken into account while developing the OS. In addition, this chapter also details the Windows GUI Network Specifier and the NTU-Configurer tools, developed to assist the developer in programming parallel applications.

Chapter 5 gives an analysis of the system using benchmark programs, and presents several tests conducted to measure the software overheads of the system. These tests examine the message passing overheads, for both global and local communications. A test is also performed to analyse the effect of the DMA access on the CPU's performance.

Chapter 6 explains a project undertaken to port an existing end user application, originally targeted at a transputer system, to the new StrongARM platform.

Chapter 7 discusses the work presented in this thesis. This work is then compared to previous work and other systems in the research literature. Finally, the conclusion of this thesis is presented and possible future work is discussed.

2. LITERATURE REVIEWS.

This chapter presents the OS's surveyed, in reference to the feasibility of adoption on the SARNET[13] hardware platform. Some of these OS's are CSP[107] based runtime system. Since a pre-requisite of the targeted system is support for CSP based communication, those OS's which do not inherently support CSP could be made to support channel communications with the inclusion of a CSP based library. A preliminary study was needed to assess the need to design and build a novel OS for the SARNET system as opposed to adopting another OS with similar functionalities. Some of the OS's reviewed are commercially available, and some are research systems. Each has its own advantages and disadvantages.

The second part of this chapter will study various message-passing mechanisms, which form the basis of inter-process and inter-processor communication. The aim is to determine which mechanism is suitable for the SARNET system based on the feasibility and practicality of adoption.

The third part of this chapter will a description of a few mechanisms for synchronisation.

The focus of the discussion and this thesis on the OS is in the area of embedded systems, involving parallel networks.

2.1 Commercial and non-commercial embedded parallel OS.

The approach of using an OS implies a high overhead on applications. The OS implementation will involve the execution of more instructions than that of the transputer T425's[48] microcoded facility, but this will be balanced by faster processing power. The StrongARM could process up to 266 instruction per microsecond (based on the throughput of 1 instruction per cycle, with core speed at 266MHz, with full cache hit), as compared to 17.5 offered by the T425 transputer (based on the single instruction per cycle, with clock speed of 17.5 MHz) [48]. Some of the processing speed will be lost through cache miss for the StrongARM, but comparatively, approximately 30%[32] of transputer instructions are prefixes (where normal instructions require longer processing time), resulting in slower execution time. Therefore, the overheads may not be as high as anticipated (these overheads will be analysed later).

An OS[33] is generally defined as a software program, which after being loaded into a computer by another program, manages other programs that run on that computer. Those other programs are called applications, and applications interact with the OS via a set of extended instructions that the OS provides. These instructions are known as 'system calls' or 'OS calls'.

There are many OS's for embedded systems, commercial and non-commercial. Non-commercial ones are used primarily in research and academia, and some of these proved to be competitive with commercial OS's. OS's have been evolving over the years, to suit contemporary embedded applications. The evolution has been evident as the demand for the type of services and level of abstraction grows in tandem with the hardware, as technology progresses. The continuing explosive growth of processing power (as predicted by Moore's law) means that modern OS's have to be more resilient, reliable, and fit for purpose. Since OS's are linked to the hardware, there is a need to either upgrade an OS to take the full advantage of any improved hardware, or in more extreme cases, to develop a new OS.

In the view of various OS's methodologies and functionalities, there is a need to review those OS's that are relevant to embedded and parallel systems for potential utilisation on the SARNET platform. This review also investigates the advantages and disadvantages of developing a customised OS as compared to adopting a commercially available OS. The main issues of interest are the portability (in line with the aim of maintaining the possibility of changing the microprocessor) and functionality of each OS. The following OS's are surveyed:

1. Communicating Sequential Process for C (CCSP)[35].
2. Communicating Thread for C++ (CTC) and Communicating Threads for Java (CTJ)[36].
3. Real Time Linux (RTL)[40].
4. VxWorks[46].
5. OS9[106]
6. Transputers.

All the OS's surveyed support multi-threading, which by definition is the ability of the system to run multiple threads or processes simultaneously. There is a particular focus on Transputers as the SARNET was designed to exploit embedded applications which were (or would have been) targeted at Transputer Networks.

2.1.1 Communicating Sequential Process for C (CCSP).

The Computing Laboratory from the University of Kent developed CCSP[35], a highly portable run-time system. It implements the channel communication ideas of CSP[107]. The run-time system provides thread management and inter-process communication facilities to processes written in both Occam and C, targeting modern desktop PC's. The system was designed to complement the Kent Retargetable Occam Compiler (KROC), previously developed by the same research group[9]. KROC was a part of Occam-for-all project, funded by EPSRC, to enable programs written in Occam to run on modern processors. CCSP retains most of the attractive transputer micro-coded features, and currently supports the Intel x86 PC platform, running Linux.

In terms of functionalities, CCSP comes close to the SARNET requirements. CCSP provides channel communications and basic support for scheduling and other OS facilities. These features are attractive and well suited for the intended objective. However, it is not feasible to adopt CCSP for the following reasons:

- A major part of the kernel is written in x86 assembly language for optimisation, hampering the effort of porting it to the SARNET. To port it would require understanding, and re-writing of these routines. Additional testing would also be required to verify the correctness of the routines on the new platform.

- The targeted systems are uni-processor desktop PC's, and not parallel-embedded systems. Further work would need to be carried out to port it to the distributed StrongARM system. This would include developing library facilities for inter-processor communications and modifying the kernel to achieve low communication overheads.

- The system adopts a co-operative approach scheduling mechanism. This is deemed not suitable as the SARNET has hardware interrupts, which require immediate servicing. Furthermore, co-operative scheduling removes the real time functionality of most systems. To cater for generic embedded real time applications, a co-operative scheduling mechanism is not preferable. Hence, the kernel would need to be modified to also incorporate a pre-emptive scheduling mechanism.

- The system is confined to operating in user mode, since the system can only access user mode via Linux. To run on the SARNET, certain routines must be run in privileged mode, and substantial modifications need to be carried out to do so.

- Certain routines in the system, like the timer, rely on the underlying OS, Linux[34], to run. Therefore, to port the whole system requires providing these routines to the SARNET. In addition, the system should run on StrongARM hardware, without any underlying software support such as Linux.

- The hardware events are polled continuously since the system is not interrupt driven. This mechanism does not suit the SARNET, which is interrupt driven especially in inter-processor communication.

2.1.2 Communicating Threads for C++ (CTC) and Communicating Threads for Java (CTJ).

The research group from the University of Twente, The Netherlands, developed both packages, which are similar in functionalities, but the implementation is different. The CTC was written mostly in C++, and the CTJ[36] was written in Java. Both packages are targeted at desktop computers and take an object-oriented programming (OOP) [37] approach. They rely less on assembly language, but more on the readily available low level libraries provided by the standard compilers. To port either of them to the SARNET would require re-writing of these low-level libraries.

The design of the CTJ system incorporates the CSP model for channel communications in Java[38]. The processes could be executed in sequence, parallel or by choice. For both systems, a slightly different syntax is used in the application code since both rely on objects for communications and thread management. The OOP approach in programming parallel applications raises the issue of backward compatibility with the transputer systems, which use mainly procedural calls.

The CTJ package is not considered practical for the SARNET, since the use of Java will require the underlying Java Virtual Machines (JVM)[39] for execution. JVM support for StrongARM was unavailable at the time and it would have high overheads, associated mainly with the translation of Java bytecodes to the processor's native code during runtime.

The use of OOP as low-level support denotes higher overhead for embedded systems. Higher overheads are not desirable, since the OS must provide real-time functionality for time-critical applications. Furthermore, since the CTC package raises the backward compatibility issue, and that there is substantial effort involved in porting to the new hardware, it is deemed unsuitable for the SARNET.

2.1.3 Real Time Linux (RTL).

RTL[40] is spawned from Linux [42], providing real-time functionality in the application code when running Linux. Another variant of RTL is available, called The Real-Time Application Interface for Linux (RTAI)[41]. Since both have similar advantages and disadvantages, only the RTL will be discussed here. Linux without real time functionality is not deemed suitable for embedded applications[43]. Linux typically runs on desktop PCs, and is used by many as a substitute for Microsoft Windows. Linux was developed as an academic project, and has grown since to a mature OS with the assistance from developers around the world, since the source code was freely available. RTL was developed by FSMLABS [44] under the GNU licence[45], which makes RTL source code available. RTL is a small and fast real-time OS that uses a technique which enables the general purpose OS Linux to run under it. The major appeal of Linux and RTL is the availability of source code in C, which gives developers an option to modify both Linux and RTL to suit their applications. Furthermore, there is a wide range of programs which are readily available to run on RTL, thus providing additional attractive features on the system.

In the embedded industry, some developers use RTL based embedded systems in the form of add-on cards, which can be inserted into the PC. These systems function as Digital Signal Processing (DSP) units, motor controllers or data acquisition units and need real-time functionality, in order to operate correctly.

Before RTL can be used, Linux must be running on the system. RTL is then added by using the procedural call, 'insmod'. This will push Linux out from the system foundation and makes the RTL kernel the owner of the system. It could be viewed that after inserting RTL, Linux runs as a thread under RTL. RTL is multi-threaded, and the real time components of RTL applications are threads and signal handlers (interrupt handlers). Essentially, periodic tasks that are real-time could then be executed within a guaranteed time frame by RTL. Figure 2-1 illustrates the hierarchy between Linux and RTL (courtesy of FSMLABS [44]).

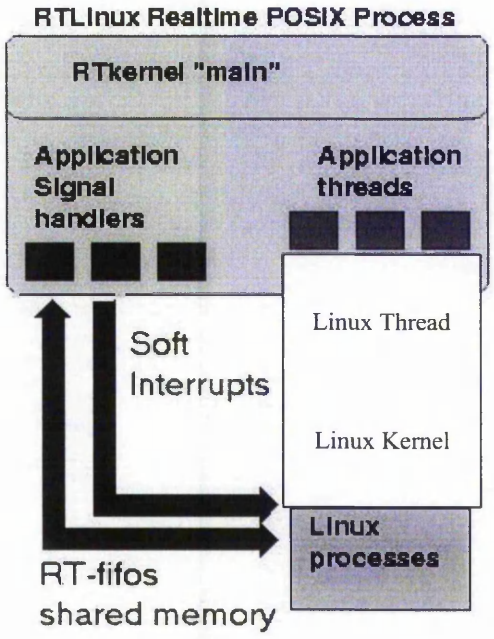


Figure 2-1 hierarchical view of Linux and RTL.

Unfortunately, RTL is not suitable to be ported to the SARNET, due to the following reasons:

1. The OS has large memory footprint. Therefore it is not suitable for remote embedded systems with constrained memory. It could be stripped down, but to perform the task would require in-depth understanding of Linux and RTL.
2. If Linux and RTL were stripped down to suit the targeted platform, there is a risk that many of the vast program libraries developed based on Linux would not be able to run, thus diminishing one of the attractive features of using Linux.
3. Linux's performance may not meet the requirements of the targeted application.
4. The end result may not be comparable to the new OS, in terms of flexibility and efficiency.

Hence there was a perceived risk of adopting Linux at the beginning of the project. If Linux were to be found not viable after extensive investigation and development, this would seriously compromise the potential success of the project.

2.1.4 VxWorks.

VxWorks[46] is a high-end RTOS from WinRiver[115], which has most of the attractive features of an embedded OS. It could target StrongARM systems, has real-time functionality and multi-tasking support. In addition, VxWorks has two types of optional scheduling policy; the pre-emptive priority scheduling based on FIFOs, and the round-robin scheduling by priority. Further advantages include:

- It is popular and well known among embedded developers.
- It has a widely supported hardware platform.
- It has optional additional facilities.
- It has a range of development tools.
- It has good vendor support.
- It is POSIX[63] compliant.

Another feature includes 256 priority levels for a user process.

However, the main disadvantages of using VxWorks are:

- It is designed for a single-processor environment.
- It is too generalised for wider applications; sacrificing performance.
- It is proprietary, hence no source codes are available, which hampers efforts to modify the OS.

Most of the system calls are different from the transputer system, but this could be easily rectified by re-declaring certain routines to match the 3L library[47] syntax. However, more work is needed to incorporate drivers and CSP based libraries for inter-process communications. Therefore, after due consideration, it was decided that VxWorks was not suitable for the intended application.

2.1.5 OS9

Although OS9[106] is among the popular embedded OS's, however, the support for the StrongARM processor was not available at the time the review was conducted. However, OS9 would have disadvantages similar to VxWorks, and as such, also would not be suitable for the intended application.

2.1.6 Transputer

The transputer system discussed here focuses only on the first generation Transputers that use OSLinks. The transputer system offers scheduling, inter-process communications and timer management facilities in the micro-coded firmware[48]. The review of the transputer is partly conducted in order to enable the understanding of the SARNET in terms of backward compatibility. The transputer system has the following functionalities:

- 2 levels of priority for a process.
- Low priority processes are co-operative and pre-emptive and they are time-sliced.
- 2 types of pre-emption method; active context-saving and non-active context saving.
- Automatic insertion of de-scheduling point instructions in the application code.
- High-priority processes are non-interruptible
- High-priority processes are not time-sliced
- Interrupts by external events will cause full-context saving (only for lower priority processes as high priority processes are non-interruptible).
- Uses a FIFO scheduling mechanism.
- When a process is de-scheduled, the scheduler will first look at the high-priority run list, and if it is empty, it will run through the lower-priority run list.

Other features which are not deemed attractive includes the following:

- The time slice quantum is not configurable.
- A constrained number of priority levels.
- Reliant on inherent properties of hardware which may not be supported in modern processors.

Transputer systems unfortunately are no longer commercially available. However the insight gained from the investigation is beneficial in determining the design of a new OS especially in addressing the issue of backward compatibility, since the SARNET was constructed in part to replace transputer networks.

The behaviour of the scheduler has to be taken into account, for backward compatibility. However, certain functions, like co-operative de-scheduling points for low priority queues, cannot be emulated in the new platform, due to the lack of support from the StrongARM cross-compiler and the hardware. These points have to be inserted by the compiler during compile time. Furthermore, in the view of the inherent architectural difference between the transputer and the StrongARM processor, should a new OS be developed, due consideration must be given during the design phase in providing most of the transputer functionalities.

2.1.7 Summary and Design Choice.

Since most of the OS's surveyed are deemed unsuitable for the SARNET StrongARM platform, a new OS has to be designed and built. The building of a new OS is not unusual[49], since embedded systems application requirements tends to differ in functionality and performance. The technical insight gained from these reviews however will assist in the design and implementation of the new OS.

The next section will discuss message-passing mechanisms.

2.2 Message-Passing mechanisms.

Inter-process communication implies passing messages from one process to another[116]. These processes could either reside on the same processor, or on different processors. The message-passing action involves the sending of a message from the sending process via a message-passing mechanism, and the receipt of the message on the receiving process. These message-passing mechanisms have developed in many forms over the years, with bias towards the OS used. The message-passing mechanism is the basis of communication for multi-processor systems.

The mechanisms discussed are:

- Memory sharing[33].
- Pipes[50].
- Files[51].
- Sockets and TCP[108].
- Remote Method Invocation (RMI) [51].
- Channels[48].

The intention of this review is not to adopt these mechanisms, rather it serves as a guide as to what mechanisms are available. The idea is to gather and analyse these implementations to assist in the development of an effective communication module that has low-latency in inter-process communication[29]. The following sections will discuss the merits of each form of message passing.

2.2.1 Memory sharing.

Memory sharing[33] is one of the most popular and widely used methods of inter-process communication. Data is stored in a particular memory location, to be accessed by both the sending process and the receiving process. Depending on the application requirements, shared memory can be used in various ways. One scenario is the operation of Multiple Instruction Multiple Data (MIMD) parallel programs, accessing a shared memory for a network of multiple processors, by sharing a single bus to a main memory. Another scenario would be two threads running on a single processor exchanging data synchronously, via a shared memory. Essentially, both scenarios use a known memory location, which is shared by two or more processes, as a means of communication.

Memory sharing is typically implemented with semaphores, to provide synchronisation between two communicating processes. Memory sharing is typically carried out on local communication (where both communicating threads reside on the same processor). Hence, one main disadvantage of it, as compared to channel communication, is that it is often difficult to change the code from performing local communication to global communication (where both communicating threads reside on different processors) when a need arises to assign two communicating threads to different processors.

2.2.2 Pipes.

Pipe is a method of passing a message from one process to another. Pipes[50] are used by Unix[50] systems as a means for channelling data streams from the output of one process to the input of another, on a uni-processor system. The pipe takes the standard output from one command and uses it as the standard input to another command. A command is considered a process executing in the Unix shell. A Unix system could run multiple shells simultaneously. Pipes can involve more than two stages, where connections are established within several shells. A pipe is uni-directional.

The concept of pipes makes the task of programming procedures easier in the Unix system. Every program can be written independently, performing its assigned task to tackle a specified portion of a larger problem. These programs could then be combined using pipes to create a coherent and unified application. Another advantage is that since each program was designed in modular fashion, they could be reused for other purposes. The flexibility introduced by the pipes mechanism is one of the most powerful traits of the Unix system.

Pipes development is still confined to Unix systems. Other modern systems will tend to use different message passing mechanisms, as the structure of the OS differs from one platform to another.

2.2.3 Files.

Communication via files[51] is the most primitive form of inter-process communications. Essentially, any message that wishes to be sent from one process to another is stored in a file, which could be accessed by both processes. The message could be of any type, and the message length is only limited by the disk space available. The sending process will first create a file and store a message in it. The file is then closed. The receiving process will then open the message file and read the message. However, some form of synchronisation needs to be used to prevent the sending process from deleting the previous message before the receiver has the chance to read it. The receiver could generate semaphores or headers in the file, to indicate a read message. The synchronisation mechanism must be recognised by the sending process. The whole message-passing system is illustrated in Figure 2-2.

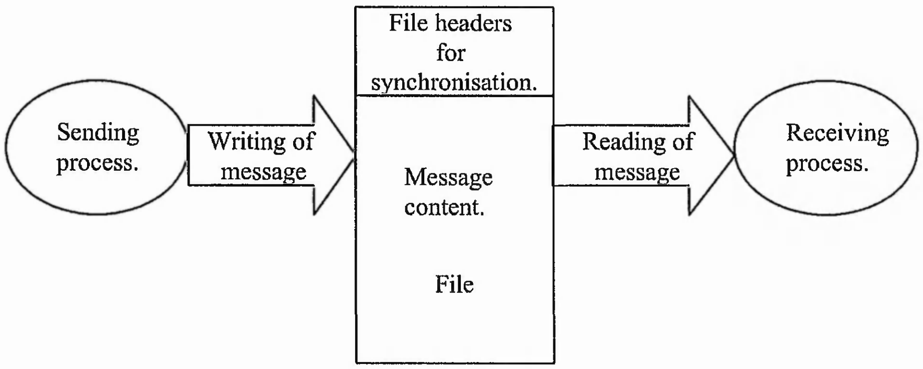


Figure 2-2 message-passing via the file mechanism.

The file communication system is too primitive to be adopted in modern real-time systems. However, it will still be used when the message is too large to fit into memory. Messages like those representing video data will often be stored in a file for the intended process.

2.2.4 Sockets and TCP.

TCP[108] is a mechanism typically used by a PC to communicate with other PC via a network, or the Internet. To establish a connection, a socket from one PC is used to connect to a port of another PC. In network terminology, a port and a socket are not physical devices, but merely an abstraction used by the software to facilitate inter-computer communications, i.e. between a server and a client. A client typically uses a socket to connect to a port, used by a server. This type of communication forms the backbone of the Internet structure, where computers communicate with each other via sockets and ports. For instance, a web server will run continuously on a remote machine, waiting for any network traffic which wishes to chat with port '8000'. Upon receiving a requesting packet to chat to port '8000', the server software will establish the connection, and start a session with the remote client. This session may involve the sending of textual information contained in a specified web page to the client, and the client may, in return, send additional packets requesting different services from the server. The session will be terminated after both sides have engaged in a dialog and any one side closes the connection.

Sockets are typically used on the TCP protocol. TCP is a protocol designed to provide a reliable delivery of packets of data. TCP relies on IP to provide routing from one endpoint to the other endpoint in a communication path. TCP/IP is developed to be robust and provide automatic recovery in an unreliable network, where the network condition is unpredictable.

The perceived risk of losing a message is low in the targeted StrongARM embedded platform. Therefore, a highly fault-tolerant network protocol such as TCP is not desirable, as there tend to be higher communication overheads associated with it.

2.2.5 Remote Method Invocation (RMI).

RMI is a mechanism used by Java [51], as a means for communicating between two remote objects running on different machines. This implies distributed collaborating objects that communicate through a network. Java uses an OOP approach towards writing applications. RMI introduces a high layer of abstraction, relieving the programmer from worrying about the underlying protocols for communicating via any network. Java's implementation of RMI is divided into two parts, the transmitting part and the receiving part. In this section, the transmitting process will be referred to as the 'client', while the receiving process will be referred to as the 'server'. A procedural call will be referred to as a 'method'.

The client will have a transport layer that handles data encoding, and the transmission protocol. A 'stub' will be created on the client to represent the remote object on the server. A stub, sitting on the client side will packetise any method invocation and parameter as a block of bytes in RMI protocol. The server side will unravel the packet using the 'skeleton' object.

All information contained in the packet is retrieved after 'unmarshalling' the bytes contained in the packet. The server will then call the desired method on the real remote object residing on its side. It may decide to send back some reply, depending on the application. Thus, all communications are carried out using RMI. Figure 2-3 illustrates the internal system of the RMI mechanism (taken from CoreJava[51]).

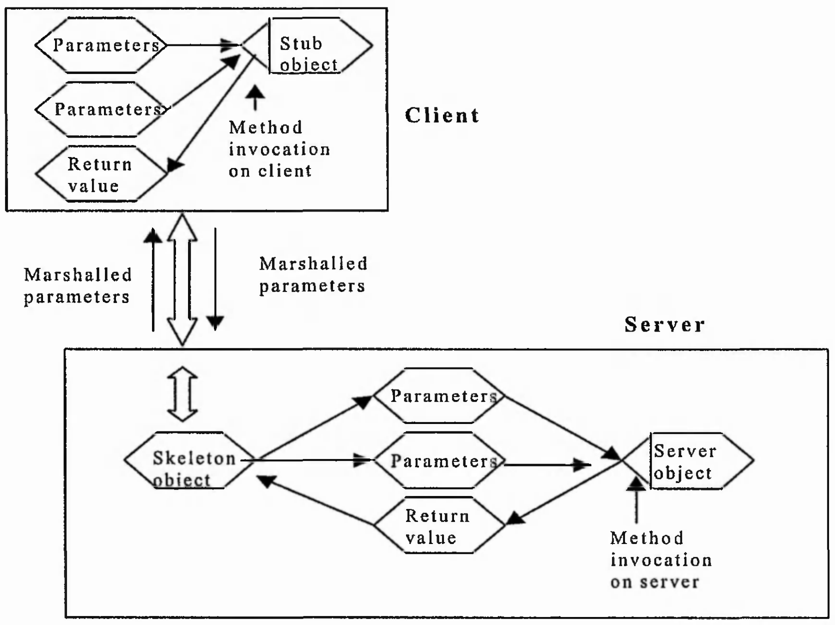


Figure 2-3 RMI internal structure.

The RMI communication mechanism is transparent to the developer. The disadvantage is that the overhead involved is tremendously high and unsuitable for real-time applications.

2.2.6 Channels.

Channel[48] communication stems from the CSP[107] principle. A channel is either an internal channel (memory location), or an external channel denoting an external link. It is a one-way communication system, whereby a message sent via one channel will be received on the other end of the channel. They can only connect two processes. A communication path represented by one channel is exclusive to one sending process and one receiving process.

In the Occam programming model, channel communication takes place only when both the receiver and the sender are ready. If any of the two processes are not ready yet, any process that is ready will become inactive or blocked while waiting for the other. Both sides are only allowed to resume after the communication takes place.

Channels have two functions:

1. As a mean of inter-process communication, whereby they provide a communication path between two communicating processes. Messages are exchanged in this way.
2. As a synchronisation mechanism between two processes, since neither process can continue until both communicating processes are ready.

For a programmer, the logical behaviour of communicating via external links, or internal links, is the same. The only difference is the performance and perhaps using different syntax or system calls. The logical behaviour for both external and internal communication is illustrated in Figure 2-4.

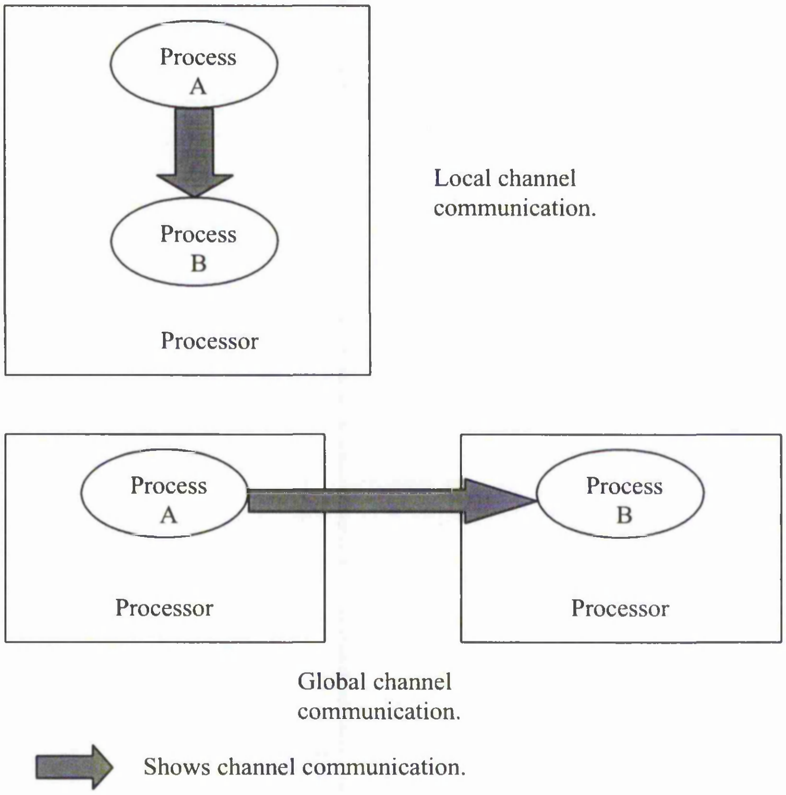

Process A
Process B
Processor
Local channel communication.
Process A
Processor
Process B
Processor
Global channel communication.
Shows channel communication.

Figure 2-4 logical similarity between local and global channel communication.

Essentially, when performing a channel communication, the following information is needed on both sides:

- Message length.
- Channel name.

- Message pointer.

A procedural call marks the communication point in the application code, and may also be used as a de-scheduling point.

The transputer system implemented channels in four possible ways[82]:

1. Soft channel – a channel between two local processes (two processes that reside on the same processor).
2. Channel edge – a channel that provides communication between a processor and the network, via routers.
3. Direct channel – a channel that is connected directly between two processors.
4. Virtual channel – channel that exists over a virtual link, where there could be several virtual channels over a single physical link.

The Transputer system uses the Channels mechanism for inter-process communications. Since one of the aims of the project is backward compatibility to the transputer system, when designing a novel OS, incorporating the channel communication mechanism is preferable.

2.3 Synchronisation mechanism

In some systems, communications are used as a mean of synchronisation. Synchronisation is needed in concurrent system, to ensure orderly system behaviour. There are, however a few other mechanisms that are readily available for this purpose. These mechanisms include:

1. Semaphores.
2. Monitors.
3. Spin locks.

2.3.1 Semaphores.

Semaphores are typically used as a way of synchronising access to shared resources. One such example of shared resources is the UART, where multiple threads may try to output messages via it. A semaphore is a value allotted in a designated area which threads could read and change, or may have to wait for a period of time before performing any operation, depending on the value of it. The operation of semaphore is 'signal' and 'wait'. Typically, a thread that manages to get hold of the resource will change the value of the semaphore associated with the resource, preventing other threads from gaining control of the resource since they will have to wait until the value of the semaphore allows them to do so.

2.3.2 Monitors.

Monitor is an object which encapsulates variables and procedures, which are typically labelled as 'critical sections', whereby only a single thread could execute this section at any given time. A monitor may keep a queue of threads that are waiting to access it, depending on the status of the monitor. Any threads that make a request to the monitor will be deactivated first, and put into a queue. If the queue is empty, the thread will get an exclusive access to the synchronised section, otherwise it will have to wait for its turn. In this way, synchronisation is achieved.

2.3.3 Spin locks.

Unlike other mechanisms, spin locks are used by processes, which constantly polls for the availability of the lock. A thread could only proceed only if it manages to acquire the spin lock. If the spin lock is unavailable, the thread waiting will continuously polls for the spin lock it is released by the thread that holds the lock. It is sometimes more efficient to perform polling for the following reasons:

1. Less context-switch as there may be less active pre-emption and de-scheduling.
2. Fewer overheads in maintaining a structure for housekeeping for processes which wish to acquire the lock.
3. Less overheads in the mechanism of trying to maintain a degree of fairness in the system.

For most CSP based system, channels are inherently available as a mechanism for synchronisation. However, others are also preferable depending on the circumstances.

3. SYSTEM DESIGN STUDY.

This chapter describes the underlying hardware support and the environment for the design of a new OS. The parallel StrongARM network is linked with the use of the ICR-C416 router [52]. The understanding of the underlying hardware is needed as every OS design is historically tightly linked to, or co-designed with[53], the hardware platform. Furthermore, there is a need to fully utilise the facilities available from the hardware, to achieve full functionality, efficiency and performance.

The requirements of the new OS are stated, including the feasibility study performed.

3.1 Hardware Overview.

The hardware described in the following sections is the targeted system for the novel OS. The routing system, SARNET was similar to the previous transputer network system (connected with the ICR-C416 router), except for the processing node, which the StrongARM system replaces. A major difference from the previous transputer systems is that the SARNET separates out the communications system from the processor, providing a more adaptable, modularised design where processors or other elements can be upgraded easily without affecting the rest of the system components.

The SARNET will be described first, followed by the custom device. The last section will describe the generic ISA Interface.

3.1.1 The ICR-C416 Router and Routing Protocol.

The ICR-C416 [52] is a dynamic hardware routing switch, with 16 ports that are connected via OSLinks capable of running at 10 or 20 Mbits/sec. The ICR-C416 router was originally designed to enhance the connectivity of first generation transputer systems.

The router has two operating modes; the dynamic and the static mode. By default, it would operate in the dynamic mode. However, the router has a control port, which could be used by any processing node connected to it (to program the router's behaviour). In a dynamic routing mode, the router could route 16 messages concurrently. Each message, in the form of a packet, carries the routing headers, message length and the payload. The size of the message length is 1 byte, and the size of the payload ranges from 1 to 255 bytes. Messages larger than

255 bytes must be packetised into multiple packets of 255 bytes. By cascading multiple routing headers, a message could be routed via several routers. Each routing byte dictates which path the message is to be taken.

The packet format is illustrated in Figure 3-1.

Routing headers	Message length.	Payload.
(Variable)	(1 byte)	(Up to 255 bytes)

Figure 3-1 Overview of a packet.

At the beginning of the packet are the routing headers, followed by the message length, and the actual message. To determine whether the subsequent byte after the routing headers is the message length, bit 7 in the routing headers is checked. If bit 7 is set, the subsequent byte is a routing byte. However, if it is not, then the subsequent byte is the length byte.

The breakdown of a packet format is shown in Figure 3-2 (courtesy of IC-Routing Ltd[54]).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3 to bit 0
1				First Output Link
1				Next Output Link
0				Last Output Link
Payload length.				
Payload (up to 255 bytes).				

Figure 3-2 Packet format for a message.

When a message arrives from an input port of a router, the first destination byte is stripped by the router and used to determine the output port. The router then processes subsequent bytes until the message length byte is found. A connection is established between the input port and the output port of the router. As the message is routed via the router, each byte routed is counted until the count matches the value of the length byte. The router will then disconnect the connection path.

In the static mode, the router functions merely as a fixed connection path. To set a fixed path, the router must be explicitly programmed via its control port. Any message sent from a pre-set input port will be routed directly to another pre-set output port. There will be no distinction between routing headers and message payload as any bytes arriving will simply be relayed to the corresponding port. Figure 3-3 illustrates two fixed connection paths.

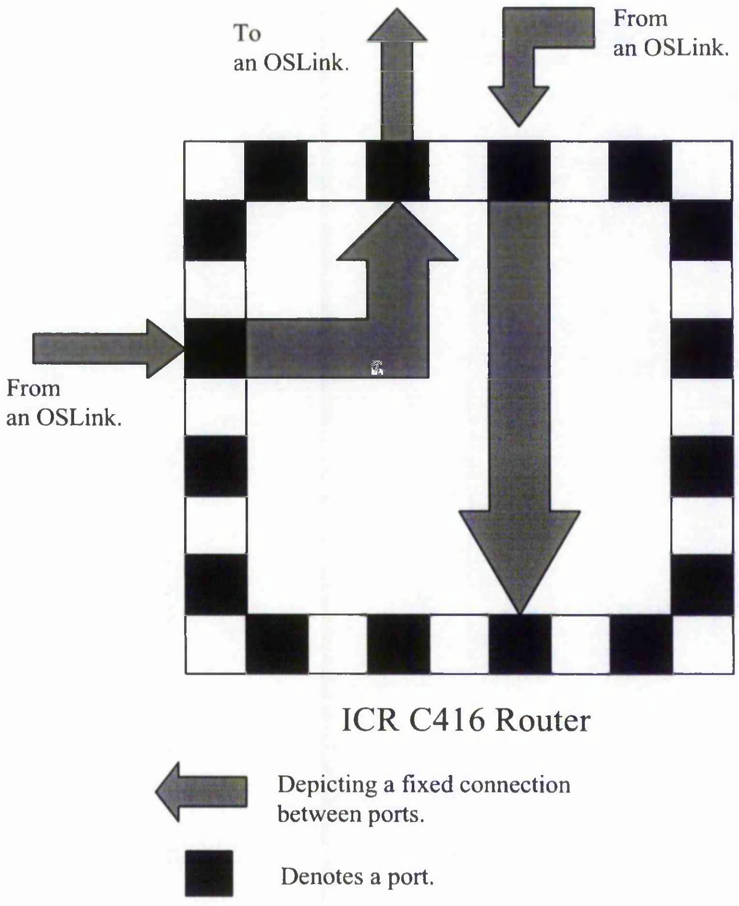


Figure 3-3 fixed routing path.

The static mode is useful when there is a need to boot a transputer. If two transputers are connected to a router and one is required to boot the other, and the router must operate in static mode. Transputers can only be booted if the image file is downloaded via one of their links without packetising the image file.

3.1.2 Custom StrongARM Interface Device.

As discussed previously, the StrongARM processing node contains a custom interface device, the SARNIC. The custom device, implemented in Field Programmable Gate Arrays (FPGA)[55], provides inter-processor communications and other functionalities. Features of the SARNIC are:

- Booting options.
- Communications support.
- Timer.
- UART.

Booting options.

Two booting options are available; through the OSLink or via the ROM embedded in the custom device. Since the size of the ROM is only 2Kbytes, the only program that could fit into it is a bootloader. The other option is via the OSLink, where there is no size limit imposed on the program. Booting via an OSLink is preferable, since the hardware system could boot directly by sending the file image of the OS via the router.

Communications support.

Traditionally, every processor that performs a communication has to wait until the bus is free before the processor is allowed to resume its activity. This method results in inefficiency, as precious time is wasted when the processor remains idle, while waiting for the communication process to complete. Therefore, the Direct Memory Access (DMA) technique is introduced. DMA is a capability that allows data to be sent directly from an attached device (such as an external link) to the memory on the processing node. The microprocessor is freed from involvement with the data transfer, thus speeding up the overall computer operation.

In the SARNET, communications initialised by the software will result in DMA being active. The DMA Engine, which is responsible for the communications will then access the main bus whenever the processor is not accessing it, to transmit or receive a message. An interrupt will be generated whenever the communication is completed. The OS will interact with these DMA engines to set up the communication, and clear the interrupt via memory mapped registers.

There are two OSLinks on the StrongARM Processing Node. Internally, there exists 4 DMA engines to handle communications; 2 engines for transmitting, and 2 engines for receiving. These engines are independent from the physical OSLink, i.e. any of the 4 engines could be active using any of the two OSLinks. The independence of DMA engines from the physical link has increased the network efficiency, as a free engine could be used to service the communication request on another link while the other engine is busy.

The communication system on the StrongARM node is capable of receiving interleaved messages. Interleaving of messages is described as two or more messages contained in two or more packets, arriving at the same physical link, at different times. The interleaving of messages is illustrated in Figure 3-4, where two large messages are packetised to conform to the router's protocol and each packet that forms part of the actual message arrives at the same destination at a different time.

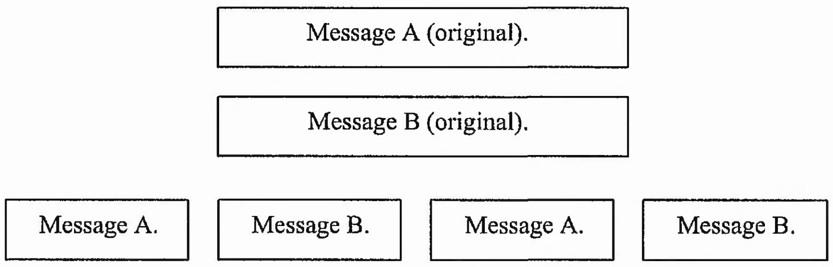
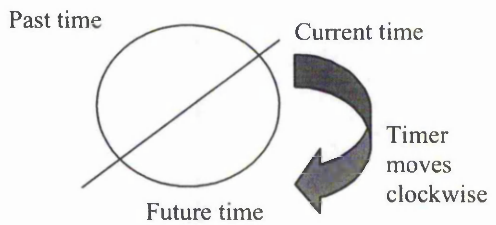


Figure 3-4 two large messages packetised and interleaved.

The hardware is able to receive two interleaved messages, transparently to the software. If the software had set up the necessary registers prior to the arrival of the two messages, the hardware would receive the expected incoming messages readily. However, if there were more than two interleaving messages, the software would need to deactivate whichever DMA engine that was not actively receiving a packet, and start another communication. Actively receiving a packet is described as the arrival of data streams within a packet. Hence, the OS must be able to support the communication mechanism.

Timer.

A timer is essential in any OS development, as there is a need to measure time quantum, to be used for various purposes including time-slicing a process. The StrongARM node has one 32-bit free-running timer register, with a resolution of 1 microsecond for every clock tick. The movement of the timer is cyclical, from 0x0 to 0xffffffff in hex values. The timer will wrap around after reaching 4,294,967,295 microseconds. However, the timing event has to be divided into two; future time and past time. Figure 3-5 shows the timing event.



The 32-bit timer.

Figure 3-5 the StrongARM node timer.

All times between the current time plus approximately 35 minutes and 47 seconds (4,294,967,295 microseconds) are considered to be in the future, and the rest of the times are in the past.

The timer could be configured to generate a timer interrupt, by writing a future time into a 'timer compare' memory-mapped register. When the time written matches the current time, an interrupt will be generated. However, the maximum length of the timing event is half of the maximum timer magnitude which the software could set. Longer timing periods could be set by additional software support.

UART.

A UART is a device that enables communication via a serial link. Specifically, it provides the StrongARM Processing Node with the RS-232 capability to communicate with the host PC via a low speed serial link. The UART is the only facility for dynamic debugging, as messages from the node could be displayed on the host PC, to ascertain which state the software is in. Hence, software modules have to be created to access the UART for debugging purposes.

3.1.3 Generic ISA Interface.

The new StrongARM network on its own is unable to interact with the user. Therefore, a generic ISA Interface device[109] was developed. The generic ISA Interface device is a standard I/O interface developed by the same research group for general purpose processor systems (aimed at PC implementation), using an Altera Flex PLD[56] to connect a PC to the embedded parallel computing system. Handling message routing with the ICR C416 router, the board provides connectivity between the host system (PC) and the StrongARM processor nodes. Consequently, image files residing on the PC could then be downloaded via the ISA interface to the intended processing node to boot the network. Application software had to be developed by the author to support downloading of software through this interface to the StrongARM processor nodes.

3.2 Software toolchains and language.

For constructing an OS, the choice of language largely depends on the availability of the toolchains, and the objectives involved. In the case of designing a new OS for the StrongARM platform, the objectives are:

- Language and toolchains have a wide appeal to the embedded community.
- Portability of the OS.
- Backward compatibility with the transputer systems.

Therefore, C/C++ seems to meet the requirements for the choice of language. Coupled with the availability of cross-compilers for C/C++ targeted at the StrongARM, ANSI C was chosen for both the kernel building and applications development.

The toolchains available for StrongARM are:

- The GNU cross-compiler, which is freely available[45].
- The ARM SDT 2.5 cross-compiler, which is proprietary[110].
- The GreenHills cross-compiler[111].

The new OS was initially developed using the ARM SDT 2.5 toolchains. However, the development toolkit was switched to the GNU toolchains, since the GNU is considered more attractive to the development community.

During development, some assembly language was used when necessary, however, again for portability, this was kept to a minimum.

3.3 Initial Feasibility Design Study.

This section will describe the initial stage of design and implementation of the basic internal structures of the new OS. A few descriptions are given as to why some of the techniques were used during development.

3.3.1 Virtual Channel Support.

The use of channels for external communication, via links, limits the number of global channels available to the local processes. Hence, a system with only two external links can only support four channels, two for transmitting, and two for receiving. To overcome this limitation, virtual channels must be supported. A virtual channel is a software support to enable multiple channels[57] to share one or more physical communication links. Each virtual channel supports unidirectional communication. Virtual channels are essential in supporting multiple, simultaneous inter-processor communications due to the limitations of hardware links. In transputer systems, virtual channels were not supported, and each link could only carry one pair of channels (virtual channel support was later provided by the T9000[48]).

In theory, there could be a large amount of channel communications taking place simultaneously over the two OSLinks provided by the StrongARM hardware. However, in practice, the number of channels is limited by the memory size of the processing node. Figure 3-6 depicts the virtual channel communications between processes running on different processors. Each virtual channel is exclusive to one process.

Since there is a need for simultaneous multiple channels communications, virtual channel support has to be provided by the new OS.

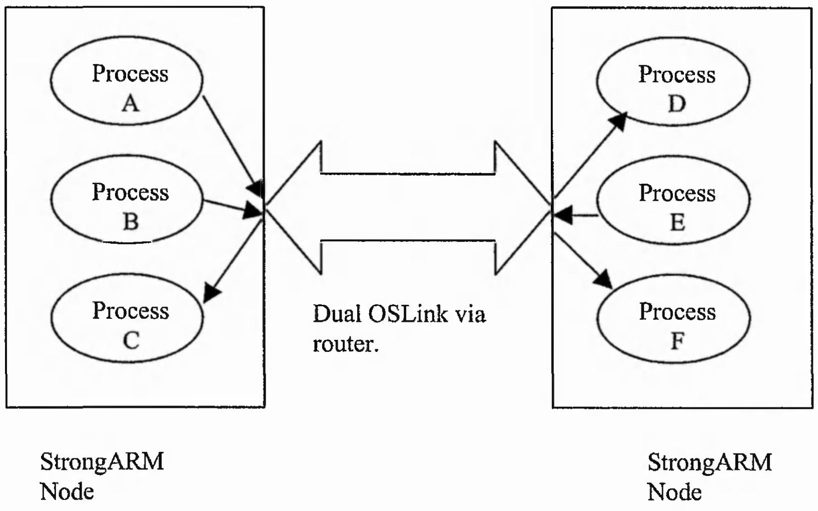


Figure 3-6 virtual channels communication.

The support for virtual channels has raised doubts about the backward compatibility issue. In the transputer system, most application designs used a single process functioning as a device driver to handle communication via links when there was more than one process that performed global communication. This is illustrated in Figure 3-7.

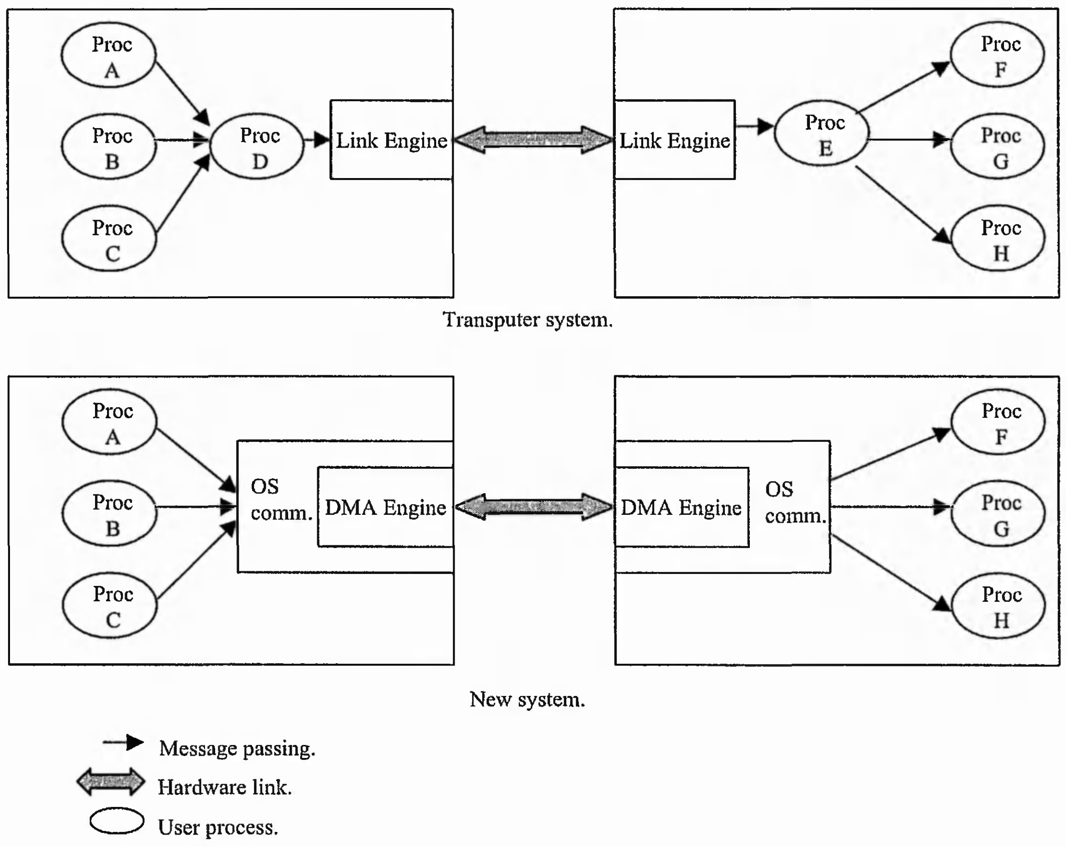


Figure 3-7 different message passing systems.

In Figure 3-7, process D and process E are acting as device drivers for the external link. Their role is taken over by the OS, when virtual channel support is made an inherent part of the OS.

3.3.2 Scheduling Policy.

When there are two or more processes that are runnable, the OS must decide which to run. Runnable is described as a state which the process is in, whereby the process is ready to run but not running yet, as it awaits its turn to execute on the processor. The decision is made by an integral part of the OS, called the scheduler. The decision algorithm used by the scheduler is called the scheduling policy.

There are many known scheduling policies. Choosing one of those policies, or formulating a new scheduling policy will depend on the objective [58]. Among the criteria that outline a good scheduling policy include:

- efficiency (to achieve a high CPU usage).

- fairness (to ensure that all processes get a fair share of the time slice).
- throughput (to achieve the maximum number of tasks processed over a period of time).

The following sections will briefly discuss the various scheduling algorithms available[33]:

- First come first served scheduling.
- Round robin scheduling.
- Priority scheduling.
- Lottery scheduling.
- Guaranteed scheduling.

First come first served

The First come first served scheduling mechanism will execute the first process until it has completed its task, before running the next process. The disadvantages include:

- processes get to monopolise the CPU until they have finished running.
- the OS will not entertain processes that require CPU time within a specified time.

Deemed as a primitive solution in modern systems, this scheduling mechanism is no longer considered attractive and thus hybrids with other rules are introduced.

Round robin scheduling.

One of the widely used algorithms, round robin will schedule each process in the process queue until all the processes have completed their tasks. Figure 3-8 illustrates the round-robin scheduling policy.

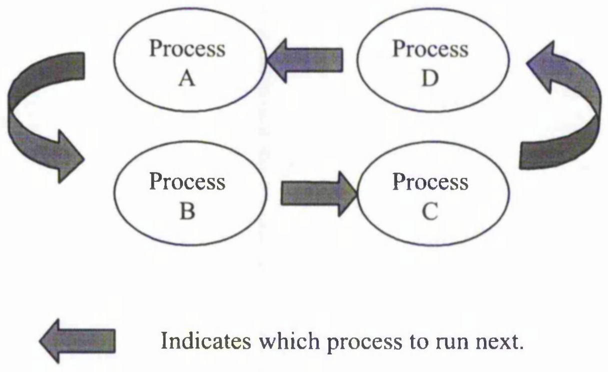


Figure 3-8 Round robin scheduling.

Each process is given the same time quantum to run, and upon expiration of the time, the next process will be scheduled to run. This scheduling policy ensures that every process gets a fair chance to run.

Priority scheduling.

Priority scheduling could be considered as an extension to the round robin scheduling policy. For the priority scheduling, each process has an associated priority level. The process with the highest priority level will be executed first, before the lower priority processes are allowed to run. There is however, a vast variation on how the priority scheduling mechanism is implemented[33]. Some may prefer to dynamically change the priority level of any process, depending on how long it has not been run, while others may choose to run lower priority processes even though there are high priorities processes that are runnable.

Lottery scheduling.

In lottery scheduling[33], all processes have the chance to run, but the process chosen to run is selected randomly. However, processes that are more important have higher chances to run. Other OS developers have increased the complexity of this scheduling mechanism with the introduction of various properties in the lottery system. For instance, any process that has just been run may have its chance to run again reduced to 0 within a period of time, or co-operating processes may exchange the chance to run with each other, when either one has to be descheduled.

Guaranteed scheduling.

This scheduling solution is to guarantee each process queued in the system, a fair share of the CPU time. This is more suited to dynamic task allocation, where the number of tasks to be run during the actual runtime has a huge variation. To deliver the guarantee, the amount of CPU time each process has is recorded by the system. A ratio between the amount of CPU time each process has and the amount of CPU time each process is entitled to, is calculated. The entitled CPU time is derived by dividing one time quantum by the total number of processes running. This CPU time will denote how much time a process is entitled to run within a time quantum.
For example,

T = 1 sec/n,
where 'T' is the time entitled to each process (per second), 1 sec is 1 second, and 'n' is the number of processes.

R = t/T,

where ‘R’ is the ratio between the amount of CPU time each process has run (t), and the amount of CPU time each process is entitled to (T).

The OS will then run the process that has the lowest ratio, until its ratio is higher than other processes in the system.

3.3.3 Selection of scheduling.

Scheduling policy is likely to remain as an active research area. With various circumstances and requirements stemming from an immense number of applications, no specific scheduling mechanism has the ‘one-size-fits-all’ solution. Moreover, it was shown that any scheduling mechanism that is suited for one specific application might be detrimental to another [59].

In the lack of the ‘general-rule-of-thumb’ for favouring any specific scheduling mechanism, the new OS adopts a scheduling policy that best suits the intended applications whilst providing backward compatibility. The intention was to apply certain traits of the transputer round robin scheduling for low priority processes, while allowing high priority processes to be time-sliced. The OS is designed to embrace the round-robin pre-emptive and co-operative scheduling policy for all priority queues. Pre-emptive denotes time-slicing for a process, while a co-operative strategy will rely on the current executing process to give up its CPU time. When both strategies are combined, this will produce a mechanism where a process is time-sliced, and yet it has the option to give up its CPU time before its time is up. The option to perform co-operative scheduling is provided in de-scheduling points, and exists when the process performs various functions, including: communications, timer events or system calls.

At a later stage, the OS design had to be modified to minimise the risk of functionality failure when porting transputer applications to the new StrongARM system. The modification involved changing the scheduling mechanism to a first-come-first-serve basis for high priority queues only, allowing high priority processes to monopolise the CPU. Since there is a risk of preventing the OS from regaining control of the system when a process is able to monopolise the CPU, the responsibility of guarding against this risk lies with the developer. All processes running on high priority must be kept short, and free from deadlock. Failure to do so is considered a design fault of the developer. The project of porting transputer codes will be described in chapter 6.

3.3.4 Number of priority levels.

A priority level is applied to a process, where a process with the higher priority level will get to run first. The transputer implemented only two levels of priority, high and low[48]. Normal application processes typically ran at low priority by default. These processes are time-sliced and have de-scheduling points. High priority processes were used for interrupts, device drivers and tasks that involve high-resolution timing.

The new system attempts to overcome the restrictions imposed by the transputer. Two levels of priority are considered too restrictive when programming modern applications. Too many levels of priority will introduce a high level of overheads during context switching, and make the application programming prone to errors. A trade-off is needed and therefore, the best method is to give the developer a choice to specify the number of priority levels needed in the targeted system during compile time. The option is easily implemented as macros, which could be easily modified to suit the need.

The current implementation of the OS has 4 levels of priority. The highest priority level is not time-sliced. The lower ones are time-sliced, and have a co-operative scheduling policy. Higher priority levels are run first. In this case, the backward compatibility is preserved, while overcoming the restrictions imposed by the transputer.

3.3.5 Buffering and non-buffering of messages.

In the transputer system, a channel provided one-way, unbuffered and synchronised point-to-point communications[48]. Any process that wished to perform communications could only resume once the other communicating end was ready.

In the SARNET, this inter-processor mechanism will not produce an efficient network. In the event that the receiving process on the targeted processor is not ready, the hardware link has to be stalled. On the sending side, the communications cannot be cancelled, since a few bytes have already been transmitted as the router is buffered. Therefore, one of the transmitting engines on the sending side and one of the receiving engines on the receiving side would be stalled. If a similar case happens on the other engine for both sides, then both nodes will be stalled and this may result in a deadlock.

Figure 3-9 illustrates the stalling of the whole system.

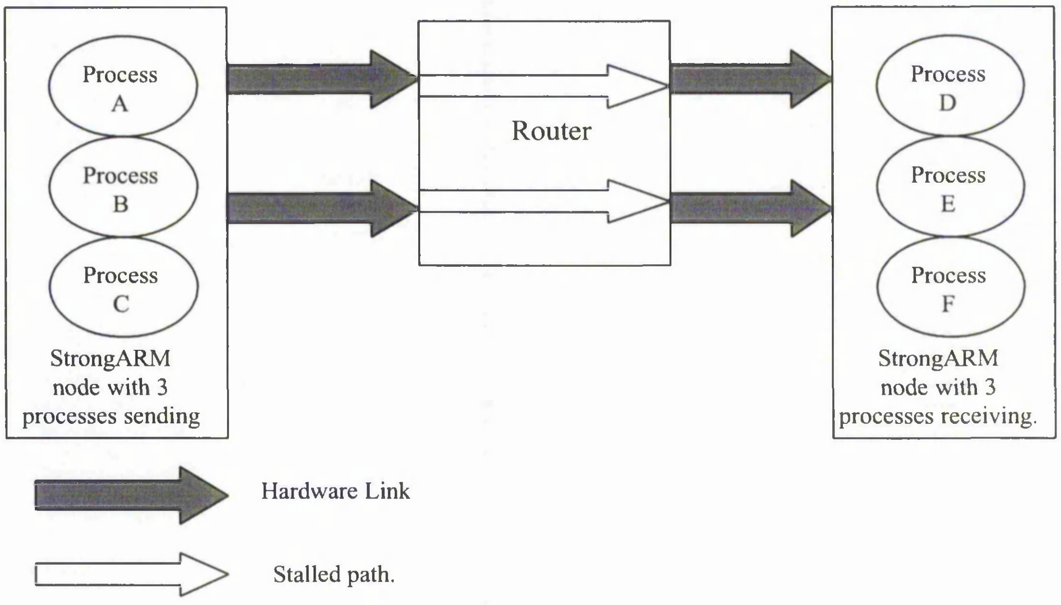


Figure 3-9 stalling of hardware links.

When Process A and B attempt to communicate with the busy processes, D and E, the links are stalled. Consequently, Process C cannot send any messages globally, and Process F cannot receive any messages globally until the links are cleared. In other words, the whole communication system for the affected node is temporarily halted, making the network inefficient. The situation worsens if only one hardware link is made available.

To keep the links free, buffering must be provided on the receiving side. There is no need to provide buffering on the transmitting side since any sending of messages is known by the system during runtime, in contrast to the receiving side where messages could arrive at any time. Furthermore, on a one-way communication, buffering on the receiving side will be sufficient to keep the communication path free.

Ring buffers, which are simple software implementations of buffers, are used in the OS design. The number of ring buffers for each message on the receiving side is configurable during compile time by the developer. The size of each receiving buffer will be the same as the size of each expected incoming message for a specific channel. Hence, each channel has different buffer size, determined in the procedural call during communication. This removes the inefficiency, which would have occurred if the same buffer size were used for every message (as large buffers tend to be wasted for buffering short messages). If there is any buffer overrun during runtime, a fatal error will be generated. Buffer overrun is attributed to a design fault, as the developer has failed to estimate the system's requirements. Another possible error is the designation of the low priority to an active receiving process, resulting in the failure of the process to clear its ring buffers within a short period of time. This design error has to be guarded by developers, as buffer overrun should not occur under normal runtime.

For soft channel communications, no buffering is provided and the model of communication follows closely that of the transputer.

4. Sarnux implementation and structure.

Section 4.1, 4.2 and 4.3 discusses the design and implementation of the OS, SARNUX. An explanation of the system design methodology is first given, before a general overview of the system. Detailed information is then given on the core modules that form SARNUX. Other issues affecting the design decisions are discussed, with their implications on the implementation given. Development of SARNUX is focused on the embedded parallel environment.

Section 4.4 will describe two tools developed in aiding parallel programming; the Windows GUI and the Configurer. The reason why these tools were developed is because the difficulty in programming a parallel application is further increased by the concern about the low-level functionality of specifying the routing headers for inter-processor communications. A network specification tool based on a Windows GUI and a Configurer, which is similar to the transputer's Configurer[21], is used to assist in automatic generation of headers for channel communications. In this way, the application is made independent from the hardware during development, which enhances the ease of portability across different locations.

4.1 System Design.

Development of a new OS is technically challenging. The reasons are due to

1. lack of standard software engineering notation (e.g. Booch[60] or UML[61]) and a proven methodology,
2. lack of mathematical models like Formal Methods[62] to evaluate the OS design in terms of correctness, functionality and design decisions.

The reasons for this lack of standards may be attributed to the different historical practices of system development, hardware support, and toolchain availability.

It is desirable then to have a common framework to achieve:

1. easier maintenance,
2. shorter design time,
3. and reduced costs.

However, before a common framework could be derived, there is a need to standardise kernel interfaces, for instance providing common interfaces for various components in the OS as opposed to programming without a proper structure, or 'spaghetti programming'.

One such standardised implementation for applications programming would be the Portable Operating System Interface (POSIX)[63]. POSIX is a set of standard OS interfaces based on the UNIX OS. The interface is common among various OS's, allowing a common interaction of a single application with different OS's that comply with the POSIX standard. Using POSIX enables applications developed on one OS to be ported to another without re-coding. This standardisation has proved difficult and challenging to implement, and thus was not adopted in this work. The issue of formulating a common framework for developing an OS remains an interesting area of research for the global community.

An easier and more practical approach is to adopt a general modular design, or a top-down design methodology. This design methodology was used to develop SARNUX, where the OS structure was broken down hierarchically into several modular parts before linking them together to form a unified and functional system. Every module was further broken into smaller parts, allowing the smaller parts to be tested independently for functionality and correctness before being integrated to form higher level module. This approach also enabled the reuse of components in other modules.

4.2 System Overview

In some OS internal structures, like Linux[34], the functionality of each sub-system is provided in modules, which overall gives the OS functionalities. In the case of SARNUX, there are six major modules that form the core backbone of the OS:

1. The Process Manager, for handling process scheduling and queuing.
2. The Transmitting Manager for handling outgoing external communication.
3. The Receiving Manager for handling incoming external communication.
4. The Software Channel Manager for handling internal communication.
5. The Exception Handler for handling software and hardware errors and interrupts.
6. The Timer Manager for handling timer events or timer notifications, for example, time-outs.

Most of these modules are coded in C language to provide portability. However, some low-level functions have to be coded in ARM assembly language for efficiency or usability since a high level language will incur performance penalties with higher overheads. Portability is still maintained, provided there is extensive documentation of these low-level functions, which are essentially machine dependant. Optimisation could be carried out for high-level functions, by generating the assembly codes using an ARM compiler and analysing them to discard overheads.

All the modules interact with each other to provide the abstraction needed by the application program. The application program makes calls or sends interrupts to the kernel, requesting services. Each program, which is termed a process, will have its own 'living' space and stack space.

4.2.1 The Process Manager.

The function of this module is to manage the process queues. With a configurable number of process queues during compile time, the process manager will search each queue hierarchically from top to bottom, starting from the highest priority queue until an active process is found. The active process will then be executed, and if the process has the highest priority, it will not be time-sliced. Otherwise, all other processes with lower priority will be time-sliced.

The process manager will co-exist with other modules, and it will perform the following functions as requested by other modules:

- To block a process.
- To un-block a process.
- To search for a particular process, identified by certain attribute of a process.

The design implementation is by using a chaining mechanism, or linked-list. Three pointers are used for each priority queue:

- A pointer to the beginning point of a queue.
- A pointer to the end point of a queue.
- A pointer to the current position, or the current process being run in the queue. This third pointer is to enable the insertion of another process into the queue at the appropriate location.

A process has its own living space: its own private registers and stack space. The size of the default stack space is again, configurable during compile time. Another option is to request for a specific stack size during runtime. A typical stack space would be 512 bytes[65], including the space needed for a context save. The stack requirement is higher, compared to the transputer and is attributed to the 4 byte stack access in the StrongARM architecture. The context of a process is described as the values contained in its registers, which include scratch registers and status registers. The context must be saved if the process is to be de-scheduled and re-scheduled at a later stage. The act of saving the context and re-scheduling another process is called context switching.

The process space implemented in SARNUX is illustrated in Figure 4-1.

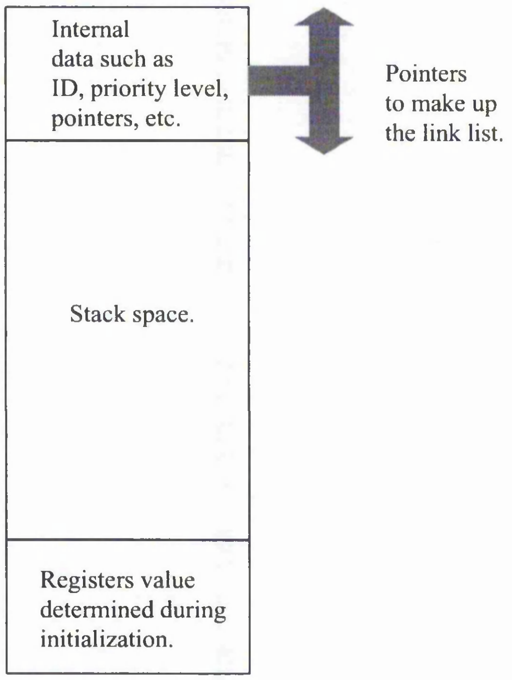

Internal data such as ID, priority level, pointers, etc.
Pointers to make up the link list.
Stack space.
Registers value determined during initialization.

Figure 4-1 implementation of a process in SARNUX.

The stack grows upwards, as determined by the cross-compiler. The internal data is maintained by the OS for housekeeping and contains pointers to make up a linked list.

To achieve short latencies during context switching, assembly code was used at the expense of portability. However, the usage of assembly code has been kept to a minimum. Assembly code is used only during context saving, and context loading.

In comparison, the transputer avoided full context saves by using de-scheduling points in the application code. These points indicated to the processor that all the 4 scratch registers were empty, thus a full context-save was not required. Should a high priority process pre-empt a low priority process, a full context-save would still be performed onto a temporary stack. High priority tasks would not be interrupted, and would continue running until finished. After that, the context from the temporary stack would be loaded and the low priority task interrupted previously would be resumed.

For the StrongARM processor:

1. There are no de-scheduling points generated by the ARM compiler.
2. There are 16 registers to be saved for each process.
3. High priority tasks must be interruptible by the OS to enable the OS to attend to hardware interrupts and other communication related events. Interruption denotes full context saving of the high priority process, to allow the OS to use the scratch registers.

In view of the differences in the architectures, context saving in the StrongARM has higher overheads compared to the transputer. However, the transputer has the disadvantage of passing the overheads to the user process in terms of a performance penalty. A reduced register count forced the application to access memory, which had slower access time. In contrast, the StrongARM has a higher number of registers available to the application software, thereby reducing the memory access but it has to save all the registers during a context-switch.

4.2.2 The Transmitting Manager.

The Transmitting manager, as its name implies, handles the global transmission of messages to the StrongARM network. It handles requests by local processes that wish to send messages, to other processes running on different processors, via channels. A channel is implemented as a structure in memory and provides the application a synchronisation mechanism. Channels are used in all the communication modules in the system, for both global and local communications. Hence, the calling process must pass a channel reference to the OS whenever it makes a system call for communication.

In the event of a process requesting transmission, three parameters supplied by the calling process are provided:

1. The message pointer.
2. The address length.
3. The message header.

These parameters are needed by the hardware registers[16][64] to start the DMA engine. The message headers are provided in a different file, as macros and are included into the system during compile time. Message headers are generated automatically by the Configurer, which will be described in a later section. Message headers include the routing headers and message ID. The unique message ID must match the message ID on the other communicating end. To start transmission on a DMA engine, the following 32 bit registers must be initialised:

- Two header registers, the low header and the high header registers. By combining both, the hardware could support up to 6 header bytes.
- The Address register, which points to the physical memory location of the message. This 24-bit register could only cover 16 megabytes of main memory. The other 8 bits are ignored.
- The Transmitting length register, to count the number of bytes transmitted. The 16 least significant bits are the counter, while other bits are used as status and control bits.

After receiving a request from a local process, this module will determine whether any one of the two DMA engines is busy transmitting. If both are busy, then the process is de-scheduled and blocked, and the request to send is placed in the transmission queue (which is implemented as a linked list in SARNUX). The message will be sent once any one of the two DMA engines is free, and upon completing the transmission, the process will be unblocked through a request to the Process Manager module. An overview of the operation is illustrated in Figure 4-2.

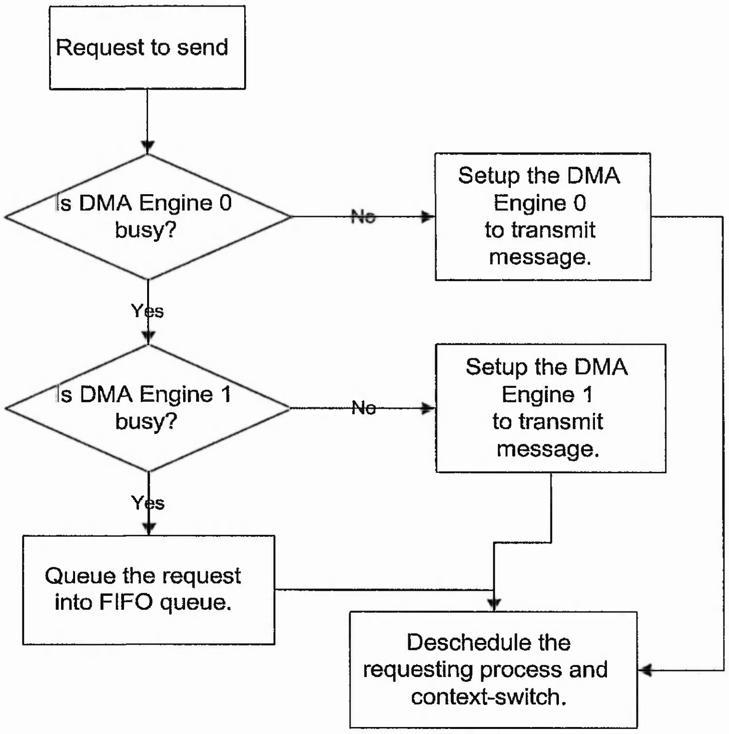

Request to send
Is DMA Engine 0 busy?
No
Setup the DMA Engine 0 to transmit message.
Yes
Is DMA Engine 1 busy?
No
Setup the DMA Engine 1 to transmit message.
Yes
Queue the request into FIFO queue.
Deschedule the requesting process and context-switch.

Figure 4-2 flow chart for request to send global message.

Upon completing the transmission of a message, the OS will unblock the communicating process, and will then process the next request to send in the queue. This mechanism is illustrated in Figure 4-3.

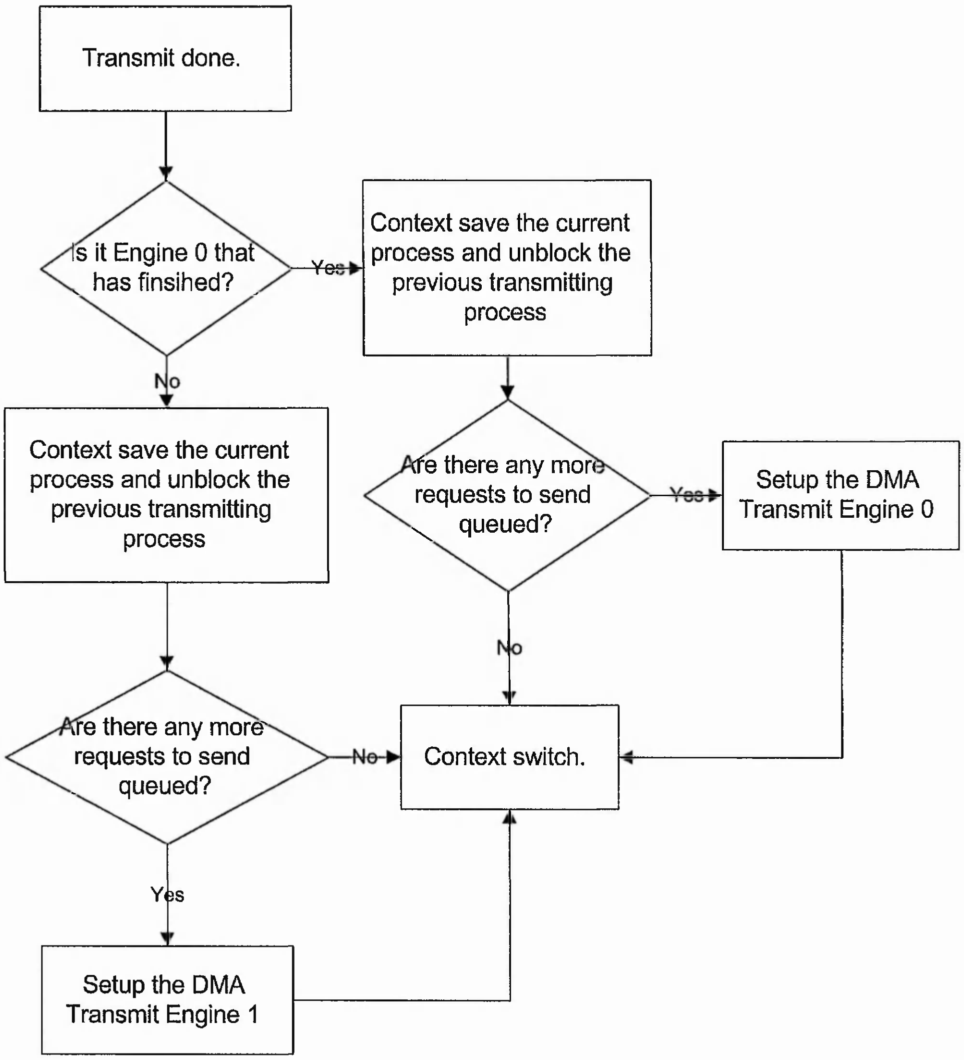

Transmit done.
Is it Engine 0 that has finsihed?
Yes
No
Context save the current process and unblock the previous transmitting process
Context save the current process and unblock the previous transmitting process
Are there any more requests to send queued?
Yes
Setup the DMA Transmit Engine 0
No
Are there any more requests to send queued?
No
Context switch.
Yes
Setup the DMA Transmit Engine 1

Figure 4-3 transmit done flow chart.

Two pointers are used to denote the activity of both transmitting DMA engines. Each pointer correlates to a specific DMA engine[64]. Pointers point to channels, and a pointer value of null denotes that the DMA engine is free. In the software design, a transmitting DMA engine is tied to a specific OSLink, although the hardware does not enforce this rule. In effect, each transmitting DMA engine will operate via a specific hardware link by specifying the 'PHYSICAL' bit in the Length Register. A bit '0' will mean link 0, and bit '1' will mean link 1. The reason for tying a DMA engine to a link is that any OSLink which is busy must have a DMA engine attached to it, and hence it would be inefficient to have both engines sharing the same link. There is no difference in the routing headers when both links are connected to the same router, as messages with the same routing headers will be sent to the same output link even though it comes from two different input links. Figure 4-4 illustrates the internal view of the queuing system.

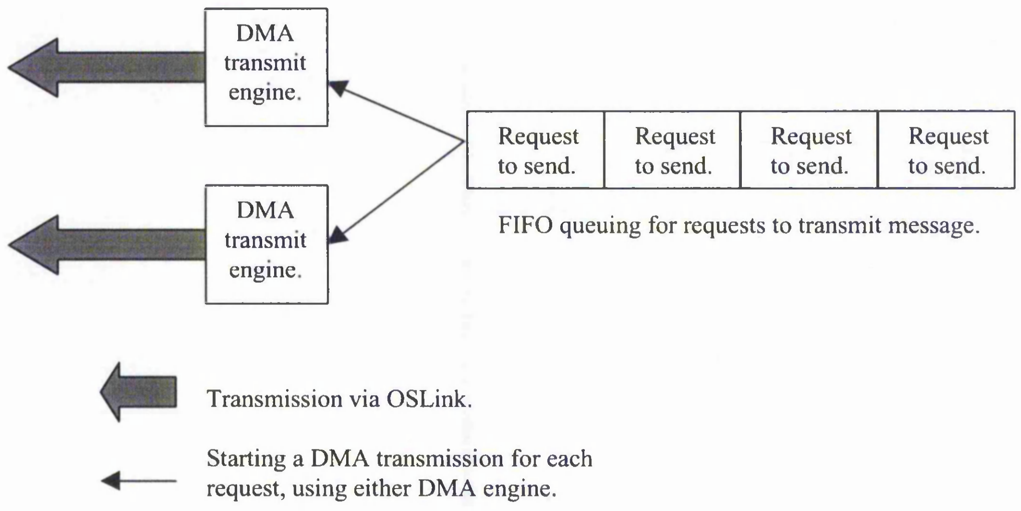


Figure 4-4 internal queuing for transmitting via DMA engines.

Buffers are implemented for this transmitting module because of the caching strategy used in the system. To perform DMA to send a message, the memory region must not be data-cached. However, since most of the data area in the application is unknown until runtime, caching these scattered memory regions is deemed unsuitable during runtime. Therefore, a buffer is used to copy the outgoing message to the non-cacheable memory region. For the whole of the transmitting module, only 2 buffers are used, since each engine can only send 1 message at a time. The size of each buffer is user-specific, and could be changed to suit the user's needs.

For debugging purpose, an optional channel with a time-out is available. This channel (that has a higher operating overhead) is designed to wake up the calling process if the maximum time set for the completion of communication expires whether the communication completes or not. A value would be returned to the calling process to indicate the success of the communication, and the course of action to be taken is left to the developer. The same mechanism was offered by the transputer.

4.2.3 The Receiving Manager.

The Receiving Manager module is responsible for handling incoming messages from the network. The implementation of this module is more complex than that of the Transmitting Manager module because Ring buffers are implemented to improve the efficiency of the network. Buffers are used whenever a message has arrived, and the process is not yet ready to receive the message. The total number of buffers for each channel is user-specified, and this value is contained in a header file.

The following parameters are supplied by the application, and used by the DMA engines:

- Message length.
- Address pointer to the message.
- Message ID.

Message ID is used to determine which channel the message belongs to. The ID value is obtained from a file generated by the Configurer.

In the system, two separate queues are maintained. One queue is for those requests to receive messages by the local processes while the other is to register the existence of any incoming messages throughout the runtime. A process must inform the OS of the incoming messages it expects during runtime by registering them. This is done transparently whenever a channel is initialised during startup. However, the developer must place the initialisation of all channels at the beginning of the code. This is because during initialisation, the receiving interrupt is disabled. It will be enabled again after all processes have initialised their respective channels. The OS is able to determine whether all channels in the system have been initialised by comparing the number of initialised channels with a figure contained in another file. The Configurer will provide this figure, which determines the total number of channels for a particular system. This method is needed since the system cannot buffer incoming messages without prior knowledge of the size of the message. The downside is that this approach requires a specified coding practice for the developer. This practice is not too difficult to enforce, since the same principle, that all soft channels must be initialised, applied in the Transputer 3L library. A possible different approach would have been to use a maximum message length for each buffer, but since a message length could reach 64Kbytes for each channel, this approach was deemed impractical.

The OS has to support the interleaving messages with the exception that the interleaving of 2 messages is handled transparently by the hardware. Hardware support of channel switching is desirable since it reduces the OS intervention in supporting virtual channels. Since in practice, there is a limit to the number of virtual channels supported by the hardware, the OS has to handle any number beyond this limit.

In the case of 3 messages interleaving, the Receiving manager would:

- Deactivate the DMA engine that is currently receiving interleaved messages.
- Save the message context of the previous interleaved message, including the address to be stored (the address is incremented after each word stored by the hardware) and the remaining length of the message (the length of the message is decremented after each word received).
- Start receiving the new interleaved message.

Should the arrival of the previous interleaved message continue at a later time, the context saved previously would be restored to continue the disrupted communication. This support for virtual channels and interleaving marks an important difference to that of the transputer system. The flow chart for this Receiver Manager module is illustrated in Figure 4-5.

From the software point of view, the link the message arrives at is insignificant. The hardware defines that receiving of messages will be done transparently to the software if the following registers are initialised:

- Receiver Header Register. If this register has been initialised prior to the arrival of the message, no new message interrupt would be generated.
- Address Register. This should be the buffer address, where the DMA engine will store the incoming message.
- Receiver Length Register. To inform the DMA engine of the maximum length of the incoming message. The 16 LSB are used for the counter, while the rest are status and control bits.

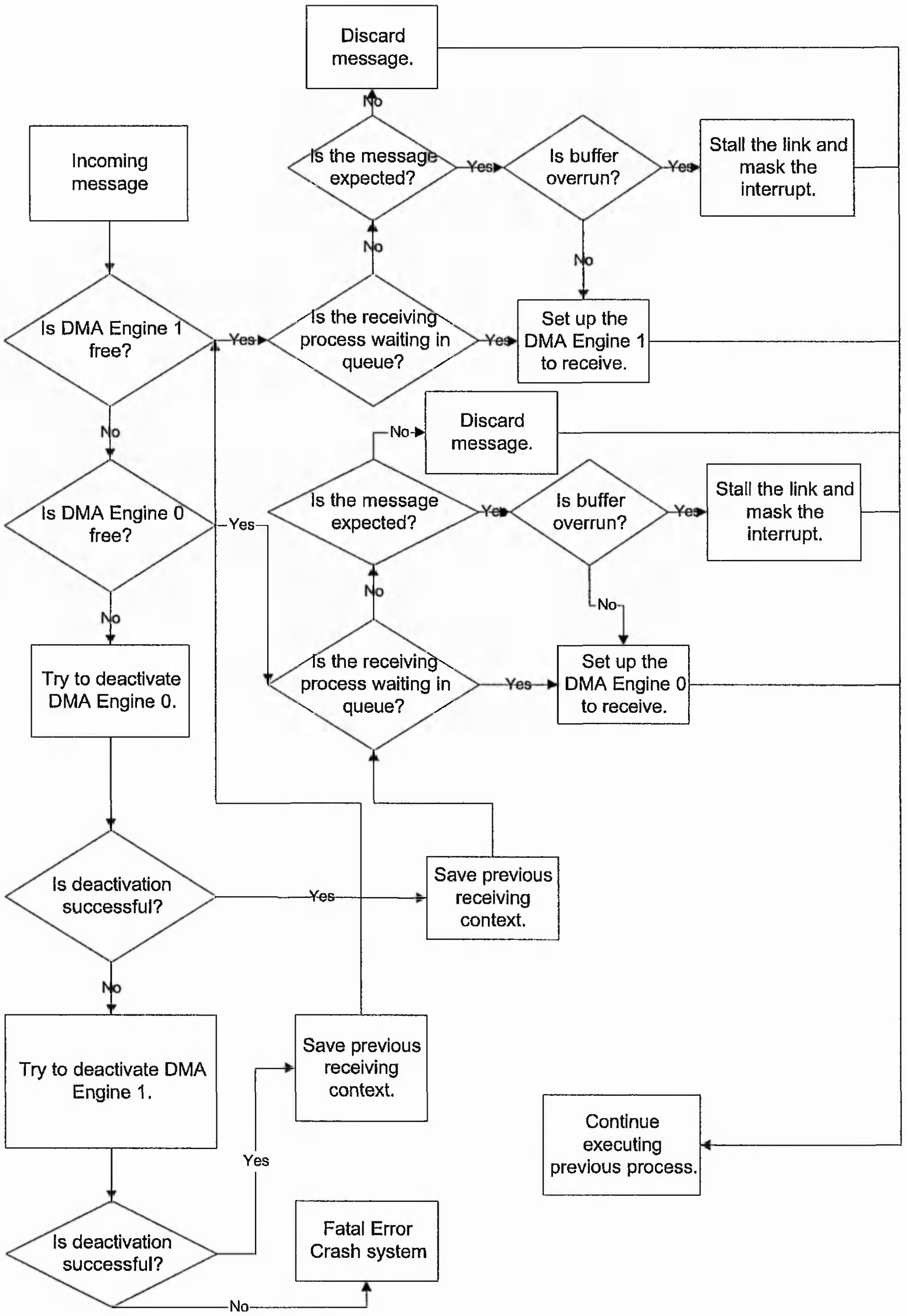

Discard message.
No
Incoming message
Is the message expected?
Yes
Is buffer overrun?
Yes
Stall the link and mask the interrupt.
No
No
Is DMA Engine 1 free?
Yes
Is the receiving process waiting in queue?
Yes
Set up the DMA Engine 1 to receive.
No
Discard message.
No
Is DMA Engine 0 free?
Yes
Is the message expected?
Yes
Is buffer overrun?
Yes
Stall the link and mask the interrupt.
No
No
No
Try to deactivate DMA Engine 0.
Is the receiving process waiting in queue?
Yes
Set up the DMA Engine 0 to receive.
Is deactivation successful?
Yes
Save previous receiving context.
No
Try to deactivate DMA Engine 1.
Save previous receiving context.
Yes
Continue executing previous process.
Is deactivation successful?
Fatal Error Crash system
No

Figure 4-5 flow chart for incoming messages.

A different flow chart is used when a process requests to receive. This diagram is illustrated in Figure 4-6.

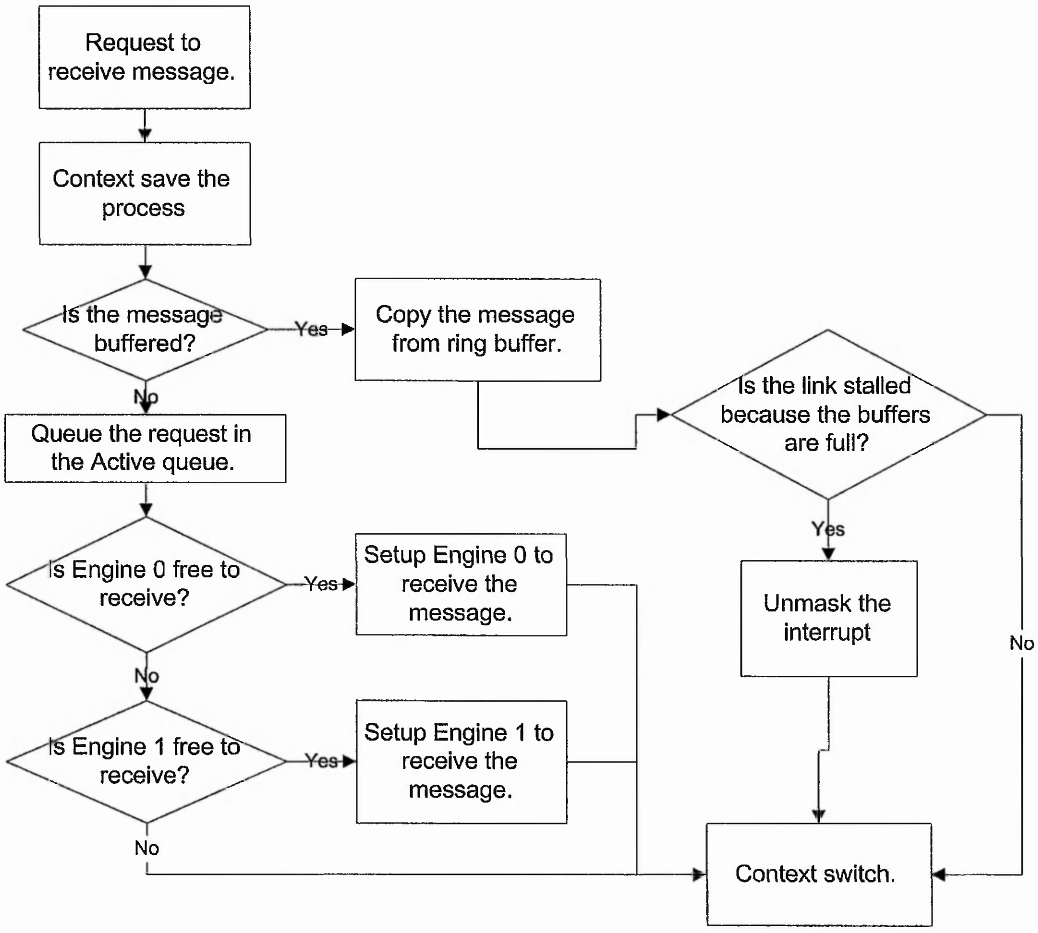

Request to receive message.
Context save the process
Is the message buffered?
Yes
No
Copy the message from ring buffer.
Queue the request in the Active queue.
Is the link stalled because the buffers are full?
Yes
No
Is Engine 0 free to receive?
Yes
No
Setup Engine 0 to receive the message.
Unmask the interrupt
Is Engine 1 free to receive?
Yes
No
Setup Engine 1 to receive the message.
Context switch.

Figure 4-6 request to receive flow chart.

When the system has completed receiving a message, it will first check whether the process, which the message belongs to, was previously blocked or not. If it were, then the process would be unblocked. The method is illustrated in Figure 4-7.

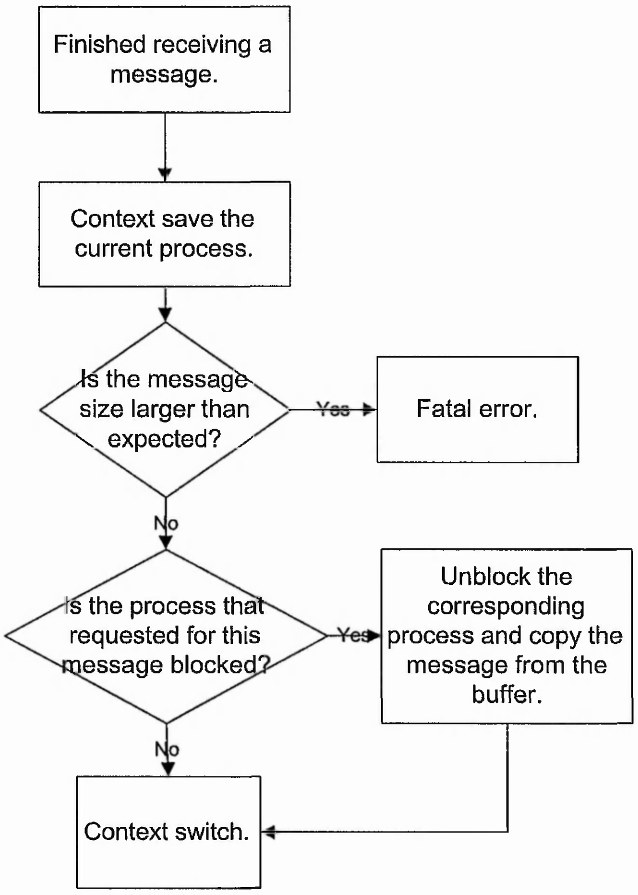

Finished receiving a message.
Context save the current process.
Is the message size larger than expected?
Yes
Fatal error.
No
Is the process that requested for this message blocked?
Yes
Unblock the corresponding process and copy the message from the buffer.
No
Context switch.

Figure 4-7 receive done flow chart.

Similar to the Transmitting module, there is an option to receive messages with a timeout for debugging. This higher overhead option will wake up the receiving process even though it has not completed the communication. Since the term 'completion' does not include communication that has started, the calling process might wake up while the message is actively being received. In such a case, the developer has to handle the situation explicitly.

To determine errors, the receiving module will test for the message length received. If the message received is longer than expected, a fatal error will be generated. The DMA engine would discard any extra bytes, preventing vital information from being overwritten in the memory. Even though no information is lost, the error must be generated to ensure rules are not broken. The OS will not take any chances on relying on this feature in the hardware, as different hardware may have different features.

4.2.4 The Software Channel Manager.

The Software Channel Manager handles soft channel communication. Communications between local threads are handled via this module. This module synchronises the unbuffered message passing via channels transparently.

In this module, the system relies solely on the flags in the channel structure for synchronisation. Each channel is shared by two processes for communication. When a request to send is called, the system will first determine whether the receiving process is ready, by checking a flag in the channel. If the other end is ready, the system will then copy the message (using two memory pointers provided by both processes) before requesting the Process Manager module to unblock the receiving process (indicated by a pointer in the channel). However, if the other end is not ready yet, the calling process would be blocked and the pointer to it would be stored in the channel structure. The flag indicating the sending process is ready will then be set. This mechanism is illustrated in Figure 4-8.

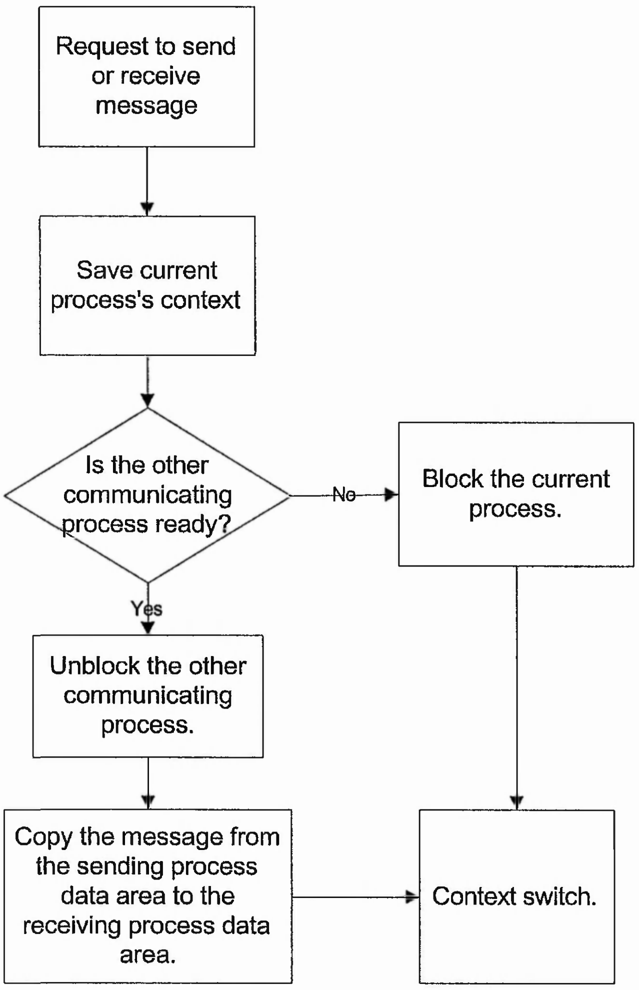

Request to send or receive message
Save current process's context
Is the other communicating process ready?
No
Block the current process.
Yes
Unblock the other communicating process.
Copy the message from the sending process data area to the receiving process data area.
Context switch.

Figure 4-8 Soft channel flow chart.

In addition to the facilities, alternation is supported. Alternation is described as a 'guarded input' against multiple channels, where a process is expecting several incoming messages simultaneously. The sequence of message arrival is unknown, and the process will accept the first incoming message. This mechanism is essential in building a deadlock-free system, and is provided by the transputer system as well.

Optional channel communications with timeout is available similar to the Transmitting Manager module. Again, this is similar to the facility provided by the transputer system.

4.2.5 The Exception Handler.

The Exception handler module handles the entire hardware and software interrupt module. It is written in assembly language for fast performance, and relays control to other modules for handling these events. Control is passed, after determining which procedure to call with the corresponding hardware or software bits. Hardware and software bits are obtained from the hardware register, denoting an appropriate event.

4.2.6 The Timer Manager.

The Timer Manager handles all timing events. The implementation of this module uses a linked list, to keep track of all the timing events. Other modules could request various timing events; for instance the Process Manager to time-slice a process or the Transmitting Manager to obtain a timeout for channel communication.

Since the hardware timer implements a wrap-around timer, the software has to take note of several issues including whether the time requested has already passed. In addition, a minimum tolerance is inserted, to ensure that any timing event (less than the minimum tolerance is considered past time). The reason for this is because the overheads in the switching may result in expiration of the time before any process could run. Expiration of time may result in context switching, thus incurring a performance penalty. The minimum tolerance is configurable, and a typical time would be less than 5 microseconds[65].

The time resolution on a transputer differs on different priority queues. For high priority queue, the time resolution is 1 microsecond. On a low priority queue however, the resolution is 64 microseconds. The reason is backward compatibility with the transputer, which has two different timers, each with a different resolution for different priority queue.

The list of timing events include:

- Time slicing.
- Sleeping period of a process.
- Timeout on transmitting a message globally.
- Timeout on transmitting a message locally.
- Timeout on receiving a message globally.
- Timeout on receiving a message locally.
- Waiting period of a process. This period differs to that of the sleeping period, as the process will wait until the timer reaches the time specified, as opposed to the time quantum used in the sleeping period.

4.2.7 Other system support.

Apart from those modules that form the core backbone of the OS, there are other smaller supporting modules to handle other miscellaneous tasks. These small modules include:

- Heap Manager.
  This module handles the memory system, allowing other processes to request for freeing up memory space. Implemented as linked list, the heap is managed by chaining all the free space together. The dynamic memory allocation facility was a standard requirement for most applications developed based on the transputer system.

- Semaphores.
  Semaphores are used in some operations to guard against concurrent access to certain resources or critical regions. They should not be used for inter-process communications, as channels are available. One example of the usage of semaphores includes 'par_printf' for concurrent access to the UART.

- UART Driver.
  The hardware provides a UART communication port to enable the display of text messages via a serial link connected to the serial port of a desktop PC. The OS supports the UART as a form of debugging facility, whereby developers could perform a printout to a PC to determine the flow of their application program. This debugging facility is also used by the OS to display warning messages during trial runs.

- Stack walkback support.

  For debugging purpose, a stack walkback is provided. The debugging facility is provided by building additional tools to complement the tools available from GNU[66][67].

4.3 Other OS development issues.

During the development of the OS, there are additional issues that have to be considered, which will affect the design phase. These include stack-overflow checking and non-caching of the DMA memory region.

4.3.1 Stack Overflow checking.

The difficulty in debugging an embedded system failure makes stack overflow checking an important feature. At the moment, the OS checks for stack overrun explicitly and thus the overhead of constant stack checking degrades the performance. However, the deterioration in performance is justified by the benefits gained, saving valuable time during system development. In the final release version after the system is deemed stable, these explicit checks could be removed by re-compiling SARNUX and specifying different options to exclude the checks, and hence improve performance.

The automatic stack overflow-checking feature has not been implemented in the GNU compiler. It involves automatic generation of stack checking codes upon each entry into a function and is important in preventing system failure during runtime. Modifying the GNU compiler source code could provide for the automatic generation of stack checking by the compiler.

An alternative to the explicit stack overrun checking is implementing memory protection[68], which also has the advantage of protecting the system against malicious users. However, since no protection is needed against malicious users in embedded systems, and there is a performance penalty in switching the page table for each process during a context-switch, the benefits gained may not justify the performance degradation. Furthermore, third party software cannot be included in the application since the memory location used by variables in the third party software package is unknown during compile time. Consequently, the permission to access certain parts of the physical memory by each process could not include these unknown locations, and thus the memory protection mechanism will generate an error whenever an access to these variables is performed.

Critics[69] have argued that implementing a memory protection mechanism to guard against stack overrun is more elegant, since stack overrun could be prevented before vital information is overwritten. However, the outcome for both mechanisms under normal operation mode should be the same: the system should crash and an error would be flagged, with no chance for recovery. This error is considered to be a design fault, as the developer has failed to estimate the required stack size. Hence, memory protection is not implemented for stack overrun checking.

4.3.2 Non-caching of the DMA memory region.

Caching of a DMA memory region on the receiving buffer may result in a thread accessing old data. This could occur even after the communication has taken place via DMA, in the event of a data cache-hit. However, if the same memory region were not data cacheable, there would be higher overhead for a thread to process that data.

In some modern processors that have a cache coherency capability, like the PowerPC [70], DMA does not pose any difficulty since there is hardware support to snoop the bus to maintain cache coherency. In the PowerPC, activities between processor cache, the memory unit and the device logic are co-ordinated. Each designated bus master must follow certain rules during a bus access. When a DMA write to a cached location occurs, the processor invalidates or overwrites the cached copy. When a DMA read from a cached location occurs, the processor sends a signal to the memory telling it not to put data on the bus: that it is supplying data from the cache instead. This mechanism removes the need for additional software support to ensure cache-coherency.

However, since the StrongARM does not have this advanced capability, additional software support is required. Caching for the DMA memory region is disabled, and the message received is copied into the thread's cacheable memory region. In this case, the overhead is substantially lower if the same data is accessed several times during execution.

The strategy is implemented by specifying the control bits in the page table, which is used by the processor in forming a logical to physical address translation. The memory layout is partitioned into a few regions, all of which are cacheable except for the receiving buffers.

4.4 New Tools for aiding parallel programming.

In parallel programming, low-level abstraction often results in difficulty in writing codes for routing messages from one processing node to another. For applications using the Internet, this problem was solved by using a domain name server (DNS) [78], which resolved a host name to the actual unique IP[79] address in digits, denoting the other end of the communication. The destination was represented by a unique name, which had a higher abstraction level than the digits that were actually the destination address. However, for embedded systems, this solution is impractical, as a dedicated node must be available to resolve the names. Moreover, the DNS technique eliminates the real-time response for an embedded system. Therefore, an approach to produce the routing headers is to provide a high layer of abstraction to the programmer to ease the burden of providing them manually.

Transputer[80][81] systems in the late 80's provided a tool called Configurer[82], where the network layout was expressed in syntax, or scripts. To adopt this user-friendly feature for the SARNET, two tools were developed: a Network Specifier based on the Windows GUI to assist in graphical mapping of parallel StrongARM systems, and a Configurer tool similar to that of a transputer systems, called the NTU-Configurer. The main difference between the two Configurers is that the new tool employs Artificial Intelligence (AI) to generate headers for channel communications. The software system developed is meant to port previous applications, targeted at transputer systems, to the new SARNET platform.

The development of these tools complements the underlying developed OS, running on the SARNET in providing a greater level of abstraction in parallel programming to the end user. The Network Specifier and NTU-Configurer were developed since existing GUI tools like Magnum Plus Manager[83] and Peakware [84] are either generally tailored for dynamic applications using TCP protocol (targeted at graphical network management applications) or do not support routers and virtual channels implementation. Therefore, for the intended platform and application, tools have to be developed to enhance the ease of parallel programming in static embedded multi-processors systems, which support routers and virtual channels.

4.4.1 The Network Specifier.

The Network Specifier is a front-end tool, written in Visual C++ 6.0 [85][86] and using Microsoft Foundation Classes (MFC) for graphics manipulation. This tool will give developers an easy way to describe the hardware network, specifying any connecting channels between two processes. The channels could be external or internal, and have unique channel names. In the GUI menu, a user could specify one or more of the following:

- A processing node, with two links available for connection to any router;
- A router, with 16 links available for connection to any router or processing node;
- An OSLink, as a hardware link between any of the processing nodes or routers;
- A thread, representing a single user process, that must reside on a specific processor;
- A channel linking between two threads, representing the actual message communication.

Each item could be added, removed or dragged dynamically around the viewport. A viewport is a small window, contained in the GUI to enable the developer to represent their hardware layout graphically using predefined figures. Additional information such as a filename is required when adding a processor. For all the items, except the OSLink, a unique name must be provided. This unique name is generated automatically for routers and processing nodes. For the thread, and the channels, the user has to specify their names, corresponding to the names in the application code. The approach provides the independence of each item for the ease of changing the network specifications.

Two examples of a viewport are given in Figure 4-9 & Figure 4-10. In Figure 4-9, the hardware layout is specified in the Windows GUI. Figure 4-10 shows the communications between processes. By clicking on each icon, the developer is able to specify the characteristics of the icon (for instance the image file where a process is located), ultimately defining implicitly which processing node the process resides on.

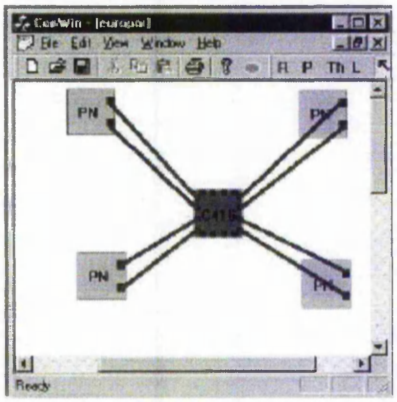


Figure 4-9 Example of a network layout.

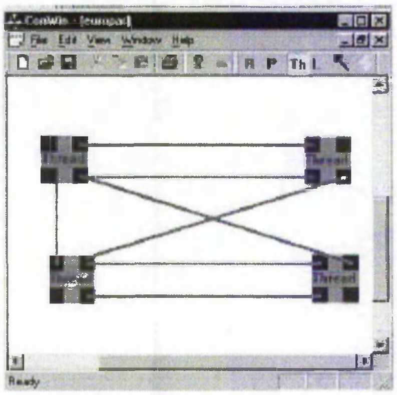


Figure 4-10 Communications between processes.

In these figures, PN represents a processing node, C416 represents a router, and a thread represents a process (in this case, each thread does not necessarily map onto a different processor). The line connections between processes and processors (or between both figures shown) are different as well, as connections between processing nodes and routers represent the hardware link, whereas the connections between processes represent channels.

In the network mapping, certain rules are enforced to prevent deadlock. The network must have horizontal and vertical mirror symmetry and both links to any processing node must be connected to the same router. These rules of having symmetrical network are yet to be explicitly enforced by the software, and any deviations from the rules will result in undetermined behaviour of the software. In essence, a message has strictly one path to route to one destination, and the same routing headers must be able to reach the same destination for both links on the StrongARM node. Since each processing node has two links connected to a router, these two links could be grouped together by programming the router explicitly, via the control port. A control port of the router is an additional port, which could be connected to another dedicated control link from the StrongARM system. The grouping of links in the router has to be programmed explicitly by the developer. In this case, any free link of these grouped together will be taken for any of the routing headers.

To obtain certain values from the user, dialog boxes are used. Dialog boxes are provided under the MFC as a means for the GUI system to interact with the user. In this way, the user could type into the box to specify an attribute to a specific item in the view port. Certain dialog boxes implement a drag list, to prevent the user from erroneously selecting invalid options. An example of a dialog box is shown in Figure 4-11.

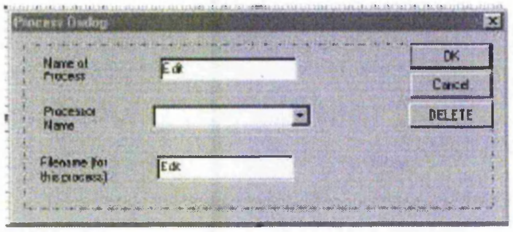


Figure 4-11 An example of a dialog box

The GUI system is divided into two parts. The first part deals mainly with drawing objects in the view port, which will interact with the user. The second part deals with the representation of objects drawn by the user, and updates the necessary information when an object is modified dynamically by the user. Both parts interact to function as a whole GUI system. The rest of the system functions, like saving and loading, rely on the MFC provided by the development toolkit.

After specifying the network graphically, the developer needs to press a button in the GUI to generate the textual representation of the network, which will be stored in a file. This could be followed by pressing a different button to call the NTU-Configurer, and the GUI task ends here. The next section will discuss the role of the NTU-Configurer.

4.4.2 NTU-Configurer

The script file generated by the GUI will be fed into the NTU-Configurer. The NTU-Configurer is the tool, written in C, which will take a syntax description of a network, and generate the appropriate headers to get a message from a particular channel to another channel over the network. The developer could also manually write a script file for this purpose. Keywords are used in the script file. An example of a script file would be as follows:

```
ROUTER ROUTER1              //to specify a router name, which is //ROUTER1
PROCESSOR PROC0   //to specify a processor name, which is //PROC0
//to specify the link connections in the hardware
SETLINK ROUTER0.5 ROUTER1.12
/*to specify which image file name to be placed on a particular processor in this case,
kendo.axf is the file name, and PROC0 is the processor name*/
PLACEFILE kendo.axf on PROC0
/*to specify which thread (or process) is to be placed in a particular filename, in this case,
kenneth is the thread, and kendo.axf is the filename*/
PLACEPROC kenneth on kendo.axf
/*to specify where the channel from one process is connected to another channel, in this case,
allan is the outgoing channel name from the thread kendo, and sue is the incoming channel
name from the thread kenneth*/
CONNECT kendo.allan to kenneth.sue
```

Keywords are “PROCESSOR”, “ROUTER”, "PROCESS", "PLACEFILE", "PLACEPROC", "CONNECT", "SETLINK", "on", "to", and ".".

The user could also provide this script file directly, without using the GUI tool. The NTU-Configurer works in several processes, which are described as follows:

1) A parser is used to extract the textual information contained in the script file, into a node structure understood by the system. All comments in the script file are removed in this process.

2) Another module in the NTU-Configurer will map the structure extracted by the parser into a table. A table entry is only created when a “CONNECT” keyword is encountered, indicating a hardware routing path.

3) The next process will resolve the dependence of each node to all others. This dependence arises when there is a channel between two processes. The AI module will use these nodes to search for the routing path for external channel communication. In the case of an external channel, where two communicating threads reside on different processors, routing headers and message ID will be generated.

4) In the final phase, a text file is generated, which will be used during program development. This text file is normally included in the application as a header file, which will then be compiled using the cross-compiler, targeting the SARNET.

The AI module uses a “depth first search” [87] to search for a path between two processing nodes. The “breadth first search” [87] method was not used due to high-memory requirements. The depth-first search only applies to find a path from one router to another destination router. Next, the path to the destination processing-node is searched using a normal method through the node structure. It will find which link of the destination router is connected to the destination processing-node. An example of the node structure traversed by the search mechanism is illustrated in Figure 4-12.

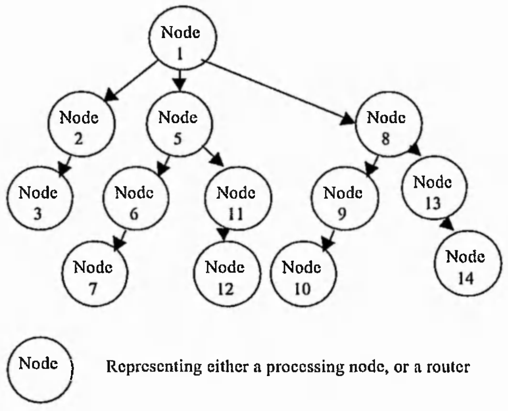


Figure 4-12 Tree-like structure in the software representation of the network.

The actual representation of the network from a source processor’s point of view is illustrated in Figure 4-13, after removing possible recursions.

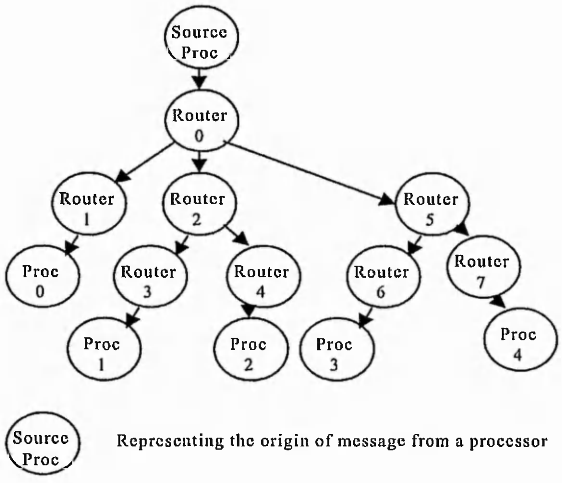


Figure 4-13 Tree-like structure for getting a message across the network from a source point of view after removing possible recursions.

When traversing the tree, any wrong path taken which does not lead to the correct destination must be backtracked to the previous node. Care must be taken to prevent creating an infinite loop when the search path is repeated indefinitely, or recursively (for instance, if the route path is not marked as searched, it may be searched again). In Figure 4-13, to get a message from a source processor to the destination processor involves searching through the tree. For instance, if the destination is Processor 3, the system will first searches through Router 0, Router 1, Processor 0, and then backtracked to Router 0 upon failure, and continues on until Processor 3 is found. If recursions are not removed, the algorithm may find that Router 2 is actually connected to Router 0, and Router 0's path to Processor 0 may be searched again, creating an infinite loop.

Since the collective system tools function independently from each other, a user could specify their own hardware map in three stages, namely:

- at a graphical level, provided by the GUI tool;
- at text level, using the script which would then be fed to the NTU-Configurer;
- at text level, by defining the headers themselves.

Such options provide the user the flexibility they need, from debugging to development purposes.

4.4.3 Discussions.

After integrating the OS with the GUI and the NTU-Configurer as a final development toolkit, the system is found to be operating according to the specification. The ease of programming a parallel application is enhanced, compared to the transputer systems. The NTU-Configurer is guaranteed to find a routing path from one processor to another, providing a solution exists.

In the current implementation, the first correct path found by the AI module to route a message would be accepted. This may result in an inefficient network, when all the messages directed to a particular processor would take the designated link (first correct path) connected from the processor to the router. The other hardware link of the processor connected to the same router has different routing headers. One approach would be to alternate the two links used in the NTU-Configurer. This would balance the usage against using one particular link, and avoid an inefficient network.

However, if there are 4 messages for the intended processor, with 2 long messages and 2 short messages, there is still a chance for the 2 long messages to be assigned to the same link, again resulting in inefficient network. To deal with this, another feasible approach would be to group these links together in the router. The router is programmed from a control link connected to a node; resulting in the same routing headers being used to route through two different links to the same destination processor. This method may defeat the idea of a high level of abstraction, since it would involve the user in programming the links. However, the trade-off of the inconvenience versus inefficient network is tolerable, as the user is only required to program each router from any of the connected processors only once during the initial phase. Furthermore, routines are made available to ease the task of programming the router. Therefore, this method described is preferable and the architecture of the NTU-Configurer remains unchanged.

In the AI module, a depth first search is used. The method is adopted since the network must be symmetrical to provide the ease of designing distributed algorithms for the network: only one path can exist between one processor to another via single or multiple routers. Furthermore, the rule where both the processor's links can only be connected to the same router to reduce deadlock will result in only a single path between two communicating processors. This applies when the dual links are grouped together in the router. Therefore, other more intelligent algorithms are not needed.

In the system described, the hardware only supports a maximum of 6 message headers, the routing headers and the message ID. The message ID is used to identify a unique global channel, and is typically a byte wide, thus limiting the number of global channels to just 255. Although this figure is deemed enough for current applications, at a larger scale in a massive parallel application it may not be sufficient. Therefore, at a later stage, an additional improvement may be needed by the software to partition the length of the message ID appropriately, to use up any unused routing headers in circumstances when the number of global channels exceed 255. To perform this task, a more intelligent algorithm is needed, and this is to be investigated further as future work.

Some independent GUI tools for mapping the transputer's network are available, but none incorporate or integrate with other automatic routing header generation tools and none support virtual channels over routers[117]. This said, it is hoped that the tools developed will be adapted to assist similar hardware platforms which require wormhole routing support, thus providing the ease of mapping channels over the network links for static and real-time applications.

Additional features could be added to the GUI and the NTU-Configurer in the future, to enforce the rule of network symmetry, and to perform checks on the validity of hardware connections. Further work could be done to enhance the graphical features of the GUI. This work will ultimately produce a toolkit to develop parallel applications for the ease of parallel programming.

5. Testing, Analysis and Performance Assessment.

During the course of the system development, software tests were performed regularly to ensure modules were functionally correct. The software tests are described in the first section.

The second section measures the performance of the OS in terms of context-switching and communication overheads.

5.1 Testing the software.

The apparatus used to perform software functional tests included:

- An OSLink network interface board and the transputer IMS B008 board
- A StrongARM processor node
- A host computer running a 'hyperterminal' program, which essentially captured the messages displayed by the StrongARM processor node through UART port (COM 2 for the host PC)

The OS was tested with both dummy programs and standard programs. Standard programs included benchmarking applications[88] e.g. Whetstones (that test the performance of a processor for floating-point operation) and Dhrystones [89] (that test the performance of a processor for integer operation). These benchmark programs were obtained in the form of machine independent source code. Hence, they are able to compile for the targeted StrongARM node. Additional programs, which perform inter-process communications, were run as well. The following major functionalities were tested:

- Inter-process communication, for hard channels and soft channels
- Inter-leaving of message handling capability
- Both pre-emptive and co-operative scheduling

Each process included in the test displayed a message after reaching certain points. These messages were displayed through the UART port, directly to the host PC running the 'hyperterminal' program. Therefore, the display could distinguish which process was currently running. Additional messages were displayed for various events, including communications, inter-leaving of messages and timer-interrupts.

5.1.1 Testing for interleaving support.

For simulating the communication module using a single link and DMA engine, the transputer IMS B008 board and the PC Interface were involved. This board was able to receive messages transmitted from the StrongARM node, or send messages with appropriate message identification to it. To simulate the interleaving of messages during receipt, two different programs running on the host PC each attempted to transmit multiple packets of 256 bytes, but pausing after each packet, until all the packets were delivered. A message in this scenario was considered to be more than 256 bytes and the receiving processes running on the StrongARM processor expected a message of the same size as the one sent by the transmitting process.

For simulating the communication module using dual link and dual DMA engines, a different setting was required. Both the generic ISA Interface and the transputer B008 board were required, and the hardware setting is illustrated in Figure 5-1.

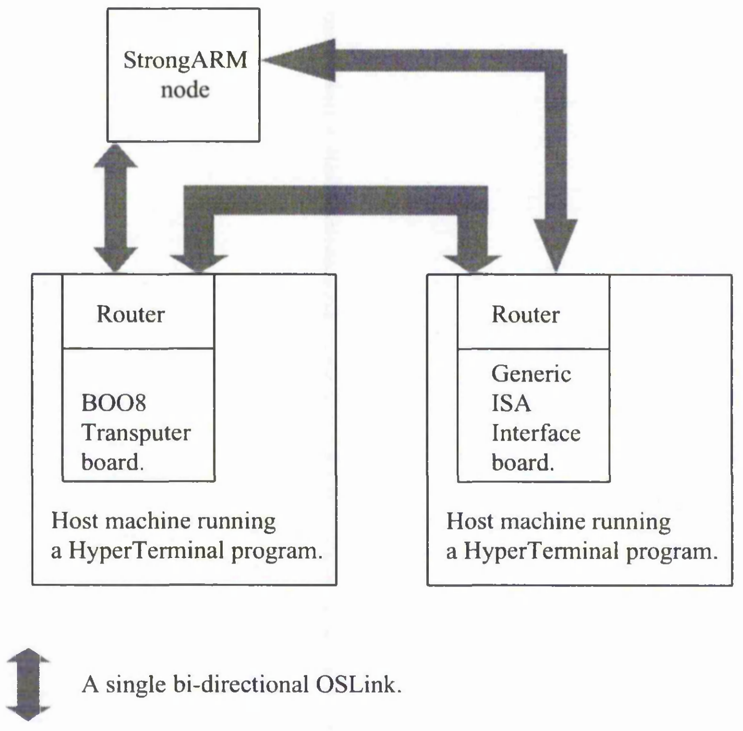

StrongARM node
Router
BOO8 Transputer board.
Host machine running a HyperTerminal program.
Router
Generic ISA Interface board.
Host machine running a HyperTerminal program.
A single bi-directional OSLink.

Figure 5-1 setting for testing dual OSLink for interleaving of messages support.

In the test, different channels were set up between the StrongARM node, the transputer and the ISA Interface. The StrongARM node was the primary recipient of messages. There were altogether three different messages being communicated in the system. Two were from the

ISA Interface, and one was from the transputer. On the StrongARM side, three different channels were used to receive these global messages.

Figure 5-2 shows the arrival of 3 different messages, for three different channels. In the test, a message of 60 bytes was first sent from the ISA Interface to the StrongARM node, via the direct OSLink connection. However, instead of sending the whole body of the message (message M1), only the first 4 bytes were sent, to create an artificial stalled link. In this way, the routing path was temporarily opened, and the receiving DMA engine (Engine 0) for the StrongARM node would remain busy. This busy engine could not be deactivated due to the 'Packet Active' status.

Another different message (message M2) of multiple packets was sent from the transputer to the StrongARM node. The other free DMA engine (Engine 1) then started receiving it. Again, instead of sending all the packets, only the first packet was sent. The ISA Interface then sent the first packet of a different message (message M3) to the StrongARM, this time via the transputer's router using a different path. When this happened, the software support deactivated the DMA engine (Engine 1), and set it up to receive the third new message (message M3). This switching was repeated several times, alternating between a packet of the message (M2) from the transputer and a packet (M3) from the ISA Interface, until both messages were sent. The stalled message (M1) was then allowed to be fully delivered.

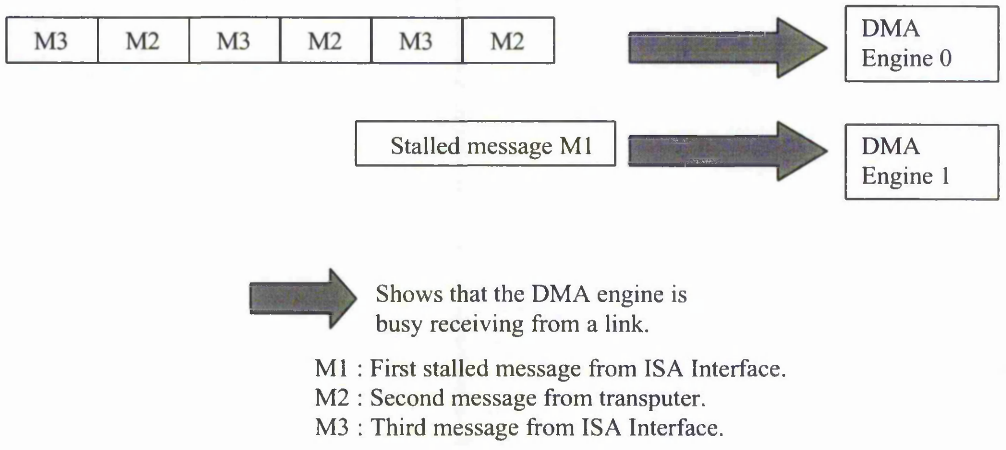

M3
M2
M3
M2
M3
M2
DMA Engine 0
Stalled message M1
DMA Engine 1
Shows that the DMA engine is busy receiving from a link.
M1 : First stalled message from ISA Interface.
M2 : Second message from transputer.
M3 : Third message from ISA Interface.

Figure 5-2 arrival of interleaved messages.

The receiving processes then displayed the received messages via the UART, and a visual comparison was made to detect any discrepancies. It was found that the messages sent were the same as the messages received for all the processes. The same process was repeated by swapping the links. In this way, the software functionality in handling interleaved messages was tested to be functioning correctly for both links.

5.1.2 Results and Observations

Following the observations gathered from the software functional tests, the OS was tested to function correctly under these situations, according to the specification. These test programs, written to test various OS functionalities, were executed on the StrongARM processing node, being scheduled as expected: both pre-emptive (due to time-slicing) and co-operative (due to communications). This was deduced after observing the messages displayed on the host PC through the UART port. The sequence of messages displayed was in accordance with the communication events taking place on the host PC.

In the case of communication tests, communicating processes residing on different processors were observed to have transmitted messages successfully, when the message on the receiving process matched the message from the sending process. The same applied to the internal channel communications. Moreover, the benchmark programs were completed successfully for the single link test, with the desired results.

Further technical details of SARNUX and a comparison to the transputer system are included in Appendix A at the end of this report. The performance measurements and analysis for both software and hardware will be presented in the following section.

5.2 Measuring Performance.

Tests were performed to gauge the system's performance, by running:

- the local Commstime[90] program.
- the global Commstime program.

Subsequently, the communication and the context switching overheads were measured, and the effect of DMA transfers on the CPU processing ability was analysed.

5.2.1 The Commstime measurement.

Commstime[90] is a benchmark program, developed by the University of Kent at Canterbury as a form of measuring a system's performance in terms of context-switching and communications. This code, originally written in Occam, has an ANSI C equivalent to enable measurement of implementing channels and co-operative scheduling mechanisms on other platforms. The overview of this program is illustrated in Figure 5-3.

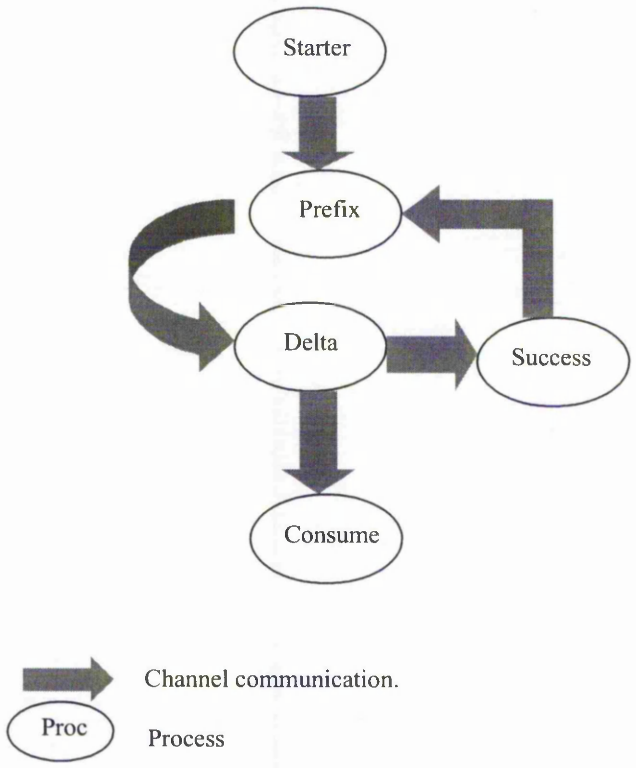


Figure 5-3 Commstime mechanism.

In the Commstime program, there are five processes, and three of these processes run in a loop. The number of loops executed depends on the developer. A higher number of loops will yield better results, as it will make fixed overheads during startup negligible. The 'starter' process will initiate the loop during the beginning of the program. Once started, the processes in the loop will communicate, and de-schedule until the number of loops specified is reached. The 'Prefix', 'Delta' and 'Success' processes execute an infinite loop to perform communications, to provide a feedback to the system. Each message passed in the system is a byte in length. The 'Consume' process will collect results after each loop and count the number of loops executed, before displaying the results after the specified number of loops has been reached. It is important to note that for this experiment, the dual outgoing channel communications performed by the 'Delta' process was sequential. Some systems performed this dual channel output in parallel, which might result in lower overheads, since the two requests by 'delta' were processed using only a single system call. When the 'Consume' process stops receiving the message, the whole loop will be halted.

The time taken for each loop was then calculated, by dividing the total execution time by the number of loops executed. From there, the time taken for each channel communication and context-switch was derived. In total, there are five context-switches and five channel communications for each loop. Therefore, the time for one context-switch and one channel communication is the time taken for each loop divided by 5.

The codes written in C were modified to suit the 3L libraries, which were implemented in SARNUX. The modification involved only minimal syntax changes.

The number of loops was set to 1,000,000, and the results obtained are shown in Table 5-1.

1. When those processes were executed in high priority:
   Time per loop: 17.4μs,
   Time per context switch and communication: 3.5μs.

2. When those processes were executed in low priority:
   Time per loop: 32.6μs,
   Time per context switch and communication: 6.5μs.

Table 5-1 results for local Commstime benchmark.

The difference between the two cases is due to the fact that when high priority processes are context-switched, no request is made to the timer module to generate a timer interrupt, since high priority processes are not time sliced. For the low priority processes, a request is made to the timer module, since these processes have to be time-sliced, and consequently they incur a higher overhead.

The Commstime program has been criticised by many as it reflects only ideal figures. Most of the instructions would have a higher cache hit if the system has a large cache, resulting in artificially low overheads. However, the program could serve as a basic benchmark for the general performance of a system using channel communications.

Appendix B shows the results obtained from other systems[91], from the University of Kent. It is observed that most of the results from other systems are faster than the results obtained from the StrongARM system. It could be explained that the other systems may not be performing full context-switches, and are thus able to achieve lower times for communications.

5.2.2 The global Commstime measurement.

The global Commstime program is similar to that of the Commstime. This test was to gauge how the system would perform when the Commstime program was run, but instead of performing communications using internal channels, it used external channels. The setup is illustrated in Figure 5-4.

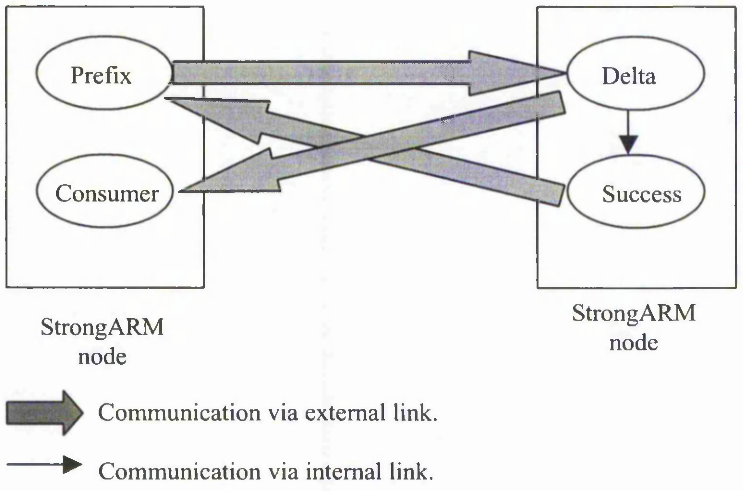

Prefix
Consumer
Delta
Success
StrongARM node
StrongARM node
Communication via external link.
Communication via internal link.

Figure 5-4 global Commstime benchmark.

Inherently, the only difference between the global Commstime and the Comsmstime is the different system call used when performing communications among all the processes. There is still a single internal channel communication between the 'Delta' process and 'Success' process, because only two StrongARM nodes are used.

All the processes ran at high priority. The results obtained are shown in Table 5-2.

Number of loops	Time taken	Time per loop	Average time per context switch/ communication
1000	48397	48.4	9.7
100000	4841721	48.4	9.7
1000000	48398480	48.4	9.7

All time are measured in microseconds.

Table 5-2 results for global Commstime.

In the test, there were about seven context-switches, and four pairs of channel communications, including a pair of internal channels. By using the previous result, whereby 1 pair of internal channel incurred an averaged 3.5µs, the remaining time used to perform global communications and context switches was estimated at 48.4 – 3.5 = 44.9µs. Therefore, the average was obtained by dividing the time of 44.9µs for the remaining six loops (by 6 instead of 7). This gives the figure 7.5µs. This figure obtained for time per loop is higher than the local Commstime program (which yields 3.5 µs), since there are hardware overheads and higher software overheads involved in performing external communications. Moreover, there are more context switches in this test.

In summary, this global Commstime test is performed only to compare the performance when the program performs global communications instead of local communications. It gives an idea of the overheads involved.

The results could serve as a guideline in gauging the performance of the StrongARM system when a parallel solution is chosen. When partitioning a task into distributed tasks, the benefits (in terms of speed up) must outweigh the cost of doing so[92] (this does not applies to distributed control systems). It is expected that the time per loop for the execution of global Commstime should be less than the time per loop for local Commstime if there are heavy computations used for each process on the message, as there is more processing power available in a parallel system for the global Commstime.

5.2.3 Measurement of scheduling overheads.

Context switching time was measured from the start of a timer interrupt, to the point where the new process program counter was to be updated. To obtain the highest possible accuracy in measuring the time taken for a context switch, minimal assembly codes were inserted at the beginning point of the interrupt routine and the end point prior to executing another process. This method is illustrated in Figure 5-5.

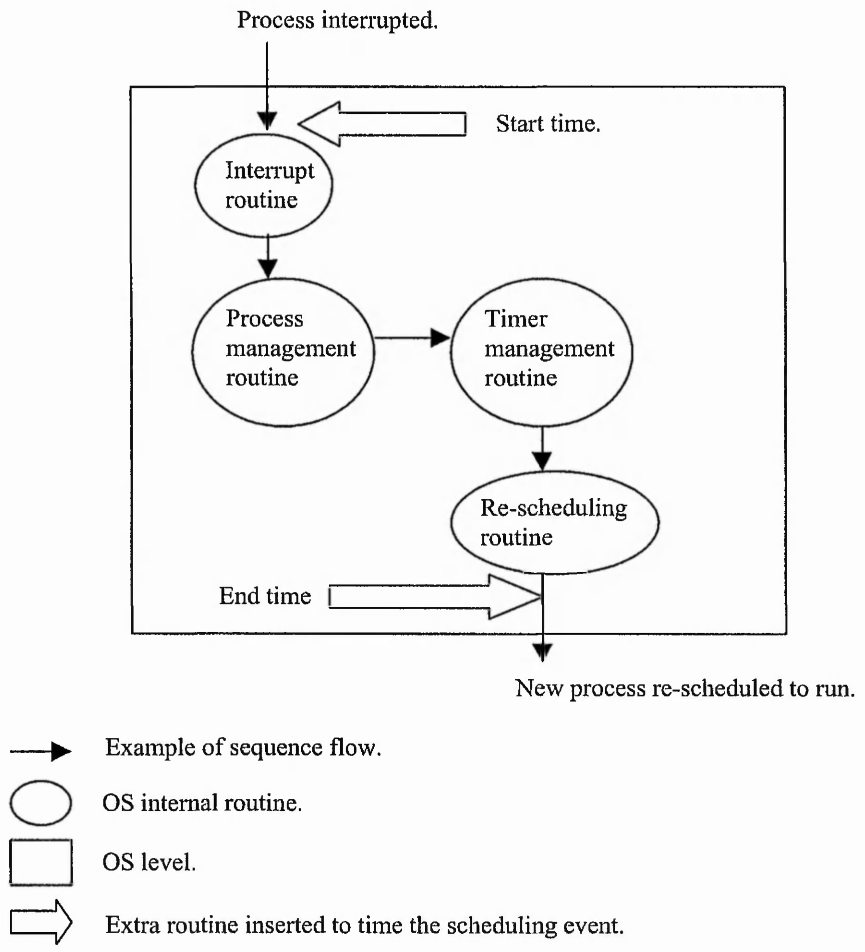

Process interrupted.
Start time.
Interrupt routine
Process management routine
Timer management routine
Re-scheduling routine
End time
New process re-scheduled to run.
Example of sequence flow.
OS internal routine.
OS level.
Extra routine inserted to time the scheduling event.

Figure 5-5 measuring the scheduling overhead.

In the measurement, processes were executed in parallel, in infinite loops performing mathematical functions. The number of processes was estimated at 15[65], from an actual Quantel application. All the processes had low priority, as high priority processes were not time-sliced. Both the data cache and the instruction cache were enabled to represent an actual executing environment. To obtain a realistic figure, the routines only reported the scheduling overhead (via the UART) after a predefined number of loops, representing the total number of context-switches, had occurred. A typical number of loops was 1000.

Table 5-3 indicates the time taken for each context switch, with different cache settings.

Cache setting	Time taken (micro-seconds)
DC disabled, IC disabled	68
DC disabled, IC enabled	27
DC enabled, IC enabled	4

DC : Data cache, IC : Instruction cache

Table 5-3 time taken for context-switching under different cache settings

Under normal operation, the cache setting would be to turn on both the data and instruction cache, yielding a context-switch time as low as 4μs. The cache hit rate should be very high, as the codes of the OS to perform the operation should be fully cached, due to the regular context-switching. The overhead is slightly above the transputer's time of 2μs [93].

This test was repeated several times, and the time quantum for the scheduling overhead measured was consistent at 4μs. A microsecond was the highest resolution reported by the hardware timer.

5.2.4 Global Communication overheads measurement.

Three tests were conducted to analyse the system performance in external communication:

- Transmitting global messages.
- Receiving global messages, when those messages were already stored in the receiving buffers.
- Receiving global messages, when those messages were not in the buffers.

Transmitting global messages.

In this test, a process that transmits global messages via an external link was looped several times, and two sets of results were obtained:

1. When the data cache was off, but the instruction cache was on.
2. When both data and instruction caches were on.

The message was sent to another StrongARM node, where another process looped indefinitely, acting to consume this message. The overheads involved in the 'Consumer' process were negligible, as both instruction and data caches were turned on to minimise latency in setting up the DMA engine to receive the incoming message. The 'Consumer' process used polling in a tight loop to minimise software overheads. The setup is illustrated in Figure 5-6.

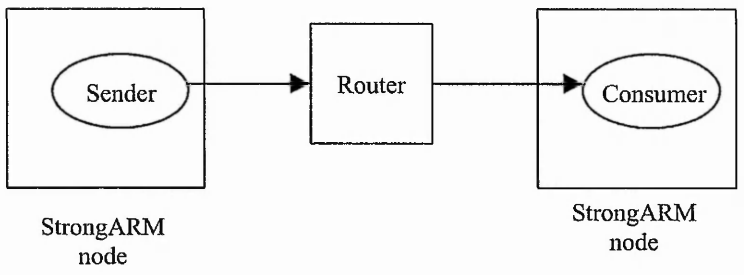


Figure 5-6 setup for measuring external communication.

The 'Sender' process sent a message 20 times repetitively, using the same channel. The time taken for each request to send, until the message was complete, was measured.

The portion of code used for the 'Sender' process is shown as follows:

```
for(i=0;i<20;i++)
    {
        value++;
        time1[i]=timer_now();                  //to get the start time
        LinkOut(&value,4,Prefix_ChanOut);      //request to send
        time2[i]=timer_now();                  //to get the end time
    }
```

The average of the time taken to transmit a message:

1. For the first set of results was : 122.7µs.
2. For the second set of results was : 11.4µs.

The figure obtained from the second set of results, i.e. 11.4µs included two context-switches. The first context switch occurred when the process requested to send a message and it was blocked while the request was processed. The second context switch occurred when the transmission was completed, and the requesting process was unblocked.

Since each context-switch typically takes about 4μs, the overhead involved for processing a request to send was:

$$\textit{Overhead} = 11.4 - 4 \times 2$$
$$= 3.4\ \mu s.$$

Receiving buffered global messages.

To measure the overheads involved for receiving a message via an external link, another setup was used. Two different tests illustrated two different scenarios. One was when the message that the process was expecting from an external link had already arrived, and buffered in the receiving buffers. Another was to show when the message was not buffered, and the OS had to initialise the DMA engine each time.

This test was to calculate the software overheads involved in receiving buffered messages. In this test, the setup was similar to Figure 5-6, except that now the 'Sender' process used a polling method in a tight loop to send messages to the 'Consumer' process. The transmitting side was written in assembly language to minimise software overheads, and was instruction and data cached.

The portion of code used for the 'Consumer' process in this test is as follows:

```
for(count=0;count<20;count++)
        {
                time_rx1[count]=timer_now();            //to measure the start time
                LinkIn(msg_temp_rx,4,JoeChanIn);        //request to receive
                time_rx2[count]=timer_now();            //to measure the end time
        }
```

For this test, a buffer size of 20 was used. This means that the system could buffer up to 20 incoming messages for a single channel. The buffer size is user defined. Before the 'Consumer' process could start receiving, all the messages would have been stored in the buffer. The results are shown in Appendix C.

The average time taken to receive a buffered global message:

1. When data cache was off, and instruction cache was on was: 44.9 μs.
2. When both data and instruction caches are on was: 3.0μs.

When the process requested the message and found that the message was already buffered, the same process was going to run again, even though its context was saved earlier. Therefore, it did not incur the penalty of a full context switch and was able to achieve an overhead lower than a context switch. The minimal overhead to receive a buffered message was 3.0μs.

Receiving un-buffered global messages.

In this test, the message arriving via the OSLink was stalled, as the system initialises itself. After the initialisation phase, the process started receiving messages, and if the process could not receive it fast enough, the incoming message was buffered. A typical buffer size of 5 was used[65]. The code used in the test for receiving buffered global messages was used in this test. The results obtained are shown in Appendix C.

The average time taken to receive an un-buffered global message:

1. When data cache was off, and instruction cache is on was: 120.3 μs.
2. When both data and instruction caches are on was: 6.2μs.

When the process requested the message and found that the message was not buffered, it incurred the penalty of a full context switch and was blocked until the message arrived. The minimal overhead to receive an un-buffered message was 6.2 μs.

Summary

For all the tests performed, the message size was kept to a minimum of 4 bytes to reduce the hardware latency involved. This minimum length was defined by the hardware. The ideal case was to demonstrate the overheads for the software only. For all cases, communication was performed using channels, and all the coding was the same as if an actual application was used. Additionally, all processes ran in high priority mode, since most of the existing applications run those processes, handling global communication, in high priority mode. Under normal execution, the test average when both caches were turned on demonstrated a more realistic figure compared to the test average when only the instruction cache was enabled, since both caches would normally be enabled. However, these figures serve only as an ideal case, to gauge the system performance in passing messages via external links. The actual run time overheads might be slightly higher, depending on various circumstances.

In comparison, the overhead of receiving buffered global message was lower (3.0 μs) than for receiving un-buffered global messages (6.2 μs). The advantage of using buffers is evident, however, the memory cost is high, in the embedded environment.

5.2.5 Effect of DMA on systems performance.

Another set of tests was carried out to determine the effect of DMA transfers on the processor performance. In this scenario, an application program was to run in the foreground, with the DMA engine running continuously in the background. The continuous transfer was done by inserting some code into the IRQ handler routine. This code would acknowledge the interrupt generated when the communication was completed, and re-transmit another message. Hence, a loop pattern for transfer of messages was established. The message size was 255 bytes, excluding the message headers. The size 255 was chosen primarily because it represented a typical message length used in an industrial application[65]. The largest message size of 64kbytes was not used since it will minimise the software and hardware latency in servicing the communication interrupts, and hence will not represent a realistic figure. The smallest message size of 4 bytes was not used either, since it will incur severe software and hardware latency in servicing communication interrupts, and may distort the results. An acceptable experiment must focus on the DMA overheads, but also some of the overheads associated with it, like the software and hardware latencies in servicing communication interrupts.

In these tests, messages sent were routed to another StrongARM node via the OSLink, where an application program residing on the receiving StrongARM node would then read the data and discard them continuously. The router resides on PC Interface located at the host PC. The aim was to provide a destination where the data transmitted is consumed. The Dhrystones benchmark program was selected to run on the StrongARM node. The time observed was the time it took to run the benchmark, in microseconds. The whole set up is illustrated in Figure 5-7.

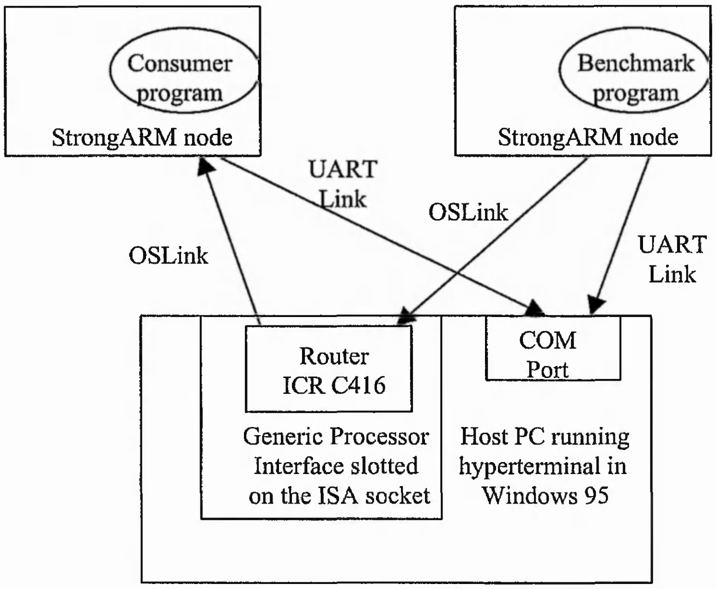


Figure 5-7 Overview of the system in the DMA performance test.

In this experiment, 4 different tests were carried out as follows:

Measurement of the time taken to complete the Dhrystone test with

- No DMA transfer in the background.
- DMA transfer, with only 1 Engine transmitting in the background.
- DMA transfer, with 2 Engines transmitting in the background.
- DMA transfer, with 4 Engines, 2 for transmitting and 2 for receiving messages in the background.

These tests were carried out with a variation of cache settings:

- Both data cache and instruction cache turned on.
- Data cache turned off, but instruction cache turned on.
- Both data cache and instruction cache turned off.

The reason for varying the cache settings was to observe the effect of different cache settings on the CPU performance. The results presented in the following tables were averaged from the complete results given in Appendix D.

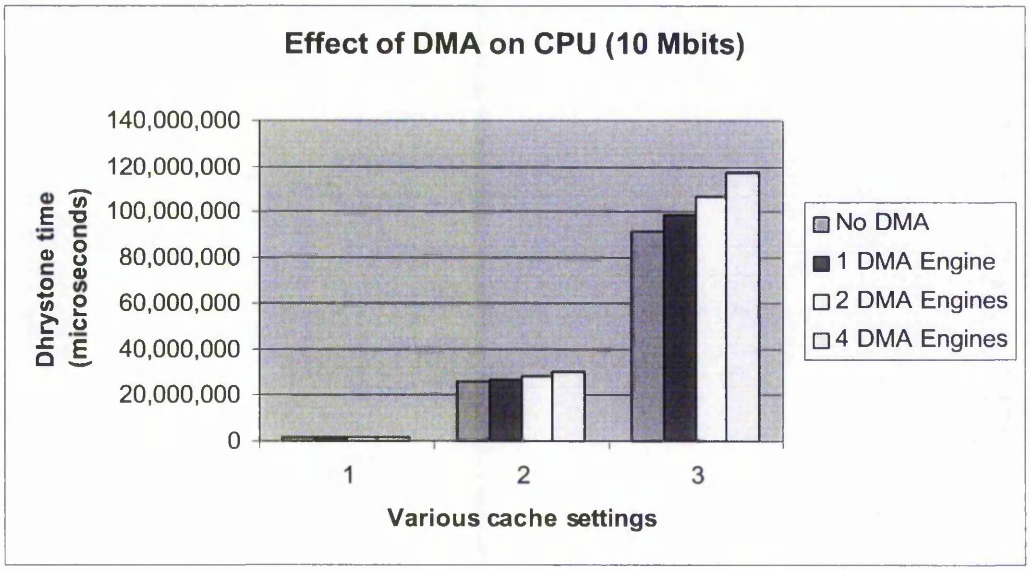
Table 5-4 gives the results for the setup, when running the OSLinks at 10 Mbits/s.

Condition	Without DMA engine running in background	With 1 DMA engine running in background	With 2 DMA engines running in background	With 4 DMA engines running in background
IC on DC on	1,901,503	1,914,421	1,922,741	1,940,337
IC on DC off	25,688,908	26,978,610	28,321,529	29,750,682
IC off DC off	91,676,961	98,725,241	107,117,843	117,619,933

(All values in microseconds, IC – Instruction Cache, DC – Data Cache)

Table 5-4 The effect of DMA transfer on the processor's performance at 10Mbits/s.

The results of Table 5-4 are illustrated in Chart 5-1.



1:DC and IC on, 2:DC off and IC on, 3:DC and IC off.

Chart 5-1 The effect of DMA on CPU performance, at 10Mbits/s.

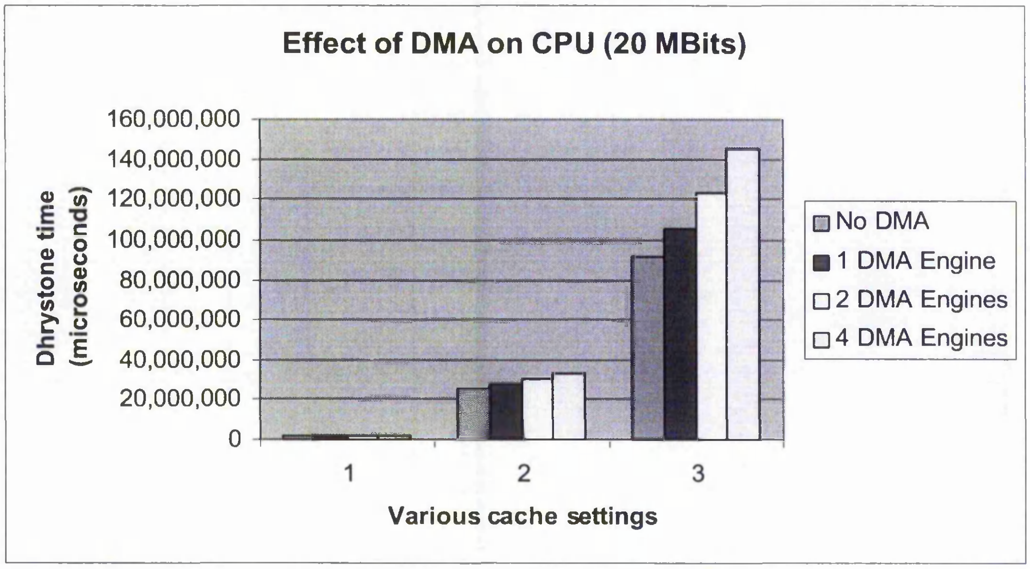
Table 5-5 gives the results for the setup, when running the OSLinks at 20 Mbits/s.

Condition	Without DMA engine running in background	With 1 DMA engine running in background	With 2 DMA engines running in background	With 4 DMA engines running in background
IC on DC on	1,901,502	1,919,036	1,940,970	1,967,312
IC on DC off	25,688,882	28,093,940	30,942,163	33,310,848
IC off DC off	91,674,674	105,007,698	122,918,752	145,275,968

(All values in microseconds, IC – Instruction Cache, DC – Data Cache)

Table 5-5 The effect of DMA transfer on the processor's performance at 20 Mbits/s.

The results of Table 5-5 are illustrated in Chart 5-2.



1:DC and IC on, 2:DC off and IC on, 3:DC and IC off.

Chart 5-2 effect of DMA on CPU performance, at 20Mbits/s.

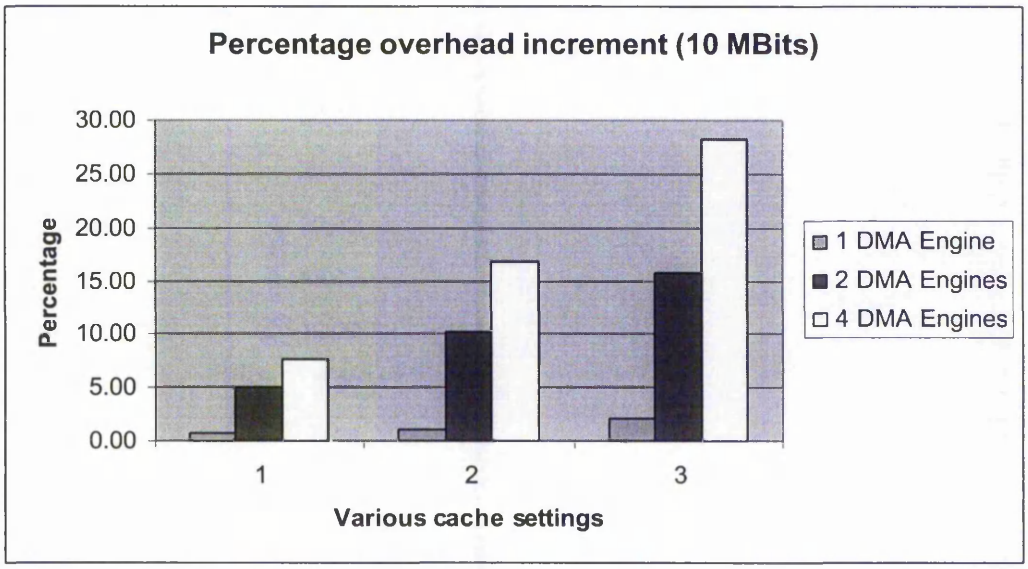
Table 5-6 gives the results for the percentage increment in overhead compared to no DMA transfer, when the links are running at 10Mbits/s.

Condition	With 1 DMA engine running in background	With 2 DMA engines running in background	With 4 DMA engines running in background
IC on DC on	0.7	1.1	2.0
IC on DC off	5.0	10.3	15.8
IC off DC off	7.7	16.8	28.3

(All values in microseconds, IC – Instruction Cache, DC – Data Cache)

Table 5-6 The percentage increment of overheads with DMA transfers as compared to non-DMA transfer at 10 Mbits/s.

The results of Table 5-6 are illustrated in Chart 5-3.



1:DC and IC on, 2:DC off and IC on, 3:DC and IC off.

Chart 5-3 percentage increment in overheads as compared to non-DMA transfer at 10Mbits/s.

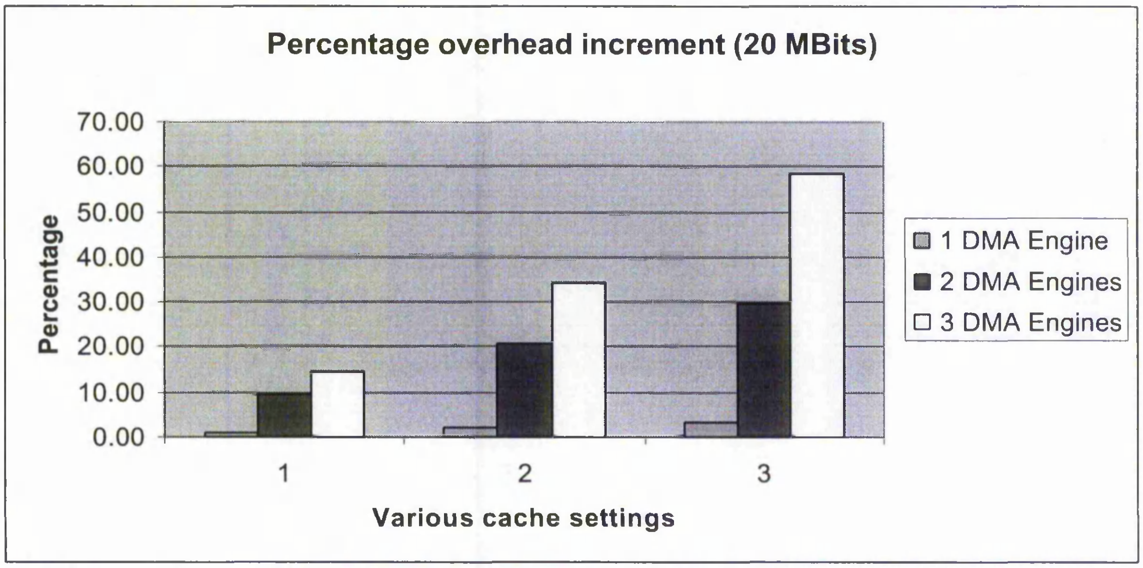
Table 5-7 gives the results for the percentage increment in overhead compared to no DMA transfer, when the links are running at 20Mbits/s.

Condition	With 1 DMA engine running in background	With 2 DMA engines running in background	With 4 DMA engines running in background
IC on DC on	0.9	2.0	3.5
IC on DC off	9.4	20.5	29.7
IC off DC off	14.5	34.1	58.5

(All averaged values in microseconds, IC – Instruction Cache, DC – Data Cache)

Table 5-7 The percentage increment of overheads with DMA transfers as compared to non-DMA transfer at 20 Mbits/s.

The results of Table 5-7 are illustrated in Chart 5-4.



1:DC and IC on, 2:DC off and IC on, 3:DC and IC off.

Chart 5-4 percentage increment in overheads as compared to non-DMA transfer at 20Mbits/s.

From both Table 5-4 and Table 5-5, cache clearly reduces the latency, which is indicated by the different cache settings and the times taken to complete the benchmark test. Delays are expected, as the processor is unable to access main memory whenever the DMA engine is stealing bus cycles, and further compounded by slight software overheads in servicing interrupts involved in the communications. For the best case scenario, where both data and instruction cache are on, the increment is:

- 0.68% when the links are running at 10Mbits/s.
- 0.92% when running the links at 20Mbits/s.

For the worst case scenario, where both data and instruction caches are off, the increment is:

- 28.30% when running the links at 10Mbits/s.
- 58.47% when running the links at 20Mbits/s.

However, the worst case scenario rarely occurs.

In the best case scenario, only minimum bus access by the CPU occur when performing the benchmark, as both instruction cache and data cache are turned on. Thus, the DMA engine is free to access the data bus all the time.

From Chart 5-1, it could be observed that as the number of DMA engines used in this test increases, the time taken to complete the test increases as well. This also applies to the results observed in Chart 5-2. As the number of DMA engines used increases, the bandwidth which is available to the processor to perform its tasks decreases, thus it takes a longer time to complete the benchmark test. The same principle applies after comparing the rise in overheads when running the links at 10Mbits/s, as compared to 20Mbits/s. The visual comparison for this case is done between Chart 5-3 and Chart 5-4. When the link bandwidth is doubled, the DMA engines perform more communications, thus it will increase its data bus accesses, reducing the CPU's chances to access the data bus. Therefore, effectively higher bus contention results in the CPU taking longer times to perform its tasks.

Figure 5-8 illustrates the relationship between DMA engines and CPU caches.

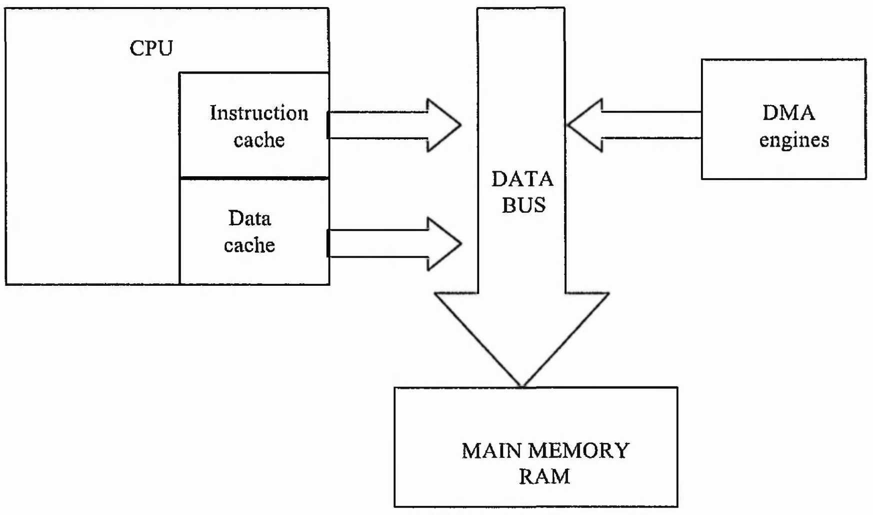


Figure 5-8 illustrates the relationship between caches and DMA engine

5.2.6 Summary

The performance tests that were conducted conformed to design expectations. The Commstime measurements gave an indication of the software overheads in performing local communications. For global communications, the global Commstime program measured the overheads involved. Further tests conducted on the global communications also showed that the software and hardware latency involved in performing global communications was fairly low.

The system also managed to achieve a low context-switching time of 4μs, which was only slightly higher than the 2μs of the transputer system. This low-latency was achieved even though there was a full context-save on all registers.

The effect of DMA engine on the overall processor performance was clear, and the latency was minimal, in all circumstances when the DMA communication took place. The results also showed that DMA effect on the CPU was at an acceptable level.

6. Porting application code.

This chapter will describe a project to port an existing end user application from a transputer system to a new StrongARM platform. This was written in C/C++ using the parallel 3L[71] libraries, based on the transputer system. The project was carried out in collaboration with Quantel[72], a company that specialises in video and audio editing technology. The project provides a practical demonstration of the benefits of the work in this thesis.

6.1 Background for the Project.

Quantel[72] used the transputer to perform distributed control system in their product, Clipbox. Clipbox is used mainly by TV stations for editing video and audio frames simultaneously, and it contains multiple transputers to service all requests from several users at any time. However, with the demise of the transputer, Quantel [72] designed a replacement system using StrongARM processors, which is similar to the StrongARM hardware designed by The Nottingham Trent University Parallel Processing Research Group.

Upgrading hardware is a major task, particularly when maintaining the existing application software, written and developed over many years. In this case, the application software was written in Parallel C and C++, based on the transputer. Consequently, for backward compatibility, an embedded OS with real-time response and functionalities is required to substitute for the transputer's micro-coded functions and to provide a layer of abstraction to hide the new hardware platform. One additional benefit of the Quantel project is because the transputer's computing power is not sufficient to meet contemporary audio streaming demands there was a need for higher processing power.

At the initial phase of this project, VxWorks was initially considered. However, after consideration of the advantages and disadvantages of using VxWorks, as already been outlined in Chapter 2, the company decided to drop VxWorks in favour of a variant of SARNUX. Even though SARNUX fulfils the requirements, however, much work was needed to fully port the old system to the new hardware. The next section will give an overview of the application software and subsequent sections will outline the work involved.

6.2 Brief description of the application software.

The general view for the application software is as follows:

Multiple users, sitting at different terminals, will try to edit different video and audio clips simultaneously, as practised in the broadcasting industry. Several processors, running in the Clipbox to extract, update or stream video clips from the single main server disk, will support these requests. As everyone shares the access to the same data storage, there is a need to maintain data storage memory coherency and speed to serve these clients. The operation must be done in real-time[73] to avoid video or audio 'glitches'. Any significant delays will result in total video and audio failure.

The low-level view of the system would be of numerous processors interconnected via routers. This is illustrated in Figure 6-1.

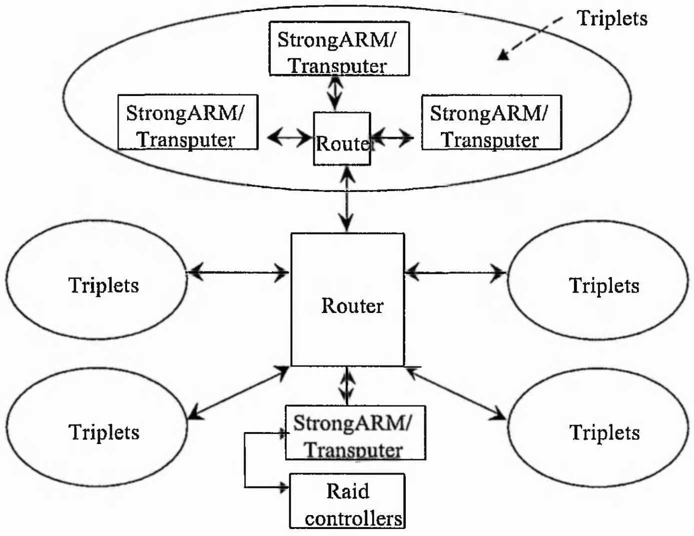


Figure 6-1 how the processors are inter-connected in the network.

In the global system, a main processor monitors all processors within its scope. When a processor is booted up correctly, the main processor is notified. The main processor will then broadcast to all processors that the service is provided for by that processor that is booted up. This occurs, by sending a message to all processors, regardless of whether they need this service or not. This is done continuously until all required processors are booted up.

During the start up, the application software uses a mechanism of 'publish and subscribe', designed and implemented by Quantel's engineers. Among the main aims of this mechanism was to ensure an orderly start-up, and the capability of an orderly restart[74]. Every task is either a client or a server. Clients must only subscribe to a service published by a server, and the communication will take place via a channel.

One of the major challenges in the project was to enable old applications written for a transputer system to run on the new StrongARM hardware. SARNUX has to provide functionalities to enable those applications to achieve similar behaviour on the StrongARM platform. Since the application involves real-time processing, this gives an additional constraints which if overlooked may result in total failure. In addition, due to the nature of the hardware, some modules in SARNUX are replaced with new modules. Care has to be taken to provide basic abstractions sufficient to meet the software needs, as if it were running on a transputer system, resulting in a small and yet efficient modified kernel.

6.3 Tasks involved in porting of software

One of the tasks involved in porting the application software is replacing all 3L[75][71] library header files. The 3L libraries provided a C-language interface to the transputer facilities. The library files that were replaced are:

- chan.h - used for inter-process communications or transferring a message across a channel; including routines for application programs to send and receive messages, regardless of local or global messages.
- thread.h - for general thread starting, allocating, de-scheduling and various other threading facilities.
- alt.h - for alternate constructs, allowing a program to input from whichever of a group of channels becomes ready first.
- par.h - for multiple parallel execution, allowing threads to access run-time libraries in a synchronised manner, and in an interlocked form (which may include semaphore operations).
- timer.h - for timer related functions.
- sema.h - for semaphore management, allowing the program to create and manipulate semaphores, which could then be used to synchronise the activity of several concurrently executing threads. It is not used for any communication purposes.

Most of these functions offered by the header files were inherently a part of SARNUX. The only differences are the calling syntax and the minor difference in thread behaviour when these functions are called. In addressing the difference in the calling syntax, slight changes had to be made to the OS. Changes were needed to protect the integrity of the OS. This includes either making several routines thread safe, or non-accessible, to user processes running in user mode. These routines are only accessible if the user process requests them via a system call. A system call implies higher overhead, but the benefits of the OS stability and integrity outweigh this.

An additional new facility was to run a portion of the lower priority user task in a non-interruptible mode. In the transputer systems, this could only be achieved by partitioning the non-interruptible portion of the code into another task, running at high priority. In this new system, a system call was inserted into the beginning of the non-interruptible codes, and another system call was made at the end of it, allowing that portion of code to run at high-priority. For both the transputer systems and SARNUX, there exist some risks to this strategy, as any unreliable code will inadvertently crash the whole system and prevent recovery.

Object oriented support also required implementing. There was a need to initialise static and global object declarations by calling the constructor in the application code that uses C++. An option of two programs, "collect" and "munch" were available from GNU to perform this function. "Munch" was used to parse the codes before generating the necessary calls to the constructor. A reference was given in a file generated, which contained addresses to call the functions. The OS performed object initialisation during initial start-up, prior to executing the application codes using this reference.

One of the primary aims of the OS was to hide the hardware architecture from the application, hence enabling this application to be ported to the new system. However, due to the nature of the hardware layout, some minor changes in the application codes cannot be avoided. One example was the software routing system. The previous routing system implementation accessed the hardware registers directly to perform inter-processor communications. This is deemed unsuitable, since the new hardware will generate interrupts whenever communications complete, and may require further immediate attention from the user tasks. Furthermore, direct links to other processing nodes are no longer available, and correct routing headers are required to route to the intended destinations. The new implementation of the software routing system is illustrated in Figure 6-2.

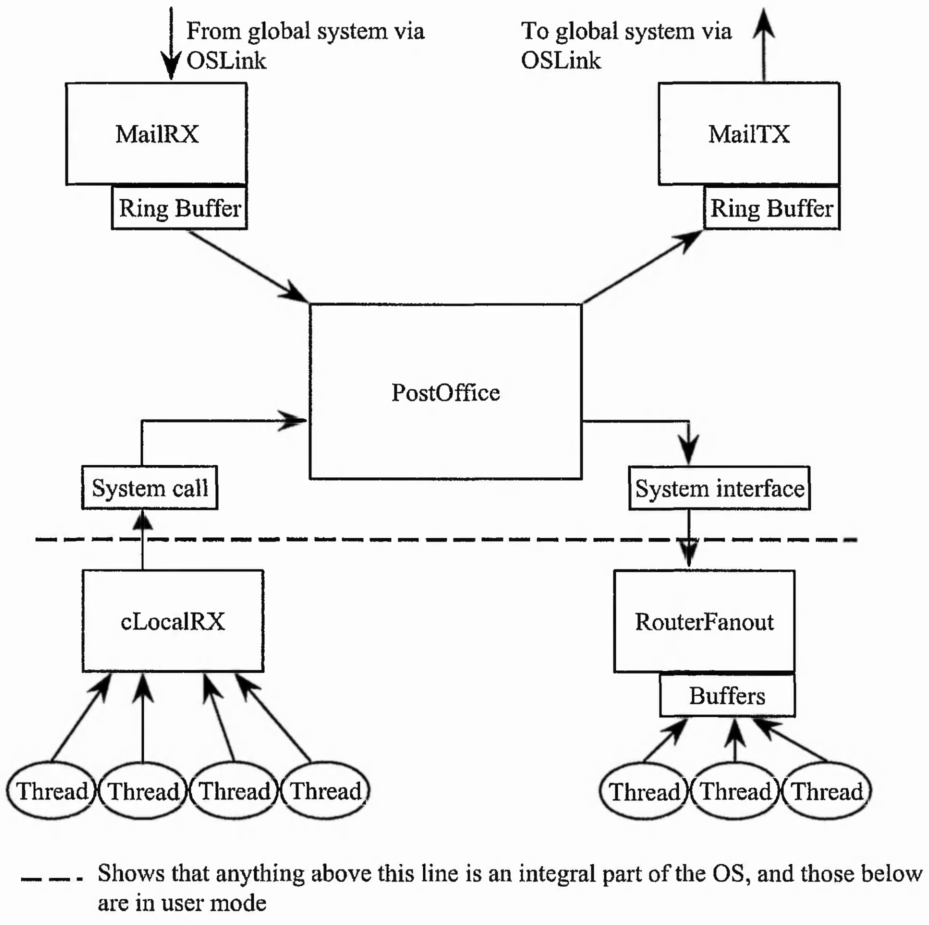

From global system via OSLink
To global system via OSLink
MailRX
Ring Buffer
MailTX
Ring Buffer
PostOffice
System call
System interface
cLocalRX
RouterFanout
Buffers
Thread
Thread
Thread
Thread
Thread
Thread
Thread
Shows that anything above this line is an integral part of the OS, and those below are in user mode

Figure 6-2 the software routing system

In Figure 6-2, MailRx will handle any messages arriving via the OSLink and forward them to the PostOffice module. This module will buffer the incoming messages into a non-cacheable memory region. The cLocalRx module will handle a fan-in of messages, and forwards them to the PostOffice. Messages in cLocalRx are buffered as well, and any threads, upon successful sending of messages to this module, will be de-blocked. Communication between local threads and the cLocalRx module is via soft channels.

Once any message arrives at the PostOffice, the PostOffice will determine whether the destination for the message is local or global. This is determined from the message header, which the PostOffice will cross-reference with a lookup table that it generates during start-up. The message will be copied into MailTx's message buffer if it is meant for a global processor, or it will be copied into the RouterFanout message buffer if it is meant for a local process.

MailTx will handle outgoing messages via the OSLink, and queue the requests to send in a FIFO manner. After each successful DMA transfer, the next message to be transmitted is fetched from the ring buffer. The RouterFanout module will forward the message to the appropriate thread, (if the thread has already requested the message), or it will continue to wait for another incoming message from the PostOffice. It is crucial that these modules are not blocked, due to the local threads being busy and unable to attend to those messages, otherwise the hardware link would stall.

The cLocalRx module and the RouterFanout module were implemented as part of the user process of high priority. The rest of the software routing modules are made part of the kernel, which were successfully integrated into the system.

Additional changes in the event system were needed as well. In the application, a process cEventTask was used to notify event handlers of certain events, which were interrupt generated by selected external sources. This was previously done through a transputer function. To simulate the transputer function in the new system, the cEventTask, which runs on high priority and communicates with other Event Handler tasks, via soft channels, will make a system call to the OS to request for events. cEventTask will then be blocked until the requested IRQ occurs. The IRQ generated could be from several sources simultaneously.

Event handlers will run in a loop, to continuously request for messages, denoting interrupts, from cEventTask via soft channels. They will be blocked when there is no interrupt pending to be serviced. Thus, when the selected sources generate the IRQ, these IRQ would be cleared and disabled. The bits representing the active event sources are then forwarded to cEventTask before this process is unblocked. After getting these bits, cEventTask will then forward messages through soft channels to the corresponding event handlers, depending on the active bits received from the OS. The event handlers will be activated upon receiving messages from cEventTask to perform their related operations before re-enabling the interrupt on the selected sources again. Another function of cEventTask is to decouple those event handlers from the OS, as some handlers may run as low priority tasks, which may potentially block the whole system.

This new system for handling events is illustrated in Figure 6-3.

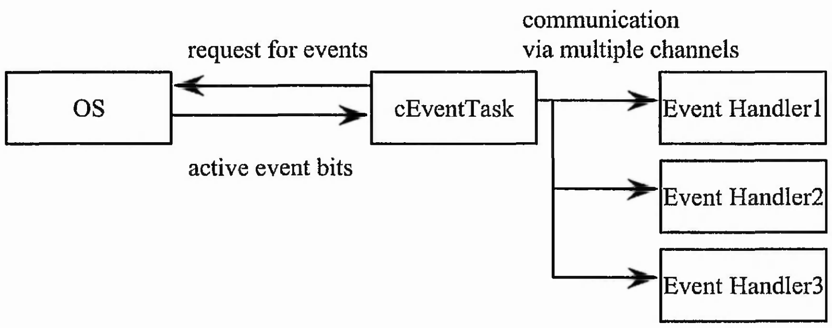


Figure 6-3 the flow for handling events in user tasks.

6.4 Debugging Facility.

For debugging, an embedded system requires constant checking for invalid circumstances, ranging from invalid message received to invalid channel addresses. One approach would be to maintain a log[76] on the system activity, and to retrieve the records should anything go wrong. Another approach would be to trace the activity using stack backtracking. Since Quantel's application code uses the stack backtracking method, support for stack walkback had to be provided.

Inherently, the application codes perform the checking explicitly by implementing 'assert' to check for false conditions. If there is an assert fail, a stack walkback is required. The stack walkback should display the sequences of source lines called, from the point of the assert call to the starting point of the thread. These include all the subnested subroutine calls. The application previously relied on the transputer system for this feature. In this project, the feature was implemented differently. The source file was compiled with the debugging option turned on. The debugging information or stabs, are then dumped into a file, and subsequently merged with the image file at the final stage. Additional tools were built to complement the tools available from GNU. At every assert fail, a list of information is displayed, ranging from the name of the source code file, to the name of the function and the line number. These are helpful in providing the developer information of where the system went wrong during trial run.

6.5 Summary.

This project lasted for almost 6 months, and most of the software components were successfully ported to the new hardware. Real time requirements from the application were met[77]. Ultimately, the system will be used to replace the transputer in Quantel's[72] production line. This phase should be completed after some minor hardware faults have been rectified.

Since the application demanded a different approach towards managing inter-processor communications, the communication modules in SARNUX responsible for inter-processor communications were not used. This is one of the differences in the variant of SARNUX.

In essence, the project carried out demonstrated the operation, application and benefits of SARNUX. Feedback obtained from this project, in return, has contributed to the refinement of SARNUX (which has already been implemented).

7 Discussion, Conclusions and Further Work.

This chapter begins with a discussion on the OS, and on the tools[98] developed to aid parallel programming. This is followed by the conclusion of the thesis, summarising the work and the original contributions made. In the final section, future work is discussed.

7.1 Design Discussions.

In the StrongARM based SARNET system, an OS is needed for software applications to run on the hardware. Adopting an existing OS is a risk, as the learning curve of the targeted OS is typically high, and the end result may not necessary meet the intended objectives. Furthermore, most parallel embedded OS's are customised systems with proprietary coding, or do not meet the requirements for the intended application. Therefore, a decision was made to develop a novel OS for the SARNET.

Recent work on parallel systems has focused on clusters based systems, using high-speed communication networks, like the Myrinet[95] with desktop PC based OS. Among the OS utilised by these systems are Linux and Microsoft Windows. Linux is deemed not appropriate for systems that are memory constrained, particularly in embedded systems, due to its large memory footprint. Although Linux could be theoretically stripped down to fit into embedded systems, the result may not achieve the performance, and Linux may also lose its attractiveness as a major platform for various applications when some functions in it are stripped off. In addition, there is a high learning curve to achieve strip Linux down to the required level of functionality. Microsoft Windows lacks the real-time response required by these embedded applications.

In the embedded systems market, more development is centred on 'high-end' OS's like VxWorks or OS9. However, although these systems offer the desired functionalities and attractive features, they have cost considerations. Generally, their source code is not available and hence could hamper debugging or optimisation efforts to achieve improved performance. In certain situations, it may even prove impossible to port the OS to the intended platform, if it is not supported, such as porting OS9 to the new StrongARM platform. In addition, most of these OS's focus on the Ethernet based communication, using the popular TCP protocols, which is not suitable for the intended platform nor channel based communication systems. The SARNUX system is deemed more feasible for a SARNET parallel processing system targeted for embedded applications, primarily to port previous applications developed based on transputer systems.

7.1.1 Comparison to the transputer system.

One of the design objectives was to maintain some similarities between SARNUX and the transputer while extending the transputer functionalities[96], the aim being to run applications previously designed for the transputer on the new StrongARM platform. One major difference is in the communication module. In the transputer system, additional software support was needed when routers were used to scale the parallel network. In SARNUX, the additional software support is already an inherent part of the OS. This includes virtual channel support for inter-leaving of messages, in accordance with the C416 router protocol. Other features, which overcome some of the transputer limitations, include a variable number of priority queues and a configurable time-slice in scheduling. In essence, SARNUX fulfils the goals in terms of functionalities[97].

SARNUX is mainly backward compatible to the transputer system and the only exception is the external communication calls. In the design, the global communication procedural call was not made the same as the local communication procedural call, primarily due to an overhead trade-off at the expense of backward-compatibility. Hence, some modifications are needed to port previous applications developed based on the transputer system, however they are only minor modifications (based on the experience gathered from porting an end user application, described in Chapter 5). The difference in the procedural call gives an evident advantage as the system allows the use of the Network Specifier, and the NTU-Configurer tools, in providing automated routing information for each channel.

SARNUX provides a high level of abstraction for ease of parallel programming [100], compared to the transputer. The user can specify graphically which channel to communicate to globally using the Network Specifier. Users are not required to specify routing headers or message identification tags. Since the transputer used a point-to-point link between two processors which supported only a single pair of channels, any use of routers to improve network scalability would have required explicit programming by the developer, including the handling of routing headers and message identification. This additional responsibility has been removed in the new system. The developer only needs to specify textually, in the Windows GUI, the connections between global channels. This high level of abstraction also allows the developer to change design easily since the binding of global channels is done at a higher level, and not at the code level, providing a degree of independence between application programs.

7.1.2 Portability of the OS.

The issues of portability and a high level of abstraction have greatly influenced the development of the SARNUX OS. However, this is at the expense of performance. Therefore, a trade-off was required. In designing the communications mechanism, the targets were low interrupt latency, and low-latency high throughput communication with a low CPU load. To accomplish this, some portions of the communications, particularly the interrupt handler, were written in assembly language; hence, unfortunately reducing the portability of the system. In the design, only 4 files were written in assembly language, which involved context saving, context loading, interrupt service identification, and software interrupt identification.

7.1.3 Reducing context-switching overheads.

In general, OS's incur a heavy penalty in overheads during context switching[33] and this is inevitable in pre-emptive scheduling, when all registers need to be saved, as the process may be context-switched during the interrupt event. The transputer system and some other OS's such as CCSP[35] support purely co-operative scheduling or make de-scheduling points available in the application code, and as such not all registers are saved, achieving lower context switch time.

Since the StrongARM compiler does not support automatic insertion of de-scheduling points into the application code, an alternative to saving all the registers could have been to limit the usage of registers by application programs. If, for instance, the number of registers available to the program is limited to only four, each context-switch will only require the OS to save the four registers, cutting the overhead significantly, but at the expense of performance. Furthermore, it would have limited the application programmer from using a third party software library (normally available in compiled code that does not place a limitation on registers used). Since the saving of all registers is required, the new OS attains performance improvement through the use of caches, as observed in the experiments conducted in this thesis and usage of assembly language in developing the kernel. The use of assembly language is kept to a minimum, for portability reasons.

7.1.4 Static compiling and linking.

Currently, any application program written for the SARNET is statically compiled and linked to SARNUX. This strategy should not pose a problem, as most applications in embedded systems where tasks do not demand dynamic uploading onto an executing OS. The primary reason to adopt a static approach is because it is easier to implement, and more reliable than the dynamic approach. However, the dynamic features of SARNUX may prove useful in certain situation, and will be discussed in the future work section.

7.1.5 Buffering of messages.

The use of buffers to receive global messages proved to be effective in reducing overheads and stalling of hardware links. Stalling of hardware links may result in deadlock. When a message arrives via an OSLink, intended for the receiving process, and if this receiving process is presently blocked while waiting for a different message, the OSLink will stall. Once stalled, an error occurs since the receiving process will never resume execution, as the prior message, that it waits for, will never arrive while the OSLink is stalled. Eventually, all the local processes may be blocked if they attempt to receive any global messages, causing deadlock. The use of buffers prevents this scenario from occurring by storing the second message into the buffer first, enabling the receiving process to receive the first message, and then process the second message in the buffer.

In the communication module, a non-blocking communication is adopted. Blocking may appear if the programmer under-estimated the application program requirements, causing buffer overrun. The non-blocking method involves the software flushing blocked messages and flagging an error, the aim being to keep the router link free. In addition, to prevent blocking, large buffers are used but again, they will overrun eventually if the receiving process cannot consume the messages fast enough. The blocking method will only flag a warning, without keeping the link free, as the system waits for the local processes to consume the messages. However, the blocking method is not preferable, since the packet router communication protocol splits messages into fixed size packets, where the maximum size is being dictated by the hardware. Had the blocking method been used, this may have resulted in wormhole blocking, thus increasing network latency and the possibility of deadlock.

The use of large buffers however results in large memory requirements. Therefore, a trade-off must be made when determining the buffer size. For buffering a message of 64kbytes in a ring buffer of, for instance five would require 320kbytes of memory, just for a single channel. If there were a few channels using the maximum length in their messages, the total memory requirement for buffers would be increased to a few megabytes. The large requirement of memory is not desirable, since in most embedded applications, the memory size tends to be limited.

With the usage of buffers in the developed system, the compatibility with CSP may be lost (subject to further investigation). In addition, the proof of correctness in a parallel program using CSP may be affected. Therefore, to maintain compatibility, further investigation is required as future work.

7.1.6 Other issues.

An additional feature of the OS is to run a portion of the lower priority user tasks in a non-interruptible mode. In transputer systems, this could only be achieved by partitioning the non-interruptible portion of the code into another task, which ran at high priority. The transputer then performed channel communication between these two tasks, to run the non-interruptible section of the code. In this new system, a system call is inserted into the beginning of the non-interruptible codes, and another system call is made at the end of it, allowing that portion of code to run at high-priority. In this way, the task is running at high-priority, and the interrupt is disabled for the whole system. There exists some risks to this strategy, as any unreliable code will inadvertently crash the whole system and prevent recovery. Since the error is deemed as a user error, the developer will have to guard against it.

7.1.7 Summary

The work has demonstrated that the portability and functionality issues have been adequately addressed. In addition, a real world application program has been ported from a transputer platform to the SARNET platform, while maintaining the integrity, functionality and real-time requirement of the program. The success in porting the code demonstrated the theoretical and practical work of this thesis

7.2 Conclusion.

The objectives of this research were to investigate the following:

- The implementation of an OS for the SARNET platform, enabling applications to be developed for the parallel network.
- The efficiency of the OS in context switching and message passing.

- The porting of application code written in Parallel C previously targeted at the transputer to the new platform.

This research has resulted in two main original contributions:

- The design and development of a new OS called SARNUX for the new parallel StrongARM platform. The OS manages the hardware resources, including the message-passing system in the StrongARM processing node, by providing support for inter-leaving of messages and virtual channels. SARNUX provides backward compatibility with applications developed using Parallel 3L targeted at transputers, and this has been demonstrated in the success in porting a large end user application to a SARNET system, with minor modifications.
- The development of additional tools that provides a higher level of abstraction [99] in assisting the end user to develop a parallel application. These tools include the Windows GUI and the Configurer, that allow the user to specify the network layout graphically, and the connection between channels across the network textually. The connection between two channels across the network is handled transparently, whereby the Configurer would automatically generate the routing headers and the unique message identification to support virtual channels.

This thesis has detailed the SARNUX implementation, which has been developed, tested and profiled. This system is able to run standard benchmark codes like Dhrystones and Commstimes after slight modifications for housekeeping purposes. Additionally, SARNUX has successfully demonstrated that its backward compatibility has been met, with the porting of an end user application written in Parallel C. The OS has also successfully managed local channel communication, virtual links, and the support for interleaving of messages.

SARNUX has been found to be working in accordance to the OSLink communication protocol. The StrongARM nodes could communicate without any problems, and the overheads involved in communication have been measured. The results showed that to send a message via an external link, the overheads incurred were only 10 microseconds for a successful transmission of 4 bytes, and 6 microseconds for a successful receipt of 4 bytes. These overheads include the hardware latency in communicating the message. The kernel achieves low context switching latency of only 4 microseconds for a full context save. Although the figure cannot match the 2-microsecond context switch achieved by the transputer, where no full context save is performed, it is considered satisfactory as other options, such as limiting the number of registers used, will result in other problems as previously discussed.

SARNUX was developed using ANSI C, and compiled using the GNU cross-compiler, targeted at the StrongARM. The modular approach adopted in designing the OS also allows ease of debugging and re-usability of modules for various purposes.

SARNUX features include:

- Pre-emptive and non pre-emptive scheduling, with configurable time slicing for low priority processes. Configurable time slicing was not available on the transputer system.
- High priority processes are not time-sliced.
- Static configuration of the number of priority levels, which can be determined by the end-user during compile time. This allows flexibility to the developer to tailor the system. The transputer priority levels were confined to two, which would be a constraining factor on modern applications.
- Event timer support for managing and handling all timer events. These events include a timeout for channel communications, and process requests to sleep for a specified time.
- Support for inter-processor communications with transparent handling of 4 DMA engines: two for sending and two for receiving messages. Essentially, the system supports a dual link.
- Support for virtual channels, allowing more than one channel to be multiplexed on a single link. This feature overcomes the hardware limitation on the number of links available. Virtual links were not supported by the transputer.
- Support for inter-leaving of more than two messages, although the current StrongARM platform supports interleaving of just two messages.
- Buffering for receiving messages, which improves network efficiency by reducing network stalling. The developer can specify the number of buffers.
- Soft channel communications, for communications within a single processor. This feature was also implemented by the transputer.

- High level of abstraction for ease of parallel programming [100] where routing headers are generated automatically.
- Small memory footprint, as low as 27 Kbytes, reduces costs as the memory space is typically constrained in embedded systems.
- Support for basic 3L library calls, allowing applications developed using 3L to be ported to the StrongARM node.
- Other support for debugging like stack walkback, semaphores, UART drivers and explicit stack overflow checking.

Communication services are integrated into the small kernel, and have a form of message passing via channels. The system has advantages like a clear system structure for applications and easy parallelisation of services and tasks [101], which is similar to the transputer. Furthermore, processes can be moved from one processor to another without major modifications to the codes, since each process is modular, and there is no direct procedural call to a local process. Essentially, both communicating processes could either reside on different processors, or on the same processor.

In this thesis, the performance of the system has been demonstrated, with and without the effect of stealing bus cycles to perform communication via DMA engines, while running a standard benchmark test. The worst case scenario, as discussed in previous chapters, is only an increase of 20% in computing time when DMA engines are active compared to when no DMA engines are active. The communication capabilities of this system are shown to be comparable to those of the transputer, without some of the associated disadvantages of that system. In the best case scenario, only a 3% increase is registered to complete the test when there is active DMA communication in the background. The tests also demonstrated that the on-chip caching of the system reduces data and instruction bus accesses, allowing a higher bandwidth for the DMA engines to steal cycles to deliver messages. The good performance figures achieved with caches in modern processor gives problems with cache coherency issues. In the developed OS, the cache coherency problem is simply resolved by marking certain the global communication area is not cacheable.

7.3 Future work.

The current OS has been successfully implemented and tested on a multi-processor network. However, a few other tasks could further refine the system to achieve lower overheads, and certain features have yet to be implemented. This future work includes:

- Integrating the whole system under a GUI development interface, with the compiler running as a background task on the desktop PC.
- Improvement to the Windows GUI and the Configurer for better graphical representation of the network.
- Modification to the GNU compiler to provide automatic stack checking.
- Port the OS to a wider range of embedded parallel StrongARM systems.
- Other improvements.

7.3.1 Integration of the whole system under a GUI.

In the current development, the integration of the whole system under the GUI has yet to be carried out. The ultimate goal is to enable the developer to specify which image file would run on which processing node. The image file must be able to be compiled, and booted down the OSLink, via the generic ISA Interface, with simple clicks of the mouse button. This task would require additional development, but it should be achievable since the basic components like the Windows GUI used to specify the network layout, the Configurer to determine routing and message headers, and the OS, have been developed. The cross compiler and the Configurer could be invoked by simple library calls, using MFC, and the routing headers to boot the image file could be extracted by the file produced by the Configurer. Therefore, with the existence of the tools described, the integration of the system under one GUI umbrella could be achieved with the objective to provide a higher user level abstraction for ease of parallel programming.

7.3.2 Improvement to the Windows GUI and the Configurer.

The Windows GUI, although performing as expected, could still be improved to have a better graphical representation. For instance, the current implementation does not graphically specify the link number for the router. Improvement to the Configurer would include generation of tables for the network mapping into a text file, for ease of debugging. Additional macros could be added into the map file generated, which would include the required routing headers for downloading a specific image file to a specific StrongARM node.

7.3.3 Modification to the GNU compiler to provide automatic stack checking.

Automatic generation of stack checking codes upon each entry into a function, to check for stack overflow has not been implemented in the GNU compiler. This feature is implemented in the ARM version of the C compiler, and is crucial in preventing system crashes due to stack overrun during runtime. The difficulty in debugging an embedded system failure makes this an important feature. At present, SARNUX checks for stack overrun explicitly.

As extra precautionary measures to prevent system crashes, additional two checks were added to the system, to check for a valid program counter and to check for word alignment in the task's stack. This increased the context-switch time by approximately one microsecond. However, the deterioration in performance was justified by the benefits gained, saving invaluable time during system development. In the final release version after the system is deemed stable, these explicit checks could be removed by re-compiling the OS again by specifying different options to exclude the checks, and hence improving performance. These checks could be removed after modifying the GNU compiler to provide automatic stack checking. This task would require in-depth understanding of the cross-compiler, and modification to the compiler's source code could then be carried out to achieve the goal.

7.3.4 Port the OS to a wider range of embedded parallel StrongARM systems.

An application running on SARNUX has yet to be extensively tested on a SARNET. Hence future work would be to run SARNUX on a wide range of SARNET topologies and parallel applications, and possibly other StrongARM systems similar to the Quantel industrial application.

7.3.5 Other Improvements.

- Future development features include the capability of the OS to be extended without modifying the kernel. Additional facilities could be included to dynamically change the priority level of a process.
- Conformance to certain proprietary standards, such as the Portable Operating System Interface (POSIX)[103] is desirable. However, it may prove difficult to follow as the standards generally do not state the specification for an embedded OS[104].

- The dynamic uploading of a task to the OS may be desirable to resolve certain run-time issues, for example debugging and network diagnosis or enabling software changes at runtime. The mechanism will require further investigation.
- Features such as minimal fault-tolerance [105] support for network failure are proposed for future investigation, to aid in error recovery.
- Improvement to the NTU-Configurer to use up any unused routing headers in circumstances when the number of global channels exceed 255.
- Improvement to the GUI and NTU-Configurer to enforce the rule of network symmetry and to perform checks on the validity of hardware connections.
- To extend the GUI and NTU-Configurer to assist other similar parallel hardware platforms which requires wormhole routing support, for automated routing headers generation.
- The OS could be made to retarget other processors in the future, if the hardware platform changes from the StrongARM to another processor that is more suitable.

Publications

1. E.W.K. Liew, B.C.O'Neill, K.L. Wong, S. Clark, P.D. Thomas and R. Cant, 'A Proposal for an Operating System for a Multi-Processor StrongARM System', Concurrent Systems Engineering, ISBN 90-5199-480-X, Vol. 57, April, 1999, pp 37-47.

2. E.W.K. Liew, D. Kaye, B.C.O'Neill and S. Clark, 'Operating System Support for StrongARM Multi-Processor Communications', Proceedings of the ISCA 13th International Conference on Parallel and Distributed Computing Systems, ISBN 1 880843 34 X, Las Vegas, US, Aug 2000, pp 334-339.

3. E.W.K. Liew, B.C.O'Neill and S. Clark, 'Porting Transputer Application to Multi-Processors StrongARM system', Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'2001) under the International Parallel & Distributed Processing Symposium (IPDPS 2001),San Francisco, US, April 2001, sponsored by IEEE Computer Society.

4. E.W.K. Liew, B.C. O'Neill, S. Clark, 'A Configurer and GUI network specifier for a Multi-Processor StrongARM network', Proceedings of the ISCA 17th International Conference on Computers and their Applications, IBSN 1 880843 42 0, San Francisco, US, April 2002, pp 283-288.

References

All webpage references are correct as of 4th February 2002.

[1]R.W. Hockeny, C.R. Jesshope, Parallel Computers 2, IOP Publishing Ltd, 1988.

[2]SETI project, see <http://www.setileague.org/general/setihome.htm>.

[3]B.M.Cook and Roger M.A.PEEL, The Para-PC, An Analysis, Parallel Processing Developments, IOS Press, 1996, pp 89-102.

[4]Inmos Ltd is now defunct, however further details about transputers, see <http://us.st.com/stonline/index.shtml>.

[5]Private discussion held with Conference Delegates during the Wotug conference, April 1999.

[6]Held, G., "Data Communications Networking Devices – Operation, Utilization and LAN and WAN Internetworking", 4th. Ed., John Wiley & Sons Ltd., 1999, pp. 300-328.

[7] Al Geist, et al, PVM: Parallel Virtual Machine The MIT Press, ISBN: 0262571080.

[8] Pacheco, Parallel Programming with MPI , Morgan Kaufmann; ISBN: 1558603395.

[9]The Nottingham Trent University Parallel Research Group, see <http://eee.ntu.ac.uk/research/parallel/>.

[10]M.D.May, P.W.Thompson, P.H.Welch,Networks, Routers & Transputers : Function, Performance and Application. IOS Press, 1993, Chapter 2(the T9000 Communications Architecture).

[11]Coulson, G., "Optimisation of a Processing Farm Using Hardware Routing", Transputer Applications and Systems, IOS Press, 1995, Vol. 46, pp. 70-77.

[12]Coulson, G., "An ASIC Implementation of A Multicast Message Routing Switch for Interprocessor Communications", Ph. D. Thesis, The Nottingham Trent University, September 1998.

[13]Wong,K.L., O'Neill, B.C., Hotchkiss, R., Ng, J.H, Clark, S., Thomas, P.D., Interfacing StrongARM Microprocessors in a Parallel Network, Postgraduate Research in Electronics, Photonics & Related Fields (PREP'99), January 1999, pp.382-385.

[14]M. Esponda, R. Rojas, The RISC Concept- A Survey of Implementations, Technical Report B-91-12, Freie University Berlin, September 1991.

[15]ARM System Architecture, Steve Furber, Addison-Wesley Longman Inc, 1996.

[16]Wong, K.L., "A Message Controller for Distributed Processing Systems." Ph. D. Thesis, The Nottingham Trent University, April 2000.

[17]"ICR C416 - 16-port Dynamic Routing Switch for Transputer Links - Data Sheet", IC-Routing Ltd., 1996.

[18]C.P.H. (Paul) Walker, Hardware for transputing without transputers, IOS Press, 1996, pp1-10.

[19]Occam 2.1 Toolset User Guide, Document Number 72 TDS 366 02, SGS-Thomson Microelectronics, 1995.

[20]Zhiwei Xu, Kai Hwang, Coherent Parallel Programming in C//, Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Computing, 1997, pp 116-122.

[21]ANSI C toolset user manual, Inmos Ltd, 1990.

[22]ANSI C toolset user language reference, Inmos Ltd, 1990.

[23]D.C. Wood and Peter H. Welch, The Kent Retargetable occam Compiler, IOS Press, 1996, pp143-166.

[24]M. Debbage, M. Hill, S.Wilkes and D.Nicole, Southampton's Portable Occam Compiler (SPOC), Progress in Transputer and Occam Research, Ed R Miles and A Chalmers, IOS Press, 1994, pp 40-55.

[25]B.M. Cook, Technical Report TR96-01, "Occam Virtual Channells over Ethernet", ISSN 1353-7776, Dept. Computer Science, Keele University, Keele, Staffs, ST5 5BG, UK.

[26]Tim Sheen, Alastair R. Allen, Andreas Ripke, Stacy Woo, oc-X: an optimizing multiprocessor occam system for the PowerPC, IOS Press, 1998, pp167-185.

[27]P. Singleton, B.M. Cook, The Globalisation of Occam, IOS Press, 1996, pp255-270.

[28]Manuel Mollar, A Small Parallel C Simulator, Proceedings of PDP, IEEE, 1996, pp123-129.

[29]E.W.K. Liew, B.C.O'Neill, K.L. Wong, S. Clark, P.D. Thomas and R. Cant, 'A Proposal for an Operating System for a Multi-Processor StrongARM System', Concurrent Systems Engineering, ISBN 90-5199-480-X, Vol. 57, April, 1999, pp 37-47.

[30]E.W.K Liew, D.Kaye, B.C.O'Neill and S.Clark, Operating System Support for StrongARM Multi-Processor Communications, Proceedings of the ISCA 13th International Conference, Las Vegas, USA, 2000, pp 334-340.

[31]E.W.K. Liew, B.C.O'Neill and S. Clark, 'Porting Transputer Application to Multi-Processors StrongARM system', Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'2001) under the International Parallel & Distributed Processing Symposium (IPDPS 2001),San Francisco, US, April 2001, sponsored by IEEE Computer Society.

[32]Jeremy Hinton and Alan Pinder, Transputer Hardware and System Design, Prentice Hall, 1993.

[33]Andrew S.Tanenbaum, Albert S. Woodhull, Operating Systems: Design and Implementation, Second Edition, Prentice-Hall International Inc, 1997.

[34]Linux, see < http://www.linux.org/>.

[35]J.Moores, CCSP – A Portable CSP-Based Run-Time System Supporting C and Occam, IOS Press, 1999, pp147-168.

[36]G.Hilderink, J.Broenink, A.Bakkers, Communicating Threads for Java, IOS Press, 1999, pp243-261.

[37]Richard Johnsonbaugh, Martin Kalin, Object Oriented Programming In C++, MacMillan; ISBN: 0023606827, 1994.

[38]Peter H.Welch, Java Threads in the Light of Occam/CSP, Architectures, Languages and Patterns for Parallel and Distributed Applications, IOS Press, 1998, pp 259-284.

[39]Jon Meyer, Troy Downing, Java Virtual Machine, O'Reilly UK; ISBN: 1565921941, 1996.

[40]Barabanov and Yodaiken, Real-Time Linux, Linux Journal, March 1996.

[41]P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, RTAI: Real Time Application Interface, Linux Journal, April 2000.

[42]Steve Shah, Linux Administration: A Beginner's Guide, McGraw-Hill Professional Publishing, 2000.

[43]J. Epplin, Linux as an Embedded Operating System, Embedded Systems Programming, Vol. 10, No. 10, Oct 1997.

[44]FsmLabs, see <http://www.fsmlabs.com/>.

[45]GNU general public license, see <http://www.gnu.org/copyleft/gpl.html>.

[46]VxWorks Programmer Guide, 5.3.1, WindRiver Systems.

[47]3L library, see <http://www.sundance.com/ext-webs/3l/>.

[48]U. de Carlini and U.Villano, Transputers and Parallel Architectures: message-passing distributed systems, Ellis Horwood Limited, 1991.

[49]Informal discussion based on the Real Time Linux newsgroup, <majordomo@rtlinux.org>.

[50]Kaare Christian, The Unix Operating System, John Wiley & Sons Ltd, 1983.

[51]Gary Cornell, Cay S. Horstmann, Core Java, SunSoft Press, 1997.

[52]"ICR C416 – 16 port Dynamic Routing Switch for Transputer Links – Data Sheet", IC-Routing Ltd., 1996.

[53]F. Thomas, M.M. Nayak, S. Udupa, J.K. Kishore, V.K Agrawal, A hardware/software codesign for improved data acquisition in a processor based embedded system, Journal of Microprocessors and Microsystems, Vol 24, No 3, Elsevier, June 2000.

[54]IC Routing, see <http://www.eee.ntu.ac.uk/research/parallel/icrltd.html>.

[55]Michael J. Smith, Application-Specific Integrated Circuits, Addison-Wesley Pub Co, ISBN: 0201500221, 1997.

[56]"Altera 1998 Data Book", Altera Corporation, 1998, pp. 21-132.

[57]E Cervera. Transparent Sharing of Links in Inmos Toolset Applications, Transputers Applications and Systems, IOS Press,1994, pp 739-747.

[58]George M. Candea, Michael B. Jones, Vassal : Loadable Scheduler Support for Multi-Policy Scheduling, Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA, August 1998, pp 157-166.

[59]L. Klienrock, Queuing Systems, Volume 1, John Wiley, 1974.

[60]Robert Cecil Martin, Designing Object Oriented C++ Applications Using The Booch Method, Prentice Hall, ISBN: 0132038374, 1995.

[61]Perdita Stevens, Rob Pooley, Using UML : Software Engineering With Objects and Components, Longman Higher Education; ISBN: 0201648601, 1999.

[62]Harry, Introduction to Formal Methods, John Wiley and Sons, ISBN: 0471958573, 1996.

[63]POSIX information, see < http://www.pasc.org/>.

[64] B C O'Neill, K L Wong, G C Coulson, R Hotchkiss, J H Ng, S Clark, P D Thomas & A Cawley, 'A Distributed Parallel Processing System for the StrongARM Microprocessor' Concurrent Systems Engineering Vol. 52, ISBN 90-5199-391-9, April 1998, pp 39-48.

[65]Private discussion held with Alec Cawley, Quantel in determining acceptable time quantum as tolerance value.

[66]Richard M. Stallman, GNU Reference, iUniverse.Com, Inc., ISBN: 0595100376, 2000.

[67]Steve Hunger, Debian GNU/Linux Bible , Hungry Minds Inc; ISBN: 0764547100.

[68]Redundancy and robustness in memory protection. Invited paper, Proc. IFIP Cong., North-Holland, 1974, pp 128-132.

[69]Conversation with conference delegates during the presentation in the Parallel and Distributed Computing Systems (PDCS) conference, chaired by Dr. Y.F. Lee (Intel), Las Vegas, August 2000.

[70]PowerPC 601, RISC Microprocessor User's Manual, Document No MPC601UM/AD, Revision 1, Mtorola Inc., 1993.

[71]Parallel C user Guide, 3L Ltd., 1991.

[72]Quantel Ltd, see <www.quantel.com>.

[73]Steven Howell, NgocDung Hoang, Cuong Nguyen, Critical Issues in the Design of Large-Scale Distributed Systems, Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems, 1997, pp28-33.

[74]Communicating distributed objects, see <http://www.cawley.demon.co.uk/>.

[75]Parallel 3L libraries information, see < http://zulu.threel.com/transput/ >.

[76]B. Wirz, E.Nett, A Generic Log-Service Supporting Fast recovery in Distributed Fault-Tolerant Systems, Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems, 1993, pp121-126.

[77]Cuong Nguyen, S. Howell, E. Lock and B.Prasad, Employing System Design Factors for Optimization and Trade-Off Analysis in Distributed Real-Time System Software Design Structuring, Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, April 1993.

[78]Paul Albitz, Cricket Liu, DNS and BIND, O'Reilly & Associates Incorporated, 1998.

[79]Thomas Maufer, IP Fundamentals: Addressing, Routing and Troubleshooting, Prentice Hall, 1999.

[80]Jeremy Hinton and Alan Pinder, Transputer Hardware and System Design, Prentice Hall, 1993.

[81]Shoichi Noguchi, Hiroshi Umeo, Transputer/Occam 4, IOS Press; ISBN: 9051990936.

[82]ANSI C Toolset User Guide, Document number 72 TDS 234 01, Inmos Ltd, 1992.

[83]MagnumPlus, see <http://www.kentrox.com/products/70455/70455.htm>.

[84]Peakware,see <http://www.matra-msi.com/ang/savoir_infor_peakware_d.htm>.

[85]Michael J. Young, Mastering Visual C++ 6, SYBEX, 1998.

[86]Stephen D.Gilbert and Bill McCarty, Visual C++ 6 Programming Blue Book, Coriolis Technology Press,1998.

[87]Alison Cawsey, The essence of Artificial Intelligence, Prentice Hall, 1998.

[88]Downloading benchmark codes from <http://128.169.92.17/benchweb/list.html>.

[89]A short synthetic benchmark program by Reinhold Weicker <weicker.muc@sni.de>, <weicker.muc@sni-usa.com>, intended to be representative of system (integer) programming.

[90]Commstime, see <http://wotug.ukc.ac.uk/parallel/>.

[91]Peter Welch, Commstime results for other system, published in the Occam-com mail group, November 2000. See also <http://wotug.ukc.ac.uk/list.shtml> for the mailing list information.

[92]Nicholas Carriero and David Gelernter, How to write Parallel Programs – A First Course, The MIT Press, ISBN 026203171-X, 1990.

[93]R. Grebe, et al, Transputer Applications and Systems 1993, IOS Press; ISBN: 9051991401.

[94]J.M. MacLaren, J.M. Bull, Lessons Learned when Comparing Shared Memory and Message Passing Codes on Three Modern Parallel Architectures, Proceedings from the high Performance Computing and Networking (HPCN), Springer, 1998, pp337-346.

[95]Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., Su, W.K., "Myrinet - A Gigabit-per-Second Local-Area Network", IEEE-Micro, February 1995, Vol.15, No.1, pp.29-36.

[96]E. Verhulst, Virtuoso: Extending the transputer concept to standard processors, IOS Press, 1994, pp869-883.

[97]C.M.Pancake and D. Bergmark, "Do Parallel Languages respond to the Needs of Scientific Programmers?, IEEE Computer, Vol. 23, No.12, 1990, pp 13-23.

[98] S.J. Turner, Tools and Techniques for Transputer Applications, IOS Press; ISBN: 9051990294.

[99]X. Ma and T. Hintz, A Perspective on Tools for Parallel and Distributed Computation – Background to the RE-Vision Project, Proceedings from the Transputer Research and Applications 5 (NATUG 5), IOS Press, 1992, pp169-177.

[100]J.C.Moure, Daneil Franco, Elisa Heymann, Emilio Luque, TransCom: a Communication Microkernel for Transputers, Proceedings of PDP 1996, pp 147-153.

[101]Wei Yuan, Yongqiang Sun, "SEQ OF PAR" Style Structure Parallel Programming, IEEE, 1997,pp82-89.

[102]A. Adir, N. Francez, S. Katz, Implementing a Language for coordinated distributed programming on a transputer multiprocessor, IOS Press, 1994, pp446-463.

[103]Bill Gallmeister, POSIX.4, O'Reilly UK; ISBN: 1565920740, 1994.

[104]Joseph Moses, "Is POSIX Appropriate for Embedded Systems?," Embedded Systems Programming, July 1995, p. 90.

[105]G.Deconinck, M. Truyens, V. De Florio, W.Rosseel, R. Lauwereins, R.Belmans, K.U.Leuven, A Framework Backbone for Software Fault Tolerance in Embedded Parallel Applications, IEEE, 1999.

[106]Mark A. Heilpern, OS-9 Primer, ISBN 0-918035-04-X.

[107] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[108] Andrew Blank, TCP/IP JumpStart, Sybex International, ISBN: 0782126448.

[109] J.H. Ng, B.C. O'Neill and S.Clark, A PC Interface Board for Parallel ARM Processor Networks, Proceedings from the Second Conference on Postgraduate Research in Electronics, Photonics and Related Fields (PREP), 2000, pp 469-474.

[110] ARM SDT 2.5, see <www.arm.com>.

[111] GreenHills cross compiler, see < http://www.ghs.com/products/RTOS_home.html>.

[112] Richard Paul, SPARC Architecture, Assembly Language Programming and C, US Imports & PHIPEs, ISBN: 0130255963.

[113] D. McKinney, et al, DEC chip 21066, The Alpha AXP Chip for Cost Focused Systems, COMPCON, digest papers, spring 1994.

[114]Inc. International Business Machines, RISC System/6000 PowerPC System Architecture, Morgan Kaufmann; ISBN: 1558603441.

[115] Windriver, see <http://www.windriver.com/>.

[116] Reed ,Multicomputer Networks Message Based Parallel Processing, The MIT Press, ISBN: 0262181290.

[117] Private discussion held with Prof Brian O'Neill, the Nottingham Trent University.

Appendix A

Comparisons between the transputer implementation and the SARNUX implementation.

Transputer	***SARNUX***
2 levels of priority	User-specified priority
Normal processes are in lower priority	Normal processes could be in any priority level
Low priority processes are co-operative and pre-emptive(if they run for too long, they are pre-empted)	Low priority processes are co-operative and pre-emptive
High priority is used for interrupt handlers (device handlers or communication handlers) and for high-resolution timing	High priority is for special processes. Interrupt handlers and timers are kept in non-interruptible supervisory routines
2 types of pre-emption method, i.e. active context-saving and non-active context saving	Only 1 type of pre-emption method i.e. active context saving
Has de-scheduling point instructions (inherent properties of hardware)	Doesn't have de-scheduling point instructions (could be done using a co-operative de-scheduling point instead)
High-priority processes are non-interruptible	High-priority processes are interruptible by the OS
High-priority processes are not time-sliced	High-priority processes are not time-sliced
Interrupts by external events will cause full-context saving (only for lower priority processes as high priority processes are non-interruptible)	Interrupts by external events will cause full-context saving (for low priority processes)
Uses FIFO scheduling mechanism	Uses FIFO scheduling mechanism
When a process is de-scheduled, the scheduler will first look at the high-priority run list first, and if it's empty, it'll run through the lower-priority run list.	When a process is de-scheduled, the scheduler will first look at the high-priority run list first, and if it's empty, it'll run through the lower-priority run list.

Communication comparisons:

Transputer	***SARNUX***
Channels are one way	Channels are one way
<u>Soft channels implementation:</u> If the other process is not ready for communication, the process is then made to wait Communication is un-buffered and synchronised When communication is completed, both processes are made active, i.e. not blocked Only 1 process can be waiting for a particular channel at any time The second process, which upon communicating finds another waiting is never de-scheduled Two processes have 2 separate data areas of memory Message passing involving copying of contents from one memory locations to the other To ensure that processes wish to communicate with each other, one process is an input and the other is output, with same number of bytes, these are checked by the compiler during compile time (usage checking)	<u>Soft channels implementation:</u> same as the transputer same as the transputer same as the transputer same as the transputer The second process, which upon communicating finds another waiting is always de-scheduled same as the transputer same as the transputer There's no checking from the compiler, hence it's left to the user to do the checking. The OS checks for these rules during run-time but only to a certain extent.
<u>External channels implementation:</u> If the other process is not ready for communication, the process is then made to wait No ring buffers are implemented to receive messages. The OS Link may be blocked if the receiving end is not ready Communication is un-buffered and synchronised When communication is completed, both processes are made active, i.e. not blocked The sender process can never proceed unless the receiver process has received the message.	<u>External channels implementation:</u> A process may communicate (transmit msg) if the message buffers of the receiving end is free, not when the receiver is ready. Ring buffers are implemented to receive messages. The OS Link may be blocked only if the receiving ring buffers are full. Communication is buffered initially, when the channels are yet initialised, and it's synchronised. same as the transputer, except that sometimes even when communications are completed, the receiving process are already active since the last message was only stored into a ring buffer and has yet to be read. Not necessary, as if the receiving end's ring buffers are free, the sender process may complete the transmission, and resume execution

The second process, which upon communicating finds another process waiting to communicate, is never de-scheduled until it is ready to communicate.	Both communicating processes will be context-switched after communicating, hence both must be de-scheduled after each communication. However, communication will take place via the receiver's ring buffers even the receiver process is not ready.
Two processes have 2 separate data areas of memory	same as the transputer (except that the receiver has ring buffers)
An output link may be accidentally used as an input link, causing error during run time. Compiler can only flag certain error during run-time.	If the NTU-Configurer is used, these errors could be avoided with the automatic headers generation.
When the same sending process wishes to send 2 consecutive messages to the same channel, but the receiver hasn't even read the first msg, the sender will be blocked until the receiver wishes to communicate for the 2nd time.	Similar to the transputer, but the receiver in this case could make a system call whenever it no longer requires the 1st msg, freeing the receive channel buffer, thus enabling the DMA of the 2nd msg. This will reduce the latency of the sender wait time. Thus when the receiver is ready for the 2nd msg, it'll find that the new msg is already stored in its msg buffer.
Data received from the hard link by the link process may be sent down to another soft channel, which is connected to another process locally. Another option if there's a lot of communication is to pass the data by reference to the actual receiving process instead of using channels, i.e. a technique called data streaming.	All data received by the irq_handler are automatically stored into the buffer provided by the receiving process (by the DMA engine), and this raw data will be directly operated by the receiver process once it's woke up. Essentially, the design is left to the user.
The instruction used to communicate either by link or soft channels are the same, with the same parameters. The only way to identify them is from the address of the channel, as link channel have special reserved address space during run-time.	The instructions are the same, but the way to identify them is from the channel properties during run-time, which will be supplied during channel initialisation

Performance comparisons:

Transputer	***SARNUX***
Scheduler codes are micro-coded on-chip, hence faster execution	OS provides scheduling, and since it's not micro-coded, there are higher overheads of accessing the slower external DRAM. It relies on the presence of cache for speeding up.
The maximum interrupt latency is affected by the operation of the current low-priority process, e.g. certain instructions like floating point or bit shifting (which can takes up to 85 seconds!) can't be interrupted halfway.	There's no significant interrupt latency, except if the interrupt is masked.
High priority process could hog the cpu and it's not interruptible. This increases the interrupt latency.	Same as the transputer.
Device drivers and link process has to share the same high priority queue with contemporary programs which may be of less importance and longer routines.	Device drivers and link process are placed at interrupt handlers, which has higher priority than contemporary programs (apart from the highest priority queue). These mechanism allows the service of urgent irq and fiq requests.
The on-board memory chip is just 4kbytes for fast memory access, severely limiting the program size for execution.	The built in cache is 32kbytes for fast memory access and slower 16Mbytes DRAM to suit contemporary program demands.
Context-saving (not for interrupt) involves saving only 2 registers, hence faster switching time	Context-saving (not for interrupt) involves saving 15 registers, and the status register, slower switching time
Higher overhead for context saving during interrupt, as all registers would be saved then.	Same overhead for context saving during interrupt
Process has convenient de-scheduling points inserted by the compiler, thus avoiding full-context saving. However, if the value needs to be used during re-scheduling, it needs to be loaded again from stack.	Process doesn't have de-scheduling points (which are inherently supported by compiler), therefore full context saving can't be avoided. But process could resume immediately during re-scheduling.
Process can only de-schedule at appropriate points, thus for low-priority queue, urgent events may not be serviced immediately.	Processes at low-priority queues can de-schedule at any point, thus urgent events could be serviced immediately.
Process has limited registers (evaluation stack) to use just 3 registers, thus every operation must involve loading and storing of registers, thus slower execution of a process.	Process has 13 registers to use (excluding stack pointer and program counter), thus a greater degree of flexibility. This implies faster execution for a process.
Processes queuing in high-level priority are not guaranteed to run within a certain period of time, since time-slicing is not enforced for high-priority queue.	Same as the transputer.
Algorithm not complicated, as during interrupt's context switching, all registers are temporarily saved into a stack and the high-priority process is run immediately. High-priority process are non-interruptible, and upon completion, the saved registers of lower priority process are restored.	Algorithm involves higher overhead, since there are more priority queues.
Effectively only 1 process priority is available to run conventional processes	Number of priority queues can be easily customised.
Does not guarantee a fixed time slice (between 1 to 2 time slice)	Guarantees a fixed time slice
If a time-consuming instruction which are atomic are encountered, e.g. shifting of bits in registers (up to 85 seconds), it must be completed before a process could be interrupted.	Immediate interruption guaranteed
Does not allow flexibility, e.g. the need of prioritisation of different interrupts to handle different events.	Allows prioritisation of different interrupts to handle different events e.g. irq, fiq and swi, therefore more flexibility.

Architectural differences:

Transputer	***StrongARM***
Operation uses only 4 registers, i.e. Areg, Breg, Creg and Oreg. These are called evaluation stack. Hence there's a limit of pushing into reg.	Every operation uses a variety of registers
Local data are stored into workspace, and the compiler could generate separate code and data area. Thus code could be used by different processes, where every process has its own data area.	Compiler can't generate separate code and data areas, i.e. these should come together
Stack is implemented by workspace, where the structure is like a downward growing stack. This is pointed at by Wreg, or workspace pointer	The conventional stack structure is applied here
Each process has its own workspace, so the processor swaps Wreg whenever it switches between processes.	Each process has individual stack identified by stack pointer.

Appendix B

The following results are obtained from Peter Welch, University of Kent. The comments too are quoted from his posting.

KROC version 1.0 targeting sparc-sun-solaris2.5.1 (driver V1.36) Ultra Sparc II - 400 MHz. (Many users)	
cycle time (SEQ delta)	850 nanoseconds (+/- 20)
cycle time (PAR delta)	875 nanoseconds (+/- 20)
context switch time	107 nanoseconds (+/- 2)
startup/shutdown	(too noisy to tell)

KROC version 1.2.3a targeting i386-pc-linux (driver V1.35) Pentium III - 500 Mhz. (Single user)	
cycle time (SEQ delta)	1272 nanoseconds (+/- 1)
cycle time (PAR delta)	1472 nanoseconds (+/- 1)
context switch time	159 nanoseconds (+/- 0)
startup/shutdown	200 nanoseconds (+/- 2)

The following is for the upcoming Linux, release that has a more efficient interface between user code and the occam kernel:

KROC version 1.3.0beta targeting i386-pc-linux (NOT RELEASED YET) Pentium III - 500 Mhz. (Single user)	
Cycle time (SEQ delta)	724 nanoseconds (+/- 1)
Cycle time (PAR delta)	868 nanoseconds (+/- 1)
Context switch time	91 nanoseconds (+/- 0)
Startup/shutdown	144 nanoseconds (+/- 2)

KROC version 1.3.0beta targeting i386-pc-linux (NOT RELEASED YET) Compiled with a new in-lining flag (==> slightly larger executables) Pentium III - 500 Mhz. (Single user)	
Cycle time (SEQ delta)	450 nanoseconds (+/- 1)
Cycle time (PAR delta)	557 nanoseconds (+/- 1)
Context switch time	56 nanoseconds (+/- 0)
Startup/shutdown	107 nanoseconds (+/- 2)

For comparison, here are the JCSP results (the CommsTime.Java code is one of the jcsp-demos in the release). Please note that the timings are downgraded to *microseconds* (not *nanoseconds*). JCSP channels are currently implemented on top of the standard Java threads model -which uses native OS threads (which have high overheads). This accounts for the 2000-fold (roughly) higher overheads. These JCSP timimgs were also conducted on a slower machine - which accounts for the further factor of 2.

JCSP version 1.0-rc2 (running under Sun's JDK1.2.2) Pentium II - 266 Mhz.	
cycle time (SEQ delta)	212 microseconds (+/- 2)
cycle time (PAR delta)	226 microseconds (+/- 2)
context switch time	27 microseconds (+/- 0)
startup/shutdown	24 microseconds (+/- 4)

Further comments: It's not that the Java version is slow - it's the occam kernel that's so quick! Work is progressing on building a special JVM that includes the basics of the occam kernel to support directly the JCSP primitives. This was presented by Jim Moores at the recent WoTUG-23 conference and is in the proceedings.

Appendix C

Results for global messages transmission:

Data cache off, instruction cache on.

Set 1	Set 2	Set 3
0000000151	0000000152	0000000152
0000000122	0000000122	0000000122
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000122
0000000122	0000000122	0000000122
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000122
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000122	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000121	0000000121	0000000121
0000000122	0000000122	0000000121
0000000121	0000000121	0000000121

Average : 122.717

Both data and instruction cache on.

Set 1	Set 2	Set 3
0000000050	0000000050	0000000050
0000000010	0000000009	0000000009
0000000009	0000000009	0000000009
0000000009	0000000010	0000000010
0000000009	0000000010	0000000010
0000000009	0000000010	0000000010
0000000009	0000000010	0000000009
0000000009	0000000010	0000000010
0000000009	0000000010	0000000010
0000000009	0000000010	0000000010
0000000009	0000000010	0000000010
0000000009	0000000009	0000000010
0000000009	0000000009	0000000009
0000000009	0000000009	0000000009
0000000009	0000000009	0000000009
0000000009	0000000009	0000000009
0000000010	0000000009	0000000009
0000000010	0000000009	0000000009
0000000009	0000000009	0000000009
0000000009	0000000009	0000000009

Average : 11.367.

Results for receiving buffered global messages.

When data cache is off, instruction cache is on.

Set 1	Set 2	Set 3
0000000054	0000000054	0000000054
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000045
0000000044	0000000045	0000000044
0000000045	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000045
0000000044	0000000045	0000000044
0000000044	0000000045	0000000044
0000000044	0000000045	0000000045
0000000044	0000000045	0000000044
0000000045	0000000045	0000000044
0000000045	0000000045	0000000045

Average : 44.933

When both data and instruction caches are on.

Set 1	Set 2	Set 3
0000000014	0000000015	0000000015
0000000002	0000000003	0000000003
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002
0000000003	0000000002	0000000002

Average:2.967

Results for receiving un-buffered global messages.

When data cache is off, but instruction cache is on.

Set 1	Set 2	Set 3
0000000157	0000000157	0000000157
0000000146	0000000146	0000000146
0000000146	0000000146	0000000147
0000000147	0000000146	0000000147
0000000148	0000000147	0000000148
0000000146	0000000147	0000000147
0000000146	0000000147	0000000147
0000000146	0000000146	0000000146
0000000147	0000000146	0000000146
0000000148	0000000147	0000000147
0000000147	0000000146	0000000146
0000000146	0000000147	0000000146
0000000146	0000000146	0000000147
0000000146	0000000146	0000000147
0000000119	0000000119	0000000120
0000000045	0000000045	0000000045
0000000045	0000000045	0000000044
0000000045	0000000044	0000000045
0000000045	0000000045	0000000045
0000000046	0000000046	0000000045

Average : 120.317

When both instruction and data caches are on.

Set 1	Set 2	Set 3
0000000025	0000000026	0000000026
0000000006	0000000006	0000000007
0000000008	0000000007	0000000006
0000000006	0000000006	0000000005
0000000005	0000000006	0000000008
0000000010	0000000010	0000000010
0000000006	0000000006	0000000006
0000000005	0000000006	0000000005
0000000006	0000000005	0000000006
0000000005	0000000006	0000000005
0000000006	0000000005	0000000006
0000000006	0000000005	0000000005
0000000005	0000000006	0000000006
0000000006	0000000005	0000000005
0000000005	0000000006	0000000005
0000000003	0000000002	0000000003
0000000003	0000000003	0000000002
0000000002	0000000003	0000000002
0000000002	0000000003	0000000003
0000000003	0000000002	0000000003

Average : 6.183

Appendix D

Results for DMA effect on CPU performance.

At 10Mbits/s.

With Data cache and Instruction cache on.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	1,914,420	6,874	1,926,104	13,832
	1,914,422	6,874	1,921,891	13,802
	1,914,421	6,874	1,921,909	13,802
	1,914,422	6,874	1,921,910	13,802
	1,914,422	6,874	1,921,891	13,802
Average	1,914,421	6,874	1,922,741	13,808

At 10Mbits/s.

With Data cache and Instruction cache on.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	1,901,503	0	1,940,363	21,760
	1,901,502	0	1,940,396	21,760
	1,901,503	0	1,940,302	21,758
	1,901,503	0	1,940,320	21,758
	1,901,503	0	1,940,302	21,758
Average	1,901,503	0	1,940,337	21,758

At 10Mbits/s.

With Data cache off and Instruction Cache on.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	26,978,647	94,582	28,321,530	199,053
	26,978,607	94,582	28,321,506	199,053
	26,978,542	94,582	28,321,467	199,053
	26,978,615	94,582	28,321,643	199,055
	26,978,641	94,582	28,321,501	199,053
Average	26,978,610	94,852	28,321,529	199,053

At 10Mbits/s.

With Data cache off and Instruction cache on.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	25,688,918	0	29,750,892	324,603
	25,688,917	0	29,750,859	324,598
	25,688,915	0	29,750,422	324,590
	25,688,905	0	29,750,924	324,599
	25,688,887	0	29,750,314	324,580
Average	25,688,908	0	29,750,682	324,594

At 10Mbits/s.

With Data cache off and Instruction Cache off.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	98,725,419	347,259	107,117,907	751,258
	98,725,476	347,260	107,117,681	751,256
	98,725,278	347,259	107,117,966	751,256
	98,725,384	347,258	107,117,699	751,256
	98,724,650	347,256	107,117,964	751,256
Average	98,725,241	347,258	107,117,843	751,256

At 10Mbits/s.

With Data cache off and Instruction cache off.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	91,678,996	0	117,621,978	1,284,561
	91,677,672	0	117,608,229	1,283,794
	91,676,596	0	117,634,001	1,284,508
	91,675,939	0	117,627,379	1,284,485
	91,675,604	0	117,608,078	1,283,790
Average	91,676,961	0	117,619,933	1,284,227

At 20Mbits/s.

With Data cache and Instruction cache on.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	1,919,036	12,606	1,940,962	25,498
	1,919,032	12,606	1,941,046	25,500
	1,919,035	12,606	1,941,081	25,500
	1,919,040	12,606	1,941,145	25,500
	1,919,037	12,605	1,940,618	25,488
Average	1,919,036	12,605	1,940,970	25,497

At 20Mbits/s.

With Data cache and Instruction cache on.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	1,901,501	0	1,968,396	37,683
	1,901,501	0	1,966,183	37,642
	1,901,502	0	1,967,531	37,668
	1,901,502	0	1,968,225	37,683
	1,901,502	0	1,966,226	37,640
Average	1,901,502	0	1,967,312	37,663

At 20Mbits/s.

With Data cache off and Instruction Cache on.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	28,093,909	178,715	30,942,591	393,263
	28,093,956	178,714	30,941,024	393,215
	28,093,985	178,718	30,942,686	393,267
	28,094,007	178,707	30,941,508	393,228
	28,093,843	178,719	30,943,005	393,253
Average	28,093,940	178,714	30,942,163	393,245

At 20Mbits/s.

With Data cache off and Instruction cache on.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	25,688,858	0	33,311,633	616,558
	25,688,872	0	33,311,988	616,556
	25,688,970	0	33,310,005	616,419
	25,688,843	0	33,310,244	616,433
	25,688,865	0	33,310,370	616,446
Average	25,688,882	0	33,310,848	616,482

At 20Mbits/s.

With Data cache off and Instruction Cache off.	1 Engine	1 Engine	2 Engines	2 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	105,007,897	658,428	122,920,765	1,528,392
	105,007,796	658,425	122,919,104	1,528,344
	105,007,718	658,416	122,917,861	1,528,236
	105,007,584	658,398	122,917,407	1,528,192
	105,007,493	658,383	122,918,625	1,528,308
Average	105,007,698	658,410	122,918,752	1,528,294

At 20Mbits/s.

With Data cache off and Instruction cache off.	No DMA in progress	No DMA in progress	4 Engines	4 Engines
	Dhrystone time	Number of packets transmitted	Dhrystone time	Number of packets transmitted
	91,674,940	0	145,288,921	2,662,129
	91,675,074	0	145,281,608	2,661,757
	91,674,448	0	145,280,930	2,661,745
	91,674,848	0	145,252,437	2,661,213
	91,674,060	0	145,275,943	2,661,507
Average	91,674,674	0	145,275,968	2,661,670

Appendix E

Headers for 3L libraries which have been ported.

File name : Thread.h

```
char *thread_create(void (*fn)(),word stack_bytes,word arg_num,...);
int thread_priority(void);
void thread_start(void (*fn)(),char *ws,int wssize, int flags, int nargs,...);
void thread_wait(void);
```

File name : Par.h

```
int par_printf(char *fmt,...);
void *par_malloc(word num_bytes);
void par_free(void *mem_ptr);
```

File name : Chan.h

```
void chan_init(word pointer);
chan_in_byte(space_ptr,chan_ptr);
chan_in_word(space_ptr,chan_ptr);
chan_in_message(length,space_ptr,chan_ptr);
void chan_out_message(word length,char *space_ptr,struct Chan_Item *chan_ptr);
chan_out_message(len,ptr,chan) ;
void chan_out_byte(word msg,struct Chan_Item *chan_ptr);
void chan_out_word(word msg,struct Chan_Item *chan_ptr);
int chan_out_byte_t(word message,struct Chan_Item *chan_ptr,word time_quantum);
int chan_out_word_t(word message,struct Chan_Item *chan_ptr,word time_quantum);
int chan_out_message_t(word length,word space_ptr,struct Chan_Item *chan_ptr,word time_quantum);
int ChanInTimeFail(void *space_ptr,struct Chan_Item *chan_ptr,word length,word time_quantum);
chan_in_byte_t(space_ptr,chan_ptr,timeout);
chan_in_word_t(space_ptr,chan_ptr,timeout);
chan_in_message_t(length,space_ptr,chan_ptr,timeout);
LinkOut(buffer_ptr,length,chan_ptr);
LinkIn(buffer_ptr,length,chan_ptr);
```

File name : Timer.h

```
int timer_after(word time1,word time2);
word timer_now(void);
void timer_wait(unsigned long int timer_value);
void timer_delay(unsigned long int time_ticks);
```

File name : Sema.h

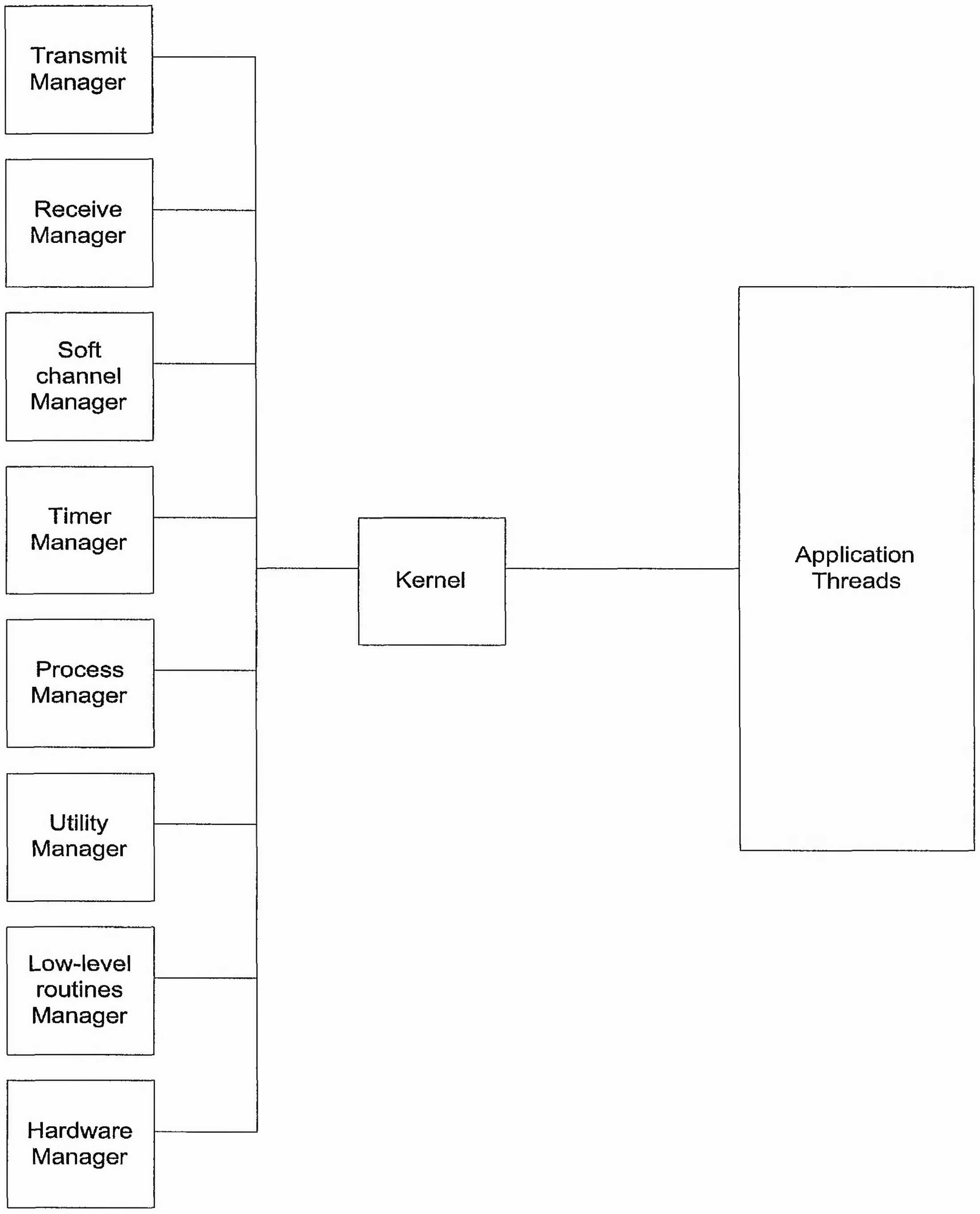
```
void sema_signal(SEMA *sema_ptr);
void sema_wait(SEMA *sema_ptr);
int sema_test_wait(SEMA *sema_ptr);
sema_wait_n(register SEMA *sema_ptr,register int n);
sema_signal_n(register SEMA *sema_ptr,register int n);
sema_init(SEMA *sema_ptr,int sema_value);
```

File name : Alt.h

```
int alt_wait_vec(int chan_num,CHAN *channels[]);
int alt_nowait_vec(int total_chan,CHAN *SOFT_Chan_ptr[]);
alt_wait(word total_chan,...);
alt_nowait(word total_chan,...);
```

Appendix F

Kenux internal structure.



Transmit manager and receive manager are for managing external channel communications. Soft channel manager handles local channel communication.