PHD

Implementation of distributed software environments for dynamic power system security assessment

Grzejewski, Jerzy Marek

Award date:
1995

Awarding institution:
University of Bath

Link to publication
IMPLEMENTATION OF DISTRIBUTED SOFTWARE ENVIRONMENTS FOR DYNAMIC POWER SYSTEM SECURITY ASSESSMENT

Submitted by Jerzy Marek Grzejewski
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
at the University of Bath
1995

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and no information derived from it may be published without the prior written consent of the author. This thesis may be made available for consultation within the University library and may be photocopied or lent to other libraries for the purposes of consultation.

Bath, October 1995
Acknowledgements

his research was only made possible by the guidance and encouragement of my supervisor, Mr A.R. Daniels, whose helpful comments were vital in forming a broader appreciation of the subject matter.

My parents and family have been a source of constant inspiration to me and have provided moral and financial support which was essential in the completion of this project and for which I shall always remain grateful.

I particularly want to thank Dr R.W. Dunn for his invaluable discussions and useful comments, as well as Dr T. Berry, Mr B. Ross and Mr V. Gott, who provided useful technical input. I also would like to thank the University of Bath and, in particular, Prof J.F. Eastham and Prof A.T. Johns for their permission to study at the School of Electronic and Electrical Engineering and for the provision of the facilities.

The research described here was assisted by the cooperation of Perihelion Software Ltd and Microway Inc. Particular thanks are due to Paul Beskeen, Dr T. King, Mark Barrenechea and Steven Fried. A further debt of gratitude is owed to Chris Selwyn and Shan Atukarala for their initial work, upon which this research is based.

This thesis would have never been completed but for the help and encouragement of James Hodgson, Dr K.W. Chan, Keith Bell and Anthony Edwards, who have proof-read its initial drafts. Finally, I would like to gratefully acknowledge my colleagues, including Chris Groom, Dave Fitton, Jim Barratt, Paul Stallard, Alex Carter, Stuart Websper, Alex Chiwaya, Umesh Patel, Grant Hainsworth and others, who have helped to provide a productive and exciting environment, without which this work would surely have been less enjoyable.
Preface

This short page will hopefully ease the reading of the remainder of this document, by explaining the notation used therein and by outlining the overall structure of the other chapters.

Perhaps due to its relatively young age and rapid evolution, the use of jargon is particularly wide-spread in computing. To aid the understanding of this report, the author has compiled an index, which may be found at the back and which should contain references to most specialist terminology used. To help with locating of the definition on each page, a different font is used for the indexed terms.

In addition to the main bibliography, this work contains a special bibliography listing relatively inaccessible technical reports. References to this bibliography are shown like this: [♦123]. Some of the references listed there are included in appendix C.

While most of the chapters of this work are independent entities, in some places knowledge of previous text is assumed to avoid unnecessary repetition. These assumptions are displayed below in graphical form, to aid the reader with viewing a fragment of this report.

*Chapter dependencies. A $\rightarrow$ B means should read A before B.*
3.4.1 Helios Namespace .................................................. 52
3.4.2 General Server Protocol ............................................. 56
3.4.3 Fault resistance .......................................................... 57
3.4.4 Security ................................................................. 58
3.4.5 Other Microkernel Operating Systems ......................... 60
3.4.6 System structure ........................................................ 62
3.4.7 Portability .............................................................. 69

4 The porting of Helios to the i860 ................................. 70
4.1 Writing and testing the executive ..................................... 70
4.1.1 Scheduling ............................................................. 71
4.1.2 Link I/O ................................................................. 73
4.1.3 Trap Handler .......................................................... 74
4.1.4 Ensuring correctness .................................................. 76
4.1.5 Configuration management ......................................... 78
4.1.6 Solutions to functional deficiencies ............................ 80
4.1.7 Debugging the executive ............................................. 81
4.2 Altering the IO server .................................................... 84
4.3 Extending the functionality of the kernel ......................... 86
4.4 The rest of the system .................................................... 87
4.5 Updating Helios to version 1.3 ....................................... 88
4.6 Adding support for an efficient floating-point compiler ........ 90
4.6.1 New GHOF patch structure ....................................... 91
4.6.2 Compiler conversion ................................................. 93
4.6.3 Microway Libraries ................................................ 94
4.7 Use of OASIS under Helios ............................................ 95

5 The Parallel Virtual Machine Environment .................... 105
5.1 Introduction .............................................................. 105
5.1.1 The p4 macros ....................................................... 107
5.1.2 PARMACS ............................................................ 108
5.1.3 PVM ................................................................. 108
5.1.4 Linda ................................................................. 109
5.1.5 Other systems ........................................................ 109
5.2 The PVM model .......................................................... 110
5.3 Implementation of PVM ............................................... 111
5.3.1 PVM library functions ............................................. 112
5.3.2 Group Extensions ................................................. 115
5.3.3 Multiprocessor machine support ............................... 115
5.4 Future developments ................................................... 116

6 Implementing PVM support for the i860 ......................... 119
6.1 Introduction .............................................................. 119
6.2 Porting Microway run-time environment to Linux ............. 120
6.2.1 OS860 link protocol ................................................. 122
Contents

6.3 Linux device driver ................................................................. 124
  6.3.1 The Numbersmasher driver ............................................... 127
  6.3.2 Tuning the driver ............................................................. 129
6.4 Implementing multi-processor support for the i860 ................ 131
  6.4.1 Modifications to PVM ...................................................... 133
  6.4.2 Intermediate communication facility – pvm860 ............... 136

7 Dynamic Security Assessment in Power Systems .................. 140
  7.1 Power System Stability ....................................................... 140
    7.1.1 Transient Stability ...................................................... 142
    7.1.2 Dynamic Stability ...................................................... 142
  7.2 OASIS ................................................................................. 143
    7.2.1 Contingency Screening ................................................ 143
    7.2.2 Contingency Evaluation ............................................... 144
    7.2.3 Contingency Ranking .................................................. 144
    7.2.4 Implementation .......................................................... 145
  7.3 Modifications for this project ............................................. 146

8 Results and Analysis ............................................................. 150
  8.1 Comparison of compiler performance ................................ 150
  8.2 Communication performance .............................................. 152
  8.3 Null kernel call .................................................................. 154
  8.4 OASIS performance ............................................................ 154
  8.5 Other results ...................................................................... 157

9 Conclusions and Further Work ............................................. 163
  9.1 Conclusions ...................................................................... 163
  9.2 Further Work .................................................................... 165
    9.2.1 Enhancing Helios ......................................................... 165
    9.2.2 Improving compiling ................................................... 166
    9.2.3 Improving PVM .......................................................... 167

A Linux device driver for the Numbersmasher card ................. 169
  A.1 Debugging Support ........................................................... 173

B Changes to Microway C compiler output .............................. 175
  B.1 Document structure .......................................................... 175
  B.2 GHOF principles .............................................................. 175
    B.2.1 Object structure .......................................................... 175
  B.3 Changes to assembler format ............................................ 177
    B.3.1 Module header ........................................................... 178
    B.3.2 Register usage ........................................................... 178
    B.3.3 Functions .................................................................. 179
    B.3.4 Initialisation routine ................................................... 181
    B.3.5 Declaring variables .................................................... 182
    B.3.6 Accessing variables .................................................... 184

vi
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Relationship between power and cost for computers.</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Electric Power System.</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Architecture of the i860 (data paths only).</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Operation of processor components without pipelining.</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Operation of processor components with pipelining.</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Delay slot instruction in the i860.</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Comparison of vector and pipelined processors for vector operations.</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Virtual to physical address translation mechanism on the i860.</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>The i860 Translation Look-aside Buffer.</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>Numbersmasher board diagram.</td>
<td>28</td>
</tr>
<tr>
<td>2.9</td>
<td>Numbersmasher memory map.</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Device driver structure.</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Time and space multiplexing.</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>A system call path in different kernels.</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>A typical Helios name tree.</td>
<td>54</td>
</tr>
<tr>
<td>3.5</td>
<td>The structure of Helios nucleus.</td>
<td>64</td>
</tr>
<tr>
<td>3.6</td>
<td>Helios module table mechanism.</td>
<td>67</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of assembler and generated GHOF.</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>Avoiding a possible race condition in I/O code.</td>
<td>72</td>
</tr>
<tr>
<td>4.2</td>
<td>Valid task states in Helios/860.</td>
<td>73</td>
</tr>
<tr>
<td>4.3</td>
<td>Structure of a part of the executive using the folding editor.</td>
<td>77</td>
</tr>
<tr>
<td>4.4</td>
<td>Hardware problems handling assembler preprocessor file.</td>
<td>97</td>
</tr>
<tr>
<td>4.5</td>
<td>Structure descriptors in the debugger.</td>
<td>98</td>
</tr>
<tr>
<td>4.6</td>
<td>Bit field descriptors in the debugger.</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Class inheritances in the i860 debugger.</td>
<td>99</td>
</tr>
<tr>
<td>4.8</td>
<td>Debugging a remote i860 card.</td>
<td>99</td>
</tr>
<tr>
<td>4.9</td>
<td>Boot time memory map for Helios/860.</td>
<td>100</td>
</tr>
<tr>
<td>4.10</td>
<td>The MMU memory mapping under Helios/860.</td>
<td>101</td>
</tr>
<tr>
<td>4.11</td>
<td>i860 instruction specification for the generic Helios assembler.</td>
<td>102</td>
</tr>
<tr>
<td>4.12</td>
<td>The PVM emulation under Helios/860.</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>Communication speed trends over the last twenty years.</td>
<td>106</td>
</tr>
<tr>
<td>5.2</td>
<td>The PVM architecture.</td>
<td>111</td>
</tr>
<tr>
<td>5.3</td>
<td>The PVM implementation.</td>
<td>112</td>
</tr>
</tbody>
</table>
5.4 An example of PVM message-passing code. .................................. 114
6.1 Microway run-time environment. ............................................... 121
6.2 A finite state machine representation of OS860 communication
   protocol. .................................................................................... 123
6.3 Scatterplot matrix of device driver performance for a sample
   program. .................................................................................. 130
6.4 Multi-processor machine model for PVM. ................................. 132
6.5 The communications within PVM/860. ................................. 133
6.6 Overview of the implementation of PVM/860. .......................... 139
7.1 Transient instability. ................................................................. 148
7.2 Dynamic instability. ................................................................. 149
7.3 Client-server design of OASIS. ............................................... 149
8.1 Dhrystone performance results under Helios/860. ..................... 151
8.2 Transmission times for messages under Helios. ........................ 152
8.3 Timings for PVM variants. ....................................................... 153
8.4 Oasis time of execution for m4b6 study. ................................. 159
8.5 Oasis speed of execution for m4b6 study. ............................. 160
8.6 Oasis time of execution for m20b100 study. ............................ 161
8.7 Oasis speed of execution for m20b100 study. .......................... 162
List of Tables

4.1 Transputer boot protocol .......................................................... 99
4.2 Boot protocol extensions .......................................................... 100
4.3 Original Helios/860 patches ..................................................... 100
4.4 Modified Helios/860 patches ...................................................... 103
5.1 General purpose PVM functions ............................................... 117
5.2 PVM message passing functions .............................................. 118
6.1 An example of OS860 protocol escapes. ................................. 138
6.2 Extensions to pvmd <-> pvm860 ............................................... 138
Abbreviations

AFS  Andrew Filing System  see page 57
AMPP Assembler Macro PreProcessor  see page 70
ANDF Architecture Neutral Distribution Format  see page 49
ART Advanced Real-time Technology  see page 237
ATM Asynchronous Transfer Mode  see page 238
BSD Berkeley Software Distribution  see pages 51, 69
CHAOS Concurrent Hierarchical Adaptable Object System  see page 237
CISC Complex Instruction Set Computer  see pages 5, 14
COFF Common Object File Format  see page 93
CORBA Common Object Request Broker Architecture  see page 52
CTRON Central part of The Real Time Nucleus  see page 236
DCE Distributed Computing Environment  see pages 51, 233
DES Data Encryption Standard  see pages 59, 234, 235
DIM Dual Instruction Mode  see page 21
DRA Defence Research Agency  see page 49
DSP Digital Signal Processor  see page 49
ELF Executable and Linking Format  see page 93
FDDI Fiber Distributed Data Interface  see page 106
FSF Free Software Foundation  see page 61
GHOF Generic Helios Object Format  see page 68
GSP General Server Protocol  see page 56
HARTS Hexagonal Architecture for Real-Time Systems  see page 237
IPC Inter-Process Communication  see page 132
IRQ Interrupt ReQuest  see page 129
MAC Mandatory Access Control  see pages 234, 234
MARS MAltainable Real-Time System  see page 237
MIMD Multiple Instruction-stream Multiple Data-stream  see page 20
<table>
<thead>
<tr>
<th>Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMU</td>
</tr>
<tr>
<td>MPI</td>
</tr>
<tr>
<td>MPP</td>
</tr>
<tr>
<td>NFS</td>
</tr>
<tr>
<td>NGC</td>
</tr>
<tr>
<td>OASIS</td>
</tr>
<tr>
<td>OMG</td>
</tr>
<tr>
<td>OSF</td>
</tr>
<tr>
<td>PARMACS</td>
</tr>
<tr>
<td>PCP</td>
</tr>
<tr>
<td>PERL</td>
</tr>
<tr>
<td>PLD</td>
</tr>
<tr>
<td>RISC</td>
</tr>
<tr>
<td>RPC</td>
</tr>
<tr>
<td>RR</td>
</tr>
<tr>
<td>RSA</td>
</tr>
<tr>
<td>RTPSS</td>
</tr>
<tr>
<td>SHARC</td>
</tr>
<tr>
<td>SIMD</td>
</tr>
<tr>
<td>SOM</td>
</tr>
<tr>
<td>SONET</td>
</tr>
<tr>
<td>SPMD</td>
</tr>
<tr>
<td>TCP</td>
</tr>
<tr>
<td>TDF</td>
</tr>
<tr>
<td>TLB</td>
</tr>
<tr>
<td>UDP</td>
</tr>
<tr>
<td>VLIW</td>
</tr>
<tr>
<td>XDR</td>
</tr>
<tr>
<td>YACC</td>
</tr>
<tr>
<td>mid</td>
</tr>
<tr>
<td>msgtag</td>
</tr>
<tr>
<td>tid</td>
</tr>
</tbody>
</table>
Chapter One

Introduction

Since the dawn of time human civilisation has progressed by developing tools to extend its command over the environment. These tools magnified man's physical attributes and allowed humans to develop far faster than other life forms thereby achieving mankind's supremacy on earth. As civilisation evolved so did the tools. What started with primitive bone-and-wood spears of the Australopithecus two million years ago, has now been developed into such intricate and sophisticated mechanisms as the space shuttle.

With time, man began to reach the limits of his intellectual as well as physical capability. Human short-term memory can deal with relatively simple tasks and therefore holds only around seven concepts. Long-term memory is often slow and imperfect. Mental processes were affected by emotions and temporary physiological conditions.

To maintain further progress, tools had to be invented to extend man's mental capabilities. Early civilisations, including the Sumerians, Egyptians, Chinese, Hindus and Greeks living between 3000 and 1500 B.C., all developed such tools including writing, formal thinking methodologies (notably philosophy and mathematics) and calculation aids. Numerous practical applications of mathematics were soon discovered and its notation and technique expanded.

Numerical calculations became increasingly important in evolving technical and scientific societies. The abacus, in its various forms, was widely used by
Introduction

ancient Egyptians, Greeks and Romans, helping them achieve many feats of engineering and architecture. The Hindu civilisation revolutionised calculation by introducing the concept of zero and positional coding of numbers. In the middle ages mechanically-minded Europeans became fascinated by the concept of an automatic calculator. Wilhelm Schickard built the first 6-digit mechanical adding machine in 1623. A flurry of developments followed, including the first four-function calculator built in 1674 by German mathematician and philosopher Gottfried Liebniz.

While all these tools were useful, they lacked the power available to closed-loop systems — they could only perform pre-programmed actions. In 1834 however, a British mathematician, Charles Babbage, proposed his analytical engine which was intended for autonomous operation. Sadly, the construction of the machine required mechanical precision which could not be achieved in the 19th century.

The discovery of electricity and its development by the likes of Charles Coulomb, Alessandro Volta, Hans Oersted, André Ampère, Michael Faraday, Friedrich Gauss and Thomas Edison made possible the creation of many useful tools. It also removed the last practical obstacles on the path to building automatic calculation machines.

Initially, various analog computers were built to aid analysis, simulation and process control [1, 2]. Despite their speed and simplicity they suffered from mediocre accuracy, which was limited by the precision of the constituent electrical components to a maximum of 0.1%. Between the years 1938 and 1941 Konrad Zuse constructs the first programmable calculators named Z1, Z2 and Z3. This was followed by developments at Harvard, construction of ENIAC and finally, in 1948, the completion of Manchester Mark I, the first stored-program computer. Over the last fifty years, digital computers have gained complete supremacy, initially in combination with analog parts in hybrid computers and

1 Although his design was lost until 1956 and Blaise Pascal was credited with the construction of the first adder in 1644.
One

Introduction

later on their own.

Digital computer technology has itself undergone numerous changes. Zuse's machines used electromechanical relays to perform its operations. Four generation changes followed changing relays for valves then transistors, integrated circuits and large-scale integrated circuits resulting in the fourth-generation machines predominant today. Progress has not stopped, although the talk of fifth-generation technology seems to concentrate more on software than hardware advances.

In fact, since the creation of the digital computer fifty years ago, the pace of technological change has not really slowed down. Currently processing speed of computer hardware increases by an order of magnitude every five years. Modern processors can perform basic arithmetic operations 1000 million times faster than Mark I, yet take up less space than one of its storage tubes and use a fraction of the electric power.

Following the description of the vast advances made, it would be natural to ask where these increases in power are being used. After all, the rate of progress in computing outstrips practically all other areas of human involvement. It could therefore be argued that further advancement is unnecessary and is driven solely by public expectation. This would be far from the truth.

Computing is being applied to many new applications every year. Many of these applications require vast quantities of processing power. Some of the problems tackled are inherently difficult; artificial intelligence, for example, although initially developed in the 1960s, is only now finding sufficient processing power to be applied to practical problems.

Even the traditional applications are placing increasing demands on systems. For example, the meteoric rise in the popularity of the computer has forced very rapid advancement in the area of human-computer interaction evolving from a text-only command line interface to a graphical direct-manipulation one. Also,

\(^2\)Addition or subtraction of the 23 digit numbers took the Mark I 0.3 s.
many problems which do not lend themselves to efficient solutions are being tackled by more brute-force approaches. A particularly important group of these includes NP-complete problems, many of which have direct practical applications [3].

1.1 Methods of achieving improved performance

Having established the genuine requirements for more processing power, we are left with choosing a method of meeting these requirements. This section outlines several such methods.

The most obvious and, at least until now, by far the most effective way of extending the available processing power is to enhance the hardware. It is past successes in this field that are responsible for the bulk of the 1000 million speed increase since 1940s described in section 1.

Probably the most obvious way of increasing the speed of operation of synchronous circuits is to increase the clock speed. This approach has paid handsome dividends in the past with a 300-fold increase over the last 15 years\(^3\). No doubt further increases in clock speeds are forthcoming, although some technological barriers will have to be overcome.

The principal limit to the clock speed which can be used with a digital circuit is the gate propagation delay. Faster digital technologies have been developed which have lower gate delays. A reduction in gate delay from 50 ns to 1 ns has been achieved through successive advances in logic technologies from the primitive resistor-transistor logic (RTL) and diode-transistor logic (DTL), through currently-predominant transistor-transistor logic (TTL) and complementary metal oxide-silicon (CMOS) to emitter coupled logic (ECL) and integrated-injection logic (I\(^2\)L).

\(^3\)From 1 MHz in the 1980s to 300 MHz in 1995.
Unfortunately, these advances have their own problems. Doubling the clock speed of a digital circuit doubles the amount of heat generated inside the transistors. With the ever smaller die sizes, it is easy for this heat to build up until the circuit is damaged. The die size is itself limited by physical considerations [4]. The faster technologies (ECL and I^2L) aggravate the problem by consuming more power and hence generating yet more heat. Some hope may lie in asynchronous technology (which does not use a clock) where promising advances have been made [5-7]. An example of this technology is the AMULET1, which is an asynchronous implementation of the ARM processor [8]. Other, more remote possibilities lie in superconducting processors [9].

In addition to much higher speeds, many of the recent advances in hardware have attempted to use the silicon available. Probably the most popular recent design strategy is that of Reduced Instruction Set Computers, (RISCs,) which contrasts with the old-style design known as Complex Instruction Set Computers. (CISCs.)

While debate on what exactly are the essential elements of RISC rages on, the key element of the RISC philosophy is more effective use of available chip space. CISC architectures use many transistors in providing baroque addressing modes and special-purpose registers, which, while useful to experts, are left untouched by compilers. Since an increasing proportion of software is programmed in high-level languages, it makes sense to remove some of these expensive extras which compilers never use.

Modern processors usually combine the RISC structure with techniques which use the transistors saved by it. Having reached the speed limits for single units, these methods usually work by allowing some limited parallel operation of multiple functional units.

^4A typical RISC processor combines an orthogonal instruction set (usually with most instructions executing in one clock cycle), a large amount of on-board memory in the form of registers and cache with a load-store architecture (or at least very few addressing modes).
Pipelining is one such method. It allows simultaneous operation of multiple stages of the processor which would normally remain idle. In a conventional architecture, the instruction decode unit, for instance, is only active for one out of five cycles, while in a pipelined processor this unit is utilised all the time.

An alternative method of obtaining simultaneous operation of processor parts is taken by superscalar and Very Long Instruction Word or VLIW\(^5\) processors. Here different functional units are provided and operated independently at the same time.

While hardware solutions dominate the commercial marketplace, they are expensive and hence are only cost-effective when widely used. Many investigations into the relationship between cost and power of computers [10] have shown some economies of scale which justified using large systems. Most of these studies were based on the so-called Grosch’s Law, which states that the cost of a computer system increases with the square root of its power. However, more recently [11–13] it has been shown that while within one family or class of computers economies of scale hold, across all classes extra power comes at a premium. This is illustrated in figure 1.1. Hence, for users who require performance far in excess of what the majority currently accept, two alternatives exist. Either they can pay the price of using state-of-the-art hardware or they can use parallel processing.

Parallel processing is only effective for software which is explicitly written with it in mind. Such software is rarely portable and in many cases will only achieve optimal performance on the machine it was originally written for. Research into practical and effective methods of writing parallel software is still continuing [14–16]. However, given the benefits of using many relatively cheap processors compared with one expensive one, this is the path that many practical designs choose.

\(^{5}\)so called because the instruction word (which effectively contains multiple instructions for the different functional units) is considerably longer than in conventional processors.
All parallel architectures share the common property of having multiple processors. However, there are many different approaches to providing the communication medium between these processors. Present-day parallel architectures may be divided into two groups: shared memory designs and distributed memory designs. Mixed designs which provide both mechanisms are also possible. For example, the custom Transputer parallel machine built at the University of Bath uses a combination of Transputer links and hierarchical shared memory [17].

In shared memory (or tightly-coupled) architectures, the communication between processors takes place via blocks of memory accessible to many of them. While this approach is most efficient, it requires care on the part of the hardware designer or the programmer to ensure that simultaneous access is not made to a single location. Also, a fault in one of the processors is usually more severe, since the processors can affect each other in a very direct way.
Distributed memory (or loosely-coupled) architectures provide each processor with its own memory and supply a different communication mechanism. This approach is often less efficient than shared memory but can allow greater scalability. Shared memory designs have a limited bandwidth, whereas distributed memory systems can ensure that the communication medium grows with the number of processors (for example by embedding the communication on the same chip as the processor as done by the Inmos Transputer or Texas Instruments TMS 320C40).

1.2 Electric Power Systems

Electricity is the most widely available and most convenient form of energy in our society. It is not surprising then that its uninterrupted supply is crucial to the correct functioning of almost every aspect of our lives. The universal use of electricity makes the system which generates and distributes it very large and complex. A national power system can have hundreds of power stations, each with several generators, thousands of nodes and lines in the transmission network and millions of unpredictable customers.

The path taken by electricity between its producer and consumers is shown in figure 1.2. The generators supply power to the transmission network which transports it near its destination at very high voltages (400 kV–132 kV). The power is then passed at lower voltages (22 kV–415 V) through the distribution network to the customers.

1.2.1 Application of computing to power systems

The continuous operation and maintenance of a power system is very demanding, due to the complexity of the network, the large distances separating the
producers and consumers and the unpredictability of demand. The application of powerful, computer-based techniques is therefore essential to the smooth operation of the system. Practically all power companies worldwide use computer systems in the control and supervision of their networks. Computer-operated equipment is increasingly finding application in on-line operation usually with added benefits. For example electronic line protection relays\(^6\) with a built-in computer running a real-time operating system are replacing mechanical ones.

One of the tasks which is made the hardest by the complexity and size of power networks is on-line operation. The many decisions which have to be taken daily during the running of the system are being currently taken by trained personnel (operators), whose knowledge is, out of necessity, approximate. In a real power system, no human can fully predict the effects of a particular action or occurrence. Hence the system has to be operated with a large margin of safety and therefore less efficiently.

The safety margins used in the operation of power systems are increasingly becoming limited by the dynamic rather than steady-state behaviour of the system. Stability analysis is a technique which can be used to determine these

\(^6\)Which detect a fault (for example lightning strike) on a line and disconnect it from the rest of the system.
dynamic stability limits. Simply put, stability analysis considers the possible events or contingencies which could occur in the future (for example lightning strikes and equipment failures) and evaluates their effect. The result is a list of contingencies with the most severe effects; it is presently then up to human operators to examine this list and adjust the safety margins as necessary.

Two types of stability analysis are distinguished: transient stability, which concerns itself with the direct effects of the contingency (i.e. effects which take place up to 2 s after the contingency) and dynamic stability which considers long-term effects which are caused by the internal system feedbacks disturbed by the contingency.

1.3 This project

This research project is concerned with developing a parallel processing platform which provides considerable processing power in a single PC-compatible computer at a relatively low cost. The project explores two approaches to creating such a platform and evaluates them using a power system stability assessment tool.

The processing hardware used in the project consists of computationally powerful nodes connected by a relatively low-performance communication channel. Such hardware is therefore well suited to parallisable problems which consists of a number of well decoupled tasks. Also, each of these tasks must require quite powerful computing resources. Therefore the parallelism supported must be relatively coarse grain.

The principal application which is used to demonstrate the practicality of this system is the On-line Algorithms for System Instability Studies (OASIS), which is a dynamic power system security assessment package described in more detail in chapter 7. The OASIS tool would particularly benefit from a
powerful yet inexpensive processing platform, since it could then be effectively applied to off-line uses like training and long-term planning.

The potential application areas of this platform include any parallel problem which has low communication requirements and high processing requirements. This includes any data-parallel programs as well as other algorithms requiring little communication. However, the time constraints on this work have not permitted an extensive benchmarking and investigation of alternative applications. Instead, artificial benchmark results are presented to allow the estimation of performance for other algorithms.
The Numbersmasher i860 accelerator board

This chapter describes some of the recent advances in computer hardware, many of which are implemented by the i860 processor. The architecture of the processor is outlined and the details of the design of the Numbersmasher i860 card are presented. The particular strengths and weaknesses of the i860 in general and the Numbersmasher board in particular are also discussed.

2.1 The int\textsuperscript{el} 80860 processor

The int\textsuperscript{el} 80860 is a RISC chip developed by int\textsuperscript{el} to capitalise on recent technological advances, which made it possible to pack the processing power of a supercomputer onto a single chip [18]. The 80860 is aimed at achieving top floating-point and integer performance and was, at the time of its introduction in 1989, the fastest commercial single-chip processor available. Sadly, the family has been abandoned by int\textsuperscript{el}, although significant user-base and support is still available [19].

The int\textsuperscript{el} 80860 (or i860 for short) is a 32/64 bit processor which uses 0.8 micron CMOS technology to pack \(2\frac{1}{2}\) million transistors onto a 9.5x16 mm die. The address bus width of 32 bits provides an ample physical address space while
the 64 bit data bus results in a very high memory bandwidth (400 Mbyte/s for the 50 MHz variant) [20].

The processor contains 32 integer and 32 floating-point registers, which are all 32-bits wide. The floating-point registers may be used in pairs or quadruples to store double and extended precision numbers respectively. All arithmetic instructions operate on three registers: two sources and a destination. The first integer register, r0, contains a unchangeable value of 0 and is used as the destination for operations which discard their result. It also reduces the complexity of the instruction set by allowing the construction of some common operations from more general instructions. For example, a move, clear and bit manipulation instructions present in other architectures are implemented using more general arithmetic and logical operations. The use of a hard-wired zero register is not new and was used in the Atlas computer developed at Manchester University in 1963 [21].

2.1.1 RISC

The i860 is designed in accordance with the principles of the Reduced Instruction Set Computer (RISC) design paradigm [22]. An overview of the architecture of the chip, including all main components and internal data paths may be seen in figure 2.1. The bandwidth of the cache and memory interfaces for an i860 XR operating at 40 MHz is also shown. In addition to the elements shown, the chip contains address generation logic (in particular including the Memory Management Unit, described in section 2) and control logic.

The original aim of the RISC architecture was to remove from the processor instruction set some of the complex commands and addressing modes, which were rarely used by most programs. This followed investigations which showed that even human programmers could not efficiently use some

1 The term RISC was coined by Agerwala [23].
complex instructions [24,25] present in the Complex Instruction Set Computers, (CISCs,) which dominated in the 1970s. Most compilers generated code which used an even smaller subset of the available instruction set [26, 27], despite the large body of on-going research into generating efficient CISC code [28]. Since the vast majority of programs are presently written in high-level languages and subsequently compiled [29], the result was a very inefficient use of silicon in processors. In fact, most modern CISC processors use over 50% of the chip area to implement the control unit. For instance, in the Motorola 68020 the control logic occupies 68% of the die area.

While the smaller control units of RISCs result in a considerable saving of the die area and the hard-wired logic is faster than microcode, the processors do suffer from increased CPU→memory traffic. Typical RISC program is at least 30% larger than an equivalent CISC program, due to being composed from simpler instructions [30]. Since the RISC instructions are also on average shorter than CISC ones, a typical RISC program may contain twice as many instructions as the CISC equivalent. The superior performance of the RISC is achieved due to its much shorter execution time for each instruction, which is of the order of five times shorter [31].
The Numbersmasher i860 accelerator board

The higher instruction processing rate results in substantially increased memory traffic, which soon becomes the bottleneck of many systems. RISC processors combat this problem by increasing the amount of on-chip state, multiplying registers and cache sizes. While this does help reduce the number of memory accesses, it does cause a problem in multitasking environments, where context switches are becoming increasingly complex and time consuming, due to larger size of on-chip state which needs to be swapped. These difficulties will be further discussed in section 2.

Of course pure RISC and CISC philosophies represent the extremes of architectural design. In practice, many high-performance chips fall somewhere between the two ideals and are much harder to classify [32]. In any case, emulation of CISC architectures on an underlying RISC implementation is becoming increasingly common [33, 34].

2.1.2 Pipelining

Pipelining is a technique which is commonly used in both RISC and CISC processors, to increase the efficiency of use of the circuits comprising the processor. In a traditional design, any component of the processor, for example a data fetch unit, will only be active for a fraction of the entire instruction processing time, since a typical instruction needs to be processed by several units. This is illustrated in figure 2.2.

Pipelining allows high utilisation of the component units, by permitting the operation of a unit to continue processing an instruction even while previous instructions are being processed by subsequent units. This is shown in figure 2.3.

As can be seen figure 2.3, pipelining allows for continuous operation of all component units, resulting in one instruction being processed every clock cycle. However, it can also be seen that pipelined operation has to be delayed under some circumstances. This is known as a pipeline stall. It occurs when an in-
The Numbersmasher i860 accelerator board

construction needs to be completely evaluated before any further instructions can be fetched. This occurs quite frequently in the case of conditional branches, where, until the branch condition is evaluated, the processor cannot decide on the further flow of execution. This is known as a control dependency.

In addition to control dependencies, pipeline stalls also occur in case of data dependencies. If a register, which is the target of an unfinished instruction is used as the source in the subsequent code, the pipeline has to be held until the value in the register is updated. The i860 uses a scheme called scoreboard, originally used in the CDC 6600 [35], where every target register is marked as unavailable on the “scoreboard” while the instruction which calculates it is being processed. Any instruction which tries to use a marked register is held up until its value becomes available.

The i860 makes extensive use of pipelines to improve its performance. The floating-point unit contains two pipelines, one for each of the adder and multiplier units. The adder pipeline has three stages, while the multiplier pipeline length varies between two and three, depending on the precision of the operation. In addition, the processor allows for pipelining of memory loads, using a
three stage pipeline.

To minimise the possibility of pipeline stalls, the i860 uses a software-oriented mechanism known as delayed branching. Rather than hold execution when a slow branch instruction is encountered, thereby wasting clock cycles while the branch is being processed, the chip continues executing the instruction stream following the branch. While a fixed number of instructions below the branch (in positions known as delay slots) are processed, further code is fetched from the target of the branch. The compiler (or assembly programmer) has the job of ensuring that some useful code is placed in the delay slots [36]. Often the simplest approach is to move the first instructions from the target of the branch to the delay slot.

The i860, like all modern RISCs which use delayed branching, has a single delay slot associated with some of its branch and loop instructions. Some architectures [37] require multiple delay slots, but since it is very difficult for the compiler to fill these with useful code in most cases, such processors tend to use other mechanisms to minimise branch delays. A diagram showing the

---

2 More exotic architectures [38] use a variable number of delay slots, which produce more compact code, but have no performance benefit.
execution of a delay branch with one delay slot is shown in figure 2.4.

Other modern processors use different schemes to avoid pipeline stalls [39–41]. This is because delay slots can cause problems with forward compatibility of the code. New implementations of the architecture often change the branch delays and hence the number of delay slots, forcing the users to recompile their code. Since the introduction of the i860, other solutions to the branch problems have become more popular. These include parallel execution of the branch (used in IBM's RS/6000 [42]), dynamic branch prediction using history buffers [43], used by the Am29000 [44,45] and the new Pentium [46] processor from Intel and speculative execution (applied to many most recent chips, including MIPS R10000 [47], Power PC 620 [48] and Motorola 88110 [37]).

This trend matches the current move away from software-based techniques and towards hardware based ones, which includes the shift in popularity from VLIW to superscalar architectures.

---

\[3\] This is known as branch folding and resulting in no penalty for unconditional branches.
2.1.3 Vector processing

A different method of obtaining maximum performance from processors is the use of vector processing. In many scientific calculations, the data processed consists of large structures of numbers, usually vectors or matrices. Many computers have in the past achieved high performance by providing vector registers and arithmetic units capable of processing them in parallel. The result is Single Instruction-stream Multiple Data-stream (SIMD) [49–51] parallelism within the CPU.

Although vector processing can, under the right conditions, provide a staggering increase in the processor performance, it can sometimes simply result in a waste of vast numbers of transistors. The i860 approach to vectorisation reflects its VLIW philosophy: no explicit vector register or ALU is provided, but provisions are made to ensure that the processing of vectors can utilise the cache as an effective set of vector registers. This is implemented by providing both caching and non-caching load instructions, allowing a clever programmer to control exactly what data is held in the cache [52]. While this solution does not quite match performance of vector processors (since the arithmetic operations can only be processed one at a time), it has greater flexibility, since the cache is used by serial code, unlike vector registers.

Flynn [53] investigates the relation between processing speeds of vector and pipelined processors. He finds, that from a theoretical standpoint the two are very closely related, as shown in figure 2.5. A vector processor with hardware vector length of $V$ exhibits step characteristics, since operations process entire vectors, whether they are full or not. The time taken to process $V$ elements is denoted $T$ in the graph. The performance of the pipelined processor, which has a $V$-stage pipeline and the same operating speed as the vector processor, is a straight line. The exact performance depends on whether the pipeline needs to
Two

The Numbersmasher i860 accelerator board

be filled before processing can begin.

![Diagram of Processing Time vs Vector Size](image)

Figure 2.5: Comparison of vector and pipelined processors for vector operations.

2.1.4 VLIW and superscalar

In order to obtain maximal performance from today's microprocessors, parallelism between different units of the CPU, such as provided by pipelining, is no longer adequate. Instruction-level SIMD parallelism, used in vector processing, is limited in its scope of application. To maximise performance, some form of instruction-level Multiple Instruction-stream Multiple Data-stream (MIMD) parallelism must be incorporated into the processor. In other words, multiple instructions must be processed simultaneously by different units within the CPU.

Instruction parallelism can be approached in one of two ways. The instructions which are to be executed in parallel may be specified in the software, by pro-
grammer or compiler, or their selection may be left up to the hardware. An example of the first solution is Very Long Instruction Word (VLIW) [54] architecture, so named due to the large size of the resulting instructions. The hardware-based approach is represented by superscalar [55] designs.

In general, VLIW architectures provide less flexibility, since a fixed number of instructions must be provided for execution at every clock cycle. Binary compatibility is also difficult with VLIW, as all the implementations of an architecture have to retain the same number of functional units. Superscalar designs, on the other hand, require more hardware and can in some cases result in less efficient execution, although recent research seems to indicate that the two approaches yield similar performance [57].

The i860 implements a VLIW architecture with the ability to execute one floating point and one integer instruction every clock cycle. The VLIW features are only available in a special processor mode, known as the **Dual Instruction Mode** (DIM). This mode is quite difficult to use efficiently due to a complex set of timing restrictions and its interaction with other processor components, in particular the pipelines. Therefore, at present, few of the available compilers generate code which takes full advantage of this feature.

In addition to the VLIW support, the i860 also provides further mechanisms to accelerate execution, namely multiply-and-add or dual operation instructions. The combination of multiplication and summation operations is very common in a number of scientific and DSP applications and the i860 provides a number of special instructions, which allow the adder and multiplier to both operate within one clock cycle. Sadly, these instructions are so irregular with respect to their arguments, that they are only useful in hand-generated assembler. However, in combination with DIM, they allow the i860 to potentially execute three operations coded as two instructions every clock cycle.

---

4Although this problem may be partially solved by providing a variable size instructions, for instance the XIMD architecture [56].
2.1.5 Memory management

The i860 has an on-board paged Memory Management Unit, which is functionally identical to the paging MMU used in Intel 80486 processor. It provides a two-stage translation of a 32 bit virtual address to the 32 bit physical address. The translation process is illustrated in figure 2.6.

As can be seen in figure 2.6, the virtual address is split into three parts:

- **Directory**: contained in the top 10 bits of the address, which provides an offset into the page directory table.
- **Page**: which is contained in the next 10 bits of the virtual address, and offsets into the page table.
- **Offset**: which provides the 12 bit byte address within a page.
A page is 4096 bytes long, and both the page directory and each page entry fit into exactly one page. The MMU provides a selection of flags, which allow precise control of the behaviour for each page. A page may be marked as only accessible from the supervisor mode, be protected from modifications or made un-cacheable.

Since the translation of a virtual address requires two memory reads, it substantially slows down the access to memory. The caches partially alleviate this problem, but to further accelerate execution the processor uses a Translation Look-aside Buffer (TLB). The i860 uses a 64-entry TLB, which forms a four way, set associative cache for address translations. The least significant 4 bits of the virtual address are used to select a set and the top 16 bits (known as the tag) are matched against those entries. This process is shown in figure 2.7. The TLB uses a random replacement policy.

---

Figure 2.7: The i860 Translation Look-aside Buffer.

---

Ignoring the bottom 12 bits which form the offset within a page.
The virtually indexed and tagged TLBs are known to cause problems in multi-tasking environments. Since this work involves such environments, the issues involved will be described in more detail in section 2.

2.1.6 Cache

One of the most fundamental problems with the RISC philosophy, is that, as described in section 2, in reducing the complexity of individual instructions, it increases the number of instructions required to perform a given task. Therefore in order to outperform CISC processors, instructions need to be processed at a substantially higher rate. This increase in instruction throughput must be matched by a corresponding increase in processor-memory transfer bandwidth, or the processor will not be fully utilised. Therefore the memory bandwidth usually represents a scarce resource in RISC designs.

The problems introduced by RISC are further compounded by on-chip parallelism. While SIMD parallelism (vector processing) has a small impact on the memory bandwidth requirements, MIMD overlapping (pipelining) or parallelism (VLIW or superscalar) substantially increases the resulting memory traffic. Since to reach the expected performance, a modern CPU must incorporate most of these features, solutions to the memory communication throughput problem must be sought.

There are principally four possible approaches to solving this problem:

1. Increase the speed of memory and memory interface. Unfortunately fast SRAM memory is expensive and less densely packaged. However progress is being made on this front: synchronous protocols show promise of faster memories in the future. A review of progress in this area, in particular the RAMBUS synchronous bus, is given in reference [58].
Two

The Numbersmasher i860 accelerator board

© Provide multiple access ports. This option is frequently used in supercomputer applications and video RAM, where in addition to the processor, the memory has to respond to video logic. However, it is expensive to implement. A slightly different approach is to opt for a Harvard architecture of the processor. Unlike the conventional von Neumann processors, Harvard architectures separate instruction and data access paths to boost the available bandwidth. At present, most CPUs use the Harvard architecture internally (with separate instruction and data caches), but provide a single memory interface, due to the cost of connecting two separate buses to the chip. There are, however, some notable exceptions, such as the HP PA7100 [59] or the Analog Devices DSP chip 21060 Super Harvard ARChitecture (SHARC) [60].

© Provide a wider access path (data bus). The current crop of 64-bit processors is reaping the benefits of wider data memory buses. However, wider buses are more expensive to implement and only help if the data accessed is stored consecutively.

© Reduce the memory interface traffic. This method can take many forms. The CISC approach was to use variable-size instructions, which was taken to the extreme by the Intel 432, which had instruction sizes ranging between 6 and 321 bits [61]. The additional decode logic complexity entailed is unacceptable for RISC philosophy. A different solution is to use a hierarchy of faster memory, with one, two, or even three levels of caches. Care has to be taken in system design, because caches are disproportionately effective in raising the performance of benchmarks while not helping as much with real applications [62]. In fact, caching data can also be used on the other end of the memory bus, to hold data (for example Mitsubishi CDRAM [58] with a SRAM cache) or even to hold a part of the address, in page mode operation.

The i860 is a RISC, pipelined, VLIW processor, with each of these architectural features imposing further demands on the memory interface. Therefore, it uses
Two The Numbersmasher i860 accelerator board 26

a combination of all of the above techniques, in order to reduce its memory bandwidth requirements to a realistic level.

The memory interface is 64-bits wide, supports page mode and multiple outstanding requests. However, to achieve its full performance, the i860 uses split data and instruction caches. This makes it internally a Harvard architecture, although its external memory interface is von Neumann. The instruction cache is 4 K two way, set associative cache with 64-sets, making each line 32-bytes long. The cache uses a random replacement algorithm and is both indexed and tagged by the virtual address. The instruction cache is read-only, requiring a cache flush for self-modifying code.

The data cache is a 8 K, four way, set associative cache with 128 sets. This cache is also indexed and tagged by the virtual address. This provides optimal performance, but is well known to cause problems in multitasking systems. These problems are discussed in section 2. The data cache uses by default a write-back policy, which further reduces the memory traffic. Any writes to data which is held in the cache, are not immediately forwarded to the memory, but instead are stored in the cache and marked as modified. When the line in which the data is stored is about to be replaced in the cache, the data is written to main memory. Caching may of course be disabled for areas where it is inappropriate, for example memory-mapped devices.

2.1.7 Interaction of processor architecture and Operating System

Many investigations have been conducted into optimal cache architectures [63, 64]. Data in the cache is addressed by parts of the virtual or physical addresses. The index selects one set of data in the cache and the tag uses associative hardware to perform a lookup in the selected set. While virtually indexed caches provide the optimal performance [65], since the address can be looked up in the
cache without waiting for address translation, virtual tagging encounters problems with aliasing. Aliasing occurs when two different virtual addresses are mapped to the same physical address. This occurs in multitasking systems in two common cases: shared memory (between processes) and virtual memory (reusing a physical page after swapping its content to disk). Aliasing can cause memory inconsistencies, since the same location may be cached twice creating problem when modified.

The i860 XR\textsuperscript{6} does not provide an easy solution to this problem. Therefore, an operating system under which aliasing can arise, has to either take great care to avoid it, or flush the cache on every context switch. This in turn causes large performance penalties, as described in section 3, and makes Helios, with its single, common memory map, particularly well suited to the i860.

\section{2.2 The Numbersmasher board}

The Numbersmasher board, which was designed at the University of Bath [66] and marketed by Microway Inc., consists of a single i860 XR 40 MHz part with an speed optimised memory interface to 8 or 32 megabytes of RAM, a boot EPROM and external communication devices. The board has an Industry Standard Architecture (ISA) bus interface [67], since it is intended for use as an accelerator board for PC-compatible computers [68]. A overview of the board may be seen in figure 2.8.

\subsection{2.2.1 Memory interface}

\textsuperscript{6}The follow up chip, i860 XP, has a modified tagging which uses both virtual and physical addresses, in order to eliminate this problem.
Since the i860 is a VLIW processor, possibly executing two instructions every clock cycle, it requires a high-speed memory interface. The memory requirements are partially relaxed by the two on-chip caches. However, a fast main memory interface is still required, and the i860 XR meets this requirement via a 64-bit data bus which is capable of fetching 8 bytes every two clock cycles. Thus, with the i860 running at 40 MHz, the required memory bandwidth is 160 Mbytes/s. Later i860 XP versions of the processor running at 50 MHz could collect 8 bytes every clock cycle, achieving 400 Mbytes/s bandwidth.
A simple memory interface for the i860, would require prohibitively expensive 25 ns static memory, to allow time for the operation of decode logic. Therefore, the memory is divided into four banks, each 64 bits wide. The decode logic is arranged so that each bank decodes at a different pattern of the least significant two bits of the physical address. This allows the use of memory access times of about 80 ns with a DRAM cycle time of 200 ns, whilst still supplying 64 bits to the processor every 50 ns. The accelerator board thus provides optimised cache line fetches of 32 bytes as well as flexible access order at full speed, provided that the least significant address bits do not repeat within four accesses. The hardware ensures that, if that is not the case, the processor is delayed until the access can be completed.

In addition to the RAM, the Numbersmasher board holds a boot EPROM. Although the i860 memory interface is 64 bits wide, the chip makes special provisions for using an 8 bit wide data bus at boot time. This allows the use of an ordinary 8 Kx8 EPROM chip. The EPROM holds boot code which emulates the boot protocol used by Transputers [69] via the on-board link adapters. This protocol allows reading and altering the content of RAM, as well as start of execution of uploaded code.

2.2.2 External devices

The Numbersmasher board provides the i860 with two link adapters, implemented using the IMS C012 [69] chips and a special-purpose FIFO interface. The link adapters provide the primary link to the outside world. One of the links may be attached to another link adapter, which in turn is interfaced to the ISA bus. This provides the primary path of communication between the PC and the i860, via the two link adapters. Although this solution may seem sub-optimal, it was chosen to allow the external interface to be easily adapted to other bus architectures.
The second link adapter is left free, and allows multiple cards to be linked together in a chain topology. The link adapters use an Inmos serial protocol, which allows the selection of one of two communication speeds, with theoretical throughput of about 1 Mbyte/s and 2 Mbyte/s. However, since the IMS C012 link adapter chips, which are used in the design do not support overlapped acknowledges, the available communication bandwidth is about half of the theoretical maximum.

The FIFO adapter basically provides direct access to the processor, together with some arbitration and interrupt logic. It is intended for attaching external devices which can communicate with the i860 faster than the limits of the link adapters. The original aim of its design was to allow construction of a distributed memory multiprocessor, by connecting multiple Numbersmasher cards using their FIFO interfaces. Although this goal was never realised, the interface was used to develop a graphical display card and a high-speed EISA bus interface card, which increases the PC-Numbersmasher communication rate to more than 64 Mbytes/s.

### 2.2.3 Interrupt control

The Numbersmasher board delivers interrupts to the i860 from five hardware sources:

- On-board timer, which generates interrupts with a frequency of 1250 Hz for early boards and 100 Hz for later revisions.
- FIFO connector.
- The host via the ISA bus.
- Two on-board link adapters, when ready to receive or transmit.

\(^7\)The Inmos protocol allows for acknowledges to be overlapped with data communication, however this feature is only implemented in the top-range processors.
Each of the link adapters may be configured to generate an interrupt when ready to send or to receive, increasing the number of logical interrupt sources to seven. The board's logic allows these interrupts to be selectively disabled.
2.2.4 Access to devices

The i860 does not provide a separate I/O address space. Therefore, the control logic of all devices is mapped into the memory address space. The decode logic uses the top bits of the address to decode the device. However, the top-most 4 bits of the address, are ignored in the address decode, because of a bug in the early versions of the i860 XR chips, resulting in the address map shown in figure 2.9.

In summary, the Numbersmasher i860 board provides a very fast and relatively inexpensive platform for scientific computation. Despite its age, the i860 XR chip is perfectly capable of delivering very high performance, if used with hand-coded assembler or a suitably optimising compiler. Helios is particularly well suited to this environment, as the chip is not designed for fast processing of full context switches, such as those necessary with UNIX.
Figure 2.9: Numbersmasher memory map.
The principal aim of this project is to provide a compact, powerful and cost-effective parallel computing platform. The research has concentrated on creating the software infrastructure upon which portable parallel programs could be built. This infrastructure has to be well suited to the hardware platform available, which in this case consists of multiple i860-based accelerator cards and in particular the Numbersmasher device described in chapter 2. One of the explored alternatives was the use of a distributed operating system.

3.1 Introduction

An operating system selected to support multiple i860 cards must meet a number of requirements imposed by the hardware characteristics. The most important of these are:

- The operating system clearly must support multiple processors. Ideally, the system should be designed to include support such an architecture, rather than it being added-on as an afterthought, resulting in a more complex and less powerful set of abstractions.

- Portable parallel software is a thing of the future, although some work by Philippsen [71] achieves very good parallel performance from portable
Modula-2 code. Therefore, the system should at least partially support the existing standards. The only existing relevant standard is POSIX [72], the IEEE portable operating system standard. The adherence to POSIX will minimise the necessary changes to applications.

Since the communication bandwidth provided by the hardware available is relatively small (of the order of 700 KB/s) the operating system must work efficiently in a distributed environment. It should particularly avoid using centralised resources which can become a performance bottleneck.

Although intcl claim [73] that the i860 was designed to fulfill the role of a general purpose workstation processor, the chip is better suited to a single application environment. Most notably, it suffers particularly badly from the bane of RISC processors: a context switch overhead [74]. The saving and restoring of the full processor context\(^1\) takes approximately 618 instructions (559 of which are spent in changing the memory map) on the i860 [76]. Since the majority of this time is spent in writing back the 8 K data cache, invalidated by changes to virtual memory mapping, a considerable improvement in performance can be gained by using an operating system which has a single memory map shared by all processes.

Finally, the programming of any parallel system requires the knowledge of its paradigms and implementation. For the parallel i860 platform to be truly useful, the system used would have to become familiar to the potential application writers and users. Since the environment is intended for local use only, any previous experience would be of great benefit.

The Helios [77, 78] distributed operating system developed by Perihelion Software Ltd, meets all of the above criteria, as well as the practical constraint of

\(^1\)This includes swapping all of the explicit state and writing back values from cache, but not including penalties incurred in reloading it, which, as shown by Mogul [75], may also be quite significant.
the availability of source code. Previous parallel work has been done locally using it, including parallel power system applications [79]. Therefore Helios was chosen as a possible software platform for the parallel i860 system.

3.2 Brief description of operating systems

Since a large amount of the work described later requires detailed knowledge of the internal structure of Helios, the system will be described in depth. A primer on some basic principles of operating systems will be provided before the details of the system itself.

An operating system is the heart of the interface between a portable application program and the hardware which executes it. It allows the programmer to write an application without worrying about the details of the hardware and other software present on the system.

In the early days of computing, programs were developed in machine code, entered via plug boards or punched card and debugged using memory dumps. This process was labour-intensive and error-prone. Later, tools were developed which allowed easier and faster development techniques. With the advent of the subroutine, the most frequently used code migrated to libraries. Computers became more widely used, with each running many batch jobs in sequence. The benefits of separating the low-level hardware-dependent code, which was used by every application, soon became clear. This code formed a library which was used by every program.

As computers became used in multi-programming environments, enforcement of protection between different users and tasks has become vital. To provide reliable defence against incorrect or rogue programs, processors usually provide at least two modes of operation: a user mode in which instructions can only access some resources and a supervisor mode in which code is not restricted.
Modern operating system can be separated into the *kernel*, which includes all code executed in the supervisor mode, and the various system libraries, utilities and applications which are executed in the user mode.

In order to protect external devices, communication with them must be restricted to the supervisor-mode only. Therefore, processors which provide two modes of operation require that I/O and interrupt processing to be done in the supervisor mode. Similarly, any faults detected during the execution of instructions, which may involve privilege violations, must be handled by supervisor code. The kernel must therefore handle numerous unpredictable events.

The handling of external devices constitutes a large part of the activities of the kernel. To allow support for the wide variety of available peripherals, device-specific code for each peripheral is usually separated in the kernel into a distinct *device driver*. Interaction with a device often requires handling asynchronous events generated by it, as may be seen in figure 3.1. By convention, the part of the device driver which processes the synchronous software requests is known as the *top part*, while the code which handles the asynchronous hardware interrupts is called the *bottom part*.

It is worth noting that while, as stated above, the kernel part of the system is usually executed in the supervisor mode, some processors do not support two execution modes and some systems do not use them. In such systems, the distinction between the kernel and any other library becomes rather less well-defined.

The level of services provided in the kernel varies widely from one system to another. For example, compare VMS [80, 81] or Multics [82] to the microkernels described in section 3. However, in all cases, the system must impose some degree of organisation on the utilisation of resources to allow the inter-operation of applications. This usually includes a defined disk filing system and memory organisation. Most systems also provide a number of useful refinements which ease the work of users and application programmers. Users are often also given numerous utility programs, which allow them to manipulate the system and
the data stored on it. In most cases, the majority of the work of users takes place inside one of a few applications, which are often provided by third parties.

3.3 Features of Modern Operating Systems

The previous section described the basic features common to all operating systems. This section outlines some of the more modern developments, which are particularly relevant since Helios utilises many of them. While many of these developments are still in the research stages, some features have been successfully applied to commercial systems besides Helios, such as Chorus [83] or the L3 microkernel system [84]. Only the most essential or novel features are discussed below, with many other aspects, such as shared libraries and dynamic linking left unmentioned.
3.3.1 Multiprocessing

The function of the operating system could be restated as:

*matching the hardware resources available to the requirements of the software*

One place where the mismatch between these two quantities is usually greatest is in the number of processing units. Even single-user systems have a need to execute multiple programs at the same time, for example displaying a clock and processing mail while running an application, and many of today's systems support multiple users. However, conventional computer architectures contain only a single CPU. This mismatch is solved within the operating system by time-division multiplexing many applications on the single CPU. This is known as *multiprocessing*.

The process of deciding which task should have the use of the CPU at any particular moment in time is known as *scheduling*. Scheduling is separated into three levels, for historical reasons:

*long term* also known as *high level* scheduling, decides which jobs shall be allowed to start execution on the system. This level is used exclusively in batch systems, like MVS [85], since its function is performed by the user in interactive systems. That is not to say that interactive systems do not provide batch facilities. UNIX for instance supplies the at batch scheduler. However any such mechanism lay outside of the core of the system. Therefore, it will not be discussed here any further.

*medium term* or

*intermediate level* scheduling enforces the system policy of CPU usage by redistributing the processor allocation between
tasks, based on their past usage and importance. This redistribution may be implemented by temporarily suspending some tasks or adjusting task priorities.

**short term** or **low level** scheduling is concerned with determining exactly which task shall execute next and for how long and is usually performed several times per second.

Various mechanisms are in use for all scheduling levels. The medium term scheduler must implement the system policy, which may be defined by a mixture of opposing aims, usually including fairness\(^2\), efficiency, low latency, predictability and graceful degradation. Since Helios does not have a medium-level scheduler, these will not be described any further. More information may be found in reference [87–89].

A short term scheduler may be preemptive or non-preemptive. A **preemptive** scheduler may interrupt a running process before it has finished in order to allow some other task to use the CPU. A non-preemptive scheduler must let every task run to completion. A non-preemptive short term scheduler is trivial, as the decision about which task to start next will be taken by medium or long term levels.

A preemptive short term scheduler may be implemented in a variety of ways, the most common being some kind of queueing arrangement. Perhaps the most common algorithm is **Round Robin** (RR), where the tasks waiting for the CPU are granted access in order for a short time interval, known as the **quantum** or **time slice**. The tasks which are ready to execute are kept in one or more queues. A simple Round Robin it does not perform well in the presence of a mixture of CPU and I/O bound tasks. However the simplicity of the algorithm, resulting in very low scheduling overheads, has resulted in the wide-spread popularity of its variants. Many algorithms have been suggested, which at-

\(^2\)Many systems define fairness in terms of equal distribution of resources between processes. However this is unfair to users who create few processes and hence more recent solutions ensure fair allocation between users and groups of users [86].
tempt to combine the simplicity of Round Robin with greater fairness [90].

Since the short term scheduler should implement the policy decisions taken by the medium term scheduler or the user some mechanism for communicating these decisions between these two levels must exist. The most common mechanism of communicating this information is assigning every process a priority. The priority imposes an ordering on the processes which are ready to run. This ordering reflects the policy decisions taken by the higher levels. The low level scheduler has to simply ensure that the highest priority process available will execute at any moment in time.

Problems can arise if a higher priority process is being held up by a low-priority task. This phenomenon is known as priority inversion and occurs usually as a result of the higher-priority task being dependent on the lower priority one, for example if the two tasks use a shared resource which is currently held exclusively by the low-priority process. Various schemes exist for avoiding priority inversion, although no panacea is readily available [91].

3.3.2 Memory management

Management of memory poses different problems to the management of a CPU. This is because most machines contain a single CPU but many independently addressable memory locations. Memory is addressed in units of bytes which are 8 bits long, although some CPU permit bit-wise addressing and many DSP processors require word aligned addresses3. Many systems prefer to manage memory in larger units called memory pages, usually because the underlying hardware enforces such protection granularity. The operating system normally uses space-division multiplexing, that is allow simultaneous accesses to memory but at different locations. However if the combined require-

3This is true for all modern machines, although many older mainframes used multiples of 6 or other sizes.
ments of all applications exceed the available memory, a time-division multiplexing technique, similar to the scheduling used with the processor, is also used. A typical scenario showing both approaches is illustrated in figure 3.2.

Memory requests 1 and 2 are space-multiplexed, and are mapped into different areas, while requests 3 and 4 are time-multiplexed. There are two common strategies of time-multiplexing: swapping and paging. In swapping an entire application is written out to disk thereby freeing all the memory occupied by it, while paging saves to disk parts of the application memory, in units of memory pages. Paging relies on memory protection hardware to tell the operating system when the saved memory is being accessed, whereas swapping can still be useful with less hardware support. However, in order to be able to reuse the freed memory the architecture must at least provide address translation. Address translation is a hardware mechanism, which performs a flexible mapping between the addresses used by software, known as logical addresses and the values used to access the memory, called physical addresses. This translation is performed by a memory management unit (MMU) and
its granularity is usually restricted to integral page sizes. For more information see reference [92]. Since Helios does not support swapping or paging, a detailed description of algorithms used will be omitted. An interested reader is referred to [88, 89, 93].

### 3.3.3 Real Time systems

The areas of industrial process supervision and control have traditionally managed without operating systems. The main problem encountered with using an operating system in these environments is the requirement for precise and deterministic control of the timing. Traditionally, operating systems provide a level of abstraction, which separates the user from the details of the hardware, but which also separates him from the details of timing.

Real time operating systems are being developed for use in these areas, both using new designs and re-implementations of older systems, like REAL/IX, a real-time UNIX [94]. A real-time operating system is a system which allows applications running under it to meet precise timing constraints on their behaviour. The main attributes of a real-time OS include:

- Guaranteed low interrupt latencies, which enable the system to respond rapidly to external events.
- A predictable scheduler, which allows the users to prioritise the tasks based on their importance or deadline. This area is still developing, with many innovative solutions being proposed [95].
- Provision for uninterrupted execution of applications in critical regions, some of which may be quite long.

Two principal methodologies are used in real-time system design: event-triggered systems and time-triggered systems [96]. The event triggered systems are interrupt-driven by events occurring in the physical plant, while time
triggered design uses snapshots of the plant taken at fixed time intervals. This makes event-triggered systems more efficient and easier to use, but far less predictable, with extensive testing necessary to ensure reliability. In practice, event-driven designs are used for most applications, except safety-critical situations, where the reliability of time-triggered systems makes them the preferred choice.

3.3.4 Threads

Many modern operating systems separate the unit of protection or task from the unit of scheduling known as a thread. Each task may contain multiple threads, which provides a degree of parallelism within an application. The use of threads can vastly simplify some programming solutions by decomposing the overall program into a number of independent parts, without incurring the overhead usually associated with tasks. Lazowaska [97] cites an order of magnitude improvement in performance between tasks and kernel-implemented threads and another order of magnitude improvement with user-space threads.

Thread support is provided by many modern operating systems either by the kernel or user-mode libraries and has been added to older operating systems, including UNIX. Threads are used in a wide variety of roles within the applications [98] and form an increasingly important part of real-time support [99] including the POSIX real-time thread specification in ISO 9945-4.

3.3.5 Distributed Parallel Systems

While most traditional architectures contain a single processor, an increasing number of systems take advantage of parallel processing [100]. Support for multiple CPUs in operating systems varies widely. Some traditional systems
such as UNIX have been enhanced to provide some multi-processor awareness [101, 102]. However, this multiprocessing is, by necessity, tagged-on and is never fully integrated with the other facilities. Therefore, it is usually limited in both scope and efficiency. In contrast, more recent systems provide support for multiprocessing from the design stage onwards [103].

An important consideration in a multi-processor system is load balancing. This involves the appropriate distribution of processing tasks across the multiple CPUs so as to maximise performance. The scheduling of tasks across multiple processors is a particularly hard problem, especially if taking into consideration the unequal distribution of resources between the different processors and the special requirements of the various tasks.

### 3.3.6 Process migration

Noting again that the operating system must match the system resources to the requirements, one very useful feature provided by some operating systems is process migration. Most multiprocessor systems provide load balancing as a method of matching the CPUs to the requirements of the tasks. However load balancing is performed statically, that is, the processor on which a task will execute is selected at the start of the task and may not be changed afterwards, leading to suboptimal solutions.

**Process migration** is a dynamic load balancing mechanism, where a program can be moved from one CPU to another during its execution [104, 105]. In most complex systems, where the processing requirements of the tasks cannot be determined before the execution process, migration provides the only reliable method of load balancing.

It must be noted, however, that process migration is still very much a research issue [106]. Many problems are associated with selecting the task to move, its destination processor and ensuring that the environment of the tasks remains
unperturbed by its migration. Since a task uses other resources in addition to the CPU, care must be taken to meet all of its resource requirements in migration. The issues concerned with migration in a heterogeneous system are even more complex.

Although some work has been done elsewhere on providing a transparent process migration mechanism under Helios [107], it was of highly experimental nature and unsuitable for general use.

### 3.3.7 Microkernels

In a traditional operating system, the kernel performs a multitude of resource management functions. The alternative approach which has lately become very popular, is to only provide the management of only the basic resources and message-passing functions in the kernel and move most of the resource management code to separate programs known as servers. This design is known as a microkernel since the resulting kernel is very small in contrast to traditional monolithic kernel designs. The idea of the microkernel is not new and similar systems were developed in the late 1960s, for example the RC4000 [108].

The microkernel approach has many advantages. The kernel, due to its key role in an operating system, cannot be easily manipulated while the system is running. The removal of code from the kernel to server tasks allows its behaviour to be easily altered. This also has the benefit of the normal system management mechanisms, including protecting the code from accidently affecting other servers and ability to swap it to disk if necessary.

However, microkernels also have their faults. The most significant one is the time overhead incurred while switching the processor context between a number of tasks as shown in figure 3.3. The majority of the delay is caused by the

---

4 Other paradigms of OS design, such as virtual machine used by VM/370 [103]
Three protection information and may in fact be thought of as the price of the extra safety provided by this compartmentalisation. Careful message passing design can of course minimise this overhead. Liedtke [109] found substantial performance improvements from small changes to the inter-process communications mechanism, ranging between 10% and 160%. However to eliminate the problem, two possible solutions exist:

- The system context size may be reduced, by using, for example, a single memory map with no protection between tasks, which is the path taken by Helios. While this loses the safety benefit of microkernels, it still retains the configurability and modularity.

- Tasks may be dynamically, that is, at run time, moved between being executed in the kernel or in a separate server task, as done by Chorus [83]. This gives the maximum flexibility with the user able to run most often used servers in the kernel while retaining the safety of separate servers for more experimental modules.

Microkernels also suffer from secondary effects, such as the reduced effectiveness of caches due to code more spread out across the address space [110].
3.3.8 Fault Tolerance

Traditional operating systems were developed for localised hardware environments, which could easily be controlled. Any hardware failure would have a substantial impact on the capacity of the system and would almost always be handled immediately by operators. However, as large distributed systems developed and users' expectations of reliability were raised, it became necessary to cope with hardware faults while minimising their impact. Fault tolerance or the ability of a system to remain functional in the presence of faults, is a complex subject, which is described in detail in references [111].

Fault tolerance usually begins with detecting the error. Then an appropriate corrective action has to be taken. This may include limiting the extent of the effect of the failure (containment) or taking corrective action, which hides the occurrence of the error from higher layers (masking). This is followed by repair of the faulty component or its replacement and recovery of system state before failure [112].

In real-time systems, errors may be classified in the following hierarchy [113]:

- omission failure no result is ever produced.
- timing failure the result is produced too late.
- response failure the wrong result is produced.

3.3.9 Heterogeneous systems

Most multi-processor systems consist of an ensemble of identical processors. Even if the processors are not exactly identical with respect to the communication medium or do not have an identical specification, they usually retain bin-
ary compatibility. Some architectures however combine many completely different processor architectures. Such systems are known as heterogeneous and require special support [114–116].

One problem is that the different elements of the system may have different capabilities which should be taken into account in load balancing. In particular, the hardware may contain general-purpose processors, graphic accelerators or Digital Signal Processors (DSPs.) The communication between systems which use different data formats, protection mechanisms and naming schemes is also a complex problem [117, 118]. Another issue is the format of executables. In systems with few processor types, separate binaries may be stored. This approach was used by Apollo in their 68000/88000 mixed machine networks running Aegis operating system [119]. However, with a large number of architectures, this results in cumbersome binaries with lots of duplicated data.

A different technique is to store the programs in an intermediate form and translate them to the target language at load time, which is the approach taken by Taos [120]. The main problem here is discovering an intermediate representation which can both be quickly translated into varying architectures and which also results in efficient code. This is a truly challenging task considering the diversity of various processor architectures. For example, the number of registers available in various processors, which is central to efficient code generation, can be as high as 250 or as low as 2. However, some research done on the Oberon system by Franz [121] suggests that the load-time generation of code may have performance benefits even on a homogeneous system. Most recently, the progress made in standardising architecture independent object formats has culminated in Architecture Neutral Distribution Format (ANDF) specification, developed by Open Software Foundation (OSF) and Defence Research Agency (DRA) [122].
3.3.10 Wide Address Spaces

The development of new processors with 64-bit data and address buses, such as the MIPS R4000 [123] and the DEC Alpha [124], has given the operating system developers vast virtual address spaces. In a 32 bit system, the amount of physical storage is likely to approach or even exceed the virtual address space. This forces the designer to provide a separate address space for each process and limits the use of other mechanisms, such as memory-mapping of files. Wide address spaces free the designer from these restrictions and allow the use of single address space within a machine and even within a network, resulting in a simpler yet more powerful systems. New operating systems based on this approach are currently being researched and include Opal [125] and Pegasus [126].

It may seem that the 18 quintillion bytes provided by a 64-bit address space will be adequate for the foreseeable future. However, much the same could be said about 32-bit address spaces at the time of their introduction. Even the huge 64-bit space is not adequate for global object addressing in a world-wide network and will undoubtedly be eventually replaced by 128 bit.

3.3.11 Standardisation

Traditionally the operating system for a piece of hardware has been developed by the manufacturer in order to provide the users with some basic functionality. More recently, portable operating systems have been written using high-level

---

5 In a network of 10 million computers, each generating ten 10 Kbyte objects every second, 18 Ebyte address space would be used up in under 7 months. This calculation does not take into account the inefficiencies resulting from orthogonal address assignment, which in the case of a similar allocation on the Internet is less than 1%.
programming languages, and used on a number of different architectures. One of the first systems to be implemented in high-level language was Multics [82], which is written in PL/I. Unfortunately, the system required a number of unusual hardware support features, which are only present in the GE 645 machine for which it was written. On the other hand, UNIX, which is written in C, makes very few assumptions about the hardware, allowing it to be ported to a vast array of architectures. Presently, the standardisation committees are coming round to the task of standardising operating systems.

Various efforts at standardising an operating system have centred mostly on the UNIX community, due to its history of proven portability. As UNIX gained wide-ranging popularity, a large number of mutually incompatible versions developed. Efforts to standardise these were undertaken by AT&T, who are the originators of UNIX, in the "System V Interface Definition" or SVID standard [127,128]. However, large numbers of UNIX users were running the Berkeley Software Distribution (BSD) version [129], which provided enhanced features. Still other recommendations for portable software were produced by X/Open, an international consortium dedicated to the advancement of open systems [130]. This resulted in considerable incompatibility between different variants of UNIX [131]. Fortunately, the national standardisation bodies have stepped in with the "Portable Operating System Standard" or POSIX. POSIX is based on a mixture of AT&T and BSD featured, and is defined by the International Standards Organisation standard 9945 [72].

Other bodies are attempting to standardise parts of operating system services, with particular view to inter-operation between different systems. The Open Software Foundation (OSF) has specified its Distributed Computing Environment (DCE) [132].

In addition to operating systems, various efforts are under way to develop inter-operation standards based on other paradigms, in particular the object oriented approach. Although such standards do not specify a conventional operating system, the functionality defined by them is traditionally associated
with operating system services making them relevant to this discussion. The most prominent among the object oriented standards include Object Management Group’s (OMG’s) Common Object Request Broker Architecture (CORBA) [133] and IBM’s System Object Model (SOM) [134].

3.4 Helios

Helios

The Helios operating system is rarely used on stand-alone computers. It is mostly installed on additional processors which are used to improve the performance of a general-purpose computer. Therefore, a typical Helios configuration includes a front-end machine, which is usually running its own operating system like UNIX or MS-DOS, with Helios controlling only the accelerator processors.

Helios is built around a message-passing microkernel, which provides only basic task control and message passing facilities. All other facilities are provided by a collection of servers. Most of these servers are executed as normal Helios tasks. Some servers provide processor-specific functionality and hence are duplicated on every CPU. Others provide global services and may be run on one processor somewhere in the network. The only exception is the IO server. The IO server is a server task which is run on the front-end processor and provides a number of external interface services, such as access to devices.

3.4.1 Helios Namespace

At the time of its development, the main innovation of the design of UNIX was its simplicity. This arose as a result of the limits of the target architecture, which was a PDP-7 computer. The enduring popularity of UNIX stems largely from its
Three

*The Helios operating system*

53

minimalistic approach, which made the system easy to use and understand. One of the simplifications introduced by UNIX, was a reduction in name-spaces. Rather than provide a separate mechanism to access devices, the system uses special files to represent peripherals [135]. Each *device file* is treated by the operating system much like an ordinary file, but any accesses to it are forwarded to the corresponding device driver instead. This unification of two previously separate name-spaces reduces the number of concepts which the operating system and its users have to handle and results in a more regular and powerful architecture.

Helios extends this philosophy while retaining backward compatibility with UNIX. Under Helios, all system objects share a single name-space. This includes networks, processors, servers, tasks, modules, users, devices and files. This approach results in a very conceptually simple system. For instance, terminating a task can be performed by deleting the corresponding task object. Most of the operating system designers now agrees that a single name space is a good idea. In fact, modern versions of UNIX usually include a `/proc` filesystem which maps processes into the file name-space.

To maintain backward compatibility with UNIX, Helios uses a hierarchical namespace. However, the UNIX namespace forms only a sub-tree of the Helios one. A typical Helios name tree starts at the network level, containing processor entries, which in turn contain server entries and finally the objects within servers. The entire UNIX file-space forms a tree under the file server entry. An illustration of the overall structure may be seen in figure 3.4. Different servers contain different types of objects. For example the entries in the `sm` (*session manager*) server will correspond to the user sessions.

However, simply using a similar structure to UNIX does not provide Helios with the desired level of POSIX compatibility. To achieve that the system must allow the definition of similar standard path names. For instance,

---

6 Unfortunately, modern UNIXes have suffered badly from lack of coordinated development, resulting in the kernel increasing in size one hundred times.
Three
The Helios operating system

the POSIX password file should be named by the path name /etc/passwd. Also, the users would rarely appreciate the very long pathnames necessary to name objects in the Helios namespace. While the full name of a task, like /net1/proc01/tasks/worker, may often be necessary to uniquely specify the desired program, similar pathnames are unacceptable when referring to frequently accessed entities, such as user files.

Therefore Helios needed to modify the UNIX name resolution rules. UNIX recognises two types of paths: **absolute paths**, which start with a / character and **relative paths** which do not. Relative paths are processed starting from user's current directory, while absolute paths are processed starting from the top of the system name tree or root directory using UNIX terminology. Helios uses a very similar scheme, however its absolute paths name objects starting at server level or above. Therefore, the object ls in figure 3.4 can be specified by using any of the following paths:

- /subnet/10/files/bin/ls
- /10/files/bin/ls
- /files/bin/ls

Any ambiguity resulting from this scheme is resolved by using the object nearest to the process which is using the name, where distance is rather loosely
Three

The Helios operating system

55

defined in terms of effective communication delay. In fact this ambiguity is extremely useful in allowing simple extension of the functionality of the system servers. Any server may be in effect “masked” by an identically named server, which is closer to the client program. For instance, any client executing on processor /subnet/02 which request object /ram/file1 will obtain a connection to /subnet/02/ram/file1 instead of /subnet/01/ram/file1. This is because the closer /subnet/02/ram server masks the identically-named /subnet/01/ram server. Such a server can provide extended functionality, for instance caching, although it should be noted that a caching server would violate the principle of statelessness, adhered to by Helios for all its servers, as described in section 3.

This scheme alone does not provide POSIX name compatibility, since the aforementioned password file would still have to be referred to as /files/etc/passwd. However, Helios does in addition provide an alias server, which allows the creation of an arbitrarily named server, which acts as an alias for a specified path name. Hence a server named etc may be easily created, which forwards requests to the path /files/etc, resulting in the correct behaviour. Such name translation may be performed for all POSIX-mandated directories.

The implementation of the naming policy described above is done in a distributed manner. This is essential to avoid the performance and reliability penalties of a centralised name server. Various different methods of implementing naming schemes are currently in use [136]. Helios uses a combination of a flood-search\(^7\) for the name combined with a local cache to accelerate the process. The search returns the first matching object found, which results in the somewhat loosely-defined ambiguity resolution described above.

\(^7\)Simultaneous search through every connected link which “floods” the network of processors and stops when all nodes have been interrogated or the requested name has been found. This process is a distributed equivalent of broadcasting.
3.4.2 General Server Protocol

Since most of the facilities provided by a micro-kernel are supplied by the server programs, the system operation depends on a large number of servers and hence Helios defines a standard communication protocol for interacting with the servers. This protocol is known as the General Server Protocol (GSP) and defines a standard structure for messages. The protocol operates on Objects which represent the entities manipulated by the particular server. For example, requests to the task server will manipulate tasks using the same protocol as requests for files to the files server.

Each GSP message contains a 32-bit request code, if it is a request or a 32-bit error code, if it is a reply. The request codes are defined by composition of three fields, which describe the protocol class, subsystem and function code of the request. The protocol class allows separation of GSP and other, user-defined protocol messages. The subsystem defines the standard server, for which the request is destined and the function code specifies the operation to be performed. An example of a simple request code, in human readable form, is GSP:IO:Open, which represents a GSP request for the Input/Output Open function. The error code is hierarchically organised into the error class specifying the severity, subsystem which caused the error, general error code which identifies the kind of error and error object identifying the object type for which the error has occurred. An example of an error code is Fatal:Task:Protected:Module.

In addition to the error codes, the message itself contains the data necessary to perform the requested operation. Since the hardware on which Helios is used cannot guarantee reliability, GSP is designed to cope with unreliable communication network and frequent server crashes. This is achieved by making the servers stateless, so that if they should crash no state will be lost, and the re-
quests idempotent\(^8\), so that in case of a suspected communications failure, the message may be repeated without undesirable side effects.

The use of stateless servers to achieve robustness has many precedents including, most notably, the UNIX Network Filing System (NFS) [137]. However, while relatively simple to use, such protocols result in considerably larger messages. This is because a server state is normally used to hold the data common to multiple messages which with a stateless server has to be replicated in every request. For instance the smallest NFS read request is at 138 bytes long. Therefore, completely stateless protocols are becoming replaced by more complex protocols, which retain some state in the server yet allow for recovery in case of failure. Examples include the Spritely NFS system [138] and Andrew Filing System (AFS) [139].

For the above reasons, GSP is also not completely stateless. In addition to manipulating Objects, GSP allows the creation of a reference to the data content of an object, which is called a Stream. The stream is obtained by issuing an Open request to the relevant server. This allows the object location and access permission checking to be performed once for each stream and then cached in the server. Although this makes the server strictly-speaking state-full, the recovery of this state is trivial and done automatically by the client library by reopening the object, should the server fail to recognise the stream reference.

### 3.4.3 Fault resistance

To achieve some degree of fault resistance in a system with no hardware memory protection, the Helios design follows some general guidelines. Firstly, as described above, all system servers remain almost stateless, allowing a crashed server to be restarted without loss of information.

\(^8\)That is, defined in such a way that multiply repeated requests have an effect identical to a single request.
Helios also provides for a checker task which scans the code of all modules loaded into memory and verifies their checksums. If corruption is detected, the offending module is reloaded from disk and a warning is issued. Other systems use similar schemes to ensure errors are caught as early as possible [140].

However, the lack of memory protection cannot ensure the survival of the operating system in the case of a serious software failure. In fact, the only protection provided by the hardware is between two communicating processors. Therefore, Helios provides processor recovery as the last line of defence against catastrophic failures. If a processor fails to acknowledge messages, its neighbour will, hardware permitting, reboot it.

### 3.4.4 Security

Security is even harder to provide than integrity or fault resistance, without memory protection, since measures which are likely to detect random corruption can easily be evaded by an intelligent perpetrator. Additional difficulty stems from the distributed nature of the system. Many secure access schemes developed use a central authorisation server. Such approach works admirably well in systems which verify authorisation on a per-server basis, such as the Kerberos system [141] developed for Project Athena [142], a distributed workstation environment. However, Helios uses a single protection mechanism to authorise access to every object and consequently the number of permission checks issued would swamp any central authorisation resource.

For a security-conscious environment, Helios requires that the system servers execute on a separate processor. Furthermore, the processors must be allocated on a per-user basis, since any user application running on a processor may compromise security of any other program on the same processor. The authorisation system is fully distributed and uses encryption to ensure security. Each object on every server is associated with a secret identifier which, together with
the access restrictions, is used to build-up a capability. This capability is then passed to the client which owns the object in question and may then be distributed or stored by this client. Every request, including in particular an Open request, must contain the capability in the message body. The capability is then verified by the server before processing the request. This of course means that any processor which is used to forward messages between the client and server may intercept and copy the capability. Therefore, to ensure security, every untrusted processor must have a direct link to the trusted processors which it uses.

Capability-based security mechanism are used in many distributed operating systems, including Amoeba, which is described in section 3. Unlike many systems, Helios allows the clients to handle the capability data directly [143]. This does result in some potential security problems, since the client may try to guess a valid capability. Also, the distinction between granting a client access to an object and allowing it to grant access permission to other clients cannot be enforced [144]. However, given that the Helios hardware is not expected to provide memory protection, the alternative of handling capabilities securely by the kernel is impossible to implement. It should be clear that in a context where security is important, for instance the Department of Defence Orange Book classification scheme [145], a single processor Helios can only achieve the minimal rating, on par with unprotected systems like MS-DOS, while even multi-processor Helios configuration cannot match secure versions of UNIX.

The Data Encryption Standard (DES) [146] is used encrypting the capabilities to minimise risk of successful guessing or cryptographic attack. DES is deemed adequately safe for this application, although doubts remain as to its long-term cryptographic strength [147]. Many other authentication schemes, including Kerberos, prefer Rivest-Shamir-Adleman (RSA) cipher [148], due to it sounder mathematical basis and un-symmetrical nature\(^9\). See Simmons [149] for further information on crypto-systems.

\(^9\)That is, uses different keys for encryption and decryption. This property is extremely useful, as it eliminates the difficulties with key distribution, which normally hamper crypto-systems.
When a new user wants to enter the system, he must have his identity verified by the login program, which uses a standard UNIX user name and 8-character password. The user’s shell is then started and given capabilities for the home directory. With this capability, the user may explore his directory space. Helios uses a quite complex scheme for allowing hierarchical protection of objects within directories, but its description is deemed to be beyond the scope of this thesis.

3.4.5 Other Microkernel Operating Systems

Helios is not the only operating system which incorporates modern features. In fact, the design of Helios is based on a previous operating system developed by Prof. Tannenbaum at the Vrije University in Amsterdam called Amoeba. Amoeba incorporates many of recent developments in operating system thinking.

Many other operating systems are being developed around the micro-kernel principle. The most notable of these are:

*Amoeba* as mentioned above, is developed by Vrije University uses a microkernel-based design. The system assumes that the hardware consists of a processor pool in addition to the user’s workstations and controls the distribution of CPU-intensive tasks across this pool [150]. Amoeba uses capabilities for protection and defines its own programming language (Orca [151]) and file server (Bullet). Many of Amoeba’s features are present in Helios.

*Chorus* operating system implements many novel features including load balancing and process migration. Chorus minimises the overhead problems, which haunt microkernel designs by allowing movement of servers between user and kernel spaces dynamically. This allows the user to decide at run time which serv-
ers are robust enough to be placed in the kernel space. Chorus achieves UNIX compatibility via an interface layer.

**Mach** kernel, developed at the Carnegie Mellon University, is arguably the most popular microkernel around today. The system integrates virtual memory with its message passing facilities. Mach provides a UNIX compatible server and is used as the basis for new operating systems, including Hurd [152] developed by the Free Software Foundation (FSF).

**Oberon** is not strictly-speaking a microkernel system. Instead the system consists of a collection of cooperating objects which interact to provide the services [153]. The system is written in the Oberon object-oriented programming language and provides a unique windowing system. Oberon has achieved remarkable portability and performance, partially due to its design principle, which was to discard all non-essential features.

**Plan 9** is under development by the AT&T laboratories as their next general purpose operating system [154]. Its computational model is similar to that used by Amoeba and is based on a processor pool connected to a user's workstation. Like Oberon, Plan 9 defines its own windowing environment called $8\frac{1}{2}$.

**V system** provides virtual-memory based file access and process migration facilities [155].

**Windows NT** or New Technology is the current commercial system developed by Microsoft, mainly for the PC-compatible market. It uses few interesting solutions and has limited portability, but is included here due to its certain future widespread popularity [156].
A recent trend in research operating systems is the development of a system programming language or a windowing system together with the operating system with the *de facto* standards, such as ANSI-C and X window system supported only later, for backward-compatibility. Systems which have followed this approach include Oberon and Plan 9 systems described above as well as the Photon windowing system designed for QNX [157]. While many of such systems introduce minimal innovations and are unnecessary, their proliferation reflects certain inadequacies in the standard tools, many of which have sacrificed efficiency for generality. For example, X window system is notoriously demanding on resources, with the X11R5 server for a DEC Alpha occupying 22 Mbytes of virtual memory.

Many even more exotic systems are under development, including Synthesis [158], MUSE [159], work done as part of the Japanese TRON project [160], ARTS [161], Spring system developed by Sun Inc [162], HARTOS [163], MATURI [164], CHAOS [165], MARC [166], Hawk [167] and Pegasus [168].

The reasons for selecting Helios, which have already been described in section 3, included technical criteria and availability of source code. In addition, most of the systems described above would require considerable modifications to operate in a host–accelerator environment, like the one used in this project.

### 3.4.6 System structure

As already stated, the Helios system consists of a kernel and a number of servers. The Helios kernel is itself split into two components: the *executive*, which contains the hardware-specific parts, and the remainder of the kernel, which should be relatively portable. The separation of the architecture-dependent code took part during the ARM and Motorola 68000 ports of Helios.
The interface between the executive and the kernel is defined by Beskeen [2]. The executive provides low level scheduling, synchronisation and communication primitives and is described in more detail in section 3. The functionality of the executive and its interface to the kernel is based on the facilities provided by the Transputer processor. Simply put, in non-Transputer ports of Helios the executive replaces the extensive scheduling and communication support provided by the Transputer hardware.

Just as the executive forms a part of the kernel, the kernel itself forms a part of the nucleus. The nucleus is the name used by Perihelion to denote all the code loaded into the processor at boot-up time or resident in on-board ROM. This code is essential to the correct operation of the system and it must remain resident to all times. In a conventional system, this would consist solely of the kernel, but microkernel systems delegate a large part of the kernel functionality to servers, without which the system is virtually useless.

In the case of Helios, two vital servers must be loaded before the system can proceed: the task server and the loader. Without the task server, further tasks, including other servers, could not be started. The loader server is responsible for managing modules which are loaded into memory. Notably, it is responsible for dynamically loading and linking executed code, unloading unused modules, verifying module integrity and performing some simple startup manipulation.

In addition to these two servers, the nucleus contains the utilities, system and server shared libraries, which are used by the two servers. The resulting nucleus is illustrated in figure 3.5, with arrows in the diagram representing uses relationships.
As was mentioned above, the executive forms the inner-most part of the kernel. Despite it being at the lowest level of the operating system hierarchy, large sections of the executive are written portably in ANSI C [169], with only the most low-level functions implemented in assembler. Some other routines, which are vital to performance are also implemented in assembler. For instance, block memory operations, which account for 20-30% of time taken in most kernels by I/O intensive operations [110], are hand-coded in assembler. Unfortunately, no standard reference implementation of the portable part of the executive is available and consequently the entire code has to be recreated with each port, although past implementations may be used for guidance.

A large part of the executive is concerned with task scheduling. While any operating system requires some low-level support in order to swap processor state, Helios demands quite extensive scheduling support from the executive. This is because the Transputer [69], for which Helios was designed, implements task scheduling in hardware\textsuperscript{10}. 
The Helios kernel does make some assumptions, which dictate some parts of the executive implementation. The tasks which are waiting for the CPU have to be held in queues, one for each priority. The tasks which are running at the highest priority are not pre-emptable, which again imitates the behaviour of the Transputer hardware scheduler. This allows the kernel to use such tasks where it needs to perform a sequence of operations which should not be interrupted. Such operations are known as *atomic* and their interruption may result in a race condition.

A *race* occurs when the exact behaviour of the system is dependent on uncontrolled timing behaviour of some components. This is undesirable, since the exact timing of a software component depends on many factors and is not predictable, resulting in unpredictable system behaviour. In addition, the interruption of an atomic sequence can result in corruption. To avoid these problems, most systems use an *exclusion mechanism*, the most common of which include semaphores, monitors [171] and rendezvous [172]. The Helios approach is functionally equivalent to a single global semaphore, which is locked whenever a high-priority task is scheduled. Although this solution is less efficient than multiple locks, it does reduce system complexity and eliminates some possible errors.

Despite making some assumptions, the executive interface does attempt to achieve hardware independence wherever possible. For example, the executive may implement an arbitrary number of *physical* priority levels, which the kernel manipulates by using a set of 65536 *logical* priorities.

All currently available ports of Helios use a *multiquue round-robin* scheduler, much like the one used by UNIX [128,129]. Unlike UNIX which usually implements around one hundred priority levels, the number of levels used by Helios varies between two and eight. This relatively low number of priorities

---

10The Transputer is by no means unique in providing such high-level mechanisms. Intel 80286 and later processors also provide task scheduling, while other architectures, such as the intel 432 provide even more complex hardware support [170]
is justified by Helios' lack of a high-level scheduler.

**Shared library structure**

The benefits provided by shared libraries are well understood and mechanisms used for providing them are not new. However, efficient and flexible implementation of dynamic linking is still generating further research [173, 174]. Also, in Helios, the vast majority of system interaction takes place via the shared library mechanism, including the interaction between tasks and the kernel, which may therefore be viewed as just another shared library.

Under Helios an executable program consists of a sequence of modules. Each module either contains code or a reference to the code in a shared library. Each module has a slot number which is assigned at link time. Each shared library uses a reserved slot number. For example the Kernel, being a shared library, occupies slot 1. When the program is loaded into memory, the data block required by each module is allocated and the pointers to these blocks are put in the module table. The data blocks contain pointers to code in addition to the data, which allows the code blocks to be loaded at arbitrary locations in memory.

When accessing a data or code defined in another module, or in case of a shared library even in its own module, the address has to be calculated from the module table. This allows two programs to share the library code while using two separate data areas for the library as shown in figure 3.6. This combined with relocatable code generation allows the use of shared libraries even without hardware memory management unit.

Obviously this approach incurs at least one extra memory access overhead on every static data access and every inter-module call. This penalty is minimised by keeping the module table pointer for a task in a reserved register. In any case, an equally large overhead is caused by dynamic or late binding used
in some programming languages, such as C++ [175], where the dynamically dispatched functions are known as virtual. Other operating systems which provide shared libraries suffer similar penalties. For instance, modern versions of UNIX, such as Sun OS and IRIX, use a Global Offset Table much like the module table which requires the same indirection [176].

The object file format

An object file, in addition to storing the compiled machine code, has to hold information for the linker. This information is used by the linker in combining multiple object files into an executable. The linker simply uses the values of symbols defined in one object to alter the code in another object. Therefore,
every object file must contain definitions of its symbols and the description of which bits of data to alter in response to the values of other symbols.

Helios defines a standard object file format which is common across all architectures. The format used is known as *Generic Helios Object Format* (GHOF) and allows sharing of code for large parts of the loader server, assembler, linker and other utilities between different Helios implementations. The format is defined in reference [3]. In GHOF, the object file consists of a header followed by a sequence of items. Each item is either a definition of a symbol or a patch. *Patches* define the changes which have to be made to a particular bit of code and may be nested to considerably increase the expressiveness of the scheme.

A simple example of a hypothetical object file in readable form is shown in figure 3.7, together with pseudo-assembler from which it would be generated. The GHOF information is shown in sans-serif font, while literal data, such as strings and machine code, is displayed in *italics*. The nesting of patches is shown by indentation.

```
label1:
  DEFINE label1
  EXPORT label1
  load #10, a
  CODE load #10, a
  jump label2:
  PATCH_JUMP jump 0
  SYMBOL label2

(a) assembler

(b) GHOF
```

Figure 3.7: An example of assembler and generated GHOF.
3.4.7 Portability

While there are few universally accepted parallel system standards, Helios provides better than average portability by supporting a number of UNIX-like standards. Compatibility libraries allow emulation of standard UNIX *Berkeley Software Distribution* (BSD) calls and POSIX calls. The X window system is also supported on hardware which provides bit-mapped displays. All these facilities allow for relatively easy porting of serial applications between UNIX and Helios, and, in fact, a large proportion of Helios utilities are public domain UNIX utilities which have been compiled under Helios.

The Helios operating system provides support for distributed hardware environments, very much like the platform available for this project. The system meets all of the criteria imposed by the circumstances of this work and provides a software infrastructure under which applications may be developed and tested.
Chapter Four

The porting of Helios to the i860

This chapter describes the porting of Helios operating system to the i860 processor. For brevity, the ported version of Helios shall be called Helios/860. The description follows chronological order and it is assumed that the reader is familiar with the details of the system structure, covered in chapter 3.

The Helios source is written using ANSI C and assembler. The ANSI C is compiled using NorCroft C compiler to generate Helios-format object files for i860 prior to this project\(^1\). The assembler uses a macro preprocessor, which forms a part of the Helios distribution, known as Assembler Macro Preprocessor (AMPP) and an i860 assembler which was written before this project begun.

4.1 Writing and testing the executive

The porting of any operating system must include the implementation of its core hardware interface functions, which, by their very nature, are not portable between architectures. In the case of Helios, all the hardware interface code is concentrated in the executive, which forms the inner-most part of the kernel. Hence the first step in the porting of Helios to the i860 was the implementation

\(^{1}\text{Although various modifications and corrections had to be made during the project}\)
of the executive.

### 4.1.1 Scheduling

The Helios/860 port uses four priority levels, which allows some degree of prioritisation (for example, raised priority for the servers, which improves overall system performance) while maintaining relative simplicity. The processes which are waiting for the CPU are kept in separate queues for each priority

For each priority queue, there is a corresponding interrupt queue, which is used to hold processes made ready to run inside an interrupt handler routine. A separate set of queues is used because the interrupt may have occurred at a time when the main priority queues are in an inconsistent state. The contents of the interrupt queues are added to the content of the normal scheduling queues by the scheduler, just before the next process is scheduled. Since this is performed with interrupts turned off, no race problems can occur.

As mentioned in section 3, the highest priority processes are not pre-emptable and hence are used for atomic operations on kernel structures. However, in addition, care must be taken to avoid race conditions created by interrupts as described above, since high-priority processes are not immune to them. The executive is designed to ensure that the periods when the processor interrupts are turned off are kept to a minimum, minimising interrupt latency. This aim is achieved by:

- Using high-priority processes, wherever possible, instead of turning off interrupts to achieve atomicity.

- Disabling interrupts from a specific device, rather than all the processor interrupts, if the protected structure is only accessed by code responding to the one device.

---

2It is basically a prioritised round-robin scheduler, very much like the UNIX scheduler [128]
However, in some cases, the disabling of processor interrupts cannot be avoided. Probably the longest period over which interrupts are disabled is the time during which the trap handler code is saving or restoring the processor state, which is discussed further in section 4. Processor interrupts also have to be turned off to avoid race conditions. For example, consider the device handling code, illustrated in figure 4.1. Unless the processor interrupts are turned off before the device interrupt is enabled, the device can complete the request and interrupt before the suspend request, which would then suspend the process indefinitely.

```plaintext
oldint := disable_processor_interrupts ()  
this statement prevents 
a possible race

enable_device_interrupts ()  
suspends current process 
and enables processor ints

suspend (current_process)  
this reached once process is restarted after interrupt

restore_processor_interrupts (oldint)  
restore original condition 
of processor interrupt mask
```

Figure 4.1: Avoiding a possible race condition in I/O code.

The processes in the i860 implementation have six valid states. These are illustrated in figure 4.2 and consist of:

- **Starting**: initial state of the task. Each task assumes this state at the time of its creation, to allow all the manipulation of process state to be performed by the scheduler.

- **Running**: task currently executing.

- **Ready**: task currently waiting for the CPU, linked in one of the priority queues.

- **Waiting**: task sleeping for a time period.

- **Blocked**: task waiting for an I/O operation to complete.
Interrupted task is processing an interrupt.

![State Transitions](image)

Figure 4.2: Valid task states in Helios/860.

Perihelion advise [2] that, on processors which support a protected supervisor mode, the entire kernel be run in this mode. However, the protection afforded by the hardware is not effective against problems with code run in supervisor mode. Hence, in order to maximise robustness and allow better debugging support, only some sections of the executive operate in supervisor mode. This does result in more traps, as control is being passed within the executive, but since Helios inflicts relatively small trap penalties, this was not deemed to be a problem.

### 4.1.2 Link I/O

The principal communication between the i860 and the outside world takes place through the two link adapters. There are two separate device drivers for these adapters: a simple polling driver used during boot time and a combined polling-interrupt driver used under normal operation.
The device driver used during system boot blocks further execution until the entire message is transferred. This driver operates correctly before the majority of the system (in particular the interrupt handlers) are initialised. After the kernel is started, the main device driver is used instead. This driver attempts to transfer each byte of the message for a fixed number of times, before giving up and suspending the current process until an interrupt is received. Care has to be taken to avoid race conditions as described in section 4.

4.1.3 Trap Handler

The i860 processor uses a single trap handler to process all interrupts, traps and exceptions. The trap handler is written predominantly in ANSI C, with entry and exit routines coded in assembler. Since some early versions of the Numbersmasher card generate timer interrupts 1250 times every second and the i860 has a large amount of state to save, care has to be taken to ensure that the interrupt handling overhead does not slow the processor. Fortunately, the RISC approach used by the i860 mandates that all state saving and restoration be done in software. While this does require large amounts of assembler and is somewhat slower than the hardware equivalent, it allows more flexibility in saving partial CPU state.

Most RISC processors are very fast when executing linear code, but have varying degrees of difficulty with control transfers, ranging from jumps to interrupts. Exception and interrupt handling pose a particular problem, since the entire processor state has to be exchanged. This problem has been discussed in section 2. The i860 is particularly slow in saving the full processor state. Fortunately, Helios uses a single-address space for all of the tasks, which means that the extremely expensive cache flushes can be avoided for normal context.

---

3Using Motorola terminology, where an interrupt is an external hardware signal to the processor, a trap is a synchronous or deliberate software interrupt and an exception is an asynchronous software interrupt, as a result of a instruction fault.
The porting of Helios to the i860

switches.

The original Helios/860 trap handler saved all the integer and some control registers, but none of the floating-point state, since the NorCroft C compiler which was used with the executive did not generate floating-point instructions. However, this was later expanded to include saving the remainder of the state, including all floating point registers, pipelined operations as well as providing some extended floating-point support, in anticipation of full floating point support.

The extended floating point support is necessary, because the i860 CPU does not implement full IEEE standard 754 floating point handling [177] in hardware. Instead, in a number of special cases, an exception is generated and the software is required to complete the processing. For example, denormal\(^4\) operands for some of the floating point operations cause exceptions and need to be handled by the trap handler software [178]. Special processing is required by the i860 for these cases, well as several functional deficiencies of the hardware. The trap handler design was obtained from Microway Inc., although it had to be rewritten for use within Helios.

Many of the more recent processors consider the performance and complexity effects of precise exceptions to be unacceptable. These architectures [47,124] implement imprecise arithmetic exceptions\(^5\), where an exception caused while processing a pipelined instruction is not processed until later in the pipeline. This avoids having to save and restore partially processed instructions. Unfortunately, this does cause severe problems with debugging, since some instructions following the fault may be executed by the time the exception is received.

\(^4\)That is, a floating point number which are valid, but are not in the standard representation format mandated by IEEE 754, which permits multiple valid representations for some numbers. A denormal number may be normalised without difficulty.

\(^5\)Imprecise exceptions are not new. They were used by the IBM 360/91 computers in the 1960s [179]
4.1.4 Ensuring correctness

The executive was originally written using ten ANSI-C and four assembler modules. It consists of approximately 1700 lines of C source and 1200 lines of assembler. Hence, considerable effort has gone into ensuring that the resulting code is readily readable and easy to debug. To that effect, the C source contains over 600 comments including, in particular, a comment header on every file and function, as well as every non-trivial component of a structure or union.

Like other recent Helios ports, the assembler uses the macro pre-processor AMPP to maximise readability. Common multi-instruction operations, for example loading a register with a constant value, are implemented using macros. Further macros are used to hide the bug work-arounds and restrictions imposed by the processor architecture, as described in section 4.

When the system was upgraded to Helios 1.3 (see section 4), a folding version of the Microemacs editor was obtained from Perihelion. A folding editor allows the representation of a file as a hierarchy of regions, each of which may be shown in its full, expanded form, or compacted to a single line title. This feature adds greatly to the readability of source, which can then be viewed in a top-down manner. Both the C and assembler modules of the executive were converted to structure the source using this facility. An extract from the resulting code structure is shown in figure 4.3, with nesting shown by indentation. Folding is used to organise the code, by folding source of each function into a block named by the prototype of the function and then organising these blocks into functional areas. While not as readable as literate programming\(^6\), the

\(^6\)Literate programming is a technique where the documentation about a piece of code is embedded in the code. This provides a much more readable program than simple comments. The term was coined by D. Knuth during the development of his \LaTeX{} and METAFONT programs using \textit{Web} [180].
Headers & preprocessor definitions...
Prototypes...
Initialisation...

Process Control...
void Suspend (SaveState *)...
void Resume (SaveState *)...
void Restart (SaveState *)
{
xroot->queues[p->pri].tail->next = p;
xroot->queues[p->pri].tail = p;
}

Timing functions...

Figure 4.3: Structure of a part of the executive using the folding editor.

resulting source is considerably easier to absorb.

In order to ensure that any problems with the executive code are detected as soon as possible, numerous assert statements were placed in the code. These took various forms: a plain ASSERT which has similar semantics to the ANSI C assert, an ASSERTMEM, which partially verifies the validity of a memory address or pointer and various asserts tailored to particular structures, which check the invariants and assumptions specific to their particular argument. While the mechanism used is not as comprehensive or efficient as some others, for example the compile-time verified invariants described by Rosenblum [181], it is adequate for this application.

The executive code is written to ensure that every function checks, wherever possible, the validity of all of its arguments, as well as any global elements it
uses. Further checking is also performed at critical points. For instance, the scheduler verifies the validity of a saved process before scheduling it. Like the ANSI assert, all the checking features are implemented through macros and can be disabled resulting in no run-time overhead for optimised versions of the code.

The use of run-time checks in real-time programs can potentially create some very subtle problems resulting from small timing changes which alter the system behaviour. While theoretically difficult to avoid, in practice such difficulties are very rare, since the time taken by most debugging is relatively small as is the amount of timing-sensitive code.

4.1.5 Configuration management

The basic hardware platform for the Helios/860 implementation is the Numbersmasher i860 card. However, this card exists in several variants, all of which had to be supported. The most significant variations between cards are:

1. Memory size varies between 8 M bytes and 32 M bytes, with the hardware able to support up to 128 M bytes in the future.

2. Timer interrupt frequency is either 100 Hz or 1250 Hz.

3. Different masks of the i860 XR chip are used, which possess different hardware problems, described further in section 4.

In order to allow all variants of the board to be used with Helios, particularly in configurations where multiple versions are mixed in a single network, the first two parameters in the above list are automatically detected on boot.

The memory size is detected by assuming that the amount of memory is a power of 2 between 1 M byte and 128 M bytes and that the memory is decoded only on the least-significant bits. The detection works by storing a known
value at a low memory location and then finding the next power-of-two address which contains this value. For example, assuming 8 M bytes of RAM, after writing a magic number\(^7\) (for instance 12345678) to word at address 0, possible memory sizes are checked by reading words at addresses 1 M byte, 2 M bytes, 4 M bytes and 8 M bytes. This last read will return the original value (12345678 in this example), because the memory decoding only pays attention to the bottom 23 bits of the address. To avoid problems with initial value of a memory word accidently containing the magic value, the value is then modified (by writing to address 0) and checked again (by reading from address 8 M bytes). Care has to be taken to flush the data cache after memory modifications. However, since the auto-detection takes place only at boot time, the cache penalties do not impact the performance of the system. The interrupts frequency is detected by a special-purpose routine, which times the number of loop iterations executed while polling the clock interrupt bit. The routine then calculates the interrupt rate in terms of the number of executed instructions and converts that to the frequency by assuming the chip clock rate.

While the memory and interrupt frequency can vary between boards, other parameters are not expected to change so dynamically between different Numbersmasher devices. However, in order to make the porting to other i860 accelerators easier, these less variable parameters are defined in a configuration file, `iconfig.h`, using the C preprocessor macros. They include:

- **Board manufacturer** permitting future selection of different hardware platforms.
- **Board version** which allows the selection of a particular configuration, including memory size and clock rate, in a case where the auto-detection fails or is inappropriate.
- **Chip frequency** which is used to calculate the timer interrupt frequency and cannot be easily obtained from any other source.

\(^7\)That is an arbitrary number picked at random by me – see [182] for further explanation.
Executive options which are described in more detail below.

Memory layout in particular physical addresses of hardware devices and where they should be mapped in the virtual address space.

The executive options include the following:

Debugging options including control of the assertions described in section 4 and the amount of debugging generated.

Link I/O selection of polling or interrupt driven approaches.

Protect nucleus from being modified, using the MMU.

4.1.6 Solutions to functional deficiencies

The complexity and novelty of the i860 XR chip resulted in many early versions being hampered by various problems. This is because, despite embodying the RISC philosophy, the i860 is a very complex device using over two million transistors, unlike some other RISC designs. The solution to most of these problems involves replacing a single assembler instruction with code which checks for the presence of a particular problem and, if necessary, correcting it in software.

The assembler preprocessor (AMPP) was used to retain readability of the assembler code, while at the same time providing the complex structure of version-dependent workarounds. Wherever possible, macros corresponding to the problematic instructions were defined. The macro names were formed by changing the corresponding instruction name to upper case. These macros were then used in the assembler source in place of the assembler instructions. Unfortunately, it is not possible to make this process transparent to the programmer. Defining macros which completely replace the faulty instructions is

---

8Such as the Tiny RISC, which uses only 12,000 transistors [183]
The porting of Helios to the i860

insufficient, since the macros can not be used everywhere where an instruction can. For example, since the macros can expand to multi-instruction sequences, they cannot be used in delay slots.

In order to minimise the effort involved in customising the code to a different version of the processor, a two-stage translation scheme is used. In the relevant preprocessor include file, i860ih.m, a macro corresponding to the required CPU mask version is defined. Then a list of macros for the bugs is defined, using the bug numbering scheme developed by int@l in their documentation. With the help of another macro, a relatively high degree of readability is achieved, as can be seen in figure 4.4. The figure shows a (fictional) bug in the ld.c instruction, numbered BUG_1, which is active in processor masks before B1. If compiling the code for a different processor version, the user has to only change the definition of STEPPING, with the correct work-around being applied. The programmer must obviously ensure that he uses LD.C in place of ld.c.

While many bugs were bypassed using this mechanism, the subtlety of many problems resulting from complex interplay of instructions made them much more difficult to solve. The cache flush function, which was changed a few times in int@l's documentation, was upgraded three times before achieving desired reliability.

4.1.7 Debugging the executive

Like almost any piece of software⁹, the initial executive code contained various bugs, many of which have remained undiscovered. Bugs in operating system kernels are notoriously hard to locate for a number of reasons:

⁹Notable exceptions being some of the work done on combination of semi-formal specification and "cleanroom" approach to development done at IBM [184]
In order to get the debugging information out, some basic communication routines must be working. If, for whatever reason, they fail, the debugging information will never be transmitted.

By its very nature, the kernel is very multi-threaded, with frequently occurring interrupts. Under these circumstances it is quite hard to maintain a consistent flow of debugging information about what is happening.

The difficulties involved in debugging the kernel were one of the reasons for attempting to ensure correctness, as described in section 4. However, significant debugging support is still essential.

Helios provides a mechanism, which is extremely useful for debugging, namely the debug message. These messages are handled in a priority manner by the kernel and displayed in a separate window by the IO server. However, this mechanism is only functional once the kernel has initialised itself and returned an acknowledgement to the IO server. Therefore, in the very early stages, the second link adapter on the Numbersmasher board was connected to a transputer card in a second PC-compatible. All data received via this link was displayed on the screen and logged to a file. This enabled initial debugging to proceed.

However, in some cases, the use of link debugging is not possible. This is because Helios provides many real-time facilities and the effect of the relatively slow debugging output could alter the system behaviour. In particular, every message passing primitive has a timeout value. The debugging message output has to take control of the processor, until the message is fully output. Otherwise, code which executed in parallel with the debugging could crash the processor before the transmission is complete. However, the debugging holding up the execution for a significant period of time can result in some messages timing out.

Although Helios is not strictly speaking a real-time system, since it does not guarantee interrupt latencies, it does utilise some real-time mechanisms.
To relieve these problems, a post-mortem link debugger has been developed. This debugger allows the examination of the memory after the processor is reset. Upon a reset, the processor starts executing the bootstrap code from ROM, without altering the memory. The debugger is written in C++ and has numerous helpful features. As well as allowing examination of memory, the debugger has more advanced facilities, including:

**disassembly** The debugger displays the memory, word at a time, in three formats: hexadecimal, ASCII and i860 disassembly.

**stack backtrace** During some stages of development, support for debugging was added to the compiler. This allowed the debugger to display a backtrace of the stack in symbolic form. This support was later removed due to recurrent problems resulting from complex interactions with the system.

**displaying structures** The debugger is aware of the format of many of the structures used within the executive. Given their address, it can display the values of the fields in a symbolic form. The structure description format is generalised to allow easy extension. Each structure is described by an array of strings. Every string describes the name of a word in the structure. The array is terminated by a NULL pointer. To allow more complex formats, a convention using the first character of the name is used. If the first character is a >, then the word is a pointer to another structure of the same type as the one being displayed, for example a Next pointer in a linked list. A + in the first character slot denotes a pointer to a different structure. The address of this structure’s describing array is then stored in place of the next string. An example of this can be seen in figure 4.5. A third special form, starting with a # is also supported to allow bit-fields. Like the + form, the name has to be followed by a pointer, this time to a bit field describing structure. An extract of the code can be

---

11 This code is valid, because ANSI-C mandates char * as a generic pointer, in order to retain compatibility with K&R C
The bit-field describing structure consists of an array of entries, each with a name, mask and value. If the mask is non-zero, the field is only active if the bitwise AND of the word with the mask is equal to the value, which is useful for decoding values of bit fields. If, on the other hand, the mask is zero, the field is active if all the bits of the value are set in the word. This variant is used for decoding value of enums. An example is shown in figure 4.6.

debugging of chained cards To allow debugging of more complex systems, in particular multiple Numbersmasher cards which are linked together using the second link adapter, the debugger supports addressing of chained cards. For that purpose it uses an arrangement of C++ classes (see figure 4.7, which allow the local and remote link adapters to be accessed using the same interface. A local adapter access will result in a simple I/O instruction, whereas the access to a remote adapter will take place through a number of memory reads and writes to the location of the memory mapped link registers on the remote card. The process is illustrated in figure 4.8.

4.2 Altering the IO server

Although, as mentioned in section 2, the Numbersmasher card starts the i860 by executing a boot ROM which emulates most of the Transputer boot protocol, modifications had to be made to the IO server to accommodate the differences between the i860 and the Transputer. The most important alterations were due to the different layout of memory, which resulted in the code being loaded to a different address. The Helios boot memory map can be seen in figure 4.9.

In addition to these changes, in order to speed up the uploading of the nucleus, the original boot protocol was modified. The original protocol required the
sending of 9 bytes (1 byte protocol and two words denoting the address and data) for every word written to memory. This was extremely inefficient.

Some thought went into accommodating both the original and modified versions of the boot code and especially the possibility of the new code being used in the i860 EPROM. The default Transputer protocol is based on transmitting a one byte protocol descriptor, followed by the data. Valid protocols are described concisely in table 4.1. The only request which requires a reply is a read, which receives the four bytes of the word read\textsuperscript{12}.

The modified protocol takes advantage of the fact that the default protocol only defines behaviour for protocol bytes 0 and 1. The new protocol uses other byte values to provide extra functionality. These extensions are described in table 4.2. The \textit{version number} function returns a four byte version number, with major number in the top two bytes. The versioning scheme used is similar to that used by many UNIX shared libraries, including those of SunOS and Silicon Graphics' IRIX [185], a major version number increment signifies backward-incompatible interface changes, while minor number changes retain backward compatibility. The use of this function allows the IO server to identify which boot ROM version it is talking to. If the code currently active does not implement this protocol, as for example the original version of boot EPROM, the server times out the request after one second. To ensure that the boot EPROM remains in a valid state for further interaction, the server completes the request as if it were a read request. This approach is possible because the boot EPROM assumes that any non-zero protocol byte signifies a read. The result of this dummy read is obviously ignored. If an older version is detected, a copy of the updated code is uploaded into RAM and used for future transfers.

\footnote{Since the Transputer, the PC and the i860 are all little-endian, no problems with byte ordering arise. The IO server code is, of course, capable of being configured for big-endian hosts.}
4.3 Extending the functionality of the kernel

In addition to the porting of existing Helios code to the i860, some modifications were made to enhance the operating system and make a fuller use of the provided hardware. In particular, the i860 is equipped with an on-board MMU, which is not used by standard Helios 1.2.1. This MMU is used by Helios/860 to provide basic memory protection and hence a more robust environment.

A diagram of the details of the memory mapping used can be seen in figure 4.10. A full description of it may be found in [♦4]. The MMU mapping is used for two primary purposes:

- Mapping the trap handler address to real memory. The i860 requires its trap handler code to be installed at address FFFFFF00h. However, the Numbersmasher card does not decode any RAM at this address (in fact, it has to decode the boot EPROM there, since the trap handler is also called on reset). The virtual memory re-mapping is used to map this virtual address to the physical address of F0001000h.

- Protecting the code from being overwritten. This includes the nucleus code and the uploaded code for the executables, which is protected from accidental modification after the loader finishes altering it. This approach provides more reliable error trapping than the usual Helios solution of using a background checksum verifying program described in section 3.
4.4 The rest of the system

The other element of the Helios system which required substantial modifications was the compilation tools. These consist of the ANSI-C compiler, in this case the NorCroft compiler and an assembler and linker which were hand-modified from the MC68000 port of Helios. The compiler, assembler and linker were converted before the start of this project. However, considerable effort went into fixing various problems and enhancements. While numerous bugs were found and fixed, some problems escaped detection and required work-arounds. For instance, extern declarations which are inside a block are handled incorrectly and require moving to file scope and resolving any name clashes.

Some other problems, proved time-consuming in their correction. For example, it was found that calls from assembler to C functions were not executed correctly. An initial investigation found that the compiler generates labels which point one instruction before the start of the relevant function. This was initially "fixed" by adding an assembler macro, i860br_toC, which compensated for this problem [♦5]. Later [♦6] it was discovered that all compiler-generated branches use a delay slot and the compiler behaviour compensates for the machine-code counting branch offsets from the delay-slot instruction. Therefore, the i860br_toC macro was removed and the assembler modified to output appropriate patches for its delay-slot branches.

One area requiring a large amount of work, was the modification of the compiler for building shared libraries. This required altering the following parts of the compiler code generation:

- Data should be allocated only for local variables. Global variables have data allocated in a central assembler module, which ensures the correct
structure and ordering of the data area.

- Global functions should generate the export code, but not reserve the data required. This is so that the exact ordering can be established by the assembler module.

- External functions should not be imported, as a common version of the import code is present in the assembler module.

- Other cosmetic changes, such as suppression of the module header and trailer, were also necessary.

The above modifications are intended to allow the assembler module full control over the layout of the module's data area. Every shared library needs to be linked with such an assembler module and careful modification of this module allows future versions of the library to retain backward compatibility. A more detailed description of the work required for supporting shared libraries can be found in [♦5].

4.5 Updating Helios to version 1.3

Towards the end of 1992 a decision was taken to upgrade the system to Helios version 1.3. This involved some substantial modifications. The most essential changes included the move from the custom-built assembler and linker to a Generic Helios Assembler and Generic Helios Linker. These two packages were developed for the non-Transputer versions of Helios, and in particular, supported the TMS 320C40 version of the operating system.

The addition of i860 support for the assembler was relatively straight-forward. The most awkward job was the generation of the grammar for the parser. The generic assembler uses the UNIX parser tools: LEX, the lexical analyser and YACC, the parser generator\[^{13}\] [188]. In order to maintain compatibility
with other versions of the assembler, a decision was made to stick with this approach. However, the instruction set of the i860 results in a very large grammar, as most instructions can have a suffix specifying their precision and some additionally have a prefix.

Two solutions to this problem were possible: the lexical analyser configuration could have been modified to split the instruction into multiple words: the core and the prefixes/suffixes. However, this would require changes to the lexical analyser specification, which would make the i860 version of the assembler incompatible with other variants. It was therefore decided that, in order to maintain full compatibility with the standard approach, the decoding of the numerous instruction formats was to be done by the parser. However, this in turn posed the problem of generating the extensive grammar required, while retaining a low error rate.

This problem was solved by automatically generating the main part of the grammar using a stand-alone program. The program in question was written in PERL\textsuperscript{14}, which is extremely efficient at processing text files.

A description format was developed, which allowed the construction of an instruction name from a prefix, core and suffix, each of which is specified separately. An example of the input format and the resulting output file is shown in figure 4.11. In the figure, the two specified instructions (ld and fadd) are expanded to six resultant forms. The full grammar expands the specified 89 descriptions to the complete set of 585 instructions.

\textsuperscript{13}To be exact, it uses the public domain versions of these two programs, namely FLEX [186] and BISON [187]

\textsuperscript{14}The Practical Extraction and Report Language is written by Larry Wall at JPL [189]
4.6 Adding support for an efficient floating-point compiler

Although the actions described so far represent progress, they do not result in a useful system. This is because the performance of the i860 cannot be exploited without a compiler which is capable of optimisation. Some work was done to slightly improve the performance of the NorCroft compiler, namely to attempt scheduling a useful instruction in the delay slot of the branches. This attempt was successful. However:

- No major performance gain was achieved. In order to attain near-optimal performance, many much more sophisticated optimisations were needed. It is well known that RISC architectures require sophisticated optimisers to achieve maximum performance [190].

- The compiler is a large and complex application. While the author has gained some familiarity with its structure, necessary to complete the modifications and enhancements performed, lack of support and shortage of time prevented any major alterations to the compiler structure.

In order to achieve the rated performance of the i860, a decision was made to move over to the compiler produced by Microway Inc. This coincided with the upgrade to generic Helios assembler and linker and a corresponding change in the object format for Helios/860. The new object format uses a different set of patches, which form a better match with the i860 architecture.

The Microway C/C++ compiler is written in an enhanced version of Pascal, which may be compiled with the Microway Pascal compiler. Unfortunately, although the back ends for the two compilers are very similar, they are distinct. Due to shortage of time, the Pascal compiler was never converted to generate
GHOF, and hence it was impossible to compile Microway C/C++ compiler for native use under Helios. Throughout this project, the C/C++ compiler has been used as a cross-compiler and executed under Microway’s OS860 environment.

4.6.1 New GHOF patch structure

Every object file under Helios uses the Generic Helios Object Format, described in section 3. As described, the objects contain embedded information about modifications which need to be made to the code by the linker. These modifications (known as patches) were performed in the old version of the linker using an ad hoc mixture of MC68000 patches and the two special purpose patches described in table 4.3. The upgrade to Helios 1.3 resulted in a more regular and robust patch structure.

The new patches always consist of a value-type patch applied to a value-calculation patch, as listed in table 4.4. For example, the task of loading the value of a 32-bit symbol into a register (r30) is normally performed by the two instructions:

```
  orh  high, r0, r30  ;; load the top 16 bits of address
  ld.1 low  (r30), r30  ;; load the value of symbol
                      at address
```

where the low and high values stand for the top and bottom 16 bits of the value of the symbol respectively\(^1\). Using the old patch system, this would be encoded as:

```
  orh  M68K.SHIFT (16, symb) r0, r30
  ld.1  i860.LOW (symb) (r30), r30
```

\(^1\)Please note that the above example is actually simplified, as the i860’s address calculation mechanism necessitates some added complexity in the first instruction patch. However, this is not central to this discussion hence is omitted.
where `symb` is a patch resulting in the value of the symbol. The new style patches use an i860-specific patch, producing:

\[
\begin{align*}
\text{orh } & \text{i860.UVAL (i860.HIA (symb)) } r0, \ r30 \\
\text{ld . l } & \text{i860.SVAL (i860.L0 (symb)) (r30), r30}
\end{align*}
\]

Initially, the difference between the two approaches may not be apparent, except for the obvious verbosity of the new solution. However, in a number of less common situations, the new approach allows the detection of link-time errors, which would otherwise remain hidden. For example, a similar load which assumes a 16-bit symbol address (which is actually generated by the compiler) would use:

\[
\begin{align*}
\text{ld . l } & \text{i860.SVAL (i860.VAL (symb)) (r0), r30}
\end{align*}
\]

If the symbol value is larger than \(2^{16}\), the linker will trap the resulting error at link time.

Due to lack of time and various assumptions, mostly related to alignment, the Helios source was not converted to use the Microway compiler. This approach does have its benefits. Since the NorCroft compiler does not use floating point, the floating-point part of processor state does not have to be saved for every interrupt. This would not be true if Microway compiler was used to build the kernel. However, some parts of the Helios nucleus, notably much low-level floating point handling, had to be converted to the new GHOF format. Therefore a conversion program was written to translate from the old to new patches, allowing the NorCroft compiler generated objects to be linked with the new assembler output.
4.6.2 Compiler conversion

The Microway environment uses the *Common Object File Format* (COFF), which is commonly used by the majority of UNIX systems\(^{16}\). While Helios 1.3 provided a GHOF assembler and linker, the Microway compiler had to be modified to output in GHOF instead of COFF-compatible format.

The Microway C/C++ environment follows the UNIX tradition of separating the compiler and assembler stages. Therefore, the compiler generates assembler text, which is then converted to the binary in a separate stage\(^{17}\). This is quite convenient, for a number of reasons:

- Conversion and, in particular, debugging is far easier when the output consists of text assembler files, which may be viewed and operated on by available tools.

- The difference between COFF and GHOF is lesser for the text assembler than for the binary files, so less work has to be done to convert the output.

The actual changes which needed to be made to the compiler output are described in detail in appendix B. The document was produced since originally the conversion work was to be done by Microway Inc, due to their reluctance to release the required sources. However, lack of available man-power had prevented that and the work was performed by the author.

It may appear that the compiler conversion consisted of simply altering the names of the assembler directives output. Unfortunately, this was not the case. The structure of the code generation and output stages of the Microway

\(^{16}\)Although it is being superseded by *Executable and Linking Format* (ELF), which is specified by the System V Application Binary Interface [191]

\(^{17}\)Unlike, for example, the NorCroft compiler available under Helios/860, which can only output binary object files
The porting of Helios to the i860 compiler were designed for the style of directives and structure of code used by the COFF assembler.

Some code which consisted of a single instruction in COFF, expanded to multi-instruction sequences under GHOF. If the optimisation stage was unaware of this expansion, it would place such an instruction in a delay slot of a branch, making it impossible to generate correct GHOF output. Therefore, the expansion of instruction sequences had to be performed at a fairly early level of code generation.

The assembler output modification were also not as straight-forward as at first appeared. The COFF patch syntax was attached to the operands, whereas GHOF patches enclosed the entire instruction. For example, the COFF instruction:

\[ \text{ld.1 hi\%sym(r0), r20} \]

is equivalent to:

\[
\text{patchinstr \ (PATCHI860L0,}
\text{\quad datasymb (sym),}
\text{\quad ld.1 0(r0), r20)}
\]

The instruction line is generated by various functions, whose structure had to be significantly altered to implement the drastic change in syntax.

### 4.6.3 Microway Libraries

Once the Microway compiler has been altered to work with GHOF tools, the Microway libraries had to be converted. The Microway software splits the libraries in a traditional UNIX fashion into: \texttt{libc}, containing all integer ANSI-defined functions and \texttt{libm}, containing all floating-point math functions. The
integer library was ported first. This involved some minor work on interfacing lowest levels of libc to Helios' Posix and C libraries, combination of headers from the two environments and some Helios modifications.

The last point was most time-consuming. The Microway compiler uses the quad floating-point operations and hence requires a 16-byte alignment for its stack and data segments. However, NorCroft compiler, which does not generate floating-point, only ensures a 4-byte alignment for its stack. Therefore, wherever a Helios function compiled with NorCroft calls a Microway function, its has to do so via an alignment-fixing wrapped coded in assembler.

The next step involved compiling and testing the math library. Unsurprisingly, this uncovered a number of bugs in the compiler modifications. These were slowly removed and the Plum-Hall compiler test-suite was used to verify the compiler changes. Unfortunately, due to lack of time, only one part of Plum-Hall was successfully used to verify the compiler and libraries.

## 4.7 Use of OASIS under Helios

The OASIS application was written for use with PVM. Further details of its implementation are outlined in section 7. The system consists of two parts: xoasis, which is the user interface and master, and timsim, which is the simulation worker. The problem with running OASIS under Helios was the requirement of X window system by its user interface. While a version of X11 has been ported to Helios, it was beyond the scope of this project to provide it under Helios/860. Therefore, xoasis had to be executed on the Linux front-end machine, while timsim run on the Numbersmashers under Helios.

To facilitate communication between the two sides, the Helios server, which already provides Helios tasks with access to the peripherals on the front-end, was extended to support a communication channel. From the Helios side, the
The porting of Helios to the i860

link appeared as another server, while on the Linux side it was implemented by a UNIX-domain socket, named /tmp/helios.i860.

The PVM implementation for Helios/860 is based on the PVM/860 port described in chapter. The nxlib.c module, which implements the emulation of NX2 library in terms of lower-level primitives has been largely reused. Since the Helios server has been modified to support only a single communication channel between the Helios and Linux sides, a multiplexing program pvmmux is used on the Helios side to forward requests to the processors. A diagram of the overall structure of communication under Helios may be seen in figure 4.12. The Helios side is labeled HEL860, while the Linux end is named HEL860LINUX.

The overall structure of the code is very similar to the PVM-only version, as may be seen by comparing figure 4.12 with 6.4. In addition, a large proportion of pvmux code is also shared between PVM/860 and PVM under Helios. This overlap is a result of the decision to maximise code reuse and hence reduce errors.

The Helios parallel operating system was successfully ported to the i860 hardware and, in particular, the Numbersmasher boards. The system provides an environment which allows up to three i860 accelerators to be used in parallel and makes effective use of the available resources. Floating point calculations are supported allowing real applications to be implemented and executed using the system.
The porting of Helios to the i860

-- stepping numbers define the chronological order of steppings
_defq 'STEP_A2 2
_defq 'STEP_B1 11
_defq 'STEP_B2 12
_defq 'STEP_NOTYET 999

_defq 'STEPPING STEP_B2 -- stepping used for the code

-- a utility function, which returns TRUE if the STEPPING number is less than the value of its argument
_defq 'fixed[ver]
 [ _lt STEPPING [eval [STEP_$ver]] ]

-- specification of the stepping in which the bug was fixed
_def 'BUG_1 fixed B1
_def 'BUG_2 fixed B2

_if BUG_1 [
 _def 'LD.C ['from 'to] [
   work-around for bug
  ]
 ] [
 _def 'LD.C ['from 'to] [
   ld.c from to
  ]
 ]

Figure 4.4: Hardware problems handling assembler preprocessor file.
The porting of Helios to the i860

Actual structure

```
// The SaveState structure
struct SaveState {
    struct SaveState * Next;
    word Priority;
    word Wakeup;
    word State;
    struct TrapData * TrapData;
};
```

Debugger's description

```
// Description of SaveState structure
char *struct_ss[] = {
    "Next",
    "Priority",
    "Wakeup time",
    "State",
    "Trap Data",
    NULL
};
```

Figure 4.5: Structure descriptors in the debugger.

Enumeration

```
// An example State enum
define STATE_IDLE = 1,
#define STATE_RUNNING,
#define STATE_WAITING
```

Debugger's description

```
// An example State description
BitField bit_state[] = {
    {"Idle", 0x03, 0x01 },
    {"Running", 0x03, 0x02 },
    {"Waiting", 0x03, 0x03 },
    {"INVALID", 0, 0xffffffffc },
    { NULL, 0, 0 }
};
```

```
// Description of the bit field
BitField bit_psr[] = {
    {"IM", 0, 0x0010},
    {"PIM", 0, 0x0020},
    {"U", 0, 0x0040},
    {"ITT", 0, 0x0100},
    { NULL, 0, 0 }
};
```

Figure 4.6: Bit field descriptors in the debugger.
The porting of Helios to the i860

Figure 4.7: Class inheritances in the i860 debugger.

Figure 4.8: Debugging a remote i860 card.

<table>
<thead>
<tr>
<th>Message bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>0 ← Address →      ← Data →        Write word to memory</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0            ← Address →     Start execution</td>
<td></td>
</tr>
<tr>
<td>1 ← Address →                                Read a word</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Transputer boot protocol
Figure 4.9: Boot time memory map for Helios/860.

<table>
<thead>
<tr>
<th>Message Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td></td>
</tr>
<tr>
<td>17 Address  Size Data...</td>
<td>Long write (a block of words)</td>
</tr>
<tr>
<td>33</td>
<td>Return version number</td>
</tr>
</tbody>
</table>

Table 4.2: Boot protocol extensions

<table>
<thead>
<tr>
<th>Patch name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i860.LBR0FF</td>
<td>Long branch offset (bottom 26 bits)</td>
</tr>
<tr>
<td>i860.LOW</td>
<td>Low 16 bits of the value</td>
</tr>
<tr>
<td>M68K.SHIFT</td>
<td>A Motorola 68000 patch which bit-shifts its argument</td>
</tr>
</tbody>
</table>

Table 4.3: Original Helios/860 patches
Figure 4.10: The MMU memory mapping under Helios/860.
### Valid suffixes and prefixes

*Fields are: name, suffix/prefix, format, value*

<table>
<thead>
<tr>
<th>Format</th>
<th>Prefix/Suffix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>s, b</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>s, s</td>
<td>2</td>
</tr>
<tr>
<td>dim</td>
<td>p, d</td>
<td>4</td>
</tr>
<tr>
<td>nodim</td>
<td>p</td>
<td>0</td>
</tr>
<tr>
<td>ss</td>
<td>s, ss</td>
<td>0</td>
</tr>
<tr>
<td>dd</td>
<td>s, dd</td>
<td>3</td>
</tr>
</tbody>
</table>

### Valid combinations of prefixes/suffixes

*Fields are: name, suffix/prefix, names*

<table>
<thead>
<tr>
<th>Integer</th>
<th>Byte, Short, Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>dual</td>
<td>p, dim, nodim</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Float</th>
<th>Suffix/Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss</td>
<td>s, ss</td>
</tr>
<tr>
<td>dd</td>
<td>s, dd</td>
</tr>
</tbody>
</table>

### Instructions

*Fields are: name, prefixes, suffixes, opcode value*

- `ld`, int, MEM_REG (50000000, SS)
- `fadd`, dual, float, FLT_INST (48, 0, 0)

Figure 4.11: i860 instruction specification for the generic Helios assembler.
### Patch name | Description
--- | ---

**Value type patches**

- i860_UVAL: Unsigned 16-bit value
- i860_SVAL: Signed 16-bit value
- i860_OFF: 16-bit offset
- i860_BR: 26-bit branch offset
- i860_IMM: 5-bit immediate for b1a

**Value calculation patches**

- i860_VAL: entire value
- i860_HI: top 16 bits of the value
- i860_LO: bottom 16 bits of the value
- i860_HIA: top 16 bits of the value, adjusted for the address calculation signed offset

---

*a This does not modify the value. The patch is provided so that an i860 value patch is always followed by an i860 calculation patch

Table 4.4: Modified Helios/860 patches
Figure 4.12: The PVM emulation under Helios/860.
Chapter Five

The Parallel Virtual Machine Environment

This chapter describes an alternative to providing a parallel operating system, namely a Message Passing Interface (MPI) with some extended management facilities. The commonly available MPI systems are described and the selected interface is described in detail.

5.1 Introduction

The use of parallel computers and, particularly, massively parallel machines is becoming increasingly widespread. These parallel systems invariably use distributed memory, since shared memory does not scale well. Many of these systems do not have an operating system which supervises all the nodes, and can only be used for specially adapted applications which use message passing facilities to communicate between segments running on different processors. A message passing environment was investigated for this project as an alternative to providing a complete operating system such as Helios. While the message passing interfaces lack many of the programmer-friendly facilities found in distributed operating systems, they consume less resources resulting in an improved performance.
A number of message-passing systems are at present in wide-spread use. Many of these, like int\_nx2 interface [192], are propriatory. However, a large number of interfaces is publically available, most of which have the additional advantage of widely proven portability.

The speed of communication networks has been evolving rapidly with increases of an order of magnitude each decade. The progress from distributed-protocol asynchronous protocols, such as Ethernet, through Fiber Distributed Data Interface (FDDI) [193] to high-speed synchronous schemes like Synchronous Optical Network (SONET) [194] may be seen in figure 5.1. These increases in network bandwidth have made it practical to construct a parallel processor from individual machines connected by a fast communication link.

![Figure 5.1: Communication speed trends over the last twenty years.](image)

The criteria used in selecting one of the alternative systems are listed below:

**Heterogeneous support** Although, for this project, the network consisted of a single processor type, it was hoped that this system would become a part of a larger research facility. Therefore, support for a network of different
interconnected machines was essential. Not only does this allow the full utilisation of available computing resources, but it also combines the development and debugging of applications on architectures with best programming support with a maximum performance.

Widespread use
Since the area of message-passing interfaces is still evolving, it was prudent to opt for a system with a large body of past experience.

Performance
Some inefficiency will be introduced by the extra layer of indirection provided by a message passing interface on top of the native communication mechanism. However, this overhead should be minimal and hence the performance of the interface should not be significantly different than that of the underlying communication system.

5.1.1 The p4 macros

The p4 system was developed at the Argonne National Laboratory in response to their requirement for usable parallel programming support [195]. The system provides support for both shared-memory and distributed-memory environments. The shared-memory support is afforded by monitors, while distributed systems are provided with message-passing. p4 provides for heterogeneous environments and uses External Data Representation (XDR), a standard for architecture-independent data encoding developed by Sun for RPC, to encode its messages.

However, since our research environment was not expected to ever contain any shared-memory machines, extra feature of p4 supporting them offered no
real benefits, especially since different software has to be written for the two paradigms.

5.1.2 PARMACS

The PARallel MACroS (PARMACS) package has developed as an offshoot of the original p4 work which was continued at the German National Research Centre for Computer Science [196]. While PARMACS does provide extensive topology-mapping facilities, it does not provide for heterogeneous networks and hence was rejected.

5.1.3 PVM

Parallel Virtual Machine [197] is a heterogeneous distributed programming environment developed from a project started in 1989 as part of the Heterogeneous Network Computing initiative [198]. The project was a collaboration between Emory University, Oak Ridge National Laboratory and the University of Tennessee.

PVM is currently the most popular package which, in addition to heterogeneous message passing, provides an unusual degree of flexibility in the configuration of the system. Both the physical machines which constitute the virtual multi-processors and the task structure may be changed dynamically, unlike the other packages which require this information to be specified before starting.

The PVM system was selected for future development. In addition to providing all the required features, its design is minimalistic, reflecting the principle of Occam's razor [199], with the aim of providing more complex features in higher-level libraries.
5.1.4 Linda

The Linda programming system [200,201], does not limit itself to supplying simple message-passing, but rather provides an entirely different paradigm for parallel programming based on shared tuple space. Linda has been very successfully implemented on both shared memory and message-passing systems and has in some cases achieved excellent performance. However, since Linda’s shared tuple-space abstraction is best suited to shared-memory computers, its performance in a distributed memory environment is highly dependent on the details of its implementation, of which there is no standard, although efficient commercial implementations are available for a variety of architectures. While the high level of abstraction used by Linda simplifies its use, it can result in inefficiency, which the programmer has no opportunity to improve. Some more recent research indicates that these fears are unfounded, with Linda achieving better performance than PVM [202].

5.1.5 Other systems

There are, of course, other notable systems, which were not selected for a variety of reasons. The Express system [203] which provides very sophisticated facilities such as dynamic load-balancing, parallel I/O and fault tolerance, was rejected due to its proprietary nature. The newly-emerging Message Passing Interface (MPI) [204,205], developed through cooperation of researchers from around the globe, is extremely likely to become the standard in the future. However its specification is still under development.
5.2 The PVM model

PVM aims to provide its users with a uniform view of a single virtual machine on which all computation takes place. This virtual machine is physically composed of a user-defined set of serial and parallel computers, connected by a communication medium.

To facilitate meaningful discussion of PVM, the term *Single Program Multiple Data* (SPMD) must first be introduced. This paradigm described by McBryan [206] is also known as *data parallel* processing and describes a system where all tasks are identical but execute independently and operate on a part of the overall data. A distinction has to be made between some data-parallel SIMD models [207] and SPMD paradigm, which is essentially a MIMD.

PVM promotes the use of SPMD programming and uses the term *instance* to denote the identical tasks. In addition, PVM also supports the use of *functional parallelism*, where the application is composed of a sequence of independent tasks, known as *component* in PVM terminology. The overall model of parallelism encouraged by PVM is illustrated in figure 5.2. However, the system does have flexibility allowing the user to select a different parallel structure. An excellent overview of different parallel processing paradigms may be found in Andrews [208].

The PVM architecture provides a set of library routines to facilitate virtual machine configuration and task management in addition to message passing and synchronisation. The message passing mechanism guarantees a reliable, efficient connection with messages guaranteed to be delivered in the same order as they were sent.
5.3 Implementation of PVM

The PVM environment is implemented by two cooperating parts: the PVM daemon, known as `pvmd3` and the library of interface functions, `pvmlib3.a`. In UNIX terminology, a *daemon* is a non-interactive process, which runs in the background performing tasks for the user. All machines which form a part of the PVM network must be executing a copy of the daemon, while every PVM program must be linked with the PVM library. The overall architecture of the system may be seen in figure 5.3.

Each PVM task communicates directly only with its local daemon, by sending it a message. The daemon then either passes this message to other local tasks or to a remote daemon running on the machine of the destination component. The daemon also directly handles some requests, for example, reconfiguring the virtual machine by starting or terminating remote daemons.
To identify the tasks in the system, PVM uses an integer *task identifier* (tid). Every running task has a unique task id, including all the daemons. The daemon ids are also used in some contexts to identify the hosts, since every host is executing exactly one daemon. To allow selection of different message streams, *message tags* (msgtag) are used. Each message has a message tag which must be matched by the recipient. The integer tags are assigned and managed entirely by the users.

### 5.3.1 PVM library functions

To avoid name collisions, all PVM library functions are prefixed with `pvm_`. The library provides a number of general-purpose functions, which allow the caller to add or delete hosts forming a part of the network as well as obtain some basic information. Some of the more common functions are listed in table 5.1

New tasks may be started by any process by calling `pvm_spawn`. This function allows the caller to specify the executable name, number of instances to start, a set of arguments and either the hosts or host architectures on which the tasks should be started. The function returns to the parent the task id of the newly-
started child.

The message-passing functions form the core of the PVM library and allow a wide variety of behaviour. The basic sequence involved in sending some information is as follows:

- Initialising a clear buffer by calling `pvm_initsend()`.

- Packing or **marshalling** the data to be sent using the `pvm_pk...` functions. For example, an array of integers may be packed with `pvm_pkint`. While the requirement for packing each data item using a different function is inconvenient, it is absolutely necessary in order to allow for data-format translation in a heterogeneous environment.

- The buffer may now be sent using one of the sending primitives listed in table 5.2. PVM provides functions which deliver the message to a single recipient (**unicast**) or a group of recipients (**multicast**). The ability to **broadcast**, that is send a message to all tasks within a group, is also provided by the group extensions.

The marshalling functions cause the majority of overhead in message passing. While there is no simple cure, various techniques can help to reduce this overhead:

- A combined marshalling and sending function, `pvm_psend`, is provided for simple messages.

- The heterogeneous data encoding may be disabled at the programmer's discretion. The `pvm_initsend` function takes an argument which specifies the encoding used. Instead of the XDR encoding selected by `PvmDataDefault`, the programmer may use `PvmDataRaw`.

- The marshalling functions allow for the packing of entire arrays and even implement a simple scatter-gather mechanism. Each function takes a pointer to an array of data, together with a **count** and **stride**. The elements which are marshalled into the array follow the pattern of: $0, \text{stride}, 2 \cdot \text{stride}, 3 \cdot \text{stride}, \ldots, \text{count} \cdot \text{stride}$
The reception of a message follows a process which is the reverse of sending.

- One of the receive primitives, listed in table 5.2, is used to receive a buffer. These functions allow the application to specify one or both of the sender task id and message tag. Only matching messages will be received, however the integer -1 may be used as a wild-card, matching any value.

- The buffer is unpacked, using the *pvm_upk* functions. Every *pvm_pk* function has a correspondingly named *pvm_upk* function.

An example of the code used in transmission and reception may be seen in figure 5.4.

```c
#define FOO_MESSAGE (101)
int foo[10];         int bar[10];
pvm_initsend (PvmDataDefault);  pvm_recv (-1, FOO_MESSAGE);
pvm_pkint (foo, 10, 1);   pvm_upkint (bar, 10, 1);
pvm_send (taskid, FOO_MESSAGE);
```

(a) Sender

(b) Recipient

Figure 5.4: An example of PVM message-passing code.

In addition to the facilities described already, the library provides various configuration and debugging functions, including an extensive tracing facility. Also, the programmer is given direct access to the transmission and reception buffers, enabling him to implement very efficient message forwarding.
5.3.2 Group Extensions

The group extensions are a separate library, implemented using PVM, which provides dynamic group support to PVM applications. The groups are identified by arbitrary strings and support various functions including a broadcast facility, `pvm_bcast`. Another group feature is barrier synchronisation. This form of synchronisation occurs frequently in parallel tasks and requires that all the group members wait for every member to reach the barrier point. This facility is provided by `pvm_barrier`.

5.3.3 Multiprocessor machine support

The main benefit of using PVM is the ability to use a number of serial computers as a single parallel machine. However, PVM also provides limited support for including multiprocessor machines in the network. Unfortunately, unlike workstations, where a standard operating environment is widely used, multiprocessors provide very few standard facilities.

Many multiprocessors can only execute a single application on each processor and hence cannot meet PVM's requirement of running the daemon on every node. Also, in order to efficiently utilise the special-purpose communication mechanisms, which often do not match the paradigm used by PVM, the multiprocessor applications must be customised to the particular architecture.

Therefore, the support provided for multiprocessor is limited to viewing the entire machine as a single node in the network. One copy of the daemon is executed on the front-end processor, which is connected to the network. In the case where the front-end processor is an entirely separate machine, this machine may not form a part of the PVM network. This is because each daemon
can only support a single architecture type, which for the front-end machine will be the multiprocessor and since each daemon must appear at a different network address, it is impossible to run two separate daemons on the front end.

### 5.4 Future developments

PVM continues to evolve with many extensions under development, including shared memory support, integrated debugging support, load balancing and fault tolerance. A number of higher-level facilities, which utilise PVM as the underlying implementation mechanism, are also being developed. These include distributed parallel I/O and transaction processing mechanisms.

While PVM itself and PVM-based applications continue to be enhanced, the system does already provide a useful scientific parallel environment. It is currently used for a wide variety of practical applications ranging from human genetics [209] to fluid dynamics [210], with some area achieving near-linear speedup [211]. The simplicity and efficiency$^1$ of the package has resulted in its wide-spread popularity which is expected to continue for the foreseeable future.

While various MPI systems are currently in use, the PVM system was the only one meeting the requirements at the time when the decision was made. PVM provides a number of extended facilities, such as task control mechanism, which are essential for use in this work. With these facilities, PVM can be effectively used in place of a parallel operating system, to provide a parallel software platform.

---

$^1$PVM achieves approximately 80-90% of the capacity of underlying software and hardware [198].
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pvm_mytid</code></td>
<td>Returns the caller’s task identifier</td>
</tr>
<tr>
<td><code>pvm_parent</code></td>
<td>Returns the task id of caller’s parent</td>
</tr>
<tr>
<td><code>pvm_tidtohost</code></td>
<td>Return the host id (i.e. the task id of the daemon running on the host) on which the argument task is executing.</td>
</tr>
<tr>
<td><code>pvm_spawn</code></td>
<td>Starts a new process.</td>
</tr>
<tr>
<td><code>pvm_exit</code></td>
<td>Terminates caller</td>
</tr>
<tr>
<td><code>pvm_kill</code></td>
<td>Terminates a specified task</td>
</tr>
<tr>
<td><code>pvm_addhosts</code></td>
<td>Add some new hosts to the virtual machine</td>
</tr>
<tr>
<td><code>pvm_sendsig</code></td>
<td>Send a POSIX signal to the specified task</td>
</tr>
</tbody>
</table>

Table 5.1: General purpose PVM functions
### Table 5.2: PVM message passing functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sending functions</strong></td>
<td></td>
</tr>
<tr>
<td>pvm_send (tid, msgtag)</td>
<td>Send buffer to a specified recipient</td>
</tr>
<tr>
<td>pvm_mcast (tid[], numtids, msgtag)</td>
<td>Send buffer to a number of recipients</td>
</tr>
<tr>
<td>pvm_psend (...)</td>
<td>Pack the data into the buffer and send it to the recipient</td>
</tr>
<tr>
<td><strong>Receiving functions</strong></td>
<td></td>
</tr>
<tr>
<td>pvm_probe (tid, msgtag)</td>
<td>Check if there is a matching message waiting</td>
</tr>
<tr>
<td>pvm_recv (tid, msgtag)</td>
<td>Receive a matching message</td>
</tr>
<tr>
<td>pvm_nrecv (tid, msgtag)</td>
<td>A non-blocking receive, which fails if there is no message ready to be received</td>
</tr>
<tr>
<td>pvm_trrecv (tid, msgtag, timeout)</td>
<td>A receive with a timeout</td>
</tr>
</tbody>
</table>
Chapter Six

Implementing PVM support for the i860

This chapter describes the work performed in extending the PVM environment to support the Numbersmasher i860 accelerator cards. Further work was done to support the Microway run-time environment under the Linux operating system and, in particular, the PVM system.

6.1 Introduction

The work performed in adding support for the i860 to PVM was split into a number of well-separated stages. Since the PVM model requires a daemon process to be executing on the front-end machine, a multitasking operating system was used in place of DOS on the host PC. This in turn required work to be done in providing support for using the i860 in such an environment. Then, finally, the work on providing i860 support to the PVM library could be done.

The front end platform selected is Linux, a publicly available UNIX-lookalike, distributed under the GNU Public Licence. Linux has been developed worldwide, from an initial implementation done by Linus Torvalds as a personal project and released in late 1991 [212]. Since then it has gained immense popularity and has evolved into a fully-fledged UNIX clone. Presently, Linux executes
Implementing PVM support for the i860

on PC-compatible computers, as well as some Motorola 68000-based machines. Ports to numerous other architectures are under way.

The main reason for selecting Linux is its free availability which includes source code. This means that any changes which may be required to the system may be done locally. Also, any lack in the documentation may be solved by examining the source code.

6.2 Porting Microway run-time environment to Linux

The initial job was to allow Microway-compiled programs to execute on the Numbersmasher board under Linux. The Microway run time environment consists of OS860, which is a simple, single-tasking kernel, controlling the i860, communicating with the run860 program, which runs on the PC. In the case of a multi-tasking operating system, like UNIX, a device driver must be used to communicate with the i860. A device driver is a program which forms a part of the operating system and controls the access to a device, as described in section 3. This arrangement may be seen in figure 6.1.

Initial exploratory work in porting Microway environment to Linux was based on the sources, supplied by Microway Inc., of the SunOS operating system. However, it was soon determined, that any further work would require a complete rewrite and the necessary device driver was written from scratch. It is described in section 6. It was found that the run860 code was not easily maintainable under Linux. This was due to a number of small variations in system structures between SunOS and Linux. Furthermore, the device driver interface used by run860 made it impossible to use this driver for interfacing to Helios. Hence a decision was made to rewrite both parts of the interface.
The new version of run860 was called mw860, since it implemented Microway protocol for communicating with OS860. The details of the protocol were undocumented, but they could be reverse-engineered from the run860 source. The overall structure of the protocol involves OS860 initiating all transactions with run860 or mw860 acting as a passive server for its requests. The mw860 must also be capable of initialising the i860 and uploading the OS860 binary followed by the application.

The implementation of mw860 was split into two main parts: librns library, which implemented all of the basic Microway protocol and utility functions and mw860 proper, which provided front-end user options and extensions to the protocol. The extension mechanism, which is present in the original Microway protocol, was used to add socket support for i860 applications. Sockets provide a local or network communication mechanism under POSIX. Very simply, each host can create a socket and connect it to another socket at another machine.

The socket support was provided by the functions listed below:
int tcpopen (char *host, int port, int listen) Depending on value of
listen, which is a boolean, either creates a local socket and waits for a
connection or connects to a specified socket at a remote host. Uses TCP\textsuperscript{1}.

int udpopen (char *host, int port, int listen) Analogous to tcpopen,
but uses UDP instead of TCP\textsuperscript{1}.

char *lastconn () Return the host name from which the last connection was
made.

### 6.2.1 OS860 link protocol

All communication between OS860 and a front-end program are transferred
using a special protocol. The protocol is initiated and controlled by the i860
and consists of an initial arbitration, which establishes the size of the transfer,
followed by the actual data. The steps involved in the protocol are illustrated in
a simplified state-transition diagram in figure 6.2. Every state change is driven
by a communication, which is indicated by a three component label which
specifies the sender, length and content. For example, the i860 sending a four-
byte length, would be labelled as i860: length(4). The description of a typical
scenario follows:

i860 sends size of transfer, len, encoded into a 4-byte integer, transmitted in
little-endian order. The transfers are limited to 1 GB, with the
top two bits used as flags. The flags indicate use of FIFO for the
transfer\textsuperscript{2} and the direction of transfer. The direction of
transfer is specified by the presence of the READ FLAG.

\textsuperscript{1}Both Transmission Control Protocol (TCP) and User Datastream Protocol (UDP) are
protocols supported by sockets, with the former providing more predictability at the price of
performance.

\textsuperscript{2}The FIFO transfer must be initiated via the link adapter, although the data is communicated
via the FIFO interface.
The i860 may also send a 0, which indicates a special condition to the PC. This is used by the higher level protocol, as described below.

PC replies with the common overall size of transfer, \( len' \). This size must be less than or equal to the transfer length requested by the i860. This arbitration stage allows both ends to limit the size of transfer to their respective buffer sizes.

If the PC is expecting a different transfer to the one which was initiated by the i860, it replies with a 0. This resets the protocol to its initial state.

i860 sends data, (or receives data), via the link adapter or FIFO as requested. When all data is transferred, the protocol returns to its initial state, ready for another transfer.

![Finite state machine representation of OS860 communication protocol](image)

Figure 6.2: A finite state machine representation of OS860 communication protocol.
The low-level protocol described above is implemented completely by the device driver. This follows the example of the original Microway code and minimises the overheads present in the transfers. On top of this protocol, OS860 implements a client-server protocol which allows it to access the resources of the front-end host. This protocol is rather straightforward, with each request consisting of an operation byte, followed by the arguments to the particular operation and expecting the result data in reply. The operations mimic UNIX system calls and provide a subset of their facilities.

One curious mechanism which uses the special condition escape described above, arises when the i860 sends a 0 instead of a requested transfer size. The higher-level protocol uses this condition to start a nested independent request, which is terminated by an ordinary operation code. This allows OS860 to perform a number of operations in the middle of transferring data for another operation. This is best illustrated by an example, such as the one in table 6.1. The arrows in the figure indicate direction of data transfer, with the i860 on the left.

The example illustrates an ordinary read request, followed by a read request which is escaped before its data is transferred to perform an access operation. The read is then resumed and completed.

### 6.3 Linux device driver

In order to access the device from Linux, a device driver had to be written. UNIX supports two kinds of devices and hence two kinds of device drivers. A character-oriented device, for example a text terminal line, provides a single two-way byte communication stream. A block-oriented device, for example a disk, provides a block storage, on top of which the kernel provides a file system.
The method for accessing a device under UNIX is through a special type of file. These special files are created by using a distinct system call, `mknod` and hold information about the device driver to use and which device number they refer to. This information is encoded as two integers, known as the `major` and `minor` device numbers respectively.

Under UNIX, the model used by character devices is that of a randomly-accessible byte stream. A character device driver has nine entry points, which are described below:

- `init()` Initialises a module to a valid initial state.
- `open (inode *, file *)` Open a device, which is associated with the inode argument.
- `close (inode *, file *)` Close an open device.
- `read (inode *, file *, char *buffer, int len)` Read a number of bytes, starting at the current offset, from the open device.
- `write (inode *, file *, char *buffer, int len)` Write a number of bytes, starting at the current offset, to the open device.
- `seek (inode *, file *, off_t offset, int from)` Change the current offset by offset, starting from the point specified by from.
- `select (inode *, file *, ...)` A function which allows implementation of the POSIX `select` system call. Its semantics are explained below.
- `ioctl (inode *, file *, int cmd, long arg)` Perform other control actions on the open device. This function implements any functionality falling outside of the domain of the other functions.

Note that in the above listing `inode` refers to the structure associated with a device-special file on disk, while `file` is the handle to the open file.
The system select call is not explained above due to lack of space. This POSIX system call implements a blocking wait with timeout and provides a mechanism for implementation of efficient communication code. Basically, the user indicates to select which of the open file descriptors he is waiting to become ready for reading or writing. An optional timeout may also be given. The select call blocks until any of the descriptors becomes ready for the specified operation or the timeout expires. The device driver select entry point must place the current process on a special queue and wake it up once the device becomes ready.

The actual behaviour of the calls depends on the model of interface implemented by the driver. For example, a device driver may support a terminal as a communication line, with read and write calls exchanging data with it. A device driver may also assume some basic characteristics of the terminal, such as availability of backspace, to provide a simple editing facility. Here, a read call requests an edited line, terminated by a RETURN from the terminal. An ordinary UNIX device driver may operate in either of the above two modes. Furthermore, if the device driver was made aware of the type of terminal attached to it, it could also treat the display as a rectangular array of characters, with the write call modifying the display and the seek call positioning the cursor. The exact level of protocols supported is a tradeoff between simplicity and efficiency.

A general design principle is to provide only the essential facilities in the device driver, since the driver code executes in supervisor mode and hence any errors have disastrous results. However, many device drivers provide extra functionality in order to reduce overheads, since on UNIX systems the cost of a system call is quite large. For example, the standard UNIX terminal driver provides simple editing facilities.
6.3.1 The Numbersmasher driver

Under many other versions of UNIX, the development of a device driver is a long and laborious task. The compile-edit-debug cycle is particularly long, since the kernel has to be recompiled and the system rebooted to test a new version of the code. Fortunately, Linux supports a mechanism for dynamically loadable kernel modules, described in detail by Welsh [213]. This allows the replacement of the driver code without rebooting the system. This is not quite as powerful an aid to development as user-space device drivers supported by many microkernel systems, since the driver code still executes in kernel space. Therefore a critical bug in the driver can easily result in a crash requiring reboot.

The device driver for the Numbersmasher/860 card is written to support three separate protocols:

**Raw protocol** where the application has direct control over data sent over the link to the processor. In this mode the `read` and `write` calls receive and transmit data to the processor, while the `seek` call is invalid.

**ROM protocol** implements the conventions used by the i860 boot ROM. The application effectively has access to the i860 memory as if it were a file, with the `read` and `write` accessing and modifying the memory and `seek` changing the current location.

**OS860 protocol** uses the communication interface used by OS860. This protocol is explained in section 6. Support by the driver for this relatively complex protocol was necessary for two reasons:

- The overheads incurred in processing this protocol in user mode by the application would be unacceptable.
The original Microway driver implemented this protocol and hence supporting it would minimise work necessary in writing mw860, as it was based on run860.

The driver also had to support access to multiple cards decoded at different addresses. Each card may be independently configured and is accessed through a separate minor device.

The driver uses two types of device files:

- A *controlling device*, which has a minor device number of 255. It is by convention named /dev/ns860.ctrl and provides a number of facilities which are not specific to any one card. It is necessary to have this device, since when the system is initialised, it is not aware of the existence of any cards and hence will not allow access to their device files.

- A *real device*, which corresponds to a Numbersmasher card in the machine. The minor device numbers start at 0, with the standard name of ns860.n, where n is the minor device number.

The control actions are performed through the *ioctl* call whose range of arguments is extensive and hence are described in appendix A. A short summary of the most important features is presented below. The *ioctl* call takes a pointer argument, which is used for input, output or both, depending on the parameters.

**Add board**
Increment the number of boards recognised by the driver. The driver initially does not recognise any boards.

**Work functions**
Include: interrupt the processor, reset the processor.

**Get and alter parameters**
Allows manipulation of the configuration of the device driver. This includes: link base address, protocol used and interrupt levels.
The driver is normally initialised by a privileged user, or by the system at boot time. The initialisation includes the configuration of all cards and their parameters. This operation requires privilege, because some of the information could be used to interfere with proper system operation. For instance, the driver allows the user complete freedom in setting the address of the link adapters and Interrupt ReQuest (IRQ) level to be used, which if set incorrectly could crash the system.

It could be argued that, instead of disallowing unprivileged users to configure the system at all, a restricted set of choices could be made available to them. For instance, the link decode address could be changeable between 150h and 170h, which are the two standard addresses, without requiring privilege, while retaining full flexibility for a privileged user. This approach was not taken, because in many cases it is very difficult to determine the validity of access to certain parts of the system, which depends greatly on the particular hardware present.

### 6.3.2 Tuning the driver

The device driver transfers bytes by polling a fixed number of times and suspending on timeout and interrupt if the polling fails. In order to investigate the performance of the device driver the interface was extended to support simple statistics gathering facilities. These collected information about the number of events and sizes of transfers. A problem was posed by the number of variables or the dimensionality of the data. In order to view the data, a scatterplot matrix was used [214]. The resulting plots of the data may be seen in figure 6.3.
The basic algorithm used by the driver to transfer each byte via the link is
\[
\text{while true do}
\]
\[
\text{for } i := 1 \text{ to } N \text{ do}
\]
\[
\text{poll (byte)}
\]
\[
\text{end for}
\]
\[
\text{if (failed)}
\]
\[
\text{then sleep (T or interrupt)}
\]
\[
\text{end if}
\]
\[
\text{end while}
\]

shown below:

The parameters of the device driver, namely the number of polls before suspension, \(N\), and the timeout \(T\) are altered. Three benchmarks were used to evaluate the performance and were run under Microway run-time environment. The scatterplot matrix shows relationships of each variable against

Figure 6.3: Scatterplot matrix of device driver performance for a sample program.

(a) Varying number of polls

(b) Varying timeout
every other. The values shown include the benchmark number (bench), total number of polls (polls), total number of sleeps (sleep), total amount of time asleep in centi-seconds (time) and the number of timeouts (tout).

The data collected for sending bytes to the i860 is separated from that for receiving data. The sending data is shown in the upper-right triangle, while that for receiving is in the lower-left triangle. Various useful information may be deduced from these plots, which constitute a very compact way of presenting the data.

For example, it is clear from the charts that the sending of data to the i860 is hardly ever held up. This is because the definition of MicroWay OS860 protocol, described in section 6, which retains all the control with the i860 side. Therefore the PC is quite frequently waiting for the i860 to initiate a transfer, but the only time it ever sends data to the Numbersmasher, is when it has been requested to do so by the i860. Hence, the i860 is always ready to receive any data sent and no waiting occurs. Various other interesting observations may be made from the data. An enlarged plot of some parameters may of course be made and were used to establish the driver tuning parameters.

The driver performance was found to be mostly dependent on the value of $N$, which was set to the value of 5000. This value may be somewhat conservative, in as much as it makes the driver somewhat inefficient. However, in most environment which use the i860, the PC is not executing any computationally demanding tasks.

### 6.4 Implementing multi-processor support for the i860

The multi-processor interface to PVM is based on the idea of running the PVM daemon on the front-end processor, which has network communication capab-
Implementing PVM support for the i860

The support of a new architecture requires the implementation of a number of low-level functions, both in the daemon and in a library, which are used for compiling multiprocessor programs. Rather than have the daemon talk directly to multiple cards, it communicates only with the multiplexing program, \texttt{pvm\_mux}, which then forwards requests to the cards. A graphical representation of the model can be seen in figure 6.4. The communication path runs in the U-shape between the \texttt{NS860} side, representing the application running on the Numbersmasher, and the \texttt{NS860\_LINUX} side, which is the PVM daemon running under Linux.

![Figure 6.4: Multi-processor machine model for PVM.](image)

The \texttt{mw860} code to service \texttt{OS860} requests had to be somehow combined in functionality with the PVM daemon. A straight-forward combination was impossible, because of the complexity of multiplexing the servicing of the two request sources. A possible solution involved converting the program to use two threads, but a simpler and more robust option of having two separate applications communicating using the UNIX \textit{Inter-Process Communication} (IPC) mechanism was selected.

A modified version of the \texttt{mw860} program, called \texttt{pvm860} was produced. This program is automatically started by the PVM daemon whenever a multipro-
Implementing PVM support for the i860 processor is in use and communicates with the daemon using a UNIX-domain socket. A UNIX domain socket provides a reliable communication between processes running on the same UNIX machine. This is in contrast to the Internet domain socket, which allows for communication between processes on different machines connected by a common network. The socket is named with a UNIX pathname, which in this case is placed in /tmp directory to avoid polluting the user's home directory. However, this means that to allow multiple users to use the system, the socket name must be derived from the user name. The overall system is illustrated in figure 6.5.

![Figure 6.5: The communications within PVM/860.](image)

### 6.4.1 Modifications to PVM

The PVM distribution keeps track of the numerous architectures which it supports through a complex configuration system, based around `make` utility. In this case, two new architectures had to be implemented to represent the front-end and Numbersmasher specific parts. The two architectures were named
Six

Implementing PVM support for the i860

NS860LINUX and NS860 respectively.

The basic implementation is centred around the emulation of NX2 message library functions, which is done by the nxlib.c module. This module is common between the front end and the Numbersmashers, to avoid repetition. A detailed representation of the various components of the implementation may be seen in figure 6.6. Both the library and the daemon had to be built for the front end, while the back end required only the library.

All functions used to implement the daemon side of the interface code have mpp_prefix, denoting the Massively Parallel Processor (MPP) interface, and include:

void mpp_init() The initialisation function, which is called before any of the other multi-processor functions.

int mpp_load (int flags, char *name, char *argv[], int count, int tids[], int) Load the number of instances specified by count of the program name into the multi-processor. The argument vector and parent task id are given and the array of newly-created task ids is returned.

void mpp_input () Receive packets from the multi-processor and put them on the task input queue.

void mpp_output () Send the packets queued for the multi-processor nodes, using the architecture-specific communication mechanism.

int mpp_probe() Return true, if there are messages from the multiprocessor ready to be read.

void mpp_kill (struct task*, int signal) Send a specified signal to the task.

In addition, architectures which support multicasting may define mpp_mcast to use the machine-specific communication mechanism. The i860 version also added void mpp_halt() function, which is called before terminating the system.
This allows the PVM daemon to correctly terminate the communication assistant program, pvm860.

The multi-processor library uses a number of macros, which again must be defined for the multiprocessor. These calls deal mostly with asynchronous message passing. The asynchronous primitives return message identifiers, \( \text{mid} \), which allow the application to later find out if a particular communication has been completed.

Since these macros were clearly designed to map one-to-one to the NX2 interface standard [192] used by intgl for some of its multiprocessors, in addition to the macro names, the corresponding NX2 function name is listed below:

**ASYNCWAIT** — void \_msgwait (int mid) Wait until the asynchronous message identified by the mid argument has completed.

**ASYNCDONE** — int \_msgdone (int mid) Return true, if the message identified by mid tag has been delivered or received.

**ASYNCSSEND** — int \_isend (int tag, void *buf, size_t len, int dest, int pty) Initiate an asynchronous send and return its message id.

**ASYNCRECV** — int \_irecv (long recvmask, void *buf, size_t len) Initiate an asynchronous receive and return its message identifier.

**MSGSIZE** — size_t \_infocount () Return the size of the message which is next to be received.

**MSGSENDER** — int \_infonode () Return the node id of the sender of the next message to be received.

**MSGPROBE** — int \_iprobe (int tag) Return an indication whether a message is pending to be received.

**MSGTAG** — int \_infotype () Return the tag of the next message to be received.
PVMCSEND - int _csend (int tag, void *buf, size_t len, int dest, int ptyp
   Perform a synchronous send of a message to the specified node.

6.4.2 Intermediate communication facility - pvm860

The PVM daemon communicates with the pvm860 application as illustrated in
figure 6.5. The protocol used for this communication is based on the standard
NX2 calls described above. The necessary modifications are outlined below:

Since there is a single pvm860 task controlling all the Numbersmasher cards in
the system, all the calls require additional arguments, which specify the card
number for which they are destined.

While the Massively Parallel Processor interface which was used for the Num­
bbersmasher port was deemed most appropriate, it is intended for custom-built
multiprocessor and not accelerator cards. This results in a couple of difficulties
stemming from the limit of one daemon per host:

- As mentioned above, because every daemon can represent one architec­
ture and the daemon running on the front-end processor represents the
accelerators, the front-end itself cannot be used effectively within PVM.
This is a recognised limitation of PVM 3.

- Since a daemon normally represents a single host, there is no provision
for suppling information about the number of accelerator cards present
in the system. This problem is solved in PVM/860 by starting copies of
the worker programs, one at a time. The daemon will only allow one
program per card (this is a limit of OS860) and hence the master can
determine when it has utilised all cards, once its requests to start more
workers fail.

The Microway run-time support was successfully re-implemented
to execute on a host running Linux. This work allowed the use of
Microway compiler for the production of PVM programs. Further work was done to interface PVM, using its Massively Parallel Processor interface, to the Microway environment, resulting in an effective PVM computing platform.
An ordinary read request

→ READ
→ file descriptor
→ data
← result

A read request with a nested access

→ READ
→ file descriptor
→ ESCAPE
→ ACCESS
→ file descriptor
→ access type
← result
→ END ESCAPE

→ data
← result

Table 6.1: An example of OS860 protocol escapes.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STARTPROG</td>
<td>Starts a named program with given arguments on the specified processor</td>
</tr>
</tbody>
</table>

Table 6.2: Extensions to pvmd ↔ pvm860
Figure 6.6: Overview of the implementation of PVM/860.
Chapter Seven

Dynamic Security Assessment in Power Systems

Operation of an electric power transmission system is aimed at maximising the system efficiency, subject to external constraints, such as pricing of generation and demand. This section deals with the issue of electric power system stability and describes the On-line Algorithms for System Instability Studies (OASIS) package developed by the Power & Energy Systems Group at the University of Bath [215]. OASIS is the principal application, for which the parallel processing platform described in this document was aimed.

A more exact evaluation of system stability has considerable financial benefits. The knowledge of stability limits allows more freedom in altering the system configuration, for instance by using cheaper, remote generation and transmitting the power to the demand area. This can result in considerable savings, with a typical example figure calculated for a large power system reaching $360,000 per day [216].

7.1 Power System Stability

System stability is defined by its response to a disturbance. Mathematically, a system is Liapunov-stable, if for every disturbance of the system from its
initial equilibrium position, there exists a bounding neighbourhood within which the system state remains. A B system is asymptotically stable, if in addition to being Liapunov-stable, it tends to an equilibrium point at time tends to infinity.

Unfortunately, this mathematical definition of stability does not take into account ongoing changes to the system, which form the operating conditions of a real electrical power network. The changes in customer demand and generation supply mean that the system is never in a steady equilibrium. Therefore, stability is usually defined in terms of engineering approximations to steady-state.

Considerations of system stability are important, because during everyday operation of an electrical power system, various naturally occurring events may disturb the system operation. Such events, which include lightning hitting a transmission line, a tree shorting out a substation connection or a power station failing due to human error, are known as contingencies.

It is important to ensure that the occurrence of a contingency has a minimal impact on the system operation. Therefore, operating limits of all components have to be met and continuity of electricity supply to all major parts of the system ensured. In the U.K. the National Grid Company (NGC) is legally obliged to ensure that their system is secure against likely contingencies [217].

An power system consists of a large number of components operating at a rated frequency, which in the UK is 50 Hz. The system stability criteria must ensure that the generators in the system stay synchronised to this frequency. If a loss of synchronism occurs, equipment ratings may be exceeded resulting in damage.

Traditionally, power system stability is viewed as consisting of two components: transient and dynamic stability. This view is taken by OASIS and hence this is the approach accepted in the remainder of this chapter. However, the separation of these two classes of stability is by no means easy and is further complicated by inconsistent uses of terminology. In fact, some sources [218]
suggest abandoning the term *dynamic stability* altogether.

### 7.1.1 Transient Stability

*Transient stability* defines the stability of the system in the face of rapid, large-scale changes, such as a severe contingency. A system is considered transiently stable if, following a disturbance, it returns to a stable operating point, which may be different to the original operating point, without losing synchronism.

Obviously, transient stability is dependent on the severity and duration of the disturbance. Hence, ordinarily, a list of credible system contingencies is considered, including all likely combinations of faults.

Since the passive transmission components of the network are relatively difficult to damage, most problems with instability occur with generators. A generator converts mechanical power input to electrical power output. However, if due to a fault, it is unable to transfer the electrical power generated, an excessive power build-up occurs in the generator which accelerates the rotor, potentially damaging it. Since the governor, which controls the physical power input, often operates only after several seconds, a transient instability may occur within such time, as illustrated in figure 7.1.

### 7.1.2 Dynamic Stability

*Dynamic stability* specifies the stability of the system when subjected to small, slow changes, such as a change of demand. An electrical power system is dynamically stable, if given such a disturbance it settles into a new steady-state condition [219] without loss of synchronism.

A dynamically unstable system may, following a small change, begin oscillating with an increasing amplitude until system limits are reached, as illustrated
in figure 7.2.

In the past, operation was limited by steady-state limits, for example maximum transmission capacity of a line. However, as systems became more interconnected, dynamic stability became the major limiting factor. Dynamic stability is normally maintained by restricting the maximum power flow between loosely connected parts of the system. Such restrictions reduce the operating efficiency by preventing the electric power utility from transferring cheaper power between loosely connected parts. This makes dynamic stability particularly important economically and was the reason for the development of OASIS.

7.2 OASIS

The Power and Energy Systems Group at the University of Bath has developed an on-line dynamic security assessor as a part of an ERCOS grant jointly funded by EPSRC and the National Grid Company (NGC). The OASIS system evaluates a provided list of credible contingencies and ranks them according to their severity. The system obtains a fully ranked list for a typical system in 10 to 15 minutes, which matches the rate of change of system state.

7.2.1 Contingency Screening

For a large power system there may be several thousand contingencies, which may cause instability and hence need to be examined. The evaluation of all these contingencies may not be possible within the time required. The version of OASIS used in this project does not perform any screening. However, other versions have been developed to include neural network based screens for both transient and dynamic stability evaluation, resulting in a twenty-fold speed
improvement [220].

7.2.2 Contingency Evaluation

While various methods of evaluating system stability are currently in use [221, 222], the most reliable and accurate approach is simulation of the system behaviour. OASIS is based around the *Real Time Power System Simulator* (RT-PSS) also developed by the Power and Energy Systems Group [223].

In order to evaluate the effect of any given contingency, the assessor simulates a maximum of 30 seconds of system behaviour following its application. If an instability arises before the 30 seconds are up, the simulation is stopped. From the simulation, the decay time constant is calculated for the key system parameters, in this case the generator rotor angle swings. The time constant is defined by the first-order equation and represents the time taken for the parameter value to decay to approximately 36.8% of the original value. The NGC suggests that a stable system should have a time constant less than 12 seconds, which is the value used by OASIS to mark unstable cases.

7.2.3 Contingency Ranking

The list of contingencies is ordered according to their severity. The *severity* of a contingency is defined by a single real number known as the *severity index*. The severity index is in turn calculated from the system parameters, which impose limits on the operation of the system. In effect, the severity index figure is a reflection of how close the system is to critical instability [224].
7.2.4 Implementation

The OASIS system is implemented in ANSI C and uses POSIX services. The system is designed to be parallel by using a client-server paradigm. The main component is a user-interface client task. It uses the PVM system, described in chapter 5, to start and communicate with a number of servers. The servers perform the bulk of the calculations. This is illustrated in figure 7.3.

The client provides a graphical user interface, using Motif toolkit and widgets on top of X window system version 11R5. The interface displays the ranked list of contingencies while the next evaluation is in progress and allows the user to examine the characteristics of the four most severely affected generators. The client also performs the sorting of the ranked contingencies.

The server tasks perform evaluation of contingencies. Since every contingency is independent of the others, this approach provides a very effective parallelisation, even in an environment which supports low communication bandwidth. Although further parallelisation would be possible, for example by using a parallel version of the power system simulator, such as the one developed in reference [79], it is not necessary. The full NGC power system requires evaluation of the effect of the order of 5000 contingencies. Since each contingency is independent, this allows a data-parallel evaluation on at least hundred computers [225].

At the start, the main process starts a copy of the worker on each of the components of the virtual machine. A description of the system is then sent to all workers. The central task then uses a "pool of tasks" paradigm, with all contingencies initially placed in the pool and the workers extracting a contingency to evaluate, performing the evaluation, returning the results and fetching another contingency.
The central resource of the task pool could become a bottleneck in a massively parallel system. However, even the largest problems currently considered require the use of only several computers to meet the timing requirements. Therefore, the central task pool is not a restriction at present, although it could become one in the future.

The last contingency is evaluated by all workers, which have finished other processing. This ensures that the overall evaluation is not held up by one slow machine and is completed as soon as possible. This is important, since results of the evaluation are not displayed until all contingencies are ranked in order.

7.3 Modifications for this project

One of the aims of this project is to provide a standard parallel environment, which would not require modifying applications, such as OASIS. However, while necessary modifications are minimised, it is impossible to avoid them completely. This is especially true, since OASIS is used in this project for operator training rather than on-line system stability assessment. This means that both the number of contingencies and the evaluation time periods are considerably smaller.

As described in chapter 5, the support for the i860 under PVM has necessary limitations. The only one which required a change in the implementation of OASIS, involved multiple i860 accelerator cards. The PVM system does not provide a mechanism for supporting multiple machines using a single PVM daemon. The original OASIS client therefore makes the assumption that each PVM daemon is a single processor and starts one copy of the server on each daemon.

While this approach does result in optimal efficiency for ordinary UNIX machines, which form the rest of the processing network used by OASIS, the i860
system may contain multiple cards, which would not be utilised under the original scheme. The client source was therefore modified, so that multiple copies of the server are started on the i860 nodes. The copies are started, one by one, until the daemon returns a failure. The PVM/860 daemon will only allow a single process per i860 card, in keeping with the restrictions imposed by Microway runtime environment. Therefore, once each card is executing a copy of the server, further attempts will fail, informing the client that all cards are now in use.

Another known limitation of the PVM system is the inability to use the front-end host processor as part of the virtual machine. This is not a limitation for the intended system configuration, since the front-end processor for the accelerators is executing the front-end interface task. However, if the accelerators are used as a part of a larger PVM network, the front-end processor will be used only to forward communications to the accelerators, which may in some cases constitute a considerable waste of resources. Hopefully future versions of PVM will free the system from this limitation, allowing the computing power of the front-end host to be fully utilised.

This chapter describes the problem of Dynamic Security Assessment in power systems and, in particular, the OASIS tool developed by the University of Bath to solve it. The OASIS package has processing requirements which match the characteristics of the parallel platform created by this project. It was therefore used as the principal benchmark in testing and evaluating of the usefulness of this work. However, as stated before, other applications with similar requirements could equally benefit from the use of the environments developed during this research.
Figure 7.1: Transient instability.
Figure 7.2: Dynamic instability.

Figure 7.3: Client-server design of OASIS.
measuring the performance of a parallel computer is a challenging task. The overall performance depends on the hardware, software and input data in a far more non-linear manner than is the case for serial environments. The aim of this chapter is simply to provide some basic results which would enable the reader to assess the system performance. An in-depth treatment of parallel performance measurement is beyond the scope of this thesis. An interested reader is referred elsewhere for more information [226].

The main performance results presented in this section relate to the execution of the dynamic security assessment program, since it represents a typical type of application for which this platform has been developed. In addition, the results of synthetic benchmarks have also been included, in the hope that they will allow the estimation of system performance for applications with other scales of computation and communication. Also, a comparison between the i860 port and more common implementations are given where appropriate.

8.1 Comparison of compiler performance

During the progress of this work, the Norcroft compiler has been replaced by Microway's C compiler. This was done to achieve better performance offered
by the superior optimisation of Microway’s compiler. In order to evaluate the extent of this advantage, the performance of the two compilers has been compared.

Since NorCroft compiler has no floating-point support on the i860, an integer benchmark had to be used. While ideally the SPEC integer benchmark should have been used, its large size made it unsuitable for this comparison. Hence, a simple synthetic benchmark of Dhrystone was used. The results of the comparison may be seen in figure 8.1.

As may be seen from the chart, the results were timed for $10^6$ and $10^7$ Dhrystones, to ensure that the steady-state performance was being correctly measured. It may be seen that the Microway compiler exhibits a two fold performance improvement over NorCroft for integer code, hence justifying the decision to upgrade the compiler.

Additionally, it should be noted that even the Microway compiler does not begin to approach the peak performance of the i860 CPU. The compiler achieves 3.6 Mflops/s for double-precision Linpack on $100 \times 100$ matrix. By comparison, the Intel Fortran compiler reaches 9.8 Mflops/s for the same conditions.
8.2 Communication performance

A major handicap of the hardware configuration used is the communication bandwidth. As explained in chapter 2, the design of the Numbersmasher card used Transputer links for convenience of design. Although the theoretical maximum throughput of the links is 1.6 MBytes/s, this is not achievable with the IMS C012 link adapters [69]. The practical throughput of the links is closer to half that value. To evaluate the performance of the communications, Helios/860 was compared against Transputer Helios, which uses a similar hardware design.

The timing was measured using the timeio program supplied with Helios, and constitutes the time taken to transmit the given message from the board running Helios to the IO-server and back. The performance of Helios/860 is compared in figure 8.2 with the results obtained from Transputer Helios, with the I/O server running under DOS and Linux.

![Figure 8.2: Transmission times for messages under Helios.](image-url)
As may be seen from the figure, the communication bandwidth of T800 Helios is considerably greater under DOS than Linux. This is due to the overhead of using a device driver to access the card as opposed to accessing it directly under DOS. The Helios I/O server is written for use under DOS and frequently transfers small-sized messages. This is not efficient under Linux, where any transfer incurs the overhead of context switch to the kernel and back. It may also be seen from the figure, that Helios/860 has a higher bandwidth and latency than Transputer variant. The higher latency is likely to be due to the higher context switch times for the i860 than the Transputer.

The apparent greater bandwidth of the i860 comes probably from modifications to the I/O server, which perform some basic buffering. The server stores small read requests, until a write is requested. A similar buffering is performed for writes. In practice this amounts to buffering entire whole messages into single device-driver requests.

The performance of PVM was compared using a supplied program timing. The program measured both packing and sending times and the results were compared with Linux version of PVM. The data may be seen in figure 8.3.

![Figure 8.3: Timings for PVM variants.](image-url)
The results show that again, the overall characteristics of the performance for PVM/860 are in line with those of PVM/Linux, although the absolute performance of PVM/860 is considerably lower. This is expected since PVM/Linux uses UNIX-domain sockets for communication, which do not suffer from the same bandwidth limitations as Transputer links.

### 8.3 Null kernel call

In order to measure the performance of Helios/860 as an operating system, the time taken to execute a simple kernel call was measured. Although the vast majority of Helios kernel functions do not involve a user to supervisor mode transition, the timed call was one of the ones which do. This was done in order to allow a fair comparison with other operating systems.

The measurement program timed the execution of a large number of traps corresponding to the Enable_Int system call, which is effectively a null operation, since interrupts are already enabled for all normal processes. The results indicated 91\(\mu s\) per call. This result was obtained during normal system operation, with the usual system tasks consuming some portion of the available CPU.

### 8.4 OASIS performance

The primary reason for developing the parallel system described in this thesis was to allow the use of sophisticated data-parallel tools, such as the OASIS program, to be available on a compact and cheap platform. It is therefore natural to evaluate its performance by using OASIS.

The system names reflect the size of the studied network and, in particular, the numbers of machines and busbars. The m4b6 system, which contains four
machines and six busbars, was used for small-scale evaluation. The training system is aimed mostly at the twenty machine (m20b100) system, which formed the core of the evaluation.

The PVM/860 system can only support up to two cards, since each card needs to decode at a distinct I/O address and only two are supported by current hardware. There are however, no fundamental reasons why the hardware could not be simply extended to support three or more cards. In the Helios/860 environment, the PC only talks directly to the first card, with the remaining cards chained using the Transputer links. Therefore, this environment can support any number of cards. The limit of three cards was however imposed by the power supply and cooling requirements of the Numbersmashers.

Each timing measurement was repeated three times to estimate the variance in results. The presented results are largely based on the article by Crowl [227].

The timing results are shown in figures 8.4–8.7. The data is shown as a log-log time plot, to allow a graphical comparison of the relative speed-up achieved in the various configurations, despite their vastly different speeds. In addition, the absolute performance may be compared on the linear speed plots in figures 8.5 and 8.7. All figures includes error bars which mark the maximum and minimum values. These are used in place of the more usual standard deviation figures, since only three measurements were taken at each point. The linear speed graphs show the speed normalised to the number of contingencies evaluated, since otherwise the range of values would make the chart unreadable.

As may be seen from the figures, the PVM/860 configuration is considerably faster than Helios/860. This difference reflects the cost of the multi-tasking services provided by Helios. Microway's OS860, which only allows a single task per CPU, does not incur these scheduling overheads, hence achieving better performance.
Another component which contributes to the slowness of the Helios variant is the relatively inefficient code used. Although both PVM and Helios use the Microway compiler, Helios requires the code to be position-independent and demands special support for shared libraries. These features result in considerably less efficient code. Some improvement in performance could be achieved by altering the interaction between the compiler and linker, so as to avoid using the expensive indirect access method where it is not absolutely necessary. However, the majority of the performance loss can not be eliminated.

The speedup of the evaluation is approximately linear with the number of processors. This, of course, was to be expected, as the algorithm is data-parallel and hence is not affected by inter-worker communication.

As may be seen from figure 8.6, the performance of the system is adequate for its function. The 35 contingencies on the 20 machine system are evaluated in under 2 minutes using two processors with PVM. Such a configuration gives the user enough time to view the results and consider their implications in between evaluations.

While this project has satisfied its original aims, the long-term usefulness of this work has been seriously reduced by Intel's decision to discontinue the i860 family. This means that the hardware currently used is over six years old. In today's environment, the entire complex hardware and software system could easily be replaced by a uni-processor. For example, a Pentium 120MHz processor has been benchmarked to take approximately 50 seconds to perform the ranking of 35 contingencies, making it twice as fast as the fastest Number-smasher configuration tested.

However, it is still possible to construct a very powerful single-box configuration based on the i860 hardware. Microway are currently marketing the QuadPuter board [7], which holds four i860 XP 25MHz CPUs together with 32 Mbytes of shared-memory and PC interface. Up to five such boards may be added to a single, specially adapted PC. Each processor also has 2 Mbytes of local RAM, which can be accessed without contention.
Although has not been ported to the QuadPuter boards, the issue of this port was considered at one stage of the project. The system would almost certainly have to emulate message passing using the shared memory. It could also use the shared memory to hold shared executable code, in particular the kernel and nucleus, although that would require some modifications to the servers.

Support of PVM on the QuadPuter would also be possible. Current versions of PVM already contain some support for various shared-memory architectures. Modifying such support to allow for the use of shared memory between processors on a single board and message passing between boards, would constitute the majority of the porting effort.

Using the QuadPuters, a single-box training environment containing twenty i860s could be constructed. Estimating the performance of such a system executing the DSA and taking into account the lower clock speed of the QuadPuter CPUs, the basic 35 contingency training scenario could be evaluated within 15 seconds.

Additionally, such a system would have a greatly improved communication performance. Microway quote the bandwidth of the shared memory access at 67 Mbytes/s and EISA bus access (for inter-card communication) as 16 MBytes/s. This would make the platform also suitable for more tightly-coupled parallel applications.

8.5 Other results

In addition to the above numerical results, the systems have a number of qualitative differences. The Helios environment is considerably less stable and more prone to crashes than MicroWay's OS860. This is a result of its greater complexity, lesser use of the memory protection hardware and real-time nature. In fact, the vast majority of problems encountered during development of Helios/860,
were due to lack of reliable memory protection and message timeouts.
Figure 8.4: Oasis time of execution for naive study for 20 and 54

Time [s] (log scale)

10 100 400

Processors (log scale)

2 3

c20 PVM/860

c54 PVM/860

c20 Helios/860

c54 Helios/860
Figure 8.5: Oasis speed of execution for m4b6

- c20 PVM/860
- c54 PVM/860
- c20 Helios/860
- c54 Helios/860
Figure 8.6: O(ais time of execution for m2db100 study for 10 and 35

<table>
<thead>
<tr>
<th>Processors (log scale)</th>
<th>Time [s] (log scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
</tr>
</tbody>
</table>

- c10 PVM/860
- c35 PVM/860
- c10 Helios/860
- c35 Helios/860
his chapter describes conclusions reached as a result of the conducted work and the further work which is suggested. The work described here was not performed because it was judged secondary to the main objective of this project and time constraints did not allow significant detours.

9.1 Conclusions

The results have demonstrated the practicality of the single-box computer for use as a DSA training platform. The hardware utilised optimised accelerator boards in a PC-compatible machine. Two software configurations supporting distributed data-parallel application were tested. Both the distributed operating system and the message passing system were ported to the i860 and their performance examined.

The performance of the message-passing system outstripped that of the distributed operating system. This was due to:

- The overheads of a fully-blown operating system.
- The inefficiencies due to use of position-independent code and shared libraries.
An operating system which does not use memory management unit, such as Helios gains substantial performance on RISC processors like the i860. It avoids the penalties due to MMU context switching and, in the case of virtually-tagged caches, also the cache flushes. However, in order to implement shared libraries, such a system must use indirection for all global data, hence losing a substantial portion of its gain.

Also some of the inefficiencies could be eliminated by a better compiler/assembler/linker implementation, the inherent overheads of multiple tasks would still heavily penalise Helios performance.

In parallel environments which require maximum performance, several choices exist for the software support framework. Traditionally, such architectures would use a simple special-purpose environment, which provided maximum performance at the cost of portability of programs. The other extreme position was occupied by fully-blown distributed operating systems, such as Helios, which provided more flexible services and a wider range of supported architectures at the cost of performance.

At present, with the popularisation of message-passing systems, such as PVM and the newly-standartised MPI, a third alternative has emerged. The special-purpose single-processor environments, like MicroWay's OS860, may be combined with a portable message passing interface, achieving optimal performance and yet retaining wide portability.

Of course, distributed operating systems can fight back by providing more advanced services, which are unavailable under plain message-passing systems. This is already the case with some systems, such as Chorus [83], which provide load balancing, task migration or distributed file systems. However, for the training DSA system, the superior performance of the message-passing environment makes it the better choice.

As mentioned in the results chapter, the system developed is adequate in fulfilling its role as a training DSA simulator. The system performance could be substantially enhanced by using more powerful hardware. In addition, some
of the latest work on the neural network contingency screens, described in section 7, could be used to increase the performance of the system by a further order of magnitude. However, in the particular application of operator training, the large number of contingencies does not necessarily offer an advantage, since the user cannot assimilate the large amount of information generated in the time available.

### 9.2 Further Work

#### 9.2.1 Enhancing Helios

The current port of the Helios operating system to the i860 is complete. However, since the architecture of the i860 is significantly different from processors for which Helios was developed, numerous enhancements to the operating system are possible. These include:

**QuadPuter support** would be a priority for Helios, in order to bring the system performance to match current state-of-the-art. While there are no conceptual difficulties with the work, various tradeoffs, in particular what to keep in the shared memory, will have to be evaluated.

**Better protection** of tasks, in particular protecting the data segments of tasks which are not currently executing from being modified could be added. The i860 contains an integrated MMU and Helios has been modified to protect code segments. The performance implications of altering the MMU mapping and the consequent cache flush have to be considered in balance to the extra reliability thus achieved. Further problems arise from some Helios programs assuming that they have access to the entire memory. For instance, the map utility which displays a memory map,
transverses the linked list of allocated memory blocks to obtain its information.

Perhaps the most flexible approach would be to use the flag field in the executable file to indicate whether while executing it the MMU should be used to protected other segments. This would allow the execution of suspect programs reliably without incurring the speed penalty or altering any software.

Faster boot facility could be provided on the i860 by noting the fact that with the MMU protecting the nucleus, it cannot be corrupted during the system use. Therefore, rather than uploading the nucleus blindly when Helios is booted, the server program can check a number of memory location to verify that the nucleus is loaded and simply restart it. In addition, in case of a failure to boot, the server should automatically re-load the nucleus, thus avoiding problems with hardware memory corruption, which can arise from stray high-energy radiation or a temporary hardware failure.

Clearly, this facility is potentially dangerous. However, many other aspects of Helios, in particular its timeouts, have a substantially larger probability of failure than an adequately large check block consisting of several bytes.

9.2.2 Improving compiling

The system does at present support two C compilers: NorCroft and MicroWay. However, neither of these systems provides a satisfactory development environment. It is the view of the author, that the NorCroft compiler should be completely replaced by the MicroWay program, which itself requires a number of enhancements. These include:
Data segment is currently provided for by generating code which initialises it. This approach was taken initially, because it directly matches the GHOF model of the object file and provides greater flexibility than the COFF solution. However, the resultant data segments take up far too much space and considerably slow down the start-up of executables. The solution would involve generating a data block containing initialised data and initialisation code which simply copied this block to the module’s data slot. Of course, more elaborate approaches, including generating code to initialise repetitive data or ultimately some form of compression would also be possible.

**Helios** support for execution of the MicroWay compiler is essential to further development. At present, the compiler has to be executed under the OS860 environment. This means that with a single i860, Helios has to be terminated in order to execute the compiler. The compiling of MicroWay C compiler for use under Helios would first require the alteration of MicroWay Pascal compiler to generate GHOF. Unfortunately, the code generation part of the Pascal compiler is distinct from the C compiler.

In addition, it might be useful to re-compile Helios itself with the MicroWay compiler. This would require work to done to enable this compiler to generate Helios shared libraries, which are provided by NorCroft compiler with the -z1 option.

### 9.2.3 Improving PVM

The PVM environment has shown best performance of the tested configurations. The main task which remains to be done is the support for three cards under PVM. This would involve modifying the address decoding Programmable Logic Device (PLD) to support at least three different base addresses. In the interests of flexibility, support for a user-definable address,
within some range, would be most appropriate. The device driver and PVM libraries would not require any significant modification to support three Numbersmasher cards.
This appendix contains a description of the IOCTL calls, which may be used to communicate with the Numbersmasher device driver for Linux.
<table>
<thead>
<tr>
<th>Name</th>
<th>Control?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>x control</td>
<td>Returns the version number of the driver. Also uses the argument (if not NULL) to return the option flags, indicating options supported by the driver. Optional features include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- FIFO interfaces supported, if any.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debugging support, see section A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Handling of non-blocking accesses, which may be slow, fast or none.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Support for non-aligned accesses using boot ROM protocol, which are not supported by the protocol and have to emulated by the driver.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Support for asynchronous accesses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Support for statistics gathering.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All of these options are implemented by conditionally-compiled code in the driver.</td>
</tr>
<tr>
<td>Name</td>
<td>Privileged?</td>
<td>Control?</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>How many</td>
<td>X</td>
<td>control</td>
</tr>
<tr>
<td>Add board</td>
<td>✓</td>
<td>control</td>
</tr>
<tr>
<td>Auto configure</td>
<td>X</td>
<td>control</td>
</tr>
<tr>
<td>Debugging</td>
<td>✓</td>
<td>control</td>
</tr>
<tr>
<td>Force release</td>
<td>✓</td>
<td>control</td>
</tr>
<tr>
<td>Reset</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Interrupt</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Which am I</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Get protocol</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Set protocol</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Get linkbase</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Set linkbase</td>
<td>✓</td>
<td>device</td>
</tr>
<tr>
<td>Get IRQ</td>
<td>X</td>
<td>device</td>
</tr>
<tr>
<td>Name</td>
<td>Control?</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Set IRQ</td>
<td>V</td>
<td>device Alter the IRQ level used by the driver.</td>
</tr>
<tr>
<td>Get tuning</td>
<td>X</td>
<td>device Get the tuning information. The tuning parameters allow to alter the driver performance-related behaviour while operational. The parameters include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ number of polls</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ timeout length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ retry after timeout ?</td>
</tr>
<tr>
<td>Set tuning</td>
<td>V</td>
<td>device Alter the tuning information.</td>
</tr>
<tr>
<td>Get flags</td>
<td>X</td>
<td>device Returns flags representing the current configuration and status of the device. Useful flags include:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Device is present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Device is enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Use interrupts in addition to polling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Use FIFO adapter instead of links</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Buffer transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Device is busy (transferring)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>○ Device is ready for a read or write</td>
</tr>
<tr>
<td>Set flags</td>
<td>V</td>
<td>device Alters the flags defined by the bottom byte. The alterable flags include the first five in the above list.</td>
</tr>
</tbody>
</table>
## A.1 Debugging Support

In addition to the above functions, the driver provides some support for tracing protocol details and debugging. The driver is organised into the following subsystems:

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>The open() driver function.</td>
</tr>
<tr>
<td>close</td>
<td>The close() driver function.</td>
</tr>
<tr>
<td>read</td>
<td>The read() driver function.</td>
</tr>
<tr>
<td>write</td>
<td>The write() driver function.</td>
</tr>
<tr>
<td>ioctl</td>
<td>The ioctl() driver function.</td>
</tr>
<tr>
<td>seek</td>
<td>The seek() driver function.</td>
</tr>
<tr>
<td>select</td>
<td>The select() driver function.</td>
</tr>
<tr>
<td>intr</td>
<td>Handling of interrupts.</td>
</tr>
<tr>
<td>low read</td>
<td>Low-level (protocol) reads.</td>
</tr>
<tr>
<td>low write</td>
<td>Low-level (protocol) writes.</td>
</tr>
</tbody>
</table>

In addition to these subsystems, there are five levels of urgency, namely:
Linux device driver for the Numbersmasher card

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbose</td>
<td>Verbosely reports progress.</td>
</tr>
<tr>
<td>normal</td>
<td>Reports all operations.</td>
</tr>
<tr>
<td>warning</td>
<td>Informs about recoverable errors.</td>
</tr>
<tr>
<td>error</td>
<td>Errors which abort operation.</td>
</tr>
<tr>
<td>critical</td>
<td>Failures which should never occur and may indicate internal inconsistency.</td>
</tr>
</tbody>
</table>

The device driver uses the Linux syslog facility, which is intended for kernel messages. The resulting messages get logged into file /usr/adm/messages, unless the system crashes. During the initial debugging of the driver, when it caused frequent kernel crashes, a more direct mechanism of viewing the syslog errors had to be used.

The user can enable any combination of levels and subsystems to generate messages, although root privilege is required to change level of the debugging. By default the critical and error level messages are reported for all subsystems. Since the messages are implemented using pre-processor macros, they can be completely removed, increasing run speed.
Appendix B

Changes to Microway C compiler output

This appendix contains a report which was originally produced for Microway Inc on the 11th November 1992. It is included in its original format.

B.1 Document structure

This document consists of a brief introduction to the Generic Helios Object Format (section B), a summary of the changes required for MicroWay's compilers (section B) and an example C program in both converted and unconverted forms (appendix B).

B.2 GHOF principles

Despite similarity of names, GHOF is quite different from COFF. The Helios concept of a module table is central to GHOF. Each module consists of a correct header which cannot be automatically generated by the assembler, since some information contained in it is only available to the compiler.

The following are most notable changes from COFF to GHOF are:

- All code must be relocatable
- All data and external functions must be accessed via the module table

- The object has only equivalents of TEXT and BSS sections. Hence all initialised data must be initialised by outputting the code to set it up at start time.

B.2.1 Object structure

The structure of a correct assembler file directly reflects the structure of the object file. The file consists of:
The order of most sections (excluding the header and trailer) is arbitrary; however the format described above is recommended. The following sections describe each of the parts of the object in detail.

**Module Header**

Each GHOF module *must* start with a module header. A module header is always 56 bytes long. The following represents the format of this header:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Contains</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0x60f160f1</td>
<td>module type</td>
</tr>
<tr>
<td>4</td>
<td>.ModEnd − .ModStart</td>
<td>total module size</td>
</tr>
<tr>
<td>32</td>
<td>&quot;name&quot;, 0...</td>
<td>module name, 0 terminated</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>module slot</td>
</tr>
<tr>
<td>4</td>
<td>0x1000</td>
<td>module version</td>
</tr>
<tr>
<td>4</td>
<td>.MaxData</td>
<td>size of module data slot</td>
</tr>
<tr>
<td>4</td>
<td>init</td>
<td>start of init chain</td>
</tr>
</tbody>
</table>

In addition, the module should set the `.ModStart` code label, which allows for the calculation of module size in the header, and enforce the correct alignment required by the rest of the file.

**Code**

This section contains ordinary i860 assembler code, mainly the definition of all the functions. The access to variables and functions in the code of this section must be modified as per instructions contained in section B.
Stubs

For each external function (one not defined in this module), which is called from within this module a stub must be generated. Stubs are described on page 181.

Data Declarations

A GHOF object has only one data space. However the COFF model generates 3 separate data spaces (Global BSS, local BSS and local initialised). Furthermore, GHOF requires some extra data space for exported functions. Hence the single physical data space is split logically into 4 separate spaces.

It is most convenient to generate the data declarations for each of these sections in a block, followed by some extra padding data to ensure correct alignment of the next logical data space.

It is also easiest to output the export directives for exported labels next to their data declarations.

Init Routine

The initialisation of all the data is performed by an initialisation routine. The assembler automatically maintains pointers to these routines provided they start with an init directive. The init routine is called like any other assembler subroutine. There are two logical blocks of data to initialise: initialised variables and exported function addresses.

Module Trailer

The module trailer contains no generated code. It should however define labels which will enable the assembler to calculate the size of the module code and data.

B.3 Changes to assembler format

Please note the following:

- The C notation is used for hexadecimal and octal constants
- The assembler recognises a number of new directives (eg init, patchinstr etc)
- The sections which describe briefly differences between MicroWay's assembler and GHOF assembler formats are highlighted by using shading.
- Each highlighted source example box is labeled with a reference number near the C source (eg Ref 2). This number will be used in future revisions of this document to mark changes. Also the example lines are numbered.
A familiarity with the basic layout of the object file will be assumed (see section B)

### B.3.1 Module header

For information on the module header see section B.

The suggested assembler output by the compiler is:

```assembly
module
  .ModStart:
    word   0x60f160f1
    word   .ModEnd - .ModStart
    byte   "file"
    space  32 - 4
    word   Modnum
    word   0x1000
    word   .MaxData
  init
```

Note: The argument to the `space` directive must be equal to 32 minus the number of characters in the string `name` and must be greater or equal to 1. Hence the name of the module must be limited to 31 characters.

Also the symbols `.ModEnd`, `.ModStart` and `.MaxData` are local code and data labels.

### B.3.2 Register usage

The Helios system requires one register which should not be used by the compiler. This register always holds the module table pointer and should never be used by the compiler. We are using `r15` for that purpose. Also, for compatibility with existing NorCroft code, the stack pointer register is `r2`. Furthermore, methods of access to data required by Helios sometimes require temporary registers. In that case, `r30` and `r31` are used in the examples below. The compiler must clearly be aware of the corruption of these registers.

Required register allocation changes are:

- `r2` should be unused
- `r15` should be unused
- `sp → r2` the register `r2` should be used instead of `sp`
- `r30, r31` temporary registers sometimes corrupted by new (Helios) code
B.3.3 Functions

Each exported function has two labels associated with it:

- `.name`  Actual code label
- `_name`  Data label associated with the function and containing its address

Helios code for defining `local` functions:
- Generate code label called `.function` not `.function`
- Local labels generated by the compiler which currently start with `.` should be altered to start with `..` to distinguish them from code labels.

C source

```c
static void local ()
{
...
```

<table>
<thead>
<tr>
<th>MicroWay compiler code</th>
<th>Helios code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.local</code></td>
<td><code>.local</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>.L3</code></td>
<td><code>.L3</code></td>
</tr>
</tbody>
</table>
Helios code for defining exported functions:

- Generate code label called `.function not .function`
- Generate the function data slot named `.function`
- Append to the `init` routine the code to initialise the function data slot

C source

```c
extern void exported ()
{
    ...
}
```

<table>
<thead>
<tr>
<th>MicroWay compiler code</th>
<th>Helios code</th>
</tr>
</thead>
<tbody>
<tr>
<td>.exported:</td>
<td>.exported:</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.globl exported</td>
<td>in data declaration</td>
</tr>
<tr>
<td></td>
<td>data exported, 4</td>
</tr>
<tr>
<td></td>
<td>export exported</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>in <code>init</code> routine</td>
</tr>
<tr>
<td></td>
<td>br <code>init.exported</code></td>
</tr>
<tr>
<td></td>
<td>ldcc fir, r30</td>
</tr>
<tr>
<td></td>
<td>word exported</td>
</tr>
<tr>
<td><code>init.exported</code>:</td>
<td>id 4(r30), r31</td>
</tr>
<tr>
<td></td>
<td>addu 4, r31, r31</td>
</tr>
<tr>
<td></td>
<td>addu r30, r31, r30</td>
</tr>
<tr>
<td></td>
<td>patchintr (PATCH60HA,</td>
</tr>
<tr>
<td></td>
<td>DATASYMB (_exported),</td>
</tr>
<tr>
<td></td>
<td>addu 0, r16, r31)</td>
</tr>
<tr>
<td></td>
<td>patchintr (PATCH60L,</td>
</tr>
<tr>
<td></td>
<td>DATASYMB (_exported),</td>
</tr>
<tr>
<td></td>
<td>st:1 r30, 0(r31) )</td>
</tr>
</tbody>
</table>
Every external function which is called in the module must have a stub appended to the module's code. The stub should have the format:

C source

```c
extern void import();
```

Ref 3

MicroWay compiler code

```c
import:
    patchinstr (PATCH8GOVAL,
                DATAMODULE (.import),
                ld.l 0(r16), r30)
    patchinstr (PATCH800OH, 
                CODESYMB (.import),
                orh 0, r0, r31)
    patchinstr (PATCH800OL, 
                CODESYMB (.import),
                or 0, r31, r31)
    ld.l r31(r30), r30
    bpi r30
    nop
```

Helios code

B.3.4 Initialisation routine

The initialisation routine must start by loading the data slot base into a register (r16)

```c
init
    patchinstr (PATCH8GOVAL, SHIFT (-2, MODNUM),
                ld.l 0(r16), r16)
...```

It should finish like any other subroutine with

```c
bpi r1
nop
```
B.3.5 Declaring variables

Space for variables is reserved using the data directive. For any initialised data the code to set up the data slot must also be output.

Uninitialised locally defined data (not exported) should be handled as follows:

C source
```c
static int var;
```

MicroWay compiler code
```
.lcomm .var, 4
```

Helios code
```
data .var, 4
```

Initialised locally defined data (not exported) should be defined as follows:

C source
```c
static int var = 2;
```

MicroWay compiler code
```
.var: .long 2
```

Helios code
```
data .var, 4
in the init routine
or 2, r0, r31
patchinstr (PATCH880HTA, DATASYM(var), orh 0, r0, r30 )
patchinstr (PATCH686GLT, DATASYM(var), st.l r31, 0(r30) )
```

Note: if the variable is initialised to the value (or address) of another variable, the code specified in section B should be used to load register r31 with its value.
Uninitialised exported data should be defined as follows:

C source
```c
int var;
```

Ref 6

**Microway compiler code**
```assembly
.comm var, 4
```

**Helios code**
```assembly
data var, 4
export var
```

Initialised exported data should be defined as follows:

C source
```c
int var = 6;
```

Ref 7

**Microway compiler code**
```assembly
.var: .long 6
.globl var
```

**Helios code**
```assembly
data var, 4
export var
.in the inst routine
or 6, r0, r31
patchinstr (PATCH660HIA,
DATASYMB(var),
enb 0, r0, r30 )
patchinstr (PATCH660LO,
DATASYMB(var),
set 1 r31, Dr30 )
```

Note: if the variable is initialised to the value (or address) of another variable, the code specified in section B should be used to load register r31 with its value.

Imported variables do not require any form of initialisation.
Because GHOF has only one common data area for both initialised and uninitialised data, it is necessary to ensure correct alignment by outputting each COFF data section as a block of data directives, followed optionally by a dummy data item padding out the block for alignment.

C source

```c
static char local = 'A';
double export;
```

MicroWay compiler code

```
.local: byte 65
.comm export 8
```

Heilos code

```
only the data directives are shown initialized local block
.data _local, 1
.data _align, 0, 1
.uninitialized global block
.data _export, 8
```

Note:
- only the data directives are shown for the Heilos part. The initialisation code would still need to be added to the init routine, as per normal.
- data should be output in a number of block equal to the number of separate data section under COFF (DATA, BSS, ...). The data directives for exported Functions must be output in a separate section following the last data section.

### B.3.6 Accessing variables

All non-local variables must be accessed through the module table.
To load the address of a variable (\text{var}) into a register (r3), generate the following code:

C source

\begin{verbatim}
int var;
...
/* load &var \rightarrow r3 */
\end{verbatim}

Helios code

\begin{verbatim}
patchinstr (PATCH860VAL, 
DATAMODULE (.var),
ld.l 0(r15), r31)
patchinstr (PATCH860HI, 
DATASYMB(.var),
orh 0, r0, r3 )
patchinstr (PATCH860LD, DATASYMB(.var),
orh 0, r3, r3 )
addu r3, r31, r3
\end{verbatim}

Note: the corresponding code should be used to store the value of a register into a variable.

To load the value of a variable (\text{ratio}) into a register pair (f6/f7), generate the following code:

C source

\begin{verbatim}
double ratio;
...
/* load ratio \rightarrow f6/f7 */
\end{verbatim}

Helios code

\begin{verbatim}
patchinstr (PATCH860VAL, 
DATAMODULE(.var),
ld.l 0(r15), r31)
patchinstr (PATCH860HI, 
DATASYMB(.var),
orh 0, r0, r30 )
patchinstr (PATCH860LD, DATASYMB(.var),
orh 0, r30, r30 )
fld.d r30(r31), f6
\end{verbatim}
B.3.7 Accessing functions

To load the address of a locally defined function `start()` into register r5, generate the following code:

C source

```c
void start();
...
/* start -> r5 */
```

Ref 11

MicroWay compiler code

```c
orh hX.start, r0, r5
or hX.start, r5, r5
```

Helios code

```c
br .tmp.label
ld .fix, r31
word .start
.tmp.label:
ld .4(r31), r30
addu r31, r30, r31
addu 4, r31, r5
```

Note: the code label at the start of function `start` is `.start` for MicroWay's compiler and `start` for Helios. The label `.tmp.label` is a temporary label and must be unique.
To load the address of an externally defined function `printf()` into register r5, generate the following code:

C source

```c
extern int printf();
```

Ref 12

### MicroWay compiler code

```
or hLstart, r0, r5
or hLstart, r5, r5
```

### Helios code

```
patchinstr (PATCH600VAL,
    DATAMODULE (printf),
    1d.1 0(r15), r31)
patchinstr (PATCH600HI,
    DATASYMB (printf),
    orh 0, r0, r30 )
patchinstr (PATCH600LO,
    DATASYMB (printf),
    or 0, r30, r30 )
1d.1 r30(r31), r5
```

Calling of a function is basically the same, but the Helios labels start with a . instead of a _.

Also any references to compiler generated labels should change the prefix from . to ..

C source

```c
start();
```

Ref 13

### MicroWay compiler code

```
call start
br .L3
```

### Helios code

```
call .start
...
br .L3
```

Note: similarly other branch instructions which take function labels should also use .label form.
B.3.8 Module Trailer

The module should end with a module trailer consisting of:

.data .MaxData, 0

B.4 Example Program C source

```c
/* Integer Variables */
static int localInt; /* localFn() localFn() */
static int localInitInt = 1; /* localFn() exportedFn() */
int exportedInt;
int exportedInitInt = 2; /* exportedFn() localFn() */
extern int importedInt; /* exportedFn2() exportedFn() */

/* Pointer Variables */
static int *localPtr; /* exportedFn() localFn() */
static int *localInitPtr = &localInt; /* exportedFn2() localFn() */
int *exportedPtr;
int *exportedInitPtr = &exportedInt; /* localFn() exportedFn() */
int *exportedInitPtr2 = &importedInt; /* localFn() exportedFn() */
extern int *importedPtr; /* exportedFn() localFn() */

/* Functions */
static int localFn (int a)
{
    *localPtr += *localInitPtr + *exportedPtr;
    exportedInitPtr = localPtr;
    exportedInitPtr2 = &exportedInt;
    *importedPtr++;
    localInt = localInitInt;
    exportedInitInt = localInt;
    return exportedInt;
}

text int importedFn (int);

int exportedFn (int a)
{
    int b = importedFn (a);
```
localPtr = importedPtr;
exportedPtr = importedPtr;
*exportedInitedPtr = *exportedInitedPtr2;
localFn (a);
localInitedInt = a;
exportedInt = a;
importedInt = b;
return exportedInitedInt;

/* Function pointers */
static int (*localFnPtr)(void);
static int (*localInitedFnPtr)(int) = localFn;
int (*exportedInitedFnPtr)(int) = exportedFn;
int (*exportedInitedFnPtr2)(int) = importedFn;
extern int (*importedFnPtr)(int);

int exportedFn2 (int a)
{
  int b = localFnPtr();
  localInitedPtr = &exportedInt;
  importedFnPtr = localFn;
  exportedInitedFnPtr(a);
  importedFnPtr (b);
  return importedInt;
}

B.5 Example Program MicroWay compiler output

/file "exl.c"

.text
.align 4
.align 8
.align 8
.localFn:

/orh ha
/ld.l 1
/orh ha
/ld.l 1
/ld.l 0(r24),r26
/ld.l 0(r25),r28
B

Changes to Microway C compiler output

17  adds r28,r26,r26
18  orh ha
19  ld.l l
20  ld.l 0(r27),r25
21  adds r26,r25,r25
22  st.l r25,0(r27)
23  orh ha
24  ld.l l
25  orh ha
26  st.l r24,l
27  orh h
28  or l
29  orh ha
30  st.l r28,l
31  orh ha
32  ld.l l
33  adds 4,r27,r27
34  orh ha
35  st.l r27,l
36  orh ha
37  ld.l l
38  orh ha
39  st.l r26,l
40  orh ha
41  ld.l l
42  orh ha
43  st.l r25,l
44  orh ha
45  ld.l l
46  // .ef
47  bri rl
48  nob
49  .align 4
50  .data
51  .L9:
52  _localIntitedPtr: .long _localInt
53  .L10:
54  _localIntitedInt: .byte 1,0,0,0
55  .lcomm _localInt,4
56  .lcomm _localPtr,4
57  // .a r16 local
58
59  .text
60  .align 4
61  .align 8
62  .exportedFn:
Changes to Microway C compiler output

64  adds -16,sp,sp
65  st.l r4,12(sp)
66  st.l r5,8(sp)
67  st.l r1,4(sp)
68  mov r16,r4
69  // .bf
70  mov r4,r16
71  call _importedFn
72  nop
73  mov r16,r5
74  orh h
75  or 1
76  orh ha
77  st.l r26,l
78  orh ha
79  ld.l l
80  orh ha
81  st.l r25,l
82  orh ha
83  ld.l l
84  orh ha
85  ld.l l
86  ld.l 0(r28),r27
87  ld.l 0(r24),r26
88  adds r26,r27,r27
89  st.l r27,0(r28)
90  mov r4,r16
91  call _localFn
92  nop
93  orh ha
94  st.l r4,l
95  orh ha
96  st.l r4,l
97  orh ha
98  st.l r5,l
99  orh ha
100 ld.l l
101 // .ef
102 ld.l 4(sp),r1
103 ld.l 0(sp),r5
104 ld.l 12(sp),r4
105 adds 16,sp,sp
106 bri r1
107 nop
108 .align 4
109 .data
110 // b r5 local
Changes to Microway C compiler output

111
112 // .r4 local
113
114 .text
115 .align 4
116 .align 8
117 _exportedFn2:
118 adds -16,sp,sp
119 st.l r4,12(sp)
120 st.l r5,8(sp)
121 st.l r1,4(sp)
122 mov r16,r4
123 // .bf
124 orh ha
125 ld.l l
126 calli r30
127 nop
128 mov r16,r5
129 orh h
130 or l
131 orh ha
132 st.l r26,l
133 orh h
134 or l
135 orh ha
136 st.l r25,l
137 mov r4,r16
138 orh ha
139 ld.l l
140 calli r30
141 nop
142 mov r5,r16
143 orh ha
144 ld.l l
145 calli r30
146 nop
147 orh ha
148 ld.l l
149 // .ef
150 ld.l 4(sp),r1
151 ld.l 8(sp),r5
152 ld.l 12(sp),r4
153 adds 16,sp,sp
154 bri r1
155 nop
156 .align 4
157 .data
B.6 Example Program Helios assembler

module

word 0x60f160f1

word .ModEnd-.ModStart
Changes to Microway C compiler output

5 byte "exl.s"
6 space 27
7 word modnum
8 word 0x1000
9 word .MaxData
10 init
11
12 // File name 'ex1.c' from 'ex1.s'
13 // IDPC -X22 -X70 -X74 -X80 -X83 -X84 -X85 -X151 -X153 -X188 -X244 -X247 -X254
16
17 align 4
18 align 8
19 .localFn:
20 // .bf
21 // (.L9).1 -> r25
22 // load value of local data .L9 -> r25
23 patchinstr (PATCH860VAL, DATAMODULE(.L9), ld.l 0(r15), r31)
24 patchinstr (PATCH860HI, DATASYMB(.L9), orh 0, r0, r30)
25 patchinstr (PATCH860LO, DATASYMB(.L9), or 0, r30, r30)
26 ld.l r30(r31), r25
27 // (.exportedPtr).1 -> r24
28 // load value of local data ._exportedPtr -> r24
29 patchinstr (PATCH860VAL, DATAMODULE(.exportedPtr), ld.l 0(r15), r31)
30 patchinstr (PATCH860HI, DATASYMB(.exportedPtr), orh 0, r0, r30)
31 patchinstr (PATCH860LO, DATASYMB(.exportedPtr), or 0, r30, r30)
32 ld.l r30(r31), r24
33 ld.l 0(r24), r26
34 ld.l 0(r25), r28
35 adds r28, r26, r26
36 // (.localPtr).1 -> r27
37 // load value of local data .localPtr -> r27
38 patchinstr (PATCH860VAL, DATAMODULE(.localPtr), ld.l 0(r15), r31)
39 patchinstr (PATCH860HI, DATASYMB(.localPtr), orh 0, r0, r30)
40 patchinstr (PATCH860LO, DATASYMB(.localPtr), or 0, r30, r30)
41 ld.l r30(r31), r27
42 ld.l 0(r27), r25
43 adds r26, r25, r26
44 st.l r26, 0(r27)
45 // (.localPtr).1 -> r24
46 // load value of local data .localPtr -> r24
47 patchinstr (PATCH860VAL, DATAMODULE(.localPtr), ld.l 0(r15), r31)
48 patchinstr (PATCH860HI, DATASYMB(.localPtr), orh 0, r0, r30)
49 patchinstr (PATCH860LO, DATASYMB(.localPtr), or 0, r30, r30)
50 ld.l r30(r31), r24
51 // (.exportedInitPtr).1 <= r24
Changes to Microway C compiler output

52 // load value of local data _exportedInitPtr -> r24
53 patchinstr (PATCH860VAL, DATAMODULE(_exportedInitPtr), ld.l 0(r15), r31)
54 patchinstr (PATCH860HI, DATAASYMB(_exportedInitPtr), orh 0, r0, r30)
55 patchinstr (PATCH860LO, DATAASYMB(_exportedInitPtr), or 0, r30, r30)
56 st.l r24, r30(r31)
57 // exportedInt -> r28
58 // load addr of local data _exportedInt -> r28
59 patchinstr (PATCH860VAL, DATAMODULE(_exportedInt), ld.l 0(r15), r31)
60 patchinstr (PATCH860HI, DATAASYMB(_exportedInt), orh 0, r0, r28)
61 patchinstr (PATCH860LO, DATAASYMB(_exportedInt), or 0, r28, r28)
62 addu r28, r31, r28
63 // (_exportedInitPtr2).1 <- r28
64 // load value of local data _exportedInitPtr2 -> r28
65 patchinstr (PATCH860VAL, DATAMODULE(_exportedInitPtr2), ld.l 0(r15), r31)
66 patchinstr (PATCH860HI, DATAASYMB(_exportedInitPtr2), orh 0, r0, r30)
67 patchinstr (PATCH860LO, DATAASYMB(_exportedInitPtr2), or 0, r30, r30)
68 st.l r28, r30(r31)
69 // (_importedPtr).1 -> r27
70 // load value of external _importedPtr -> r27
71 patchinstr (PATCH860VAL, DATAMODULE(_importedPtr), ld.l 0(r15), r31)
72 patchinstr (PATCH860HI, DATAASYMB(_importedPtr), orh 0, r0, r30)
73 patchinstr (PATCH860LO, DATAASYMB(_importedPtr), or 0, r30, r30)
74 ld.l r30(r31), r27
75 adding 4, r27, r27
76 // (_importedPtr).1 <- r27
77 // load value of external _importedPtr -> r27
78 patchinstr (PATCH860VAL, DATAMODULE(_importedPtr), ld.l 0(r15), r31)
79 patchinstr (PATCH860HI, DATAASYMB(_importedPtr), orh 0, r0, r30)
80 patchinstr (PATCH860LO, DATAASYMB(_importedPtr), or 0, r30, r30)
81 st.l r27, r30(r31)
82 // (_L10).1 -> r26
83 // load value of local data _L10 -> r26
84 patchinstr (PATCH860VAL, DATAMODULE(_L10), ld.l 0(r15), r31)
85 patchinstr (PATCH860HI, DATAASYMB(_L10), orh 0, r0, r30)
86 patchinstr (PATCH860LO, DATAASYMB(_L10), or 0, r30, r30)
87 ld.l r30(r31), r26
88 // (_localInt).1 <- r26
89 // load value of local data _localInt -> r26
90 patchinstr (PATCH860VAL, DATAMODULE(_localInt), ld.l 0(r15), r31)
91 patchinstr (PATCH860HI, DATAASYMB(_localInt), orh 0, r0, r30)
92 patchinstr (PATCH860LO, DATAASYMB(_localInt), or 0, r30, r30)
93 st.l r26, r30(r31)
94 // (_localInt).1 -> r25
95 // load value of local data _localInt -> r25
96 patchinstr (PATCH860VAL, DATAMODULE(_localInt), ld.l 0(r15), r31)
97 patchinstr (PATCH860HI, DATAASYMB(_localInt), orh 0, r0, r30)
98 patchinstr (PATCH860LO, DATAASYMB(_localInt), or 0, r30, r30)
99  ld.l r30(r31), r25
100  // (_exportedlnt).1 <- r25
101  // load value of local data _exportedlnt1 -> r25
102  patchinstr (PATCH860VAL, DATAMODULE(_exportedlnt1), ld.l 0(r15), r31)
103  patchinstr (PATCH860HI, DATASYMB(_exportedlnt1), orh 0, r0, r30)
104  patchinstr (PATCH860LO, DATASYMB(_exportedlnt1), or 0, r30, r30)
105  st.l r25, r30(r31)
106  // (_exportedInt).1 -> r16
107  // load value of local data _exportedInt -> r16
108  patchinstr (PATCH860VAL, DATAMODULE(_exportedInt), ld.l 0(r15), r31)
109  patchinstr (PATCH860HI, DATASYMB(_exportedInt), orh 0, r0, r30)
110  patchinstr (PATCH860LO, DATASYMB(_exportedInt), or 0, r30, r30)
111  ld.l r30(r31), r16
112  // .ef
113  bri r1
114  nop
115  align 4
116  ..L9:
117  ..L10:
118
119  // _a r16 local
120
121  align 4
122  align 8
123  .exportedFn:
124  adds -16,sp,sp
125  st.l r4,12(sp)
126  st.l r5,8(sp)
127  st.l r1,4(sp)
128  mov r16,r4
129  // .bf
130  mov r4,r16
131  call .importedFn
132  nop
133  mov r16,r5
134  // _importedInt -> r26
135  // load addr of external _importedInt -> r26
136  patchinstr (PATCH860VAL, DATAMODULE(_importedInt), ld.l 0(r15), r31)
137  patchinstr (PATCH860HI, DATASYMB(_importedInt), orh 0, r0, r26)
138  patchinstr (PATCH860LO, DATASYMB(_importedInt), or 0, r26, r26)
139  addu r26, r31, r26
140  // (_localPtr).1 <- r26
141  // load value of local data _localPtr -> r26
142  patchinstr (PATCH860VAL, DATAMODULE(_localPtr), ld.l 0(r15), r31)
143  patchinstr (PATCH860HI, DATASYMB(_localPtr), orh 0, r0, r30)
144  patchinstr (PATCH860LO, DATASYMB(_localPtr), or 0, r30, r30)
145  st.l r26, r30(r31)
Changes to Microway C compiler output

146 // (_importedPtr).1 -> r25
147 // load value of external _importedPtr -> r25
148 patchinstr (PATCH860VAL, DATAMODULE(_importedPtr), ld.1 0(r15), r31)
149 patchinstr (PATCH860HI, DATASYMB(_importedPtr), orh 0, r0, r30)
150 patchinstr (PATCH860L0, DATASYMB(_importedPtr), or 0, r30, r30)
151 ld.1 r30(r31), r25
152 // (_exportedPtr).1 <- r25
153 // load value of local data _exportedPtr -> r25
154 patchinstr (PATCH860VAL, DATAMODULE(_exportedPtr), ld.1 0(r15), r31)
155 patchinstr (PATCH860HI, DATASYMB(_exportedPtr), or 0, r0, r30)
156 patchinstr (PATCH860L0, DATASYMB(_exportedPtr), or 0, r30, r30)
157 st.1 r25, r30(r31)
158 // (_exportedInit2).1 -> r24
159 // load value of local data _exportedInit2 -> r24
160 patchinstr (PATCH860VAL, DATAMODULE(_exportedInit2), ld.1 0(r15), r31)
161 patchinstr (PATCH860HI, DATASYMB(_exportedInit2), or 0, r0, r30)
162 patchinstr (PATCH860L0, DATASYMB(_exportedInit2), or 0, r30, r30)
163 ld.1 r30(r31), r24
164 // (_exportedInit).1 <- r28
165 // load value of local data _exportedInit -> r28
166 patchinstr (PATCH860VAL, DATAMODULE(_exportedInit), ld.1 0(r15), r31)
167 patchinstr (PATCH860HI, DATASYMB(_exportedInit), or 0, r0, r30)
168 patchinstr (PATCH860L0, DATASYMB(_exportedInit), or 0, r30, r30)
169 ld.1 r30(r31), r28
170 ld.1 0(r28), r27
171 ld.1 0(r24), r26
172 adds r26, r27, r27
173 st.1 r27, 0(r28)
174 mov r4, r16
175 call .localFn
176 nop
177 // (.L10).1 <- r4
178 // load value of local data ..L10 -> r4
179 patchinstr (PATCH860VAL, DATAMODULE(..L10), ld.1 0(r15), r31)
180 patchinstr (PATCH860HI, DATASYMB(..L10), orh 0, r0, r30)
181 patchinstr (PATCH860L0, DATASYMB(..L10), or 0, r30, r30)
182 st.1 r4, r30(r31)
183 // (_exportedInt).1 <- r4
184 // load value of local data _exportedInt -> r4
185 patchinstr (PATCH860VAL, DATAMODULE(_exportedInt), ld.1 0(r15), r31)
186 patchinstr (PATCH860HI, DATASYMB(_exportedInt), or 0, r0, r30)
187 patchinstr (PATCH860L0, DATASYMB(_exportedInt), or 0, r30, r30)
188 st.1 r4, r30(r31)
189 // (_importedInt).1 <- r5
190 // load value of external _importedInt -> r5
191 patchinstr (PATCH860VAL, DATAMODULE(_importedInt), ld.1 0(r15), r31)
192 patchinstr (PATCH860HI, DATASYMB(_importedInt), or 0, r0, r30)
Changes to Microway C compiler output

193  patchinstr (PATCH860LO, DATASYMB(_importedInt), or 0, r30, r30)
194  st.l r5, r30(r31)
195  // (_exportedInitInt).1 -> r16
196  // load value of local data _exportedInitInt -> r16
197  patchinstr (PATCH860VAL, DATAMODULE(_exportedInitInt), ld.l 0(r15), r31)
198  patchinstr (PATCH860HI, DATASYHB(_exportedInitInt), orh 0, r0, r30)
199  patchinstr (PATCH860LO, DATASYHB(_exportedInitInt), or 0, r30, r30)
200  ld.l r30(r31), r16
201  // .ef
202  ld.l 4(sp), r1
203  ld.l 8(sp), r5
204  ld.l 12(sp), r4
205  adds 16, sp, sp
206  bri r1
207  nop
208  align 4
209  // b r5 local
210  // a r4 local
211  align 4
212  align 8
213  .exportedFn2:
214  adds -16, sp, sp
215  st.l r4, 12(sp)
216  st.l r5, 8(sp)
217  st.l r1, 4(sp)
218  mov r16, r4
219  // .bf
220  // (_localFnPtr).1 -> r30
221  // load value of local data _localFnPtr -> r30
222  patchinstr (PATCH860VAL, DATAMODULE(_localFnPtr), ld.l 0(r15), r31)
223  patchinstr (PATCH860HI, DATASYHB(_localFnPtr), orh 0, r0, r30)
224  patchinstr (PATCH860LO, DATASYHB(_localFnPtr), or 0, r30, r30)
225  ld.l r30(r31), r30
226  calli r30
227  nop
228  mov r16, r5
229  // _exportedInt -> r26
230  // load addr of local data _exportedInt -> r26
231  patchinstr (PATCH860VAL, DATAMODULE(_exportedInt), ld.l 0(r15), r31)
232  patchinstr (PATCH860HI, DATASYHB(_exportedInt), orh 0, r0, r26)
233  patchinstr (PATCH860LO, DATASYHB(_exportedInt), or 0, r26, r26)
234  adds r26, r31, r26
235  // (.L9).1 <- r26
236  // load value of local data ..L9 -> r26
237  patchinstr (PATCH860VAL, DATAMODULE(.L9), ld.l 0(r15), r31)
B  Changes to Microway C compiler output  199

240  patchinstr (PATCH860HI, DATASYMB(.L9), orh 0, r0, r30)
241  patchinstr (PATCH860LO, DATASYMB(.L9), or 0, r30, r30)
242  st.l r26, r30(r31)
243  // _localFn -> r25
244  // load addr of external _localFn -> r25
245  br .tmp.0001
246  ld.c fir, r31
247  word .localFn
248  .tmp.0001:
249  ld.l 4(r31), r30
250  addu r31, r30, r31
251  addu 4, r31, r25
252  // (_importedFnPtr).1 <- r25
253  // load value of external _importedFnPtr -> r25
254  patchinstr (PATCH860VAL, DATAMODULE(_importedFnPtr), ld.l 0(r15), r31)
255  patchinstr (PATCH860HI, DATASYMB(_importedFnPtr), orh 0, r0, r30)
256  patchinstr (PATCH860LO, DATASYMB(_importedFnPtr), or 0, r30, r30)
257  st.l r25, r30(r31)
258  mov r4, r16
259  // (_exportedInitiatedFnPtr).1 -> r30
260  // load value of local data _exportedInitiatedFnPtr -> r30
261  patchinstr (PATCH860VAL, DATAMODULE(_exportedInitiatedFnPtr), ld.l 0(r15), r31)
262  patchinstr (PATCH860HI, DATASYMB(_exportedInitiatedFnPtr), orh 0, r0, r30)
263  patchinstr (PATCH860LO, DATASYMB(_exportedInitiatedFnPtr), or 0, r30, r30)
264  ld.l r30(r31), r30
265  calli r30
266  nop
267  mov r5, r16
268  // (_importedFnPtr).1 -> r30
269  // load value of external _importedFnPtr -> r30
270  patchinstr (PATCH860VAL, DATAMODULE(_importedFnPtr), ld.l 0(r15), r31)
271  patchinstr (PATCH860HI, DATASYMB(_importedFnPtr), orh 0, r0, r30)
272  patchinstr (PATCH860LO, DATASYMB(_importedFnPtr), or 0, r30, r30)
273  ld.l r30(r31), r30
274  calli r30
275  nop
276  // (_importedInt).1 -> r16
277  // load value of external _importedInt -> r16
278  patchinstr (PATCH860VAL, DATAMODULE(_importedInt), ld.l 0(r15), r31)
279  patchinstr (PATCH860HI, DATASYMB(_importedInt), orh 0, r0, r30)
280  patchinstr (PATCH860LO, DATASYMB(_importedInt), or 0, r30, r30)
281  ld.l r30(r31), r16
282  // .af
283  ld.l 4(sp), r1
284  ld.l 8(sp), r5
285  ld.l 12(sp), r4
286  add.s 16, sp, sp
Changes to Microway C compiler output

287 bri r1
288  nop
289  align 4
290  //. r5 local
291  ..L30:
292
293  //. r4 local
294
295  //.localInt .localInt static
296  //.localIntedInt .L10 static
297  //.importedInt .importedInt import
298  //.localPtr .localPtr static
299  //.localIntedPtr .L9 static
300  //.importedPtr .importedPtr import
301  //.localFnPtr .localFnPtr static
302  //.localIntedFnPtr .L30 static
303  //.importedFnPtr .importedFnPtr import
304  ..L37:
305  ..L39:
306  ..L40:
307  ..L41:
308  ..L42:
309
310  //
311  // Stubs
312  // for . importedFn
313  .importedFn:
314  patchinstr (PATCH860VAL, DATAMODULE (_importedFn), ld.1 0(r15), r30)
315  patchinstr (PATCH860X1, CODESYMB (_importedFn), orh 0, r0, r31)
316  patchinstr (PATCH860LO, CODESYMB (_importedFn), or 0, r31, r31)
317  ld.1 r31(r30), r30
318  bri r30
319  nop
320  //
321  // Data section
322  // Uninitialised global
323  data _exportedPtr, 4
324  export _exportedPtr
325  data _exportedInt, 4
326  export _exportedInt
327  data __align.000, 8
328  // Uninitialised local
329  data __localInt, 4
330  data __localPtr, 4
331  data __localFnPtr, 4
332  data __align.001, 4
333  // Initialised
data _localInitedInt, 4
data _exportedInitedInt, 4
export _exportedInitedInt
data __align.002, 8

// Local Functions
data _exportedFn, 4
export _exportedFn
data _exportedFn2, 4
export _exportedFn2

// Initialisation routine

init

patchinstr (PATCH860VAL, SHIFT (-2, MODWNUM), ld.i 0(r15), r16)

// Initiating '_localInitedPtr' to address of _localInt
patchinstr (PATCH860HI, DATASYMB(_localInt), orh 0, r0, r17)
patchinstr (PATCH860LO, DATASYMB(_localInt), or 0, r17, r17)
add r17, r16, r17

patchinstr (PATCH860HI, DATASYMB(_localInitedInt), orh 0, r0, r30)
patchinstr (PATCH860LO, DATASYMB(_localInitedInt), or 0, r30, r30)
st.1 r17, r30(r16)

// Initiating '_localInitedInt' to 1 0 0 0
or 0x1, r0, r17

patchinstr (PATCH860HI, DATASYMB(_localInitedInt), orh 0, r0, r30)
patchinstr (PATCH860LO, DATASYMB(_localInitedInt), or 0, r30, r30)
st.1 r17, r30(r16)

// Initiating '_localInitedFnPtr' to address of _localFn
br .tap.0002

ld.c fir, r31

.word .localFn

.tap.0002:
ld.i 4(r31), r30
add r31, r30, r31
add 4, r31, r17

patchinstr (PATCH860HI, DATASYMB(_localInitedFnPtr), orh 0, r0, r30)
patchinstr (PATCH860LO, DATASYMB(_localInitedFnPtr), or 0, r30, r30)
st.1 r17, r30(r16)

// Initiating '_exportedInitedInt' to 2 0 0 0
or 0x2, r0, r17

patchinstr (PATCH860HI, DATASYMB(_exportedInitedInt), orh 0, r0, r30)
patchinstr (PATCH860LO, DATASYMB(_exportedInitedInt), or 0, r30, r30)
st.1 r17, r30(r16)

// Initiating '_exportedInitedPtr' to address of _exportedInt
patchinstr (PATCH860HI, DATASYMB(_exportedInt), orh 0, r0, r17)
patchinstr (PATCH860LO, DATASYMB(_exportedInt), or 0, r17, r17)
add r17, r16, r17

patchinstr (PATCH860HI, DATASYMB(_exportedInitedPtr), orh 0, r0, r30)
Changes to Microway C compiler output

381 patchinstr (PATCH860LO, DATASYMB(_exportedInitedPtr), or 0, r30, r30)
382 st.l r17, r30(r16)
383 // Initing '_exportedInitedPtr2' to address of _importedInt
384 patchinstr (PATCH860HI, DATASYMB(_importedInt), orh 0, r0, r17)
385 patchinstr (PATCH860LO, DATASYMB(_importedInt), or 0, r17, r17)
386 addu r17, r16, r17
387 patchinstr (PATCH860HI, DATASYMB(_exportedInitedPtr2), orh 0, r0, r30)
388 patchinstr (PATCH860LO, DATASYMB(_exportedInitedPtr2), or 0, r30, r30)
389 st.l r17, r30(r16)
390 // Initing '_exportedInitedFnPtr' to address of _exportedFn
391 br .tmp.0003
392  ld.c fir, r31
393  word .exportedFn
394 .tmp.0003:
395  ld.1 4(r31), r30
396  addu r31, r30, r31
397  addu 4, r31, r17
398 patchinstr (PATCH860HI, DATASYMB(_exportedInitedFnPtr), orh 0, r0, r30)
399 patchinstr (PATCH860LO, DATASYMB(_exportedInitedFnPtr), or 0, r30, r30)
400 st.l r17, r30(r16)
401 // Initing '_exportedInitedFnPtr2' to _importedFn
402 patchinstr (PATCH860HI, DATASYMB(_importedFn), orh 0, r0, r30)
403 patchinstr (PATCH860LO, DATASYMB(_importedFn), or 0, r30, r30)
404 ld.1 r30(r16), r17
405 patchinstr (PATCH860HI, DATASYMB(_exportedInitedFnPtr), orh 0, r0, r30)
406 patchinstr (PATCH860LO, DATASYMB(_exportedInitedFnPtr), or 0, r30, r30)
407 st.l r17, r30(r16)
408 // exporting _exportedFn
409 br init.exportedFn
410  ld.c fir, r30
411  word .exportedFn
412 init.exportedFn:
413  ld.1 4(r30), r31
414  addu r31, r31, r31
415  addu r30, r31, r30
416 patchinstr (PATCH860HA, DATASYMB(_exportedFn), addu 0, r16, r31)
417 patchinstr (PATCH860LO, DATASYMB(_exportedFn), st.l r30, 0(r31) )
418 // exporting _exportedFn2
419 br init.exportedFn2
420  ld.c fir, r30
421  word .exportedFn2
422 init.exportedFn2:
423  ld.1 4(r30), r31
424  addu r31, r31, r31
425  addu r30, r31, r30
426 patchinstr (PATCH860HA, DATASYMB(_exportedFn2), addu 0, r16, r31)
427 patchinstr (PATCH860LO, DATASYMB(_exportedFn2), st.l r30, 0(r31) )
Bri rl
nop
.MemEnd:
.data .MaxData, 0
Appendix C

Internal Reports

The following pages contain three internal reports, produced by the author and referenced in this thesis. They are included here because obtaining them might otherwise be difficult. The reports are shown in their original format, with each page reduced to 85% of its size to fit.
Progress report
Porting of Helios to i860

Jerzy M. Grajewski

December 13th, 1991

The following work was done between November 25th and December 13th:

• The compiler -s1 option was corrected to generate (hopefully) correct code for static variables.

• The problem with assembler functions calling C functions was tackled. The C compiler exports code labels which point to the second instruction of the function and corrects for this when generating calls to external functions by adding a patch. A macro i860br_toC was written which generates a branch to the specified label minus 4 bytes. This has to be used mostly in server and C startup code.

• The etc/init program was compiled. This now loads and runs looking for etc/inittab.

• The sources for login were restored off tape. However, to compile login a number of libraries are required. The sources for these were recovered off tape.

• The RmLib, SessionLib and Clib were converted to i860. In a few places in Clib, global functions and variables were not allocated data space in the assembler header. Since the modified -s1 option does not allocate space for exported variables, one of two solutions could be used:
  - allocate space for these functions and variables explicitly in the assembler header
  - change the C declarations to local (static), for which the data area is allocated by the compiler

A mixture of the two methods was used in the end; declaring space in assembler where functions were declared as extern in headers and rededucing them as static otherwise. However, the fact that modifications had
to be made by myself to machine-independent code suggests that perhaps the `z1' option is not working correctly. Hence section 7.7 of this document describes my interpretation of the correct behaviour of this option.

- The Clib required some floating-point support (in particular a floating-point divide routine). This was written.

1 The fixed z1 option

The following describes my interpretation of the documentation describing the behaviour of the NorCroft compiler `-z1' option.

Local Variables A DATA directive should be generated (reserving space in the module's data area). The initialisation for the data should be generated, but should use DATASYMB to obtain the offset of the data in the data area, since the actual position of the data is determined by the linker. Similarly, any code using the data should generate DATASYMB rather than assuming a specific position in the data area.

Local Functions No directives need to be generated – an unexported LABEL will suffice.

Exported Variables No space allocation or initialisation code is generated. Also, any use of the variables may not assume a particular position in the data area and so should use DATASYMB.

Exported Functions As well as a LABEL for the code, initialisation for the `fn' variable should be generated. However, since the position of the `fn' in the data area is determined by the linker, a DATASYMB directive should be generated to find its address. However code label is fixed in relation to the initialisation code, so `fn' may be referenced directly.

Imported Variables No code needs to be generated here. Access to the values should be done via DATAHMODULE and DATASYMB as usual.

Imported Functions Here no stubs should be generated. However, calls should still be generated to the code directly (`fn'), as stubs will usually be supplied by the assembler module. Since the position of these stubs is unknown at compilation time, a LABELREF should be used to obtain their location.

Other changes No module header or trailer should be generated.
Progress report
Porting of Helios to i860

Jerzy M. Grzejewski

January 26th, 1992

The following work was done between January 6th and January 24th, 1992.

- Found a re-entancy problem in the executive code. The trap routine is designed to be non-reentrant for each process, that is it may be called only once for any one current process. Unfortunately, under certain circumstances (described in section 1) it was called reentrantly, corrupting the saved state. This was corrected.

- The above problem highlighted the remaining timing problem in the executive, and in the interests of robustness, which will lead to a faster port, the majority of the C code for the executive was

- The aim assembler for the i860 was found to generate incorrect object files. When generating a branch or call instruction, the assembler used to generate the sequence:

  \[
  \text{WORD instr H68K.ADD H68K.SHIFT } -2 \text{ LABELOFF label}
  \]

  instead of

  \[
  \text{WORD instr I860.IA I860.SHIFT } -2 \text{ LABELOFF label}
  \]

  The H68K.ADD patch added the value of the offset to the instruction, without restricting the jump offset to bits 25 - 0. The correct sequence was already generated by the C compiler, so the linker recognised the I860.IA patch.

- Fixed the assembler branch offset bug. Previously, the macro I860BR.ZERO had to be used to generate a branch from assembler code to C functions. The inconsistency in the branch addressing was fully investigated and corrected, removing need for the above macro. For further information, see section 2.
1. Trap handler reentrancy problem

The trap handler for the i860 is organised as follows:

- The hardware executes a jump to location 0xfffffffff000.
- At that location, a branch to location 0xffffffff000 is located. This gives enough space for the trap handler.

- The assembler routine Trap.Entry saves the state of the process into the save state area in the current process, switches to system stack and jumps through a pointer in exec pointers structure.
- The pointer in Exec.Ptrs is set up to point to C function Trap.Routine, which handles the trap. The C code attempts to quickly switch over to a per-process stack, allocating another stack for future system calls. The per-process stack is freed before returning from Trap.Routine.
- When the C code returns, the assembler routine Trap.Exit takes over restoring CPU state.

This setup is robust and minimal amount of code has to be written in assembler, but since only one save area exists per process, the trap routine is not reentrant for any one process.

The original code used a trap call in the definition of the executive function System(). This caused problems when:

- A process was interrupted by an interrupt
- The Trap.Routine attempted to switch to per-user stack, but the current stack table is used up, so a call to AllocMem which in turn calls Allocate using System.
- Since the original process was not running at priority 0, the System generates a trap.
- The new trap overwrites the saved state from the original interrupt.

To solve the problem, the System function was re-written not to use trap.

The executive functions were examined for code which needs to be executed in processor's Supervisor mode. The only function which is called from user mode and contains such code is SchedulerDispatch(). A assembler function was set up to use a trap to call this function.

1 The save pointers are placed at the top of memory, so from 0xffffffff0 the trap entry routine would have to fit in less than 256 bytes.
2 There has never been much reason for System to use trap. As well as solving the current problem, the modification should produce performance benefits.
Also, to prevent future problems of this sort, as well as validating my solution, a number of assertions were added to the functions in the executive. Most functions were tidied up and comments added. A header file assert.h was added to the executive code to contain the assertion generating code.

2 Internal and external branches and calls

The external branches from C to assembler code and vice versa has in the past caused a considerable degree of confusion. The C compiler exported patches for external labels which contained a subtraction of 4 bytes from the label offset. The i860 trap handler needs to examine instruction previous to the one on which the fault occurred, and this forces the software writers to ensure that instruction previous to the first instruction in each function is valid. However, this did not explain the subtraction of 4 bytes.

The temporary workaround used to that time involved using a macro i860br_toC, which calculated offset -4. After a re-investigation, what is believed to be the real reason for the -4 was discovered. The i860 branch offsets are counted from the delayed slot instruction, whereas the label offset directive counts offset from the branch instruction. The subtraction of 4 bytes from the label cancels out the implicit addition of 4 bytes by the measuring of the offset from the delayed slot.

The assembler was corrected to generate patches for external branches similar to the ones generated by the C compiler. The need for the i860br_toC macro was hence removed. To verify this correction, a test of branching was conducted, testing all possible combinations in the following categories:

- C to assembler, C to C, assembler to C and assembler to assembler
- Internal and external branches
- Forward and backward branches

The corrected assembler was found to generate correct code for all tests.
The Helios memory management on the i860

Jerry M. Grzejewski
September 21, 1991

1 Introduction

This document describes in detail the memory management used for Helios 1.2.1 operating system on the i860 processor. The hardware used is a Microway i860 card, version 1.0.

2 The Memory Management Unit

The i860 has an on-board MMU, which is identical in function to that found on intel 80386 and 80486 processors. Logical address is split into three parts:

<table>
<thead>
<tr>
<th>dir</th>
<th>page</th>
<th>offset</th>
</tr>
</thead>
</table>

where

- dir is a 10 bit wide offset in page directory, which combined with the directory base gives the page table base
- page is a 10 bit wide offset in the page table, which combined with the page table base gives the page address
- offset is a 12 bit offset within a page

The main purpose of the mapping is to ensure that the trap handler, which must be at virtual location $ffff0000h$ is mapped to RAM. The final arrangement involved mapping the 8 megabytes of memory at the base address of $00000000h$ and physical devices at their real addresses. The complete overview of the MMU set-up may be seen in figure 1.

The Helios recommended generic memory map (as shown in the Executive Porting Guide) was followed in placing of code and data in memory. The key element of the Helios memory map is the ExecPtrs structure just at the top of the virtual address space\(^1\). This structure holds the pointers to other entities which may be therefore moved in memory. Figure 2 demonstrates the complete Helios memory map. The dashed area are of variable size.

Please note that the physical address decoding means that the top nibble (4 bits) of the address are ignored. This counteracted hardware problems with the A stepping of the processor, and is denoted on figure 1 by the symbol $\times$ as the most significant figure.

The ExecPtrs structure is always mapped to the top of virtual memory, and contains a number of pointers to executive areas as well as other data. The function of each of its elements is described below.

\(^1\)This structure is therefore mapped into the same page as the trap handler routine
As already described, the trap handler has to be located at address $ffff000h$. However, this only gives 256 bytes before the top of address space, which are shared with the executive pointers. This is not sufficient for even the simplest trap handler, so instead the assembler routine `installTrapHandler` installs a stub at $ffff000h$, which calls the code at $ffff000h$, and it is there that the proper entry code is located. After saving the state, the trap entry routine at $ffff000h$ calls the C function, whose address is stored in the `TRAPRTHP` member of `ExecPtrs`. 
Figure 2: Helios memory map
Annotated Bibliography

Development of analog computing in the U.K. and U.S. between 1945-1960s; brief overview of earlier development.

A description of solid-state technology advances for analog and hybrid computers.

Historical overview of developments in combinatorial algorithms.

Argues that 0.3μm is close to the practical lower limit for silicon VLSI circuits.

Turing Award lecture describing micropipelines (simple asynchronous decoupling mechanism) and outlining the limitations of synchronous logic.

Taxonomy of asynchronous circuits. Describes a locally synchronous design, which avoids many async problems.


[11] Phillip Ein-Dor. "Grosch's law re-visited: CPU power and the cost of computation". Communications of the ACM, 28(2), 142-151, February 1985. Authors split CPUs into five groups. Results suggest that Grosch's law holds within groups but not between them.


Results of 1981–1985 period suggest no significant economies of scale. Includes overview of earlier studies.

Overview of CSP, NIL, Occam, Ada, Linda, Orca and others.

Paradigms for parallel programming illustrated in Linda.

Efficient and portable programming environment called Chameleon, based around high-level abstractions implemented in C++.

Work performed on a parallel expert system with a real-time task scheduler. Includes a description of a hierarchical shared-memory computer built at the University of Bath.

Description of pipelines in the implementation of the i860.

Description of commercial efforts in porting software from the i860 to TMS320C40.
Overview of the architecture of the i860.

Description of the Mark I, Atlas and MU 5 computers.

Survey of many early RISC CPUs features and performance.

This article coined the phrase RISC.

Usage of processor instructions by assembler programmers. Indicates some redundancy and a proportion of unused operations.

General discussion of a regular structure of assembler for more efficient compiler utilisation.

Description of the RISC-I design and architecture, including register windows and delayed branching.

Overview of high-level statement usage frequencies. Proposes an architecture which uses variable-length encoding of instructions.

A comprehensive overview of optimisation techniques for high-performance architectures including superscalar, vector and multiprocessor.

Description of the features common in high-level programming languages.

Compares subsets of the VAX architecture. Suggests that RISC programs require 2–2½ more memory and even with large caches generate more misses.

Compares MIPS 2000 and VAX 8700 using SPEC benchmarks. RISC uses twice as many instructions, but its 5½ fewer cycles per instruction results in better performance.

Overview of modern RISC, including M88000, i860, i960, Sparc, AMD 29000 and MIPS R3000.
Annotated Bibliography

Describes emulation of MC 68020 using a MIPS instruction set, intended for use with a GaAs processor. Results in a mediocre performance.

Description of the AMD K5 processor, which is compatible with the Pentium.

Description of the CDC 6600 architecture.


Describes the 88110 implementation. Also suggests CMOS is better suited to superscalar than superpipelined architectures, since superpipelined needs faster devices.

Describes novel RISC architecture aimed at reducing impact on memory. Uses internal queues to decouple internal units and allows a variable number of delay slots, between 0 and 7.

Describes a number of branch optimisations common in RISCs, including branch prediction, delayed branching, speculative execution and multiple instruction stream multiplexing.
Annotated Bibliography


Describes many branch penalty avoidance methods. Introduces COBRA, which combines some of these techniques to achieve better performance.


Contains data on static branch prediction having an average success rate of 85%.


Processor architecture book, which covers CISC and RISC principles, illustrated by examples of all major contemporary microprocessors, including 80x86, M 68000, NS 32000, i860, M 88000, AMD 29000, Sparc and RS/6000.


Overview of branch prediction methods and a comparison of simulated performance.


Describes suitability of AMD 29000 to C and UNIX.


Architecture and instruction set reference for the AMD 29000 processor.


Overview of the P5 implementation, in particular with relation to the i486.
Description of the MIPS T5 (also known as R 10000) processor.

November 1994.
Description of the PowerPC 620 microprocessor.

Original Flynn's taxonomy paper.

[50] D.B. Skillicorn. "A taxonomy of computer architectures". IEEE Computer,
21(11), 46, November 1988.
Skillicorn's extension to Flynn's taxonomy.

Dasgupta's extension to Skillicorn's taxonomy.

Intel Corporation, Intel Corporation, Literature Sales, P.O. Box 58130,

Overview of architecture and O.S. concepts. Includes a section on the classification of parallel architectures and their inter-relations.

ACM Doctoral Dissertation Award 1985. Describes the development of the Bulldog compiler and in particular the use of trace scheduling to reorder VLIW code.

[55] E.S. Davidson, G.S. Sohi et al. "Better than one operation per clock:
Vectors, VLIW and superscalar". Computer architecture news, 18(2), 376,
1990.
Annotated Bibliography

Description of XIMD, a variable instruction stream, multiple data stream architecture, which can function as a VLIW or a MIMD.

Comparison of simulated performance of a superscalar CPU and VIPER, a VLIW processor. Indicates superscalar architecture requires only a small amount of look-ahead to match the VLIW performance.

A review of new fast RAM architectures, which compete with 2-level caches and include SDRAM, CDRAM EDRAM and Rambus.

Description of the HP RISC processor. The 7100 is a Harvard-architecture CPU with separate external instruction and data caches.

An overview of the Analog Devices 21060 SHARC processor aimed at DSP applications, which uses separate instruction and data buses to the 512K internal SRAM.


221
Annotated Bibliography

A balanced discussion of early RISC (RISC-I, 801) and CISC (i432, VAX) architectures.

Investigation of SPEC benchmark suite, which indicates that the benchmark's cache hit-ratio is higher than that for applications. In particular, the effects of multi-programming on caches is not well modelled by the suite.


Develops a comparison of cache indexing and tagging schemes. Discusses problems with synonyms and the requirement for cache flushes in virtually-tagged architectures. Suggests that virtually-indexed and physically-tagged cache obtains the best performance.

An article describing the design and development of the Numbersmasher board.


Annotated Bibliography

A description of the Transputer concept and implementation of the T212, T414 and T800.


Development of a modified Modula-2 language for portable parallel programs, which achieves 80% of the performance of hand-written code for MIMD/SIMD/SISD architectures.

The ISO operating systems standard.

Overview of the i860 architecture, including hardware interface details and available software.

An article discussing operating system design and suggesting that their architecture is not well suited to contemporary RISC processors.

Examination of the cost of cache flush and refill during a context switch. Suggests that cache effects amount to a significant part of the context switch and are expected to increase with faster architectures.


Examination of the impact of RISC architectures on operating system design, especially message passing, virtual memory and threads. Mentions the extremely slow context switch for the i860 and discusses problems with virtually-tagged cache.


Old Helios "black" book. Contains some general notes on Helios design as well as reference manual for all commands and libraries.


Description of a novel reordering algorithm, used to minimise number of fill-in elements in the sparse matrix during LU decomposition. Work was done on a parallel transputer system under Helios.


A complete description of the design and implementation of the components of Multics operating system.


Practical experience in using the L3 commercial micro-kernel operating system, which supports advanced features, including persistent tasks and user-space device drivers.

Development and evolution of the MVS, one of two mainframe operating systems used by IBM.

Description of SHARE, a scheduler which ensures uniform resource usage between users and groups instead of tasks.

Brief description of Mach and in particular its scheduler, which uses hints to improve overall performance.
Annotated Bibliography

An introduction to operating system principles and mechanisms, illustrated by short descriptions of UNIX, MVS, OS/2 and others.

A complete guide to operating system issues and concepts, illustrated by the implementation of Minix microkernel.

Presepts a Virtual Round Robin scheduling algorithm, which is based on the concept of shrinking quantum allocation and results in less variation than RR.

Discussion of the sources of priority inversion, including critical sections and communication queues and a novel solution – *Priority Ceiling Protocol* (PCP).

Tutorial of MMU principles and architectures, illustrated by many CISC and RISC examples (including the 1860).

A book introducing many operating system mechanism and concepts.

Description of a commercial UNIX with Real-Time support. Discusses the requirements and challenges in making UNIX a real-time operating system.

A compiler-based method of generating self-scheduling programs for real-time performance. The proposed scheduler is more deterministic and precise than interrupt-based scheduling.

Presents the differences between the two classes of real-time approaches, with the more popular event-triggered systems being more efficient and easier to design, but less predictable.

Short summary of operating system factors in network performance. Suggests that kernel-implemented threads are an order of magnitude faster than processes, but another order of magnitude slower than user-implemented threads.

Describes a taxonomy for thread applications and presents some usage statistics for Cedar environment.

Describes the increasing popularity of multi-threading in efficient, real-time micro-kernels. Illustrated by examples including pSOS and Chorus.

Annotated Bibliography

A general overview of the structure of distributed operating systems, their components and requirements. Suggests separation of local and global mechanisms.

Description of the work done in the porting of UNIX operating system to a local shared-memory parallel computer.

Requirements for and problems with implementing UNIX on a parallel system. Illustrated by the Mach micro-kernel.


Discussion of the challenges to process migration in a homogeneous environment and the available solutions, illustrated by LOCUS, DEMOS/MP, XOS and V systems. See also (105).

A review of migration mechanisms, including both transparent facilities and ones requiring a special interface. See also (104).

An assessment of two years experience with development and usage of the Sprite process migration.
Annotated Bibliography


An object oriented C++ system, which provides applications with a dynamic process migration facility. The system is implemented under Transputer Helios.


Describes the system developed for the RC 4000 computer, which is based around a minimal message-passing nucleus and a hierarchy of processes implementing the system strategies by controlling their descendants.


An investigation into various mechanisms of improving message passing performance, not involving use of virtual memory hardware.


Examines the differences between a monolithic and microkernel implementations of system kernels (ULTRIX and Mach 3.0). Derives experimental results for 13 typical applications, which demonstrate the impact of architecture on the memory system.


Tutorial on fault types and classifications as well as software and hardware methods for detection and recovery.
Introduction to fault-tolerant techniques, including masking, containment, repair and recovery.

Good introduction to fault classification and software and hardware recovery techniques.

Presents ZGL, a heterogeneous operating system.

Describes a combined software-hardware emulation of a CISC (IBM S/390) on a theoretical VLIW processor.

An overview of current problems with effective heterogeneous architectures, including algorithm partitioning, machine selection, scheduling and synchronisation.

Report on the ACM workshop discussing issues and challenges with interconnection, distributed file systems, naming, authentication and user interfaces.

The Apollo DOMAIN introductory manual.

An introduction to TAOS operating system, which supports heterogeneous architectures. The system stores binaries in intermediate form, generating machine code at module load-time.

A modification to the Oberon-2 compiler running under the Oberon system, which generates intermediate representation files. The code generation for the target is performed at load time. Significant space and speed improvements are achieved.

Specification of TenDRA Distribution Format (TDF), which is a concrete implementation of the Architecture Neutral Distribution Format.

A description of the 64-bit MIPS processor. The article suggests that super-pipelined design is cheaper than super-scalar.

Description of the Alpha processor architecture.
Description of Opal, a wide address space operating system.

An assessment of the impact of 64-bit processors on operating system architecture. Discusses a single, shared address space and problems with persistence and naming.

[127] System V Interface Definition. Customer Information Center, P.O.Box 19901, Indianapolis, IN 46219, 1986.
The standard defining the kernel interface for System V variants of UNIX.

A thorough explanation of the System V kernel implementation. Analogous to (129) for System V.

An excellent description of the design and implementation of the BSD kernel, with numerous examples and code fragments. Similar to (128), but for BSD.

The specification of a UNIX kernel interface and application standard developed by the X/Open consortium.

A description of the difficulties in porting an application between different variants of UNIX (in particular System V and BSD).
Annotated Bibliography


An introduction to *Distributed Computing Environment* (DCE), which uses a client-server model to provide higher-level facilities, like file storage, naming, threads, authentication and *Remote Procedure Calls* (RPCs).


Develops a taxonomy of name directory systems, including distributed systems.


Specification of the *Network Filing System* (NFS) protocol.


Describes the use of the idle task to verify invariants in an operating system.
Annotated Bibliography

Kerberos is an authentication system which is robust to a passive tap on the network.

Historical overview of design and experience with the MIT Athena distributed environment, based around networked UNIX workstations.

A reply to (144), citing KeyOS as an example of a capability-based system supporting *Mandatory Access Control* (MAC) by managing the capabilities by the kernel.

An article suggesting that capability-based systems are inappropriate for *Mandatory Access Control* (MAC), since any holder of a privilege may grant it to other users.

The "orange book" which defines the Department of Defence classification of computing system security.

The *Data Encryption Standard* (DES) definition document.

Annotated Bibliography

The history and development of the Data Encryption Standard (DES), including the controversy and future directions.

The original article introducing the Rivest-Shamir-Adleman (RSA) public-key encryption algorithm.

An excellent overview of the history and recent developments in private and public key cryptography.

Description of Amoeba v4.0 distributed operating system.

Description of the distributed programming language developed for the Amoeba system, which supports object sharing by replication/migration across a network of machines.

A description of the philosophy and architecture of the GNU Hurd operating system. Available as URL:.

Description of the Oberon environment, which consists of the operating system, an object oriented language and a tiled windowing user interface.
A description of the design principles and implementation of the Plan 9 microkernel operating system and the 8½ windowing system.

An introduction to the development of the V distributed microkernel. Much of the paper concentrates on V’s IPC design.


Describes the Photon windowing system developed for the QNX real-time microkernel.

Description of the Synthesis kernel, which uses some novel solutions to achieve speed, including dynamic code generation.

Description of the design of the object-oriented C++ heterogeneous distributed environment, based on a virtual processor provided for the objects.

Description of the ideas behind Central part of The Real Time Nucleus (CTRON), which forms the core of the Japanese TRON architecture.
Annotated Bibliography

Description of the Advanced Real-time Technology (ART) distributed object-based system with a predictable real-time scheduler.

Examines the architecture of a distributed kernel designed for multiprocessors (with a processor dedicated to the kernel) which provides complex guaranteed scheduling for tasks and task groups.

Description of Hexagonal Architecture for Real-Time Systems (HARTS), a research platform based on an extended pSOS kernel and a hexagonal network.

Overview of the object-oriented system with guaranteed real-time scheduling, which uses replication to provide some degree of fault-tolerance.

An introduction to Concurrent Hierarchical Adaptable Object System (CHAOs), which consists of a real-time operating system together with a programming environment.

Description of the MAIntainable Real-Time System (MARS), which provides de-
Annotated Bibliography

terministic real-time scheduling and fault-tolerance via redundancy, intended for industrial process control applications.

Examines a small, embedded, real-time system developed for a shared memory multiprocessor and used in aeronautical systems.

An overview of the PEGASUS system, which uses a persistent object architecture and Asynchronous Transfer Mode (ATM) network to provide support for multimedia applications.

The definition of C programming language.

A description of the Intel 432 processor architecture, which is an extreme example of CISC. The i432 provided extensive object-oriented and scheduling facilities in hardware and was designed for multi-processing and fault tolerance.


Comparison of the three synchronisation methods which suggests that while
rendezvous and monitors provide a higher level of abstraction, they are less efficient and require more context switches.

Description of the implementation of a library under UNIX allowing programs to link and unlink object files at run-time.


The new C++ book by the author of the language.

Description of the mechanism used to implement Sun’s shared libraries.

The widely accepted standard for the format of and operations on floating point numbers.

Discussion of the issues in implementation of portable floating point calculations, in particular using (177).


Introduction to literate programming, including a detailed description of the \texttt{noweb} system, based on D. Knuth’s Web.


An introduction to hacker jargon compiled from various on-line sources.

The description of a minimal 16-bit RISC processor, implementing 26 instructions using 12000 transistors and capable of approximately 14 MIPS.

History and status of cleanroom approach, which consists mainly of rigorous specification, separate implementation and statistical testing.

A guide to system programming under the IRIX variant of the UNIX operating system.

[186] Vern Paxson. “\textsc{flex}: The LEX-compatible lexical analyser generator”.
Manual for \texttt{flex}, which is a free \texttt{lex}-compatible lexical analyser generator.

Documentation for \texttt{bison}, which is a free \textit{Yet Another Compiler Compiler (YACC)}-compatible parser generator.

The "dragon book" containing an excellent introduction and reference to compiler techniques.


Book describing the Practical Extraction and Report Language (PERL) ideally suited to processing text files.


A discussion of the complexity of RISC compilers, illustrated by the implementation of the ANSI-C `varargs` facility on a number of processors, including the i860, which requires a particularly complex solution.


241


Outlines the development and history of PVM as well as introducing its key concepts. Also presents some general performance data and outlines current applications.


A brief overview of currently ongoing research into extending Linda.


A comparison between the two distributed systems, which suggests that Linda is more maintainable and faster for large messages, but does not support heterogeneous environments.


Describes a parallel language and compiler using data-parallel paradigm on a MIMD. Step-synchronisation of the workers, results in a virtual SIMD.


Part of the *Linux Documentation Project*.


Description of the dynamically loadable kernel modules under Linux.
An introduction to numerous data plotting techniques, including scatterplot matrix and multiway dot plot.


244


Index of Terms and Concepts

A
absolute paths, 54
address translation, 42
Advanced Real-time Technology, 237
Aegis, 49
AFS see Andrew Filing System
aliasing, 27
Amoeba, 235
    microkernel, 60
AMPP see Assembler Macro PreProcessor
    Helios/860 use, 80
    Helios/860 use, 76
ANDF see Architecture Neutral Distribution Format
Andrew Filing System, 57
Architecture Neutral Distribution Format, 49
ART see Advanced Real-time Technology
Assembler Macro PreProcessor, 70
asymptotical, 141
Asynchronous Transfer Mode, 238
ATM see Asynchronous Transfer Mode
atomic, 65

B
barrier synchronisation, 115
Berkeley Software Distribution, 69
    UNIX version, 51
    block-oriented device, 124
    bottom part, 37
    branch folding, 18
    branch prediction, 18
    broadcast, 113
    BSD see Berkeley Software Distribution
C
capability, 59
Central part of The Real Time Nucleus, 236
CHAOS see Concurrent Hierarchical Adaptable Object System
character-oriented device, 124
Chorus, 60
CISC see Complex Instruction Set Computer
course grain, 10
COFF see Common Object File Format
Common Object File Format, 93
Common Object Request Broker Architecture, 52
Complex Instruction Set Computer, 14
    Introduction, 5
    component, 110
    Concurrent Hierarchical Adaptable Object System, 237
<table>
<thead>
<tr>
<th>Term/Concept</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>containment</td>
<td>48</td>
</tr>
<tr>
<td>contingencies</td>
<td>141</td>
</tr>
<tr>
<td>control dependency</td>
<td>16</td>
</tr>
<tr>
<td>controlling device</td>
<td>128</td>
</tr>
<tr>
<td>CORBA see Common Object Request Broker Architecture</td>
<td></td>
</tr>
<tr>
<td>CTRON see Central part of The Real Time Nucleus</td>
<td></td>
</tr>
<tr>
<td>current directory</td>
<td>54</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>daemon</td>
<td>111</td>
</tr>
<tr>
<td>data dependencies</td>
<td>16</td>
</tr>
<tr>
<td>Data Encryption Standard, 234, 235</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>security standard</td>
<td>59</td>
</tr>
<tr>
<td>data parallel</td>
<td>110</td>
</tr>
<tr>
<td>data-parallel</td>
<td>145</td>
</tr>
<tr>
<td>DCE see Distributed Computing Environment</td>
<td></td>
</tr>
<tr>
<td>Defence Research Agency</td>
<td>49</td>
</tr>
<tr>
<td>delay slots</td>
<td>17</td>
</tr>
<tr>
<td>delayed branching</td>
<td>17</td>
</tr>
<tr>
<td>DES see Data Encryption Standard</td>
<td></td>
</tr>
<tr>
<td>device driver, operating system, 37, 120</td>
<td></td>
</tr>
<tr>
<td>device file</td>
<td>53</td>
</tr>
<tr>
<td>Digital Signal Processor</td>
<td>49</td>
</tr>
<tr>
<td>DIM see Dual Instruction Mode</td>
<td></td>
</tr>
<tr>
<td>Distributed Computing Environment, 233 standard</td>
<td>51</td>
</tr>
<tr>
<td>DRA see Defence Research Agency</td>
<td></td>
</tr>
<tr>
<td>DSP see Digital Signal Processor</td>
<td></td>
</tr>
<tr>
<td>Dual Instruction Mode, 21</td>
<td></td>
</tr>
<tr>
<td>dual operation</td>
<td>21</td>
</tr>
<tr>
<td>dynamic linking</td>
<td>38</td>
</tr>
<tr>
<td>Dynamic stability</td>
<td>142</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>ELF see Executable and Linking Format</td>
<td></td>
</tr>
<tr>
<td>error class</td>
<td>56</td>
</tr>
<tr>
<td>error code</td>
<td>56</td>
</tr>
<tr>
<td>error object</td>
<td>56</td>
</tr>
<tr>
<td>Ethernet</td>
<td>106</td>
</tr>
<tr>
<td>event-triggered</td>
<td>43</td>
</tr>
<tr>
<td>exclusion mechanism</td>
<td>65</td>
</tr>
<tr>
<td>Executable and Linking Format, 93</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td>62</td>
</tr>
<tr>
<td>External Data Representation, 107</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>48</td>
</tr>
<tr>
<td>FDDI see Fiber Distributed Data Interface</td>
<td></td>
</tr>
<tr>
<td>Fiber Distributed Data Interface, 106</td>
<td></td>
</tr>
<tr>
<td>FIFO interface</td>
<td>29</td>
</tr>
<tr>
<td>folding</td>
<td>76</td>
</tr>
<tr>
<td>Free Software Foundation</td>
<td>61</td>
</tr>
<tr>
<td>FSF see Free Software Foundation</td>
<td></td>
</tr>
<tr>
<td>function code</td>
<td>56</td>
</tr>
<tr>
<td>functional parallelism</td>
<td>110</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>general error code</td>
<td>56</td>
</tr>
<tr>
<td>General Server Protocol</td>
<td>56</td>
</tr>
<tr>
<td>Generic Helios Assembler</td>
<td>88</td>
</tr>
<tr>
<td>Generic Helios Linker</td>
<td>88</td>
</tr>
<tr>
<td>Generic Helios Object Format, 68</td>
<td></td>
</tr>
<tr>
<td>GHOF see Generic Helios Object Format</td>
<td></td>
</tr>
<tr>
<td>GSP see General Server Protocol</td>
<td></td>
</tr>
</tbody>
</table>
Index of Terms and Concepts

H
HARTS see Hexagonal Architecture for Real-Time Systems
Harvard,
  definition, 25
  i860, 26
heterogeneous, 49
Heterogeneous Network Computing, 108
Hexagonal Architecture for Real-Time Systems, 237
high level, 39
high-level scheduler, 66

I
imprecise arithmetic exceptions, 75
inode, 125
instance, 110
Inter-Process Communication, 132
intermediate level, 39
Internet domain, 133
Interrupt ReQuest, 129
IO server, 52
IPC see Inter-Process Communication
IRQ see Interrupt ReQuest

K
Kerberos, 58
kernel, 37

L
L3, 38
Liapunov, 140
libns, 121
link adapters, 29
Linux, 119
literate programming, 76
load balancing, 45
  and process migration, 45
loader, 63
logical address, 42
long term, 39
low level, 40

M
MAC see Mandatory Access Control
Mach, 61
MAintainable Real-Time System, 237
major, 125
Mandatory Access Control, 234
MARS see MAintainable Real-Time System
marshalling, 113
masking, 48
Massively Parallel Processor, 134
medium term, 39
memory management unit, 42
memory pages, 41
message identifier, 135
message tags, 112
other, 109
MPI see Message Passing Interface
mid see message identifier
MIMD see Multiple Instruction-stream
  Multiple Data-stream
minor, 125
MMU see memory management unit
monolithic, 46
MPI see Message Passing Interface
Index of Terms and Concepts

MPP see Massively Parallel Processor
msgtag see message tags
multicast, 113
Multics, 51
Multiple Instruction-stream Multiple Data-stream, 20
multiprocessing, 39
multiqueue round-robin, 65
mw860, 121

N
National Grid Company, 141, 143
Network Filing System, 233
UNIX, 57
NFS see Network Filing System
NGC see National Grid Company
nucleus, 63

O
OASIS see On-line Algorithms for System Instability Studies,
performance, 154
transient and dynamic stability, 141
Oberon, 61
Object Management Group, 52
Objects, 56
OMG see Object Management Group
On-line Algorithms for System Instability Studies, 140
Introduction, 10
Open Software Foundation,
ANDF, 49
organisation, 51
option flags, 170
Orange Book, 59
OS860, 120
OSF see Open Software Foundation

P
p4, 107
page mode, 25
paging, 42
PARallel MACroS, 108
parallel paradigm, 110
PARMACS see PARallel MACroS
Patch, 68
PCP see Priority Ceiling Protocol
PERL see Practical Extraction and Report Language
physical address, 42
pipeline stall, 15
Plan 9, 61
PLD see Programmable Logic Device
Practical Extraction and Report Language, 241
preemptive, 40
priority, 41
Priority Ceiling Protocol, 226
priority inversion, 41
Process migration, 45
Programmable Logic Device, 167
Project Athena, 58
protocol class, 56
pvmd3, 111
pvmlib3.a, 111
query, 40

Q
Index of Terms and Concepts

R
race, 65
real device, 128
real time, 43
Real Time Power System Simulator, 144
Reduced Instruction Set Computer,
  Introduction, 5
  i860, 13
relative paths, 54
Remote Procedure Call, 233
request code, 56
RISC see Reduced Instruction Set Com-
puter
Rivest-Shamir-Adleman, 235
  security standard, 59
root directory, 54
RPC see Remote Procedure Call
RR see Round Robin
RSA see Rivest-Shamir-Adleman
RTPSS see Real Time Power System Sim-
ulator
run860, 120

S
scatterplot matix, 129
scheduling, 39
scheduling algorithm, 40
scoreboarding, 16
server, 63
servers, 46
severity, 144
severity index, 144
SHARC see Super Harvard ARCHitec-
ture
shared libraries, 38
short term, 40
SIMD see Single Instruction-stream Mul-
tiple Data-stream
Single Instruction-stream Multiple Data-
stream, 19
socket, 121
SOM see System Object Model
SONET see Synchronous Optical Network
speculative execution, 18
SPMD see Single Program Multiple Data
Stream, 57
subsystem,
  GSP error code, 56
  GSP request code, 56
Super Harvard ARCHitecture, 25
superscalar, 21
supervisor mode, 36
swapping, 42
Synchronous Optical Network, 106
system, 63
System Object Model, 52

T
TAOS, 231
task, 44
task identifier, 112
task server, 63
TCP see Transmission Control Protocol
TDF see TenDRA Distribution Format
TenDRA Distribution Format, 231
thread, 44
tid see task identifier
time slice, 40
Index of Terms and Concepts

time-triggered, 43

Z

TLB see Translation Look-aside Buffer

top part, 37

Transient stability, 142

Translation Look-aside Buffer, 23

Transmission Control Protocol, 122

U

UDP see User Datastream Protocol

unicast, 113

UNIX domain, 133

User Datastream Protocol, 122

user mode, 36

utilities, 63

V

V system, 61

Very Long Instruction Word, 21

virtual machine, 46

VLIW see Very Long Instruction Word

von Neumann,

definition, 25

i860, 26

W

Web, 76

Windows NT, 61

write-back, 26

X

XDR see External Data Representation

Y

YACC see Yet Another Compiler Compiler

Yet Another Compiler Compiler, 240