International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 2, pages 160–187, July 2012

Virtualized Development and Testing for Embedded Cluster Computing

Ian Vince McLoughlin
School of Computer Engineering
Nanyang Technological University
Block N4, Nanyang Avenue, Singapore 639798

Timo Rolf Bretschneider
EADS Innovation Works
110 Seletar Aerospace View
Singapore 797562

Chen Zheming[1]
School of Computer Engineering
Nanyang Technological University
Block N4, Nanyang Avenue, Singapore 639798

Received: May 2, 2012
Revised: June 6, 2012
Accepted: June 29, 2012
Communicated by Koji Nakano

Abstract

Embedded cluster processing solutions can be difficult to design if quantity and type of processors, interconnectivity technology and code partitioning are not specified in advance, and yet it may not be possible to determine these without qualitative and quantitative testing which can not be performed until hardware is available. Similar issues are faced in conventional development projects for generic embedded systems, or for generic distributed systems, but the issues are exacerbated when both attributes are combined. This paper considers issues relating to custom embedded cluster processing system development in general, before focussing on a specific example of a space-borne cluster computer using ARM processors running embedded Linux. This early example, which will be shown performing distributed synthetic aperture radar image processing, is representative of a growing number of systems in the pervasive and ambient computing fields. It illustrates many difficulties in co-developing hardware and software for specific tasks. We present one potential solution – the use of distributed virtualization to simulate the final design. An ARM cluster simulator is presented, constructed using QEMU and virtual networking, and shown used for the development and validation of distributed embedded processing tasks on a cluster-type architecture. As clusters of embedded processors, pervasive embedded networks, and the "embedded cloud" become more popular in the near future, such virtualized systems can prove useful for development and testing.

Keywords: Embedded cluster; virtualization; hardware-software codesign; distributed embedded computing

[1]Chen Zheming is now with Renesas Electronics, Singapore

1 Introduction

Embedded cluster computers extend and combine the two advancing research areas of distributed processing and embedded systems. As embedded systems become more pervasive and more powerful and distributed processing increases in usefulness and popularity, it was inevitable that the domains would eventually meet. *Pervasive computing*, *ambient intelligence* and *ubiquitous computing* describe post-desktop computing paradigms that often utilise wireless communications infrastructure to supply distributed computational resource wherever and whenever it is required. Although the computer infrastructure in such systems need not be either embedded or low power, the practical requirements of *computing everywhere* tend to mandate that low power embedded solutions will be key in this maturing technology field.

A second feature of these post-desktop computing schemes is that they are naturally asynchronous. This is first due to their adoption of high level operating systems and languages (e.g. Android or Java), but also through the use of asynchronous, packetised wireless links.

This paper focuses on the use of virtualization technology for development, testing and modelling of such systems. Although it is feasible that virtualization technology will itself be part of the run-time of ambient intelligence solutions (such as in the Oxygen project [18]), the processing and latency overheads are currently significant detractors. This paper considers virtualization at *design-time* rather than run-time. We demonstrate the need for a software-only development solution by discussing a real world example; we propose a solution based upon the popular QEMU dynamic translation emulator [2], present the operation of this tool, and evaluate its ability to assist in the software design and testing phases of an embedded cluster processing solution [12].

Section 2 presents the PPU (parallel processing unit) target architecture, while Section 3 considers the distributed processing architecture and protocols for this system. Section 4 discusses virtualized emulations for modelling the example system, presenting a system solution which is evaluated in general in Section 5. Section 6 applies this to a real-world distributed ARM example for the processing of polarimetric synthetic aperture radar (POLSAR) data, before Section 7 presents final results and Section 8 concludes the paper.

2 The Parallel Processing Unit (PPU)

The PPU [13] was designed as a high speed but low power/cost/weight dedicated image processing unit for Singapore's X-Sat satellite, launched on 20 April 2011 [17]. It was designed to handle the processing of high rate multispectral imagery using low power ARM processors arranged in a cluster-like fault-tolerant configuration. For reasons of fault tolerance, the PPU employs loose indirect coupling between asynchronous processing nodes with each processor running embedded Linux. From a software perspective, this closely resembles a Beowulf cluster [20].

One of the main aims of the PPU is reliability. Although the cluster's processing nodes (PNs), which are standard ARM CPUs, have no radiation hardening, the concept of reliability-through-redundancy, and the application of single-point-failure mitigation techniques, allow the original system to match the reliability figures of a fully radiation-hardened design operating in low-earth orbit [13]. It means that any node can fail at any time, but the overall system must continue to operate (potentially with some delay). It is thus a fault-tolerant and asynchronous system (i.e. an embedded Beowulf cluster [10]).

Since the initial design of the PPU (seen in Fig. 1, and described in more detail below in Subsection 2.1), this architecture has been extended and applied to the ground-based processing of earth observation data, including multispectral images as well as synthetic aperture radar (SAR) and POLSAR data. Other embedded cluster architectures have also been designed [4], and the rising popularity of low-power embedded processors such as the ARM is likely to result in many more systems of this type, both in space and on the ground.

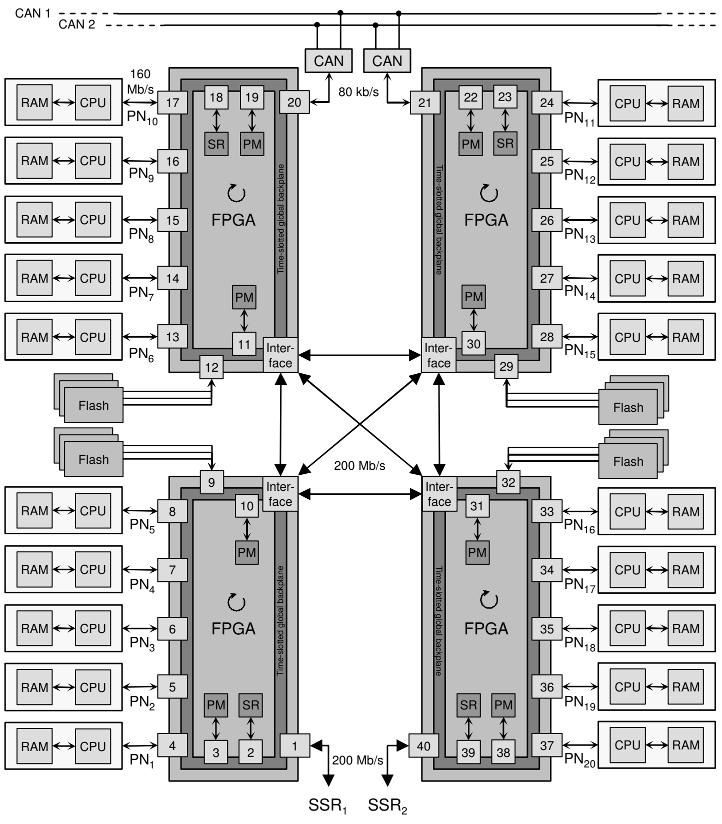


Figure 1: A block diagram of the PPU architecture showing twenty processing nodes (PNs) connected across four FPGA blocks. Inside are status registers (SR) and processing modules (PM), while a controller area network (CAN) bus conveys control messages, while fast low voltage differential signalling (LVDS) links interconnect FPGAs and link the PPU to a solid state recorder (SSR) RAM storage module.

2.1 Design Rationale

When a system such as the PPU is included in a satellite, it contributes to cost primarily in terms of lift-off mass, but also design cost, component cost, as well as having electrical power and volume requirements. Thus, it is important for it to provide sufficient value in terms of raw processing power, but also in flexibility and reliability. For X-Sat, this was the motivation for using a general-purpose operating system (Linux) running on an ARM processor. This combination supports a variety of software payloads, including image compression [25] or in-orbit disaster monitoring software [26].

However, this architecture (originally designed by the first author in 1999), has also been used for various ground-based applications. In fact it represents an early example of an emerging class of computing systems: embedded clusters. Embedded clusters can be either a single centralised unit or distributed across a cloud, in car-area network or in-home network. Systems of this type are finding use in an increasing number of niche applications [4]. High reliability or fault tolerance is often cited as being important reasons for their adoption.

2.2 Distributed Structure

The PPU's interconnection backbone consists of four one-time-programmable interconnected FPGAs (anti-fuse Actel AX1000). These enable real-time data streaming capability through a time-slotted serial network topology interconnecting 20 processing nodes (PN). In the PPU, each PN consists of one Intel SA1110 (206 MHz) processor and 64 MB of Samsung SDRAM (Samsung K42561632D) [11].

If we assume the FPGAs in Fig. 1 do nothing more than provide an interconnection, then the PPU resembles a Beowulf cluster [10] (although a packet processing service is implemented in the PMs for simple unsupervised radiometric correction in the space-borne FPGA design, we will not consider it further here). The generic PPU is therefore a cluster[2] as we can see more clearly in Fig. 2.

Every PN connects to the hosting FPGA using a dedicated 17-bit parallel bus (the 17 bits are for efficiency [13]). Dedicated power switching circuitry isolates each PN and FPGA from power faults occurring in other PNs or FPGAs. Direct memory access (DMA) is enabled over this link to connect into the central routing network, and both data and command messages share the network [13]. A photograph of the flight-ready PPU hardware as deployed on X-Sat is given in Fig. 3

Within Fig. 1, two methods of interconnection to the outside world are shown, each with a redundant backup. One is through CAN (controller area network), and the other through a high speed low voltage differential signalling (LVDS) link to a solid state RAM (SSR) block. These interfaces nominally relate to control messages and data streams respectively, but both map to packetised message transfers along the timeslotted global backplane (TGB) – a Token ring-like networking protocol developed for this system. Redundant flash memory provides for OS, ramdisc and program storage, accessed at boot time. Status registers (SR) maintain system state – a local copy of the knowledge maintained by the external PPU controller.

After reset, PNs are booted from program images that reside in redundant flash memory, controlled by the FPGAs. These flash images contain the Linux OS, plus common applications and the processing code library. PNs can voluntarily reboot, or be power-cycled by the FPGAs in which case, new code is loaded on demand. As we will see, in the virtualized solution, individual instances act identically to this, and can even be virtually 'power-cycled' or rebooted by the control process.

2.3 Operating System and Monitoring

Each PN runs embedded Linux with real-time extensions. Using Linux on this type of system is trivial since it is already commonly used this way on the ground in Beowulf networks. Extensive support is available within the OS to handle higher-level protocols, including message passing and networking stacks.

[2]Note that earlier publications describing the PPU [13] presented a two-FPGA alternative structure. Whether two, four or more FPGAs are used, this is still a cluster since FPGAs are largely only operating as network switches.

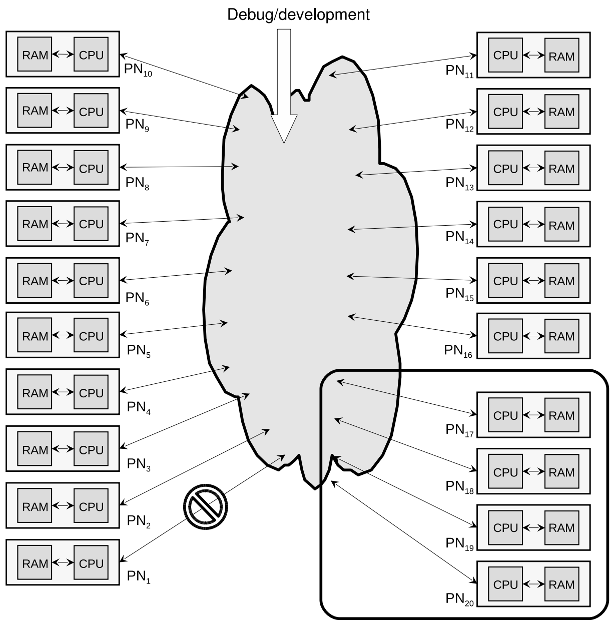


Figure 2: PPU-style architectural drawing comprising multiple PNs connected to a network cloud. Two failure modes are shown apart from node failure; the island effect where several nodes operate separately to the main network (PN17 to PN20), and an erroneous communication link (PN1).

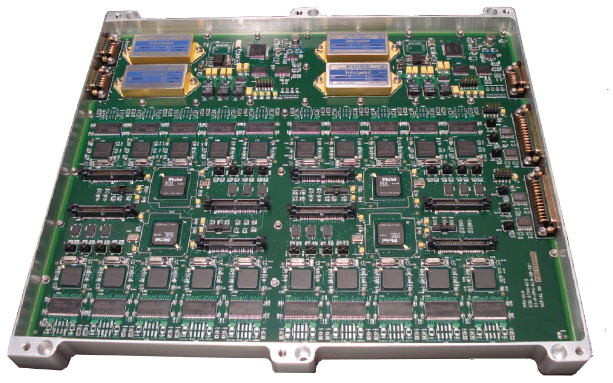


Figure 3: A photograph of the PPU and power supply (along the back edge) installed in a mounting tray for use in the X-Sat microsatellite. A row of 10 PNs is clearly visible along the front edge of the circuit board.

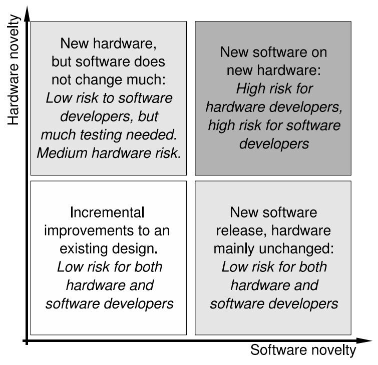

Hardware novelty
New hardware, but software does not change much: Low risk to software developers, but much testing needed. Medium hardware risk.
New software on new hardware: High risk for hardware developers, high risk for software developers
Incremental improvements to an existing design. Low risk for both hardware and software developers
New software release, hardware mainly unchanged: Low risk for both hardware and software developers
Software novelty

Figure 4: The graph of development risk with hardware and software novelty divided into four quadrants. Note that the consequence of 'risk' to hardware is costly and time consuming hardware re-spins, whereas the consequence to software is usually confined to delayed software release.

PN software is divided into four modules: identical boot code runs on each PN as soon as the corresponding FPGA commands it to turn on. Above this, drivers handle communication with the FPGA. These respond to control events from the FPGA (such as heartbeat signals), and manage buffers and queues used by the third level of software, which defines the parallel processing topology. Finally, the fourth level is application code, selected from a library of possible run-time executables.

As a parallel processing machine, this is therefore a loosely asynchronous arrangement similar to a cloud or a Beowulf cluster. This both improves reliability by maintaining greater independence between units, and significantly simplifies code development and testing.

Also, the loosely asynchronous PPU is unlike a tightly coupled hard real-time signal processing computer, and this implies that there is no shared memory maintained at run-time. Whilst this can be disadvantageous in terms of gross computational speed, it has advantages in terms of fault tolerance (for example, at the simplest extreme, processing jobs allocated to failed nodes can simply be restarted without affecting the remainder of the system), and in maintaining memory coherence. It also has great advantages in allowing the use of standard development tools (as noted above), and enabling the extensive use of simulation built upon virtualization technology – the primary focus of this paper.

2.4 System Development

We assume the existence of a cooperative multi-person project to develop a high speed fault tolerant signal processing solution. This could result in a single standalone unit like the PPU (which contains a cluster of processors), or it could result in a distributed solution of embedded processors. While many development paradigms are possible for such systems, eventually they all partition the design into hardware, firmware and software requirements. The latter includes an operating system (if used), libraries, high level languages, scripting and so on. The hardware design may be reasonably fixed when the software is being developed (i.e. the number, type and connectivity of processing elements is likely to be known before software development begins in earnest). However, it is unlikely that fully working prototype hardware is ready for the software development team to use. Generally speaking, the hardware would only be available if it is being reused, or if the software development effort has deliberately been delayed until the hardware is fixed. In a space mission where fully working hardware may only be available immediately prior to launch, the time reserved for the software development process to port code on to the final hardware may be unduly limited.

Usually a software development team requires access to hardware for prototyping and developing their code. However, often the team is provided be provided with hardware that is deficient in some aspect (see below). We can understand this by classifying a project in terms of novelty in both hardware and software as in Fig. 4. Designs where both the hardware and software are novel would encompass a higher risk than designs in which either software or hardware remains fairly constant.

As mentioned, software developers usually require access to hardware for prototyping and development purposes, especially where the hardware and the software are bespoke. If both are being developed simultaneously, then definitely no hardware can be available for the software developers until very near the end of the overall development process. Any hardware that is provided to the software team will be deficient in one or more aspects such as the following:

- CPU type/family/device, CPU speed, memory map changes.
- System-on-chip differences in capabilities or structure/addresses.
- Connectivity differs in type or packet structure.
- Different operating system [version], libraries, APIs.
- Different toolchain and development environment.

However, if common OS code and communication processes are adopted, then industry standard authoring and prototyping tools can be employed for software development. Most deficiencies are then confined to the upper few items on the list, and useful tools such as `lint`, `gprof`, `electric fence`, `strip`, `objdump`, `gdb` etc. are used to improve software reliability and accelerate development time.

Returning to the example of embedded clusters such as the PPU, and the large and intricate algorithms that these can support: Firstly, the embedded nature of the systems, and secondly their inherent distributed and loosely coupled asynchronous natures, conspire to make prototyping difficult. We can consider them to be unfriendly and rare custom hardware clusters in development terms. In fact, several of the development options for the PPU are shown in Fig. 5. Assuming that the software developers do not have access to the PPU system itself (which is likely when it is still under development, is too expensive or rare, or extremely difficult to use), then another system must be used by developers for writing, testing and validating code. In the case of the PPU, apart from all reasons mentioned, the individual PNs did not have separate debug interfaces or serial ports. Only a single serial port and a single JTAG debug interface were provided, multiplexed via the FPGAs. Firmware within the FPGA allowed the ports to be remapped to individual PNs as required. There was thus never any possibility of two PN processors being debugged simultaneously, or for their console (serial port) status messages to be accessible simultaneously. These issues seriously reduced the development friendliness of the system.

In order to improve the friendliness toward developers, the PPU was deliberately constructed to be Beowulf-like in nature. In fact it was one of the first embedded Beowulf clusters, and certainly the first to use an ARM processor. Adopting Linux opened up the possibility of having standard development tools, and allowing some prototyping to occur on a Linux PC. However, for more realistic testing, including issues of hardware dependencies, data types, memory access and so on, either a separate ARM board (i.e. similar CPU but completely different architecture), or a virtualized ARM processor solution (could be same CPU, same architecture) would suffice. In fact, precisely this aspect is explored further in Section 4.

3 Distributed Computing

Scalability and reliability are essential characteristics of a distributed computing system. So in the case of the PPU, a general purpose task allocation and messaging system was designed with these attributes in mind. The physical transport is called the timeslotted global backplane [13], a bit-serial parity-checked ring architecture allowing node-to-node or broadcast communications. Although this is a custom architecture designed by one of the authors, to the higher layers it provides a packet transfer service that mimics UDP.

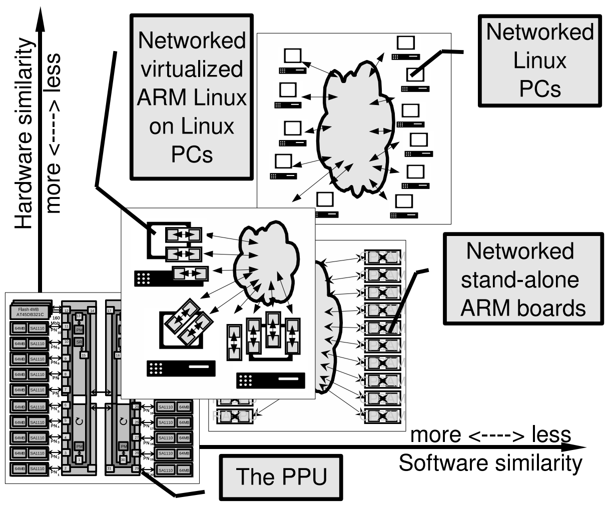


Figure 5: Several options for software development for PPU without access to final PPU hardware, in terms of hardware and software similarity.

3.1 Asynchronous Event Management

Messages within the PPU are essentially asynchronous. Therefore, computing nodes have to maintain independent processing schedules, but be sensitive to responding to messages in a timely fashion. Within Linux programs, signals can be used to convey synchronisation messages, while data transfers to the higher layers are considered to be by UDP. The actual content of messages is application-specific, as illustrated for the POLSAR edge detection application in Section 6.

4 Virtualization

The PPU built for Singapore's X-Sat low-earth orbit microsatellite, is a mission-specific and delicate unit. Only mission specialists have access to flight model hardware, or the engineering model, and very few fully working prototypes exist. Even when one of these is available, it is not easy to use: having multiple flash memory devices, several need to be reprogrammed manually to enable new code (they use triple redundancy with a binary majority voter, so at least two of the three connected to each FPGA need to be reprogrammed manually each time), and sometimes either messaging changes or boot code alterations will require reprogramming the four host FPGAs (which again is done manually, and sequentially). In summary, this platform is fragile, extremely expensive and rare. It is also difficult and time-consuming to use for prototyping: it was built originally for space, definitely not for the convenience of software developers.

As already discussed in Section 2.4, thought was needed to allow developers to write and test software for the system without requiring physical access. In fact, this was one of the main motivations behind the adoption of Linux as the PPU OS, and the adoption of a Beowulf-like architecture. Of course, the PPU is not a cluster of x86 machines, but rather a cluster of ARM processors. It would be possible to construct a cluster using commercial off the shelf (COTS) ARM hardware, but such a network would be costly, relatively large, and would still require a method to reprogram each of the 20 separate nodes individually. Thus a simulation approach was considered.

4.1 Emulation or Simulation

Cycle-accurate ARM emulators are available, such as in the famous ARM Realview suite, but these are extremely slow - simply running the ARM-Linux operating system without any processing code, could take days of execution time. However, note that when simulating or emulating a Beowulf-style cluster computer, being cycle-accurate is not necessary. Similarly, the entire PPU operates asynchronously. Thus the use of accelerated emulation or even simulation systems, especially those which can support Linux, becomes possible.

At present, several ARM architecture emulators/simulators exist, including QEMU [2], SkyEye [19], ARMware [21] and so on, with a good overview available from [5]. To simulate the system on inexpensive and convenient PC hardware requires emulation or virtualization of the multiple ARM processors, plus the interconnection network.

Several researchers have also noted the disparity in speeds between cycle-accurate emulators and simulators or emulators such as those mentioned. This has led them to perfect methods of estimating or calculating cycle-accurate timings, generally from QEMU. Hsu et al. [6] developed and evaluated several methods of extracting virtual timing information by various extensions to the QEMU source code. Nakamoto and others have gone further by using simulated processors for timing-critical control applications [14][15] for vehicle subsystems, with some success.

Yeh and Chiang [24] used QEMU, implemented in System-C, for cycle-accurate timing of ARM processors, expanding on the interface to the QEMU code [23]. Meanwhile Chylek had been collecting program statistics through monitoring QEMU [3]. At the same time, the first author of this paper had modified QEMU source code to handle GPIO (general purpose I/O) ports on ARM system-on-chip processors, something of particular importance where GPIO pins are used for simple messaging between processors in multi-processor systems.

Considering the large amount of research work in this area based around QEMU, its relatively fast speed of simulation, and the flexibility of its open source implementation, QEMU-ARM is a good choice for a cluster computer emulation target.

4.2 QEMU

QEMU is an open source emulator that uses the technique of dynamic code translation to achieve high emulation speeds [2]. It currently supports many processor architectures, including ARM, x86, PowerPC and so on. An ARM instance running on a Linux host can be emulated using a simple single-line command specifying the kernel and initrd (initial ramdisc) images[3]. It is simple to use, and has been found by many authors to work reliably and accurately. QEMU also can simulate standard peripherals including network interface cards, and use virtual local area networks (VLANs) to connect separate instances of QEMU virtually. With this, the simulation of large networks or clusters becomes possible.

The standard way to connect QEMU to a real network is via the network tap interface (TAP), although several alternative methods exist. QEMU adds a virtual network device to the host computer, which can then be configured as if it were a real Ethernet card. A non-privileged (user mode) network stack can also be used to replace the TAP device, and form a basic network connection. This ability of QEMU to choose either user-mode or kernel networking options matches the two possibilities for a cluster computer, irrespective of the physical medium over which the messages actually pass.

4.3 Other Emulators/Simulators

SkyEye, like QEMU, is open source software, providing an integrated simulation environment for both Linux and Windows hosts. The SkyEye environment simulates/emulates typical embedded computer systems, including a series of ARM architecture based microprocessors and Blackfin DSPs; RAM, ROM and Flash memory; peripherals such as timer, UART, NIC chip, LCD, touch screen; and supports operating systems including ucLinux, ARM Linux, Nucleus, Rtems, eCos etc. [19]. Although SkyEye is reportedly a fast simulator, it lacks the networking flexibility of QEMU, and is less well documented.

ARMware is another ARM emulator, supporting most ARM architecture features including all types of memory management. It uses a built-in dynamic re-compiler to translate ARM machine

[3]A typical processor instantiation command would be;
```
qemu-system-arm -kernel zImage.integrator -initrd MTinitrd.img -append "console=ttyAMA0" -m 128
-icount 0
```

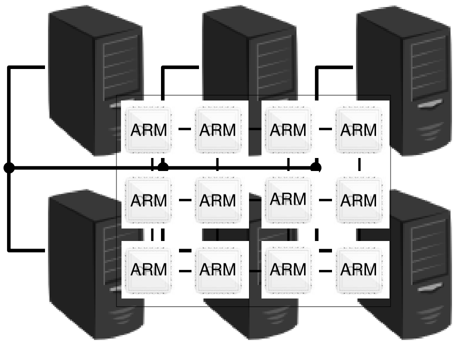


Figure 6: Several virtualized networked ARM processors emulated on several networked PC servers. Both the virtualized machines and the hosts run optimised versions of the Linux operating system.

code to x86 machine code. The whole emulated environment which ARMware provides is based upon the HP iPaq H3600. Besides emulating the CPU core, it provides various peripheral devices, including flash ROMs and touch screen. ARMware does not currently support ARM Thumb code (which would not be required here anyway), but also lacks network support [21] which is, of course, essential for cluster computation.

4.4 Virtualization Arrangement

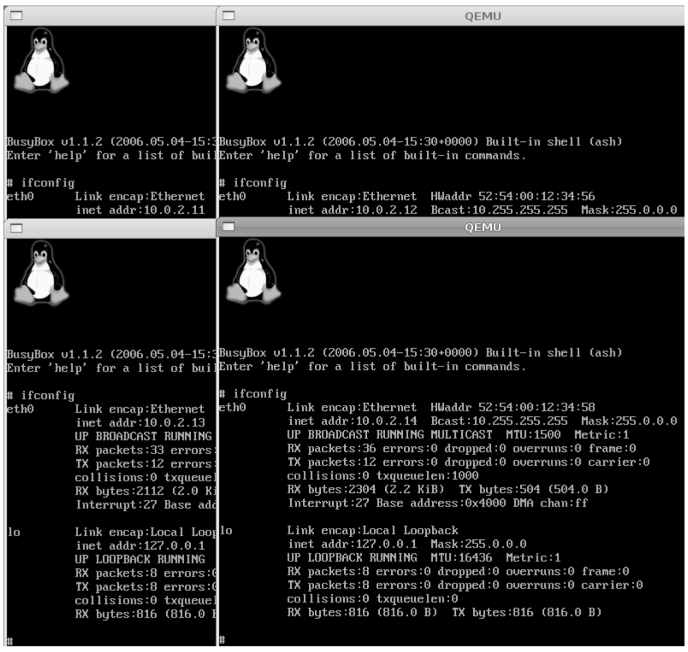
The proposed virtual distributed computing platform to model the PPU is composed of one or more PCs connected by fast Ethernet, hosting many virtual ARMs with virtual LANs interconnecting them (see Fig. 6). One host PC task operates as the server, which is responsible for assigning computing tasks to PNs, providing corresponding input data to its clients, collecting output data from all the clients, tracking progress, and generating the final processing result. In the evaluations of this system, it is the host PC task that measures the overall system timing figures. The virtual ARMs are considered to be clients, which apply for tasks and fetch input data from the server, complete the assigned task and submit the output data back to the server. The number of virtual ARMs is adjustable, both at configuration-time, and at run-time due to the nature of the communication and distribution system used. Thus new nodes can join, and existing nodes can drop out at any time.

Since connection is through IEEE802.3 network (which may or may not traverse an Ethernet link), virtual ARMs can run across multiple host PCs if necessary, although in practice several 200MHz ARM processors can be operated at near real speed on a single fast PC. Fig. 7 shows a screen capture from four ARM virtal machines (VMs), each a separate instance of QEMU, running on a single x86 computer. In this figure, each ARM has booted up into a standard Busybox shell and is waiting for input. In the experimental system, the client software would normally be automatically launched immediately after boot, and we may not run in 'graphical' mode.

The use of virtual networking means that each VM (i.e. each QEMU ARM instance) simply needs to join the virtual network switch, irrespective of whether they are resident on a single host, or across multiple hosts. A server program, running on an x86 PC, also joins the same network (but is omitted during QEMU benchmarking tests), to coordinate and control the software running on each VM. In this system, the server also provides an NFS (network file system) shared directory to distribute shared data to the QEMU ARM instances. In PPU terminology, each QEMU VM maps to a single ARM PN.

4.5 The Server/Controller

In the simulated PPU (based upon the situation in the real one), the server maintains a circular linked task list indexing the sub-tasks that need to be processed. Sub-tasks are allocated one after another from the top to the bottom of the list, and then back to the top again. Once a sub-task has



```
BusyBox v1.1.2 (2006.05.04-15:3
Enter 'help' for a list of buil

# ifconfig
eth0      Link encap:Ethernet
          inet addr:10.0.2.11
```

```
BusyBox v1.1.2 (2006.05.04-15:30+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.12  Bcast:10.255.255.255  Mask:255.0.0.0
```

```
BusyBox v1.1.2 (2006.05.04-15:3
Enter 'help' for a list of buil

# ifconfig
eth0      Link encap:Ethernet
          inet addr:10.0.2.13
          UP BROADCAST RUNNING
          RX packets:33 errors:
          TX packets:12 errors:
          collisions:0 txqueuel
          RX bytes:2112 (2.0 Ki
          Interrupt:27 Base add

lo        Link encap:Local Loop
          inet addr:127.0.0.1
          UP LOOPBACK RUNNING
          RX packets:8 errors:0
          TX packets:8 errors:0
          collisions:0 txqueuel
          RX bytes:816 (816.0 B

#
```

```
BusyBox v1.1.2 (2006.05.04-15:30+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:58
          inet addr:10.0.2.14  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:36 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2304 (2.2 KiB)  TX bytes:504 (504.0 B)
          Interrupt:27 Base address:0x4000 DMA chan:ff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:816 (816.0 B)  TX bytes:816 (816.0 B)

#
```

Figure 7: Four virtualized ARM processors executing embedded ARM Linux on a single host PC.

completed, the corresponding index is deleted from the list. Clients are able to apply to the server for tasks whenever they become idle.

During the entire processing period, the server is only responsible for handling two kinds of received message: task application and request for processing of result submission, and sends four kinds of messages: *task assign*, *submission permission*, *submission rejection*, and *shut down computing node*. Apart from these messages, the server is also responsible for handling errors and events such as node failure. Due to the flexible and asynchronous nature of the cluster, client and server software, nodes can potentially join and exit the work group at any time – with different degrees of consequence on overall processing time. For the particular test cases and results presented in this paper, no errors or node failure were modelled.

5 Evaluation

The system has been evaluated in a number of ways. Functionally, it is identical to the emulated ARM processors, it can run the same version of Linux, and in fact uses exactly the same kernel and initial ramdisc files. The same compiler, libraries, and executables can be run on both the real and the virtualized emulated system without alteration.

Two methods of performance evaluation are presented here. Firstly, pure processor execution speed, and secondly networking and interconnection speed.

5.1 Processor Performance

The Byte magazine standardised benchmarking suite [22] was used to determine the performance of the virtualized processors for computation of a number of real tasks which exercise the CPU in different ways. This test suite, called n-bench, is a well thought-out attempt at measuring processor performance. Tasks are dynamic, scale well with processor speed, are repetitive and assessed for statistical relevance as part of the run-time testing. It is also cross-platform and relatively easy to port to other architectures.

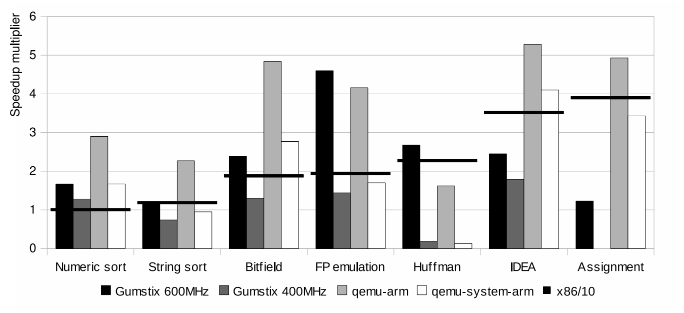


Figure 8: Byte magazine n-bench scores for several ARM-based, QEMU emulated ARM and x86 systems (the x86 results have been divided by 10 to display within the same scale, and plotted as horizontal bars), as multiples of K6/233 performance.

Release 2 of n-bench in Linux source code form, used here (nbench_byte version 2.2.3)[4], compares tested systems to a benchmark of an AMD K6/233 MHz machine with 512 MHz of SDRAM and 32 MB of L2 pipeline-burst cache. Results from such comparison obviously depend upon more than pure hardware: the operating system, compiler type, version and settings are factors, as are memory and bus speeds. However, for the comparisons presented within this section, these items are invariant.

n-bench figures were obtained for the default QEMU emulated ARM system, an integrator/CP board with the following virtual hardware:

- ARM926E processor
- PL011 UART (x2)
- SMC 91c111 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse
- PL181 multimedia card interface with SD card
- 128 MB SDRAM

Each ARM instance was running Linux 2.6.17 (compiled with GCC version 4.1.0). A custom initial ramdisc was set up to launch the n-bench test suite five seconds after boot, with results automatically collected and collated.

Fig. 8 shows scores for seven of the ten available n-bench test metrics, for a number of machines. The graph also compares results from a desktop class x86 PC (an Intel Core 2 Quad Q9550, clocked at 2.83 GHz, running Linux 2.6.27, with score divided by 10 to fit on the same scale), from 400 MHz and 600 MHz ARM-powered systems (Intel PXA255 and PXA270 respectively), from a standalone QEMU-emulated PXA255 processor (qemu-arm), and an ARM926 system with graphics and peripheral emulation (qemu-system-arm). The QEMU emulations were run alone and individually on the Core 2 machine mentioned above. Fourier and Neural Network results (not plotted in the graph) were very low; the highest results were for qemu-arm at 0.05 and 0.08 respectively. For many of the conditions in this section, the Fourier/NN results were invariant for all systems at around 0.01 or 0.00. This is because the ARM devices used and emulated do not have hardware floating point capabilities. We therefore restricted the performance comparison to fixed point operations only. LU decomposition for qemu-arm was 0.16, and lower for other tests.

At this point, it should be noted that QEMU provides an *icount* parameter that is an instruction cycle timer (but for various reasons is not entirely cycle-accurate). It allows the emulated machine

[4]Source code from http://www.tux.org/~mayer/linux/bmark.html

Table 1: n-bench performance figures for a single instance of QEMU virtualized ARM with various *icount* parameter settings, related to K6/233 performance.

Test	**-icount 2**	**auto**	**-icount 1**	**-icount 0**
Numeric sort	1.10	1.37	2.20	4.39
String sort	0.50	0.74	1.00	2.01
Bitfield	0.99	1.94	1.99	3.98
FP emulation	1.02	1.74	2.04	4.08
Assignment	1.23	2.72	2.45	4.91
IDEA	1.35	3.03	2.70	5.40
Huffman	0.40	0.12	0.79	1.58
LU decomposition	0.02	0.01	0.04	0.09
Memory index	0.85	1.58	1.70	3.40
Integer index	0.88	1.02	1.76	3.51
Floating point	0.28	0.18	0.35	0.44

instruction cycle to be set to a given number of nanoseconds: in the results presented so far, and in the networking tests below, *icount* was unspecified, leading QEMU to automatically estimate the correct cycle timing of the original processor. However, it is possible to override this by specifying a time value. The effect of overriding *icount* is explored in Table 1, where the n-bench results for a single invocation of QEMU-ARM are presented for automatic timing, plus values of 2, 1, and 0 (relating to target instruction cycle timings of 4, 2 and 1 ns). The tests were run on a Xeon X3220 2.4 GHz quad-core 64-bit machine with 4 GB of RAM and a 1.033 GHz bus, running Linux 2.6.32. Clearly, the *icount* setting significantly effects the speed of the emulated processor: Integer index results vary from 0.88 to 3.51 times the benchmark AMD K6/233 machine, reaching the approximate real-world speed of a 1.2 GHz ARMv7 architecture processor.

The speed-up indicated in Table 1 provides the very nice possibility that the virtualized system may exceed the speed of the real-world system that is being emulated. However due to the interaction between real-time (including real-time clock ticks and network latency) and actual cycle timing, it is preferable to maintain approximately realistic performance if possible. Hence the remaining tests use the automatic *icount* setting. Note that QEMU also supports debugging of loaded images, single-step operation and dumping of trace information, which could also assist in determining real-time characteristics.

Since it is clear that a desktop x86 class machine can easily emulate a single ARM processor at greater than real-time speed, for the purpose of the system explored in this paper, the question now becomes how many ARMs can a single PC emulate comfortably. Again, this obviously depends upon the power and specific capabilities of the PC hosting the emulations, and the system being emulated, as well as operating system, compiler speed, other tasks and so on. However, although the specific figures will vary from machine to machine, it is useful to explore the relationship between the two.

Thus Table 2 presents n-bench scores for between two and twelve simultaneous virtualized machines on the Intel Core 2 machine mentioned. The results show relatively flat performance up to four, and then a gradual decline beyond that. Therefore, we can see that the Core 2 host is capable of supporting four virtual ARM integrator systems without appreciably affecting performance. It should be remembered that the host was a quad-core machine with four hardware threads. The sum of multipliers (i.e. multiples of the base AMD K6/233 machine) is plotted as a stacked graph in Fig. 9. The Xeon machine (also advertised as a quad-core machine with four hardware threads), could support 16 simultaneous QEMU virtualized ARM systems at the same performance given in Table 1.

It is also interesting to see that free physical memory on the host, shown at the bottom of Table 2, reduced gracefully through the tests (which reserved 128 MB of RAM for each emulated ARM processor). Clearly, physical memory is not a limitation in this case, and virtual memory swapping is similarly unlikely to be a significant factor in test performance.

Table 2: n-bench performance figures for 1 to 12 simultaneous QEMU virtualized ARMs, related to K6/233 performances.

Test	**1**	**2**	**3**	**4**	**6**	**8**	**12**
Numeric sort	1.70	1.64	1.65	1.64	1.38	1.14	0.98
String sort	0.96	0.95	0.95	0.94	0.78	0.67	0.57
Bitfield	2.78	2.78	2.77	2.73	2.37	2.00	1.73
FP emulation	1.70	1.68	1.63	1.64	1.35	1.15	1.05
Assignment	3.44	3.43	3.35	3.35	2.70	2.48	2.15
IDEA	4.17	4.16	4.13	4.10	3.39	3.14	2.60
Huffman	0.13	0.13	0.13	0.12	0.12	0.11	0.08
LU decomposition	0.01	0.01	0.01	0.01	0.01	0.01	0.00
Memory index	2.09	2.07	2.07	2.05	1.72	1.49	1.28
Integer index	1.10	1.10	1.10	1.08	0.93	0.81	0.68
Floating point	0.18	0.18	0.18	0.18	0.18	0.17	0.15
Free mem, MiB	>495	495	455	436	360	312	162

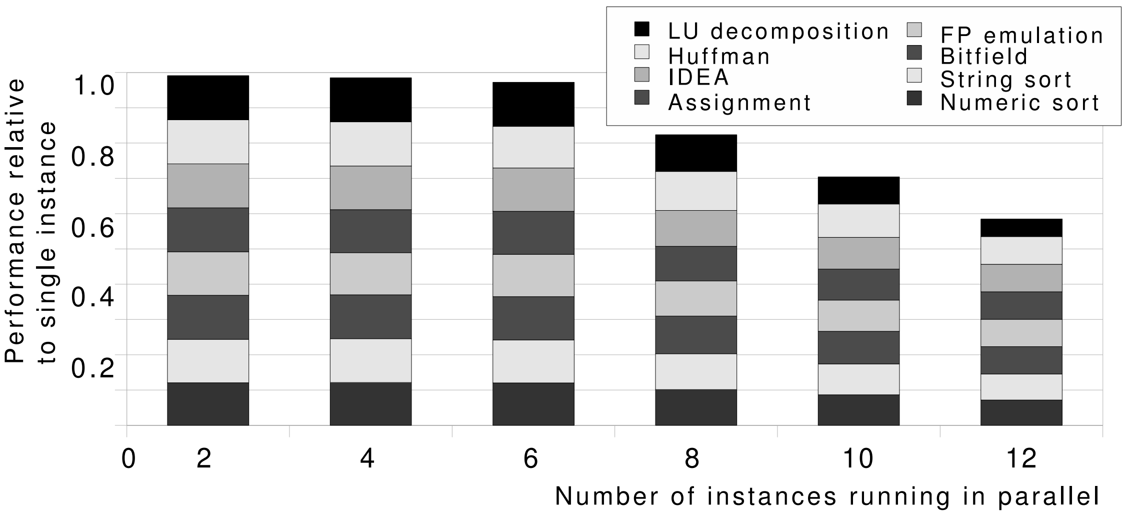


Figure 9: Overall speedup multipliers reduce monotonically beyond four instances.

Table 3: iperf UDP throughput and jitter figures for three arrangements of virtualized client-server network links.

		Server		**Client**	
Number	Pair	Mb/s	Jitter, μs	Mb/s	Jitter, μs
One pair	1	1.06	13	1.06	12
	1	1.05	11	1.05	10
	1	1.05	16	1.05	11
	1	1.05	11	1.05	11
one pair (of 2)	1	1.05	16	1.05	15
one pair (of 2)	1	1.05	11	1.05	11
Two pairs	1	1.05	12	1.05	11
	1	1.05	17	1.05	13
	1	1.05	13	1.05	12
	2	1.05	22	1.05	22
	2	1.05	51	1.05	50
	2	1.05	17	1.05	13
Three pairs	1	1.05	20	1.05	20
	1	1.05	17	1.05	17
	2	1.05	8	1.05	8
	2	1.05	12	1.05	12
	3	1.06	25	1.06	25
	3	1.06	27	1.06	27

A further point to note is that while virtualized performance decreased relatively smoothly for more than four emulated ARMs, the real world time taken to perform the tests obviously scaled significantly with the number of emulated systems – meaning that the elapsed time required to test twelve virtualized processors was approximately twice as long as it was to emulate six. A 32-way test was extremely time consuming. However the important fact is that the speed reduction – at least up to 12 way – is linear with the number of emulated ARMs.

Finally, the n-bench test results require a disclaimer; they are meant to expose the theoretical upper limit of the CPU, FPU, and memory architecture of a test system, but are presented as single-threaded tests. Many modern operating systems are capable of multitasking, and may have hardware support for multithreading, including some of the more recent ARM processors. QEMU itself can very simply emulate up to 255-way symmetrical multi-processing through a single command line argument, not explored further here.

5.2 Network Performance

Virtualized network performance was determined through the iperf2.05[5] suite. This measures either UDP or TCP performance in terms of throughput and jitter (UDP only).

For the tests presented in this section, QEMU ARM processors were connected by virtual Ethernet[6]. All tests were conducted on the Xeon machine mentioned earlier. Since the tests measured point-to-point networking capabilities, emulated systems were invoked with SMC91C111 network devices having different MAC addresses, operating on separate virtual networks (i.e. none of the virtualized machines shared a subnet with the host). First a single pair of ARM systems were emulated and tests run between them. Next, another pair of ARMs were invoked but remained passive while a further set of tests were run. Next, both pairs ran tests simultaneously, and finally three pairs (i.e. a virtualized network of six separate ARM machines) ran tests simultaneously.

[5]Source code from http://sourceforge.net/projects/iperf

[6]Using the following command switches;
`-net socket,connect=localhost:1234` and
`-net socket,listen=localhost:1234`

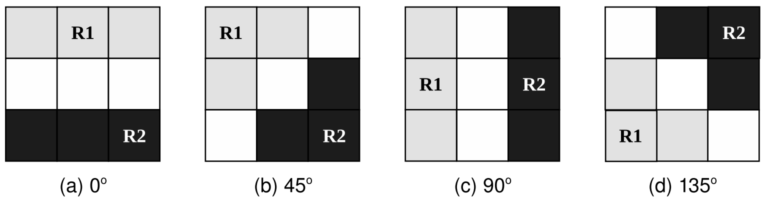


Figure 10: 3×3 edge templates of different orientations.

Results shown in Table 3 indicate that, while actual throughput was fairly constant, jitter – which was well constrained for a single pair, increased substantially as more devices were virtualized, between certain pairs of devices. TCP performance (not shown in the table) yielded consistent throughputs of around 2.5 Mb/s for a single pair. In the other cases, always one pair of ARM devices communicated faster than the other pairs, reaching around 2.2 Mb/s, while the other pairs dropped as low as 1.3 Mb/s. This variability in results when more than a single pair are emulated is similar to the jitter situation shown in Table 3 for UDP between two and three pairs. Since more than two pairs of ARM devices emulated per x86 CPU exhibit performance degradation, seen in the n-bench scores, it is likely that emulating four ARM devices per modern x86 server is optimal: they can compute at real-time speeds and communicate with constant throughput and relatively constrained jitter (no more than 51μs in the tests above). Tests between machines on different subnets, each hosting one or more virtualized ARM processors, exhibited greater network delays, jitter, and reduced throughput, all of which were heavily dependent upon the characteristics of the network infrastructure tested (and much less so than upon the QEMU performance).

6 Polarimetric SAR Data Processing Example

The example given in this paper to demonstrate the practical usefulness of the distributed QEMU-ARM emulation approach is an implementation of the Roy's largest eigenvalue based edge detector [8]. The purpose of the algorithm is to detect edges in order to produce a filtered image while preserving the image features. The polarimetric synthetic aperture radar (POLSAR) image data handled by this system represents each pixel with a 3×3 Hermitian matrix, containing nine complex floating point elements, stored band interleaved by pixel (bip).

This algorithm and data set is a good choice for illustrating embedded cluster processing for a number of reasons, including (i) it matches the original satellite-based processing domain of the PPU (i.e. for remote sensing satellite or unmanned aerial vehicle processing), (ii) POLSAR image sizes tend to be extremely large in terms of number of pixels and naturally lead to a tiling approach for processing, (iii) data size is even larger given that each pixel represents nine complex floating point numbers so that an entire image may not easily be handled by a single processor system, (iv) the edge detection task often needs to be performed in near-real-time, (v) the task involves asynchronous communications between neighbouring nodes.

The edge magnitude output module employs Roy's largest eigenvalue based detector to compute an edge magnitude from the POLSAR data using four basic edge templates of 3×3 pixels in which two different regions R1 and R2 are denoted by different colours/shades, as shown in Fig. 10 [7]. The output from this initial edge-template matching module may feed a subsequent speckle noise filter [9].

6.1 Roy's Largest Eigenvalue Edge Detector

Let p_{ij} denote a pixel in the image and e_{ijk} denote a single element of the 3×3 Hermitian matrix that is this pixel. i and j are the row and column of the pixel respectively, and k indicates the band of this element. The steps used to detect edges in multi-look POLSAR data are as follows:

1. Overlay all edge templates over pixel under test, and its adjacent eight pixels.

2. Calculate Z_{r1} and Z_{r2} for each edge template as follows:

$$Z_{r1} = \frac{1}{3}\begin{bmatrix} \sum e_{ij1} & \sum e_{ij2} & \sum e_{ij3} \\ \sum e_{ij4} & \sum e_{ij5} & \sum e_{ij6} \\ \sum e_{ij7} & \sum e_{ij8} & \sum e_{ij9} \end{bmatrix}, \; for\ each\ [i,j]\ pair\ that\ makes\ p_{ij} \in R1 \tag{1}$$

   Z_{r2} is defined likewise for each $[i, j]$ pair that makes $p_{ij} \in R2$.

3. Determine the Roy's largest eigenvalue for all edge templates using Eqn. 1, Note that $ch1(X)$ denotes the maximum eigenvalue of matrix X.

$$\lambda_{Roy} = max\left\{ch_1(Z_{r1}Z_{r2}^{-1}), ch_1(Z_{r2}Z_{r1}^{-1})\right\} \tag{2}$$

4. Record the maximum of all the computed Roy's largest eigenvalues from the previous step.

5. Repeat steps 1–4 by moving the edge templates to the next (adjacent) pixel. Continue testing pixels one-by-one until the end of the image.

Following this process, a threshold can be estimated from the maximum Roy's largest eigenvalue outputs that have been found, in order to determine a pixel as being either "edge" or "non-edge" [9].

6.2 Initial Segmentation by Watershed Transform

If the input edge magnitude image is considered a topographic surface, then the value of each pixel becomes its altitude. When flooded, this 'landscape' becomes immersed up to a particular depth. In order to do this, the watershed transform consists of two steps: sorting and flooding. Firstly, pixels are sorted according to their edge magnitude value. Flooding is then carried out based on an immersion simulation, following two rules:

1. Random access to any pixels in the image

2. Direct access to the neighbours of a given pixel.

Water begins rising from the level of the minimum magnitude pixels (local minima), and these are initially labelled as 'flooded'. Flooding then continues for each magnitude step increase k until the maximum is reached. At any time, each k-magnitude pixel, which happens to be adjacent to an already-labelled watershed region, is added into that region. A new region is constructed for any k-magnitude pixel which is not connected to existing regions.

In order to avoid over-segmentation from the watershed transform, edge magnitudes are first thresholded according to:

$$I(x,y) = \left\{ \begin{array}{cc} g(x,y), & if\ g(x,y) > T \\ T, & otherwise \end{array} \right\} \tag{3}$$

where $g(x, y)$ denotes the edge magnitude value of a pixel located at coordinate (x, y), and T is a user-defined threshold. This thresholding replaces all lower gradient values by the uniform threshold T and small regions can then be eliminated by removing insignificant local minima.

6.3 Region Merging

Over-segmentation may still occur, despite the thresholding above, and therefore a modified HSWO [1] (hierarchical stepwise optimization) algorithm is adopted to adjust the initial region mapping as described below:

1. The segmented output from the watershed transform is used as an initial image partition.

2. For each adjacent pair of regions i and j, a stepwise criterion is computed as in:

$$SC = \left\{ \frac{1}{N_i} + \frac{1}{N_j} \right\} \times \frac{1}{max\left\{ ch_1(C_i C_j^{-1}), ch_1(C_j C_i^{-1}) \right\}} \tag{4}$$

3. Perform the global best merging. To achieve this, spatially adjacent region pairs are found and merged, having the maximum stepwise criterion value over the entire image as per [9].

4. The process is terminated if no further merging is needed; otherwise, it iterates back to step (2). Termination depends on having reached a pre-set desired number of regions.

6.4 Software Structure

For a single CPU implementation, an entire image can be loaded to the local RAM of the processor, and each step proceeds in turn. However, as mentioned, POLSAR images can be extremely large in size with each pixel containing a nine-element matrix of complex floating point values. Thus any processing of such images tends to require a large amount of computational resource. Even on powerful servers equipped with large amounts of memory, it is often desirable to subdivide into a number of strips or tiles for individual processing. In fact, real-time processing of large POLSAR images is currently only possible using a multi-core or many-core implementation, or an FPGA-based processing system [16].

6.5 Distributed Processing

Assuming that an image has been divided into equal-sized sub-images to be processed using different processing cores or nodes, these nodes cannot process their own sub-images in isolation – they probably need to exchange information with neighbouring nodes.

Although the amount of incursion of regions into neighbouring images is limited only by the boundaries of the original super-image in theory, in practice a sufficient number of distinct regions is allowed such that only minimal incursion to neighbours is considered in many cases. Fig. 11 illustrates the extent of neighbouring region incursion that is typically allowed.

For ordinary sub-images, which are not located on any edge of the whole radar image, Fig. 12 identifies and enumerates the actual edge regions that any node may need to share with a neighbour. As can be seen, there are eight adjacent sub-images, indexed as [0...8]. These indices will be referred to later when describing message passing in the system.

6.6 Task Allocation

In the benchmark implementation and experiments described in this paper, multiple 128×128 pixel sub-images are processed individually from an original 512×512 pixel POLSAR image (as was shown in Fig. 11 which comprises nine-look NASA/JPL POLSAR C-and L-band images covering the region of Muda Merbok, West Malaysia). The 128×128 pixel sub-images are sized to fit easily within the local memory of individual nodes, and to complete processing in a timely fashion, and can in principle be any size, trading off processing time for communications message overhead (described later).

For the reference implementation, then, the entire processing task is to divide the original image area into 16 identically-sized sub-tasks indexed as $[i, j]$ $(i, j = 0, 1, 2, 3)$. The indices of all sub-tasks to be processed are maintained by a centralized controller (a control application running on the host PC for the virtualized simulation, or on-board flight computer for the PPU), communicating through network messages. The control application is responsible to allocate sub-tasks to client nodes, and update internal sub-task lists according to the feedback from the nodes. The controller must also take corrective action to re-launch tasks in the case of node failure (identified through loss of heartbeat response).

If there are more client nodes than sub-tasks to be processed, some of the clients can process identical sub-tasks to enhance the reliability of this platform in the face of radiation-induced single

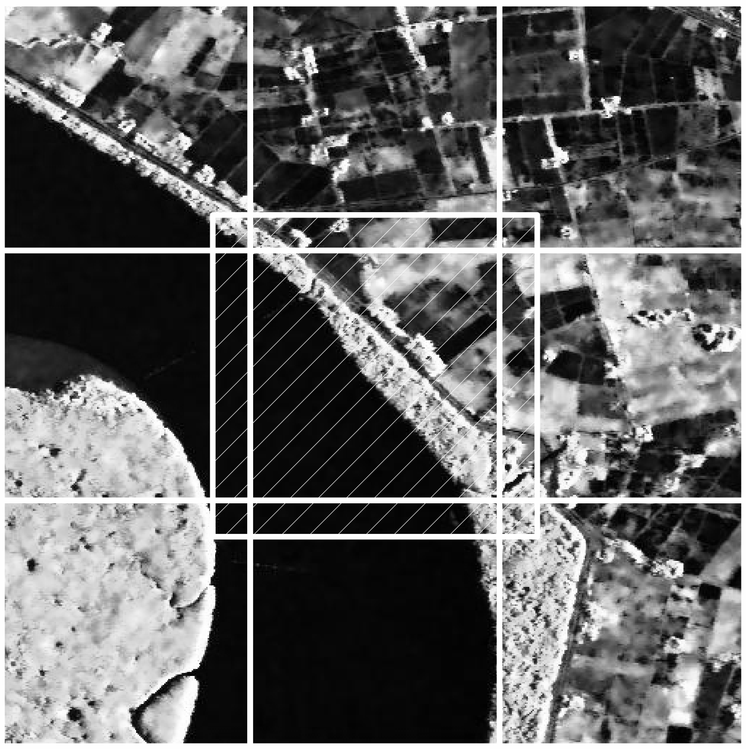


Figure 11: A POLSAR image shown divided into nine regions with the central region augmented with edge information that may be needed from neighbouring blocks for the processing of that particular central sub-image. The underlying POLSAR image is from NASA/JPL, covering the region of Muda Merbok, West Malaysia.

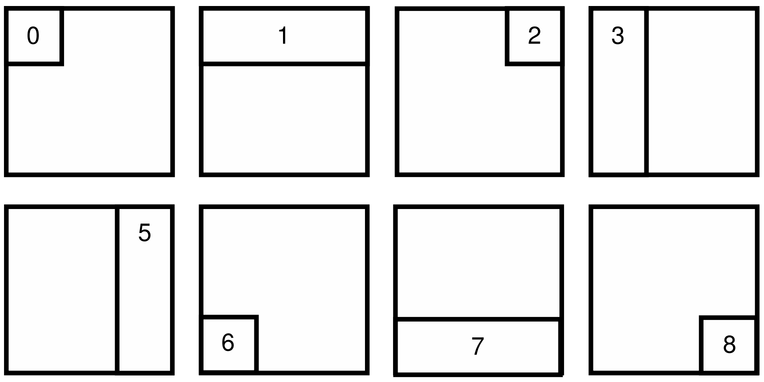


Figure 12: The eight possible edge segments of a sub-image.

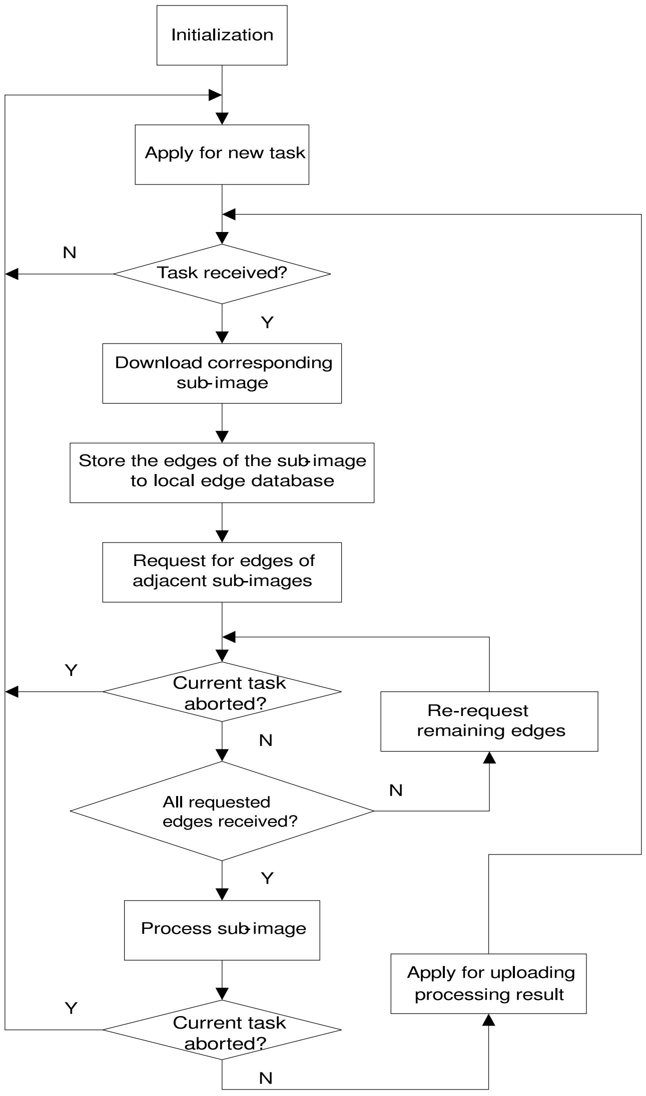

Initialization
Apply for new task
Task received?
N
Y
Download corresponding sub-image
Store the edges of the sub-image to local edge database
Request for edges of adjacent sub-images
Current task aborted?
Y
N
Re-request remaining edges
All requested edges received?
N
Y
Process sub-image
Apply for uploading processing result
Current task aborted?
Y
N

Figure 13: Flowchart for the process of image partitioning.

event bit errors. If there are fewer client nodes, then after the initial round of allocations, nodes may be issued with subsequent sub-tasks, until all outstanding sub-tasks have been allocated. Since nodes run a multitasking OS (Linux), sub-tasks can run time-shared on single nodes.

In this way, the control and partitioning algorithm is easily able to scale to larger original images, and manage different numbers of client nodes, subject to PN memory constraints. It should be noted that the main time-critical constraint to be managed is for the controller to ensure that, for each sub-image currently being processed, edge information can be made available from the neighbouring sub-images (i.e. that at least one PN is processing those sub-images).

The client workflow, shown in Fig. 13, indicates that nodes are responsible for requesting work from the controller, and for requesting edge information from neighbouring nodes. The computing nodes provide edge pixels to each other, rather than downloading from the central server in order to reduce the overall communication burden, and reduce the likelihood of communications bottlenecks with the server from causing processing delays.

Table 4: Multi-byte task-task and task-server message datagrams and meanings.

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Bytes 6,7,8...**
From client to server for task application (APP)						
‘A’	‘P’	‘P’				
From the server to a client for allocating sub-tasks (TSK)						
‘T’	‘S’	‘K’	Sub-task ID			
Broadcast from a client for requesting edge information (REQ)						
‘R’	‘E’	‘Q’	Sub-image ID		Edge ID	
From one client to another for providing edge information (EDG)						
‘E’	‘D’	‘G’	Sub-image ID		Edge ID	information
From a client to the server to apply for uploading processing result (UPL)						
‘U’	‘P’	‘L’	Sub-task ID			
From the server to a client for accepting submission (CPY)						
‘C’	‘P’	‘Y’				
From the server to a client for rejecting submission (ERR)						
‘E’	‘R’	‘R’				
Broadcast from a client to stop other clients processing the same sub-image and applying for a new sub-task from the server (STP)						
‘S’	‘T’	‘P’	Sub-task ID			
Broadcast from the server to shut down all the computing nodes (END)						
‘E’	‘N’	‘D’				

6.7 Node Communications

When processing a sub-image, each computing node stores its segmented edges, and broadcasts requests to the cluster with the index pairs of sub-image ID and edge segment ID for needed edge information of those adjacent sub-images. When a node receives an edge information request, it checks all the edges it has stored, and replies with matching edge information, if any. Either a negative or a positive reply must be made.

In case some adjacent sub-images have not yet been processed by any computing nodes in the cluster and the edge information is therefore currently not available, the requesting computing node must then continually re-broadcast its request until it has received all of the needed edge information, and then start to perform the image segmentation algorithm. Although such an arrangement could potentially lead to a deadlock situation, and thus require mitigative control structures, such situations did not occur during our tests.

After finishing processing a sub-task, the corresponding edge information stored in the node should not be simply deleted, it needs to be maintained until notified by the central server. Otherwise, such information is lost forever and all the adjacent sub-tasks carried out later may never be able to complete (this is important in the event that a neighbouring node fails after receiving edge information, but before completing its processing, in which case, another node will be allocated as a replacement – and that new node will need to request the same edge information a second time).

Table 4 identifies the set of message datagrams that have been defined to handle the intercommunications between tasks and between task and server to allocate and control tasks, track progress, exchange information, and terminate processing.

6.8 Asynchronous Event Management

Apart from a few items of request/acknowledge information (such as new task allocation), all messages are asynchronous. Therefore, computing nodes should maintain independent processing schedules, but be sensitive to responding to messages. In the current implementation, Linux signals are used to convey messages and the distributed QEMU virtualization naturally supports this.

While some message from other computing nodes, which are irrelevant to the processing of the current sub-image, such as edge information request, can be simply handled by a SIGIO handler alone, others need attention from the main processing routine. The signal handler implemented here thus filters out irrelevant messages according to the current stage of the main processing routine. Consequently, cooperation between the main routine and the signal handler becomes an important issue, requiring clear task partitioning within a node, and information exchange between processing routines and SIGIO signal handlers.

When configured to perform task processing, a computing node needs to go through all the following steps:

1. Apply for a sub-task.
2. Download a sub-image.
3. Store the edge information of the sub-image.
4. Request & receive edge information of adjacent sub-images from other nodes.
5. Provide edge information to other nodes processing adjacent sub-images.
6. Process the sub-image.
7. Apply for submitting the processing result.
8. Submit the processing result.
9. Exit

It can be seen that some steps can be directly managed by the signal handler routine, but some need the involvement of the processing routine(s). In fact, the following steps are handled entirely within the signal handler:

1. Provide edge information to other nodes,
2. Receive edge information from another node,
3. Submit the processing result,
4. Exit.

The remaining steps are handled by the main processing routine(s), and these can be categorised into three distinct stages determined by the current processing action. To proceed from one stage to the next will require certain messages from other nodes:

- **Stage 1**: Apply for a sub-task.
- **Stage 2**:
  1. Download a sub-image
  2. Store the edge information of the sub-image
  3. Request edge information of adjacent sub-images from other nodes
- **Stage 3**:
  1. Process the sub-image;
  2. Apply for submitting the processing result.
  3. Go back to Stage 1

The information needed to proceed from Stage 1 to Stage 2 is to receive a new sub-task id, and from Stage 2 to Stage 3 is to receive all of the necessary edge information from nodes processing adjacent sub-images.

To allow each step to be completed, the main processing routine(s) and signal handler need information from each other. Moreover, to enhance reliability, the signal handler filters out unexpected messages according to the current stage of the main routine using the rules specified in Table 5.

Table 5: Rules for filtering messages.

Message	Stage 1	Stage 2	Stage 3
REQ	A	A	A
EDG	F	A	F
TSK	A	F	F
CPY	F	F	A
ERR	F	F	A
STP	F	A	A
END	A	A	A

A means ‘accept’, F means ‘filter out’

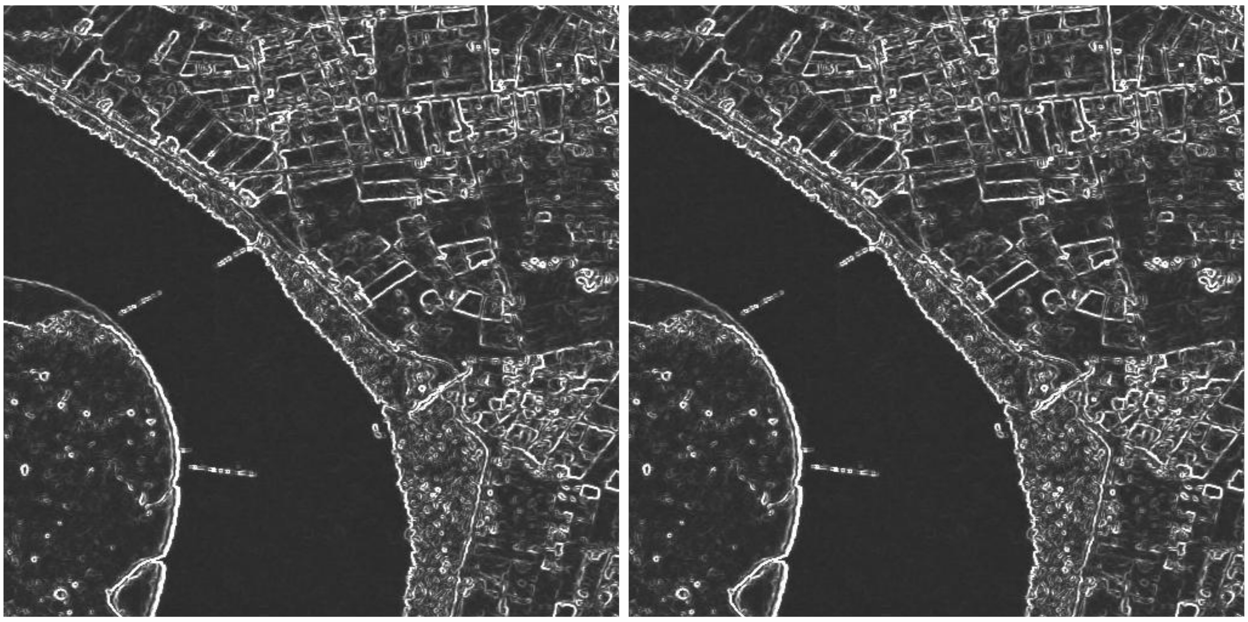


Figure 14: Comparison of reference implementation (left) and distributed QEMU-emulated ARM coded output (right).

7 Results

The virtualized platform was configured to run the Roy’s largest eigenvalue based image segmentation software, operating on real POLSAR images, using 16 virtualized QEMU-ARM processors. The output from one run of the system, is shown in Fig. 14 alongside a reference implementation [9], using a single non-distributed x86 program. As can be seen, there is little or no discernible difference in output, although slight numerical differences exist due to differences in floating point accuracy between the two systems.

To estimate the computational performance of this virtual distributed computing platform, the execution duration in different scales of processing different size of images was tracked. In the test scenario, all the computing nodes boot up and launch their client programs before the host PC launches the server program, and no node failure or message loss is experienced.

One sub-task is pre-allocated for each computing node to prevent initial network congestion. Once a computing node receives a start command from the host PC, it initiates its processing; when that sub-task is done, it notifies the host PC again and stops the client program. The host PC broadcasts a start command across the virtual LAN at the beginning of the test, and terminates the server program once all the computing nodes have finished their individual sub-tasks. Therefore, some overhead time is eliminated: the measured client program execution duration is primarily the time required for a virtual ARM instance to process one sub-image by its own clock. The server program execution duration measures the time needed to process the whole image for the platform by the clock of the host PC, which also indicates the real world time. In fact, ARM processing time for each node, PC server time, and actual CPU time on all computers, were all recorded.

To characterise the performance of this virtualized platform, there are several factors to consider:

1. Image size: P, which is the number of pixels in the whole image. The time spent on processing each pixel is likely to be roughly similar.

2. Number of CPU cores, C, where we limit a C core x86 CPU to running at most C QEMU instances concurrently (although in practice, we have seen in Section 5 that one x86 CPU can comfortably emulate several ARMs). The host CPUs are Intel Pentium IV 2.80GHz machines running Ubuntu Linux. Since these are single core CPUs, we set $C = 1$.

3. Number of virtual ARM instances, N. On the one hand, each ARM has a fixed overhead for communications, control and operating system time which is independent of the size of the image being processed. On the other hand, more instances of virtual ARMs lead to a heavier network communication burden. In this platform, each computing node processes a 128×128 sub-image. Thus the time spent on fetching edge pixels for each ARM can be regarded as roughly equivalent.

4. Processing time, T, in seconds, which can be expressed in the form of Eqn. 5, in which $a \times P$ represents the time cost for processing all the pixels, $b \times N$ represents the time cost for edge pixels fetching and pre-ARM overhead, c represents the platform overhead and d represents the overhead within each ARM. The coefficients a, b, c and d are to be determined empirically according to the test results.

$$T(P,N) = \left\{ \begin{array}{ll} a \times P + b \times N + c, & for\ N > 1\ and\ P = N \times 128^2 \\ a \times P + d, & for\ N = 1 \end{array} \right\} \tag{5}$$

For scaling purposes, the first experiment was to determine the processing time required of a single ARM operating on a single tile of POLSAR imagery. The results are shown in Table 6, where it can be seen that around 640 s is required to process a 128×128 pixel image.

Further experiments were then constructed, in which each ARM node processes sub-images of size 128×128 pixels, and between 1 and 8 ARM processors are allocated for the overall processing. In this way, when two ARM processors are employed, the overall processed image size is 128×256 pixels; four ARMs process a 256×256 pixel image, and eight ARMs process a 512×256 pixel image. Therefore, the amount of data processed increases linearly with the number of ARM processor instances being run.

When analysing the experimental results, the empirical formula of Eqn.5, yields values of $a = 0.0377$, and $d = 22.98$ (from the test run on a single ARM processor), $b = 65.065$ and $c = 78.64$ (from the multi-processor experiments). Thus the platform performance can be characterised as:

$$T(P,N) = \left\{ \begin{array}{ll} 0.0377P + 65.065N + 78.64, & for\ N > 1\ and\ P = N \times 128^2 \\ 0.0377P + 22.98, & for\ N = 1 \end{array} \right\} \tag{6}$$

The 128×128 pixel experimental results are plotted, in Fig. 15. The 'linear' plot indicates the processing time that would be expected if actual measured time for a single node (including all real measured overheads) is extrapolated upward for more nodes, while the 'model' plot shows the results from the empirically derived timing model of Eqn. 6, noting that P changes for each experiment.

8 Conclusion

This paper has discussed the issues involved in the development of embedded cluster computers which resemble a Beowulf structure, especially those involving architectures other than x86-class machines.

Table 6: Processing duration for a single ARM to perform edge detection on POLSAR images of 64×64 and 128×128 pixels, for three test runs (timed within the QEMU ARM simulation, and timed on the host PC respectively).

Test No.		64×64		128×128	
		ARM	**PC**	**ARM**	**PC**
1	real	2m 58.03s	2m 57.168s	10m 42.00s	10m 40.483s
	user	0m 7.53s	0m 0.000s	0m 29.71s	0m 0.000s
	sys	2m 49.62s	0m 0.000s	10m 10.63s	0m 0.004s
2	real	2m 58.02s	2m 57.593s	10m 42.50s	10m 42.475s
	user	0m 7.24s	0m 0.000s	0m 27.90s	0m 0.000s
	sys	2m 50.11s	0m 0.000s	10m 14.05s	0m 0.000s
3	real	2m 58.00s	2m 57.670s	10m 41.11s	10m 39.966s
	user	0m 8.01s	0m 0.000s	0m 28.33s	0m 0.000s
	sys	2m 49.41s	0m 0.004s	10m 11.62s	0m 0.000s
average of real		-	2m 57.477s	-	10m 40.975s

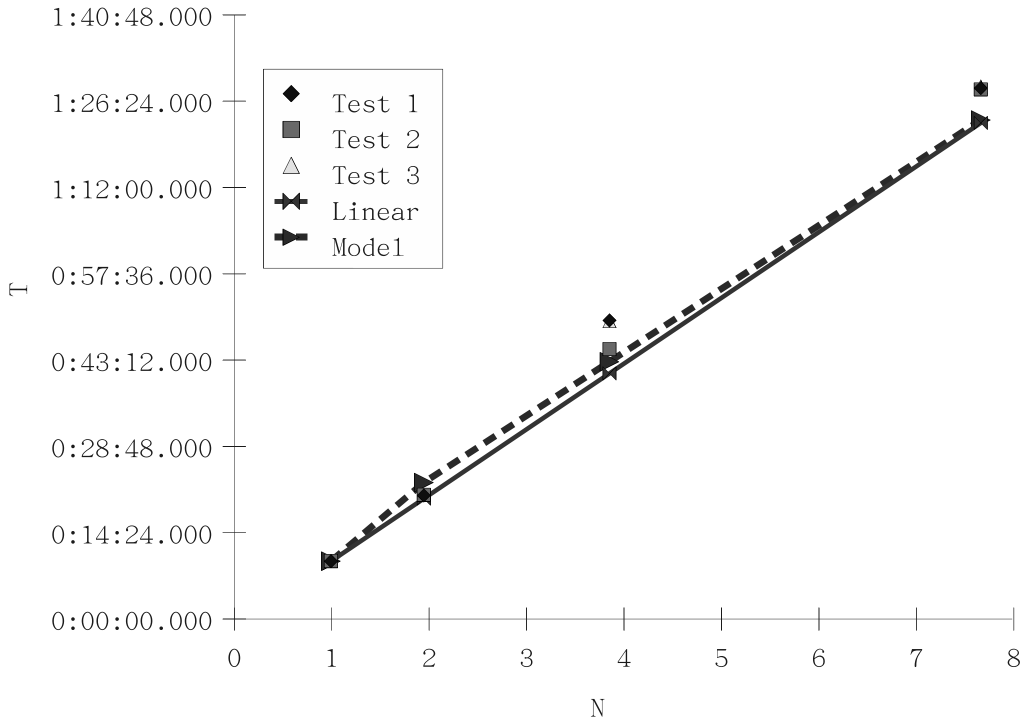


Figure 15: Measured processing speeds for N between 1 and 8 for three test runs, also showing a linear extrapolation of processing time from $N = 1$ and the results of the derived empirical timing mode.

The PPU was presented; a system developed originally for satellite-based image processing, consisting of twenty low-power ARM processing nodes in a custom interconnect network. Methods were discussed for the emulation or simulation of the PPU hardware and software environment for the purposes of code development, testing and verification.

The dynamic code translation QEMU emulator is presented and then arranged as a cluster of virtualized ARM processors running on a loose cluster of x86 machines. This is shown useful for testing and validation of code for the emulated system. The benefits of this QEMU-based system for embedded clusters include the low cost of hardware and software, the ease of prototyping, development and testing on such a system, the simple access to the system and the ability to infer or estimate timing information. Disadvantages include possible mismatch of actual system-on-chip peripherals to those required, and the fact that the virtualized system is not cycle-accurate.

Although the virtualized emulation is not cycle-by-cycle: given that the PPU is a twenty-PN custom parallel processing hardware device using a custom bus interconnect, and arbitrated by four relatively unusual FPGAs, with all processors running a high level OS (Linux), such a simulation would be extremely difficult to construct, and undoubtedly hugely time consuming in operation. However since the architecture is that of a loosely-coupled asynchronous Beowulf cluster, cycle-by-cycle simulation is likely to be unnecessary in most cases, particularly where fault-tolerance is concerned. The system has been characterised in terms of CPU performance, TCP throughput and UDP throughput/jitter for a variety of cluster sizes, as an aid to others in building and operating such systems.

The cluster is demonstrated in use modelling an image processing task designed for the PPU – the edge detection of polarimetric synthetic aperture radar images using Roy's largest eigenvector image. An empirical model is developed using measured timing data for a number of nodes over different test runs and compared to actual measured timing for between 1 and 8 PNs running on the QEMU virtualized cluster computer.

9 Acknowledgements

The authors would like to acknowledge with thanks the advice and support of Lee Ken Yoong of EADS Innovation Works related to the Roy's largest eigenvector implementation and POLSAR data handling in particular.

References

[1] J. M. Beaulieu and M. Goldberg. Hierarchy in picture segmentation: a stepwise optimization approach. volume 11-2, pages 150–163, 1989.

[2] F Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference*, 2005.

[3] S. Chylek. Collecting program execution statistics with Qemu processor emulator. In *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, pages 555 –558, October 2009.

[4] M.R. Fowler, E. Stipidis, and F.H. Ali. Practical verification of an embedded Beowulf architecture using standard cluster benchmarks. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, pages 140 –145, Oct. 2008.

[5] C. Heng. Free arm emulators, http://www.thefreecountry.com/emulators/arm.shtml, accessed May 2012.

[6] W.-C. Hsu, S.-H. Hung, and C.-H. Tu. A virtual timing device for program performance analysis. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2255 –2260, 29 2010-july 1 2010.

[7] K.-Y. Lee *Polarimetric Synthetic Aperture Radar Image Processing for Land Cover Classication: PhD Thesis.* Nanyang Technological University, Singapore, 2008.

[8] K.-Y. Lee and T. Bretschneider. On detecting edges in polarimetric synthetic aperture radar imagery. In *Proceedings of the 27th Asian Conf. on Remote Sensing*, number J1_J4, 2006.

[9] K.-Y. Lee and T. Bretschneider. Segmentation of dual-frequency polarimetric sar data for an improved land cover classication. In *Proceedings of the 31st Asian Conf. on Remote Sensing*, 2010.

[10] I. V. McLoughlin, T. Bretschneider, and B. R. Ramesh. The first linux Beowulf cluster in space. *Linux Journal*, 137:34–38, September 2005.

[11] I. V. McLoughlin, V. Gupta, S. Singh, S. Lim, and T. Bretschneider. Fault tolerance through redundant COTS components for satellite processing applications. In *Proceedings of the International Conference on Information, Communications and Signal Processing and IEEE Pacific-Rim Conference on Multimedia*, volume 1, pages 296–299, 2003.

[12] I. V. McLoughlin. Virtualized development and testing of embedded computing clusters. In *Second International Conference on Networking and Computing*, pages 17–26, December 2011.

[13] I. V. McLoughlin and T. Bretschneider. Reliability through redundant parallelism for micro-satellite computing. *ACM Trans. Embedded Computing Systems*, 9, Aug 2009. Iss. 3.

[14] Y. Nakamoto, K. Yabuuchi, and T. Osaki. Virtual software execution environments for distributed embedded control systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 288 –296, March 2011.

[15] Y. Nakamoto, K. Yabuuchi, T. Osaki, T. Kishida, T. Hara, I. Abe, and A. Kitamura. Proposing a hybrid software execution environment for distributed embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*, pages 176 –183, May 2010.

[16] Q. H. Nguyen, K. Y. Lee, M. T. Aung, T. Bretschneider, and I. McLoughlin. Hardware-accelerated edge detection for polarimetric synthetic aperture radar data. In *Geoscience and Remote Sensing Symposium,2009 IEEE International,IGARSS 2009*, volume 4, pages IV–204 –IV–207, July 2009.

[17] B. Ramesh, D. Mohan, T. Bretschneider, and I. V. McLoughlin. Centralised computation service architecture for the X-Sat micro-satellite. In *Proc. 5th International Academy of Aeronautics Symposium on Small Satellites for Earth Observation*, Berlin, Germany, April 2005. IAA.

[18] L. Rudolph. Project oxygen: Pervasive, human-centric computing an initial experience. In Klaus Dittrich, Andreas Geppert, and Moira Norrie, editors, *Advanced Information Systems Engineering*, volume 2068 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

[19] M. Kang. Skyeye - open source simulator, http://www.skyeye.org/index.shtml, accessed May 2012.

[20] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, A. Udaya. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[21] Hu Wei. Armware, http://code.google.com/p/armware, accessed May 2012.

[22] J. White and A. Pilbeam. A survey of virtualization technologies with performance testing. *Computing Research Repository*, abs/1010.3233, 2010.

[23] T.-C. Yeh and M.-C. Chiang. On the interface between QEMU and SystemC for hardware modeling. In *Next-Generation Electronics (ISNE), 2010 International Symposium on*, pages 73 –76, November 2010.

[24] T.-C. Yeh, G.-F. Tseng, and M.-C. Chiang. A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development. In *MELECON 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 1033 –1038, April 2010.

[25] G. Yua, T. Vladimirova, and M.N. Sweeting. Image compression systems on board satellites. *Acta Astronautica*, 64:988–1005, 2009.

[26] S. Yuhaniz, T. Vladimirova, and M. Sweeting. Embedded intelligent imaging on-board small satellites. In Thambipillai Srikanthan, Jingling Xue, and Chip-Hong Chang, editors, *Advances in Computer Systems Architecture*, volume 3740 of *Lecture Notes in Computer Science*, pages 90–103. Springer Berlin / Heidelberg, 2005.