Robert Taylor

Implementation of a Linux Kernel Module to Stress Test Memory Subsystems in x86 Architecture

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
27th February 2011

Metropolia

Abstract

Author(s) Title Number of Pages Date	Robert Taylor Implementation of a Linux Kernel Module to Stress Test Memory Subsystems in x86 Architecture 36 pages 27 February 2011
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructors	Antti Piironen, Dr (Project Supervisor) Janne Kilpelainen (Hardware Supervisor, NSN) Antti Virtaniemi (Software Supervisor, NSN)

The aim of the project was to implement software which would stress test the memory subsystem of x86 based hardware for Nokia Siemens Networks (NSN). The software was to be used in the validation process of memory components during hardware development. NSN used a number of software tools during this process. However, none met all their requirements and a custom piece of software was required.

NSN produced a detailed requirements specification which formed the basis of the project. The requirements left the implementation method open to the developer. After analysing the requirements document, a feasibility study was undertaken to determine the most fitting method of implementation. The feasibility study involved detailed discussions with senior engineers and architects within both the hardware and software fields at NSN. The study concluded that a Linux kernel module met the criteria best.

The software was successfully implemented as a kernel module and met the majority of the requirements set by NSN. The software is currently in use at NSN and is being actively updated and maintained.

Memory testing software within the Linux kernel has not been implemented previously due to a number of risk factors. However, this implementation demonstrated that it can be achieved successfully. The software has the possibility to be extended for other memory testing purposes in the future.

Keywords	Linux kernel, memory test, x86, memory allocation, virtual memory, loadable kernel module, memory pages

Contents

1 Introduction

The aim of the project was to develop software to be used in the Dual In-Line Memory Modules (DIMM) validation process for Nokia Siemens Networks (NSN). The software was required to stress test the memory subsystem of x86-based Central Processing Unit (CPU) blades which are used in NSN network elements, such as the FlexiPlatform, a third-generation (3G) telecommunications exchange. For the blades to function optimally it is essential that the DIMM modules are compatible with the existing hardware and software and that they are capable of handling heavy loads and high temperatures whilst maintaining signal integrity (SI).

During the NSN validation process multiple combinations of DIMM modules from different vendors are tested using a variety of memory stress test software. The key requirements for the software were automation, simplicity of use, extensive stress on the memory buses and detailed logging. The software was also to be designed to thoroughly test SI on the memory buses and generate maximal heat. The software implementation method was to be decided by the developer after conducting a feasibility study. The scope of the project did not include testing individual errors in Dynamic Random Access Memory (DRAM) cells.

Physical memory is an essential component in devices ranging from supercomputers to embedded devices such as mobile phones. In x86-based systems physical memory commonly refers to DIMM modules with mounted DRAM cells, which are inserted onto the systems' motherboard. The CPU communicates with physical memory via the Northbridge, or Memory Controller Hub (MCH), to which it passes data over the Front Side Bus (FSB). Modern CPUs also utilise memory caches when interfacing with physical memory.

Memory management and optimisation is a key field of study in both hardware and software development, particularly how an Operating System (OS) manages physical memory. In x86 systems the CPU uses techniques called segmentation and paging to manage physical memory implemented by the Memory Management Unit (MMU). Due to the limiting 32-bit address space (2^32 = ~4GB) Physical Address Extension (PAE) was introduced which added four more address lines resulting in the ability to access up to 64GB.

The Linux kernel also employs paging, and to a lesser degree segmentation, techniques to manage physical memory. The Linux kernel is responsible for managing every process' request for memory allocation, whilst simultaneously managing its' own data structures. When operating on a x86 32-bit hardware platform the kernel also utilises a 4GB address space, or 64GB if PAE is enabled. The kernel also employs a virtual memory model to manage physical memory.

NSN are one the world's leading communication services providers and operate in around 150 countries, employing 60,000 people. The company was formed on 19 June 2006 as a joint venture between Nokia Networks and Siemens Communications. NSN solutions include radio and broadband access, IP transport, mobile backhaul and Long Term Evolution (LTE). The project was undertaken in the Original Design Manufacturers (ODM) team within the Research and Development department at NSN.

2 Background to Physical Memory and Memory Management

2.1 The x86 Architecture

2.1.1 History

The x86 architecture is the most ubiquitous and successful instruction set architecture in the world today. Originally developed by Intel, the x86 architecture has been implemented by a number of manufacturers including Advanced Micro Devices (AMD) and VIA Technologies. Intel’s own documentation refers to Intel Architecture 32-bit (IA-32) and not x86. However, despite the focus of this paper being IA-32 processor based systems the more generic x86 will be used throughout.

The x86 processor family has been developed by Intel since 1978. The term x86 originates from the ending of the processor’s model number. The first x86 processor was the Intel 8086 which used 16-bit registers, 20-bit addressing and implemented segmentation to create a 1MB address space. In 1982 the Intel 286 processor introduced protected mode which provided a 16MB address space using descriptor tables, added protection mechanisms including privilege levels and also added support for virtual memory. [1, 33]

The Intel 386 (i386) was the first 32-bit processor in the x86 family and was introduced in 1985. The i386 uses 32-bits for addressing providing a 4GB address space and introduced paging with a page size of 4096KB, which remains in Intel x86 processors today. The i386 also introduced a segmented-memory model and a flat memory model [1, 34].

The x86 continued to evolve with key features added such as the first-level, or L1, 8KB write-through cache in the Intel 486 in 1989. In 1993 the Pentium processor doubled the L1 cache to 16KB split between 8KB data and instruction caches (L1d and L1i respectively). In addition the caches use the Modified Exclusive Shared Invalid (MESI) protocol and introduced the write-back method. Support for larger 4MB pages was included and later the Pentium III would introduce the MMX instruction set which utilises 64-bit registers. [1, 34-36]

Intel's x86 processors have now evolved to increase the size and effectiveness of the L1 caches, added a larger second-level (L2) cache, introduced dual cores and increased the frequency of the FSB and their own clock speeds The NetBurst microarchitecture delivered a new cache subsystem including an 8-way associative L2 cache with 64B line size and the capability to deliver 8.5GB/s bandwidth. [1, 35-43]

2.1.2 Memory Organisation

Physical memory is organised by x86 processors into a sequence bytes with each byte having a physical address associated with it. In a 32-bit processor with Page Address Extensions enabled 64GB of memory can be addressed in the physical address space. Processors provide three models dependant on configuration for accessing memory. [1, 76]

The flat memory model provides a linear address space, whereby each byte is addressed on a one-to-one basis by the processor, providing 2^32 (~4GB) of addressable memory. The segmented model presents memory as a sequence of segments which are then accessed using logical addresses. These logical addresses are then mapped into the linear address space. The segmented model provides the same amount of addressable memory as the flat model. The final memory model is the real-address model which also divides memory into logical addresses, however, this model is tailored for 8086 processors and provides a much smaller address space of 2^20 bytes. [1, 76 – 77]

Paging is employed to address memory when a one-to-one linear mapping does not exist. When paging is enabled the linear address space is divided into pages, either 4KB, 2MB or 4MB in length, which are then mapped into virtual memory and stored either to memory or to the disk. Pages mapped into virtual memory are retrieved and mapped into physical memory as required. The virtual memory simulates a larger linear address space than the physical memory can provide. Pages are accessed using the page directory and page tables which translates the requested linear address into a physical one. [1, 78; 2, 92 – 93; 3]

Therefore, when a programming instruction requires access to the physical memory its logical address as provided by the CPU must be translated via the segmentation process into a linear address. The linear address is then translated during through paging into a physical address which is passed to the CPU's MCH which interfaces directly with the physical memory.

2.2 Physical Memory

2.2.1 Physical Structure and Operation

Dynamic Random Access Memory (DRAM) is commonly employed in a wide variety of computing systems today. DRAM cells consist of a capacitor, which stores the state of the cell (either 0 or 1) and a transistor which controls access to the state for other components. A data line is connected to the transistor, which is in turn connected to the capacitor and also to an access line that governs access for reads and writes. Writes are achieved by charging the capacitor with current issued along the data line should access be granted by the transistor. Reading of the cell results in the capacitor being discharged. As the capacitor is charged and discharged capacity leaks from the capacitor and the cell must be refreshed. [4, 5 – 6]

DRAM cells are stored in matrices which can range from 128MB to 4GB per DIMM. With addressable cells reaching such vast numbers it is impossible to address every cell with its own address line. In order to access the cells a multiplexer and demultiplexer using a smaller set of address lines are required.

The matrices of cells are accessed with row and column address selection (RAS and CAS) lines. The RAS line is a demultiplexer which accesses rows of DRAM cells whilst the CAS line multiplexer accesses the columns in the matrix. The result is delivered to the data bus via the CAS. [4, 6 - 7]

2.2.2 Memory Errors

There are a wide variety of causes of memory errors in DIMM modules, especially when in constant use. The process of charging and discharging the capacitors can result in memory errors. The state of the cell can be difficult to determine as charging and discharging occurs exponentially in a capacitor rather than a square wave signal. [4; 6]

The high density of memory cells on DIMM modules increases capacitivity which in turn contributes to signal degradation due to noise. Intensive use of DRAM cells can also overheat which results in voltage errors in reading and writing. In fact memory errors can have a wide range of causes including proximity to magnetic fields and even cosmic radiation. [4, 105; 5]

In extreme circumstances persistent memory can result in an entire system becoming unstable or unusable. In the majority of cases memory errors are seen in incorrect data patterns. When these errors occur in small numbers they can be handled sufficiently using Error Correcting Code (ECC) DIMM. ECC enables error recognition in the hardware to identify and rectify errors as they occur. In extreme circumstances persistent memory can result in an entire system becoming unstable or unusable. [4, 105; 5]

2.3 Memory Management in the Linux Kernel

2.3.1 Page Frames and Memory Zones

The Linux kernel is monolithic in design, consisting of several logically different components. Linux allows kernel modules to be added dynamically when the system is running. Linux supports multiple processors, the NUMA memory model and can utilise numerous file systems. At the core of the Linux kernel is virtual memory management. [6, 2-3] Due the large number of architectures it supports the Linux kernel must describe the physical memory in an architecture-independent manner.

As discussed previously in section 2.1.1, in x86 architecture systems the CPU has a 4GB (2 ^ 32) virtual address space. When paging is enabled in the CPU the Linux kernel splits this 4GB address space into two, a 1GB split for the kernel space and 3GB for user space. The division is defined by the constant `PAGE_OFFSET` whose value is 0xC0000000 in hexadecimal, or 3221225472 in decimal (3GB). In virtual memory, the linear addresses from 0 to `PAGE_OFFSET` are reserved for user space and from `PAGE_OFFSET` to 0xFFFFFFFF, or 4GB, for the kernel space which totals 1GB. It is important to note that the linear address space above `PAGE_OFFSET` can only be addressed by the kernel and not from within a user space process. Therefore the kernel, or a process in kernel mode, is able to address the whole linear address space from 0 to 0xFFFFFFFF, whereas in user mode only 0 to 0xC0000000 is addressable. The physical memory split is reversed with 0 to 1GB allocated to the kernel space and addresses above 1GB belonging to user space. The `PAGE_OFFSET` constant is an important value and is used in the paging and addressing process. [6, 68; 8, 82]

The physical memory is divided into page frames which are the same size as the page size defined by the architecture (4KB in x86) and as the kernel constant `PAGE_SIZE`. Each page frame is represented by the `page` structure, see Listing 1 below, which itself is ~40B and is stored in the `mem_map` array. The page structure for each frame details the current status of the page frame for the kernel's use but contains no information about the data it holds. [6, 295]

```
struct page {
        unsigned long flags;
...
        atomic_t _count;
        atomic_t _mapcount;
...
        union {
            struct {
                unsigned long private;
...
                struct address_space *mapping;
...
            };
#if NR_CPUS >= CONFIG_SPLIT_PTLOCK_CPUS
```

```
            spinlock_t ptl;
#endif
        };
        pgoff_t index;
        struct list_head lru;
...
#if defined(WANT_PAGE_VIRTUAL)
        void *virtual;
...
#endif /* WANT_PAGE_VIRTUAL */
};
```

Listing 1: The Linux kernel page structure

Pages are mapped to memory zones using a bit shift in the `flags` field of the `page` structure. More importantly, the `flags` field describes the current status of the page using a number of macros defined by the kernel to retrieve and set bits within the field, for example `PG_highmem` which returns true if the page is in high memory [6, 296 - 297]. A further salient member of the `page` structure is the `virtual` pointer. This pointer refers to a kernel virtual address if the page has been mapped or `NULL` if the page is in high memory.

The physical memory is divided into nodes depending on the processor memory configuration, the node or nodes consist of page frames. In both NUMA and UMA machines memory is arranged into nodes dependent on their access distance from the processor. Each node is then split into zones representing different regions. In an x86 machine, when UMA is the memory model, there are three zones in one singular node which is defined and stored to the `contig_page_data` kernel variable.

In a default x86 UMA configuration the node is divided into three memory zones as follows `ZONE_DMA` (0 – 16MB), `ZONE_NORMAL` (16MB – 896MB) and `ZONE_HIGHMEM` (> 896MB). The `ZONE_DMA` is used for Direct Memory Access (DMA) memory operations and is limited to 16MB for legacy bus addressing reasons. `ZONE_NORMAL`, or low memory, is the most critical memory area as it is where the kernel code and data structures reside. The remaining physical memory is placed in `ZONE_HIGHMEM` and is referred to as high memory. Low memory does not account for the full 1GB kernel space as a further 128MB are reserved for mapping high memory addresses. [6, 299; 8, 26]

2.3.2 Addressing and Mapping Memory

Linux is a virtual memory system, therefore addresses which are seen by user processes do not necessarily correspond to physical addresses in the hardware. Linux has a number of different address types which each have their own semantics and related functions. Physical addresses refer to those which are passed between the CPU and the physical memory, in x86 these addresses are 32-bit values (36 bit if PAE is enabled). Kernel logical addresses are used in the kernel's address space and map a region of physical memory. Logical addresses can be viewed as physical addresses as they tend to be offset by a constant. A kernel virtual address also maps to physical address. However it may not be a linear mapping. All logical addresses in the kernel are virtual addresses but the reverse is not true. [6, 35 -36; 7, 413 - 414]

The kernel provides a number of useful functions to retrieve memory addresses, for example `virt_to_phys(address)` returns the physical address from a given virtual address and its reverse function `phys_to_virt(address)`. Likewise the page frame number (`pfn`), which is an offset in `mem_map` indexing the page structure, is also used in functions to retrieve the page address and vice versa. However, not all page frames are assigned virtual addresses and those in high memory must be mapped into the kernel address space.

As discussed previously in section 2.3.1, the low and high memory model in x86 kernels only provides addresses for the first GB of memory, the watermark for this is defined by `PAGE_OFFSET`. Beyond `PAGE_OFFSET` a number of boundaries are set to provide address space for mapping high memory. An address space is provided for memory allocated with `vmalloc()` begins at `VMALLOC_START` and ends at `VMALLOC_END`. The kernel also requires an address space to map high memory addresses, the Persistent Kernel Map (PKMAP). The PKMAP space is located at

`PKMAP_BASE` for a size dependent on the `PAGE_SIZE` and `LAST_PKMAP` constants. The `LAST_PKMAP` constant depends on whether PAE is enabled, if PAE was configured then it has a value of 512 otherwise, in a default configuration, the value is 1024; resulting in mappings of 2MB and 4MB respectively. [8, 53 – 56]

High memory is mapped into the PKMAP using the `kmap()` function and unmapped with the `kunmap()` function. The `kmap()` function works by accessing the flags field of the page structure to check whether the page is marked as high memory. If the page is not marked as high memory then the page's linear address may be returned immediately to the caller. However, if the page was marked as high memory it must first be mapped into the PKMAP space, which results in the page then having an address to access it. [6, 308 - 309]

3 Software Requirements Overview and Analysis

Prior to commencing the project in NSN premises a software requirements document was produced by Janne Kilpelainen, a senior HW specialist and supervisor for the project. The document detailed the minimum set of requirements required by the ODM team for the DRAM validation software. The document specified that during implementation improvements and enhancements to the requirements were encouraged. The document served as the foundation for the project.

The primary requirements for the software were to provide two operational modes: the SI mode, which would electrically stress test the memory subsystem whilst verifying signal integrity and a thermal mode, which should generate maximum heat in the DIMMs. From the user's perspective the software must be automated, boot-to-test in nature and easy to use. In further detail the requirements were as follows:

— Minimal user interaction after booting
— Self booting, automated operation
— A simple effective user interface (UI)
— All accesses (reads/writes) at full bus width, 64/128 bits plus ECC
— Ability to change processor speed
— Determine whether errors are in read or write operations
— Disable/enable ECC
— SI mode has a variety of test patterns
— Detailed reporting to both screen and file
— Can be ported to other CPU architectures eg PowerPC and MIPS
— Modular code design for future updates

In addition the UI should allow the user to set the processor speed, set the clock, select the operational mode, the number of loops and provide clear logging during operation. The software was to be used in x86 based CPU blades and must automatically detect the DIMM configuration of these blades and run the tests accordingly. The CPU blades are in listed in Table 1 below, in addition the software design should consider future CPUs during implementation

Table 1. Technical specifications overview of CPU blades required in implementation

CPU Blade	Processor	MCH	Memory	OS	BIOS/ USB
CPU-1	Pentium M (Dothan) 1.8GHz	E7501	DDR 200/266. 4 DIMMsockets (4GB Max)	Linux (Wind River) /NSN OS	No/No
CPU-2	2 x Intel Xeon LV 1-2GHz	E7520	DDR2 400. 4 DIMM sockets (16GB Max)	Linux (Wind River) /NSN OS	Yes/Yes
CPU-3	Penryn SP9300 1.6 –2.26GHz	5100	DDR2 533/667. 4 DIMM sockets (16GB Max)	Linux (Wind River) /NSN OS	Yes/Yes
CPU-4	2 x Intel Xeon LV 2GHz	E7501	DDR 200/266. 4 DIMM sockets (8GB Max)	Linux (Red Hat Enterprise Linux, RHEL)	Yes/Yes

As detailed in Table 1, all the CPU blades run the Linux kernel which would become a key fact during the feasibility study. Other salient facts derived from the table include the lack of a BIOS or USB slot in CPU-1 and that overall the maximum required for DRAM testing would be 16GB.

3.1 Overview of the DIMM Validation Process at NSN

During the development of a new CPU blade, the manufacturers are required to validate DIMM modules which have been selected by NSN. The purpose of the manufacturer validation is to ensure signal integrity in the memory interface and that the CPU and DIMM modules operate under various thermal conditions.

Once this work has been undertaken and reviewed, NSN begins a further DIMM validation process. This process also examines signal integrity and thermal behaviour but also acts as validation that NSN software will work in harmony with the CPU and DIMMs. The manufacturer validation does not guarantee that the DIMMs will function correctly when the CPU is running with real-world user configurations. Upon successful completion of NSN's validation process DIMMs are approved for use in the CPU blades.

The NSN DIMM validation process is conducted in a thermal chamber which can generate a sustained temperature of up to 100°C. The CPU is housed inside the chamber and serial port connection established to an external PC for operation of the memory testing software. The CPU is then tested with its minimum and maximum DIMM configurations installed, and also by using DIMMs with varied capacities, from different vendors and in combination with previously approved DIMMs.

For all DIMM configurations the memory test software tests are run whilst the temperature is simultaneously incremented to 80°C. At each 10 degree interval the temperature is stabilised for between 30 and 60 minutes. The entire test cycle can take up to 24 hours for 16GB memory when using Memtest86+, for example.

Further requirements in the validation process prior to running the software must also be configured in the BIOS. The processor must be running at its lowest frequency in order to ensure there is no throttling which may disturb the validation. ECC correction should be disabled so that errors can be detected and a normal DRAM refresh rate be selected rather than an auto refresh rate.

The ODM team use a variety of software currently during the DIMM validation process. The advantages and disadvantages as detailed in the software requirements document are summarised in Table 2 below.

Table 2. High Level Assessment of Current Software

Software	**Advantages**	**Disadvantages**
Memtest86+	Free, open source, some chipset support, supports large amounts of memory, ease of use	Only 32 bit accesses, does not stress test memory bus
UTE.img	NSN internal, source code	Max 4GB testing range, complex

	available, stresses memory bus, 64/128 bit accesses	usage, only for NSN internal OS
Intel MARSE	Intel developed, stresses memory bus, 64/128 bit modes	Runs on DOS, limited maximum memory
T3memio	NSN internal, source code available, thermal stress test, Linux based	Cannot stress test memory bus or specific DIMM

Table 2 illustrates the need for software which combines the advantages of those currently used whilst eliminating their disadvantages.

3.2 Feasibility Study

The software requirements document did not suggest how the software would be implemented in practice. However, after careful analysis of the document, particularly those sections referring to the software currently used and what OS is running on the blade (see Table 2 above), it was clear that there were two primary options for implementation: a standalone binary or a Linux-based implementation. A Linux implementation itself offered two further options, user space or kernel space (LKM). These development options would serve as the basis for the feasibility study.

The purpose of the feasibility study was to gain an insight into the advantages and disadvantages of the potential implementations from more experienced colleagues. The study consisted of a series of interviews, discussions, meetings and email exchanges with senior engineers employed by NSN. These findings from the feasibility study have been summarised in Table 3 below.

Table 3. Advantages and Disadvantages of Differing Software Implementation Options

Software Implementation	**Advantages**	**Disadvantages**
Linux Kernel Space (LKM)	- All CPU blades run Linux - Can utilise kernel functionality - Direct access and ability to address physical and virtual memory (kernel and user) - Can call the kernel directly - Easily ported to other architectures	- Kernel crashes easily when working with memory - Kernel occupies memory itself - Memory management is complex - Abstraction from HW causes SW to run slower - Debugging kernel is complex - LKM cannot be killed if it errors during operation
Linux User Space	- All CPU blades run Linux - Can utilise standard C library and Linux system calls - Debugging is simple - User space processes can be killed easily - Reduces development effort as memory management is handled by the kernel	- Cannot access kernel space memory - Cannot directly address memory - Memory allocation is random - More abstraction before instructions are executed - Context switch makes processes slower
Standalone Binary (based on memtest86)	- Improved speed - Direct access to hardware - More memory available	- Only for x86 architecture - CPU-1 has no BIOS - Increased development effort

The feasibility study results illustrated that all three development approaches met the software requirements to varying degrees. An extension to memtest86 appeared to be the most exhaustive in terms of development effort and also would prove difficult to adapt when new architectures are introduced. Whilst the development of Linux user space software would have been simpler, its high level of abstraction from the hardware and the inability to directly access memory or the kernel space was a clear disadvantage.

Whilst having an increased development load and a steep learning curve, the LKM approach met the software requirements much more closely. Using an LKM the software would be able to access all memory and address it precisely. As a developer it would also provide a useful insight into the internals of the Linux kernel and particularly its memory management subsystem. It was decided that an LKM would be the chosen implementation. It was also determined that using a USB bootable Linux image would be the best way to deploy the software.

4 Software Design Methods

4.1 Deployment

As determined during the feasibility study, Kmemtestd (kernel memory test daemon) would be developed as a dynamically loadable kernel module, operating in kernel space, which can be inserted into the kernel on demand via a user interface (UI), or manually using the `insmod` Linux command.

Kmemtestd was developed purely in the C programming language and the user interface and logging functionality were developed with shell scripting. C was chosen as the development language as it provides low level access to physical memory. Using C also allowed for simple access and use of existing kernel structures.

The software requirements stipulated that the software be boot-to-test in nature and as automated as possible. Using a USB bootable image of the Linux kernel would provide the simplest deployment method for operation as the kernel can be booted automatically by setting the appropriate BIOS settings prior to powering on the hardware.

The user interface (UI) for Kmemtestd is a shell script which collates basic system information and presents it the user, for example, CPU speed and the number of CPUs present. The UI collects data through a series of prompts and passes it as module parameters when loading the module, such as the number of loops the test routines will perform, whether ECC and the cache will be enabled or disabled. The UI is also used to unload the module itself from the kernel once complete and also produce logs and write those logs to the mounted USB drive.

4.2 Operation

When Kmemtestd has been loaded into the kernel, it is initialised as a kernel thread. Kernel threads exist only in the kernel space and provide the module with access to kernel data structures and the kernel address space. Threads also have direct access to the kernel symbol table and system call library. Therefore a kernel thread was an ideal choice for the Kmemtestd module as it must utilise the above kernel features. Another reason to choose a thread was that it can be killed cleanly by the user.

Once the thread has been started, the hardware is initialised either based on parameters passed by the UI or its default behaviour. Hardware initialisation disables/enables the L2 cache and the ECC registers. The default configuration is that the cache is enabled and ECC is disabled. Both sets of registers are accessed via kernel structures: MTRR for the cache and the PCI device list for ECC.

If hardware initialisation did not fail the thread is daemonised and begins to calculate the available memory. Kmemtestd automatically allocates the maximum available memory without disturbing the kernel and then performs stress tests on the allocated memory with a variety of data patterns. Upon completion of the test routines the allocated memory is freed and the software exits cleanly. During execution the user may send a kill signal to the software which also frees the memory and exits cleanly. The execution of Kmemtestd is described in pseudocode in Listing 2 below.

```
Kmemtestd_start() {
      initialise_software_as_kernel_thread();
      initialise_hardware();
      daemonise();
      calculate_available_memory();
      allocate_memory();
      while(USER_DEFINED_LOOPS || INFINITE_LOOPS){
                if(no_kill_signal_received){
                          do_stress_tests();
                }
                USER_DEFINED_LOOPS--;
                } else {
                free_memory_then_exit();
      }
      free_memory_then_exit();
}
```

Listing 2. Kmemtestd execution in pseudocode

4.2 Memory Allocation

Prior to allocation the size of the installed memory should be assessed and stored for Kmemtestd to handle. Much of the information about the system required by Kmemtestd is already known to the kernel and will be utilised via kernel interfaces, structures and macros. The `/proc` filesystem stores information about the current use of memory resources. As discussed in 2.4.1 Linux uses three memory zones, only ZONE_NORMAL and ZONE_HIGHMEM are assessed and a safe allocation total is calculated. ZONE_DMA is excluded as a future feature is planned whereby the DMA controller will be driven concurrently as Kmemtestd performs its routines, thereby increasing stress to the memory subsystem.

The low and high memory zones have `kswapd` watermarks set by the kernel. Should the amount of available memory go below these watermarks `kswapd` will begin reclaiming memory. In order to prevent this occurring, the watermark amounts are doubled and subtracted from the to-be allocated memory amount.

The kernel has limited access to the physical memory of the system, as detailed in section 2.4.2, and as a result memory must be mapped into the kernel map space. Kmemtestd calls the `LAST_PKMAP` macro which returns the number of page table entries available for mapping. This value is then utilised as a variable for iterating through the memory in Kmemtestd’s mapping functions.

5 Software Implementation Details

5.1 Hardware Initialisation

Hardware initialisation is executed once the module is loaded into the kernel and accepts parameters passed by the user from the UI or uses its default values. Kmemtestd utilises the kernel's PCI device list structures and functions to access ECC bit stored in the CPU memory controller register. Due to its customised nature Kmemtestd works on the assumption that the type of CPU board is one of four variants. The ECC bit value and MCH registers for each CPU board were therefore required to be pre-defined for hardware initialisation interactions. The specific registers were noted from the hardware specifications of each CPU. Interfacing with the L2 cache does not require such customisation.

Whilst the kernel PCI device list covers a wide range of CPU manufacturer's, not all registers or device ID values are defined in each kernel. Listing 3 below details some of the PCI device IDs used by Kmemtestd .

```
#define PCI_DEVICE_ID_INTEL_5100_16        0x65f0
    /* PCI dev id for I5100 */
#define PCI_E7501_DRC                      0x7c
    /* Memory Control Register for e7520 and e7501 */
#define I5100_MC                           0x40
    /* Memory Control Register for i5100 */
```

Listing 3. Kmemtestd defines for various hardware elements.

In the Linux kernel version used for development it was necessary to manually define some PCI device values taken from later kernel versions for use in Kmemtestd.

5.2 Allocation Amounts

Once executed Kmemtestd queries the available high and low memory from the kernel. This is achieved via two methods. The first uses the `sysinfo` kernel structure as used by the `proc` filesystem for example in `/proc/meminfo`. The `sysinfo` structure is then populated with the kernel `si_meminfo()` function, as illustrated in Listing 4.

```
struct sysinfo mem;
...
si_meminfo(&mem);
```

Listing 4. Kmemtestd use of the proc filesystem.

The second method is used at various points in the code and simply queries the kernel for the value of total free pages across the system. This was implemented in a simple function, shown in Listing 5, the `K()` wrapper simply converts the returned value to kilobytes.

```
/* retrieves current system wide free ram */
static unsigned long curr_free_ram(void)
{
    return K(global_page_state(NR_FREE_PAGES));
}
```

Listing 5. Kmemtestd's function to retrieve the amount of currently available memory.

The members of the `mem` structure are then accessed to provide the total available memory in the low and high memory zones. These totals then have their respective watermarks doubled and then subtracted from the total in order to avoid `kswapd` interference. The available memory is then divided by the size of the PKMAP which results in a granularity used in the allocation routines, as shown in Lisitng 6 below.

```
init_freeram = curr_free_ram(); /* get initial free RAM according to kernel */
...
/* calculate the amount of memory to reserve based on free mem and kswap watermarks */
/* then divide each amount by the granularity eg how many pages to be allocated/freed per page pointer */
lowmem_alloc_size = mem.freeram - mem.freehigh;
lowmem_alloc_size -=(NODE.node_zones[lowmem].pages_high * 2) + (NODE.node_zones[lowmem].pages_min * 2));
lowmem_alloc_size /= CUR_PKMAP_SIZE;
highmem_alloc_size = mem.freehigh;
highmem_alloc_size -= NODE.node_zones[himem].pages_high + NODE.node_zones[himem].pages_min);
highmem_alloc_size /= CUR_PKMAP_SIZE;
```

Listing 6. Memory allocation technique used in Kmemtestd.

The final totals `lowmem_alloc_size` and `highmem_alloc_size` are then passed to the allocation functions. The allocation totals essentially represent the number of `page` pointers which will point to allocated page ranges.

5.3 Page Handling

Kmemtestd utilises the kernel’s implementation of the linked list data structure to store the allocated memory. The linked list was chosen as its size is dynamic and therefore reduces memory consumption or possible overflows. The kernel’s implementation is efficient and optimised for use in the kernel. The elements of the list are structures, see Listing 7, containing a pointer to the memory allocated, a pointer to the memory address (if the address is in low memory) and the page frame number.

```
/* list for storing allocated page ranges */
struct km_allocd_pages_list{
        struct list_head list;
        struct page* km_p; /* page pointer to the range of allocated pages */
        void* km_vaddr; /* virtual address (only low memory) */
        unsigned long km_pfn; /* page frame number */
};
```

Listing 7. The allocated page list structure in Kmemtestd.

As discussed previously in 2.4.2 there are a number of memory allocation functions made available via the kernel API. However, the core of the kernel’s memory allocator is the `__alloc_pages()`function which is accessed via the macro `alloc_pages()` as defined by the prototype:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
```

The `alloc_pages()` parameter flags indicates which area of memory the kernel will allocate for example GFP_KERNEL. Order is the number of pages to be allocated, 2^`order`. The function returns a pointer to the `page` structure at the head of the contiguous range of pages which were allocated. The returned pointers are then stored to the linked list. The order at which page ranges are allocated is determined by the size of the PKMAP area in the address space. As discussed in section 4.2 the `LAST_PKMAP` macro returns the number of entries in the page table. This figure is 512 (2^9) if Page Address Extension (PAE) is enabled, the case when more than 4GB of physical memory is installed, or 1024 (2^10) if PAE is not present. The power of two, or order, is stored and then used in the allocation function, which in turn passes it as a parameter to the `alloc_pages()` function, shown below in Listing 9.

```
/* Allocate memory within gfp_t type only in ranges of order in a loop based on the */
/* calculated maximum in the init function */
static void allocate_page_ranges_in_mem_zone(u64 num_page_ranges_to_alloc, gfp_t mem_zone_gfp)
{
    int i;

    for(i = 0;i < num_page_ranges_to_alloc;i++){
        km_new = kmalloc(sizeof(struct km_allocd_pages_list), GFP_KERNEL); /* kmalloc each list element */
        km_new->km_p = alloc_pages(mem_zone_gfp, ORD); /* returns a page pointer to range of ORDER order pages and add to list */
        km_new->km_vaddr = page_address(km_new->km_p); /* try to retrieve virtual address */
        km_new->km_pfn = page_to_pfn(km_new->km_p); /* get the page frame number for sorting */
        add_sorted_entry(km_new); /* sort and add to list */
    }
}
```

Listing 9. Kmemtestd `alloc_pages()` function.

The kernel's `alloc_pages()` attempts to allocate 2^(order parameter) contiguous pages in the memory zone it receives as its' first parameter. As noted above the order is based on the number entries in the PKMAP so as not to case any overflows in allocation and mapping. For example, when PAE is enabled there are 512 entries for 4096B pages totalling 2MB, therefore each allocated page range must be 2MB also.

The allocation routine is called twice, once for low memory and once for high memory. However, instead of maintaining this assumption the allocation function calls the kernel's `page_address()` function which takes a page pointer as a parameter and returns a virtual address of the page if in low memory. This address is then stored to the allocated pages list structure even when null. The virtual address is checked during the testing algorithms to determine whether the page requires mapping.

As each page range is successfully allocated the pointer to the first memory area is sorted based on its page frame number and inserted into the list. The page frame number is a number assigned by the kernel to represent the physical position of the page in the memory. By sorting based on this number the list structure should closely resemble physical memory.

The allocated pages list is stored as global variable and is then iterated through during the memory tests using the `list_for_each()` kernel iteration function. The page pointer and virtual address fields from the structure are then accessed. If the there is no virtual address then the page is in high memory and must be mapped into the PKMAP, in the case of low memery the virtual address pointer is simply cast to a 64-bit wide pointer. Mapping high memory pages is achieved via a wrapper function to the kernel API's `kmap()` function, see Listing 10.

```
static inline void *kmap(struct page *page)
{
  might_sleep();
              return page_address(page);
}

#define kunmap(page) do { (void) (page); } while (0)
```

Listing 10. The Linux Kernel kmap function.

Whether the page and its allocated page range required mapping or not the memory testing function iterates through the pages using pointer arithmetic based on the width of the test pattern. If the page range was mapped then once is has been iterated it is unmapped. Again this is achieved via a wrapper function to the kernel's `kunmap()` function. The test function then repeats this process until the full list has been iterated through.

6 Discussion

6.1 Memory Allocation Methods in Kmemtestd and the Kernel

The Linux kernel's memory management subsystem is a complex and intricately designed piece of software. Its efficiency provides a stable platform for the Linux operating system. For the novice kernel developer exploring this subsystem provides a detailed insight into the inner workings of the Linux kernel and operating system memory management in general.

The feasibility study identified a Linux kernel module as the optimum deployment method for the required memory testing software. A kernel module has access to the kernel space and the kernel API, which in turn provides access to many low level memory functions. Using a module would also provide a method to return physical addresses where memory errors occurred and precise access to memory controller registers. However, working inside the kernel space presented a number of challenges.

In user space memory allocation and manipulation is usually achieved via standard C library functions like `malloc()` and `memcpy()` which are familiar to most developers. These tools are not available in the kernel space and the kernel API's own tools required study. The API function `kmalloc()` allocates memory in small contiguous chunks up to approximately 8KB and returns a void pointer to that location. In the case of Kmemtestd, this would have resulted in an excessive number of pointers requiring storage for future iterations. Using `vmalloc()` did not meet the requirements as it allocates discontiguous chunks of memory. Both functions use the `page` structure at their lowest level as the unit of allocation, `vmalloc()` uses `alloc_page()` and `kmalloc()` uses the page via the `bigblock` structure and the slob. Therefore, the `alloc_pages()` function which returns a page pointer was selected for development as its use reduces the processing overhead, removed some layers of abstraction and produced more readable code.

Using page sized granularity proved to be very effective due to the number of functions in the kernel API providing access to and information about pages. Pages are the basic unit of memory management in the kernel. The PKMAP also uses pages when mapping high memory.

By default the Linux kernel splits the available physical memory into zones. The first 36 MB is reserved for DMA, from here to 896 MB is low memory and anything above is considered high memory. When is use and during memory tests the hardware will have 4-16 GB memory installed. Therefore, approximately 15 GB of physical memory may require mapping into the kernel space via PKMAP prior to testing. This is achieved by mapping memory in 2MB chunks, testing and then unmapping. In the instance of 16 GB of physical memory these two procedure must be performed approximately 15300 times. Mapping high memory is an unavoidable overhead but may have had some impact on Kmemtestd's ability to stress the memory subsystem.

6.2 Using Customised Kernels in Deployment

The Linux kernel which the module was to be deployed with is used is a wide variety of other applications within NSN. Therefore, modifying the configurations of the kernel itself was not an option during development and the 1 GB/3 GB default kernel memory split had to be utilised. For future development the use of a 4 GB/ 4GB kernel split would result in dramatic reduction in the mapping requirements. In fact with configuration in a customised kernel all memory could be configured as low memory for the purposes of memory testing.

A further drawback related to the use of page sized granularity is the size of the page itself, 4096 B. As discussed previously in section 6.1, mapping high memory has an associated overhead. When PAE is enabled PKMAP has 512 page entries available. If it had been possible to configure the kernel to support hugepages or superpages of 4 MB in size then 20 MB would have been mapped per operation. Again, this feature should be considered for future development.

6.3 Memory Allocation and its Danger to System Stability

Whilst the page based allocation method works very effectively, Kmemtestd assumes that the memory is fully contiguous. In general NSN usage this will be the case as the kernel is loaded only for this purpose and a bare minimum of processes are permitted. However, if Kmemtestd is executed in an environment which has many running processes memory fragmentation and lack of available memory will cause kernel crashes. Future iterations of the software would need to implement a method to assess how contiguous the system's memory is and take necessary actions to correct it.

The kernel controls and allocates memory to user space processes from within the kernel space. The kernel's own processes are also reserving and freeing memory within the kernel space simultaneously. The kernel swap daemon `kswapd` is periodically monitoring the number of free pages and will forcefully begin to free pages if the total is deemed to low. Implementing software which could safely allocate all available free memory without disturbing other processes was difficult and perhaps the reason why no memory testing software has been written in the kernel space before. This was achieved by allocating safe amounts discovered through repeated testing. An alternative approach may have been to implement a process, similar to `kswapd`, to monitor Kmemtestd memory allocations and usage. However, implementing such a process was beyond the scope of this project.

In order to address the issue of safe memory allocation amounts, the early stages of development were spent exploring the balance between maximal allocation and minimum interference with the kernel. This proved difficult and time-consuming as a result of frequent, and sometimes spectacular, kernel panics. Through repeated testing by gradually increasing the amount of memory allocated from the absolute free memory total a safe and stable amount was determined, which resulted in approximately 36MB remaining untested from installed memory between 4GB and 16GB. An alternative approach and consideration for the future would be to use the Linux `kexec` tool to load another kernel from an executing kernel. By performing a warm reboot it would be possible to move the kernel's location in the memory allowing even more memory to be tested.

As discussed in section 4.2 Kmemtestd tests only low and high memory, ZONE_DMA was omitted in the initial design. It was planned as a future improvement that a separate function or driver would be developed to run simultaneously with Kmemtestd to drive the DMA controller and increase stress to the CPU’s memory management subsystem.

6.3 Memory Test Routines

Once the initial allocation, storage and iteration issues had been addressed, the allocated memory required testing. Kmemtestd uses a battery of well-known and established memory tests patterns to be written and read. It was not necessary nor possible due to time constraints to create and compare new test patterns to those commonly used in industry standard applications such as memtest86. Further tests, for example custom patterns chosen by the user, can be simply added to Kmemtestd due to its modularity in future iterations.

The software requirements stated that read/write accesses must be both 64/128 bit wide. Despite extensive research and effort it was not possible to determine how it would be possible to achieve this. The processor uses 32-bit instructions but it is difficult to assert how reads/writes are managed by the MCH. Whilst Intel's technical specifications provide some insight, there remains a level of abstraction. By better aligning data structures and read/write accesses to the cache line length, it is assumed that the MCH will operate optimally and drive the memory bus maximally.

Kmemtestd accesses MCH registers via the PCI device structures in the Linux kernel in order to configure ECC. The PCI device bus has access to many features of the DIMMs and could been used to add further functionality such as changing the DRAM refresh rate, determining whether interleaving is enabled and retrieving other details about the configuration of the DIMMs. These improvements could be added in future revisions.

6.4 Software Development

Kmemtestd was developed using an iterative approach to both its design and implementation. Like the development of the Linux kernel, much of the design was responsive to the real-world behaviour of the module when running on hardware. Although the software requirements and feasibility study served as a solid basis for design, the complexity of the task resulted in the design and the techniques employed having to change at various stages.

The software was developed to be modular, so future CPU blades could be added simply to the existing code. At the time of writing this document maintenance for Kmemtestd has been transferred to another development team and new CPUs modules have already been added to the code for its most recent revision. The software code itself is easy to understand and follows the coding principles laid out in the kernel. However, code reuse is an area for future improvement. The read/write functions should be refactored into more generic functions perhaps taking the test patterns as a parameter. Also, the software has very little error handling whereas all functions which can fail should return a success or failure value to its caller and inform the user of the error.

Software testing during development was excessively time consuming. Each code change required the kernel to be built and then transferred to a USB pen drive, which was then inserted into the target hardware. The system was then booted and the module was manually loaded into the kernel and the results observed. This testing cycle lasted between 10 and 15 minutes. QEMU, a CPU emulator, may have been a better option but unfortunately as the target hardware was heavily customized, this was not possible.

Whilst the software was tested directly on the hardware, it proved difficult to verify the extent to which the memory subsystem was stress tested. Manually generating controlled memory errors is not trivial. Installing fully faulty DIMMs would result in the system not booting at all and short circuiting a connector on the DIMM produced unexpected kernel panics. By applying excessive heat with a thermal-air blower across the length of the DIMMs produced real memory errors which were successfully caught by Kmemtestd. The voltage across the CPU was also measured during test runs with Kmemtestd and the existing UTE software and similar values were recorded, suggesting that the memory subsystem was stress tested to a similar extent. Ideally the software verification would have been more thorough and detailed.

The user interface for Kmemtestd was implemented using a shell script. Whilst meeting the requirements for usability, the UI could be improved in future versions of the software. Adding a Linux command to the kernel for user interaction with Kmemtestd would have been a better approach and would have also allowed for documentation via `man` pages. Currently user documentation is provided in a Word document and a help option flag should be added to the script, so the user can see the available options. Log writing is a further area for improvement as the UI script currently appends the system log to a file. By implementing a device driver, more detailed and controlling logging could be achieved. However, this would increase complexity and impact maintainability.

Due to a lack of software engineering experience during the development process, a number of software tools which would have increased productivity were not utilised. Despite there being only one contributor to the code, a subversion system such as SVN or git, would have helped track the frequent code changes and have served as a useful reference tool in future debugging. Also, learning to use the kernel debugger would have helped in tracing the causes of the more obscure kernel panics. Unit testing the code would have helped in identifying coding errors and added verification for future additions.

A Linux kernel module has not been used previously to test memory due to the complexity of memory testing inside the kernel space. However, Kmemtestd proves that it is both possible and a practical method for implementation. Some discussion has taken place with the current maintainer of the software regarding releasing it to the open-source community for analysis and further development. A kernel module provides a great deal of flexibility as it can be compiled for any architecture supported by the kernel.

7 Conclusion

The goal of the project was to develop software to stress test the memory subsystem in x86 based hardware for NSN, which met their software requirements. Due to the implementation of Kmemtestd the memory subsystem NSN's CPU blades can be stress tested effectively and the results stored for further analysis. The software is modular and can be adapted easily to future processors. Compilation for future CPU architectures is provided via the kernel. Kmemtestd is relatively boot-to-test in nature and provides a simple user interface. Kmemtestd has been taken into use by hardware engineers and is currently being maintained and developed further by an embedded software team.

The Linux kernel memory management subsystem is vast and complex. In order to develop the software a considerable effort was placed in exploring this subsystem and kernel development itself, which consumed more time than was initially anticipated. Despite the challenges involved in Linux kernel development the project was both a rewarding and valuable learning experience. Working so closely with the hardware also provided insight into how memory works and how software utilises it, which in turn illustrated how to better optimise memory usage in future software projects.

The result of the project was a software implementation, Kmemtestd, which met the primary criteria defined in the software requirements. The implementation also delivered a user interface script and extensive documentation. The software has been taken into use successfully.

In conclusion, the project was successful and met the criteria set out at its inception. NSN now have a further software tool which can be updated and modified simply. The project was challenging but provided valuable insights into profession software development.

References

1 Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture [online]. Intel Corporation; November 2008.
URL:http://download.intel.com/design/processor/manuals/253665.pdf.
Accessed 20 January 2009.

2 Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System's Programming Guide, Part 1 [online]. Intel Corporation; November 2008.
URL: http://download.intel.com/design/processor/manuals/253665.pdf.
Accessed 20 January 2009.

3 Duarte G. Memory Translation and Segmentation [online]. August 2008.
URL: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation. Accessed 23 February 2009.

4 Drepper U. What Every Programmer Should Know About Memory. November 2007.
URL: http://www.akkadia.org/drepper/cpumemory.pdf.
Accessed 25 February 2009.

5 Ganssle J. Testing RAM in Embedded Systems. April 2007.
URL: http://www.ganssle.com/testingram.pdf. Accessed 22 January 2009.

6 Bovet DP, Cesati M. Understanding the Linux Kernel. 3rd ed. United States of America: O'Reilly, November 2005.

7 Corbet J, Rubini A, Kroah-Hartman G. Linux Device Drivers. 3rd ed. United States of America: O'Reilly, February 2005.

8 Gorman M. Understanding the Linux Virtual Memory Manager [online]. February 2004.
URL:http://ptgmedia.pearsoncmg.com/images/0131453483/downloads/gorman_book.pdf. Accessed 20 March 2009.