Multithreading Issues on Contemporary PowerPC Microprocessors

Hugh Blemings

A Masters Thesis.
Part of the course requirements of COMP6702,
Masters in eScience,
The Department of Computer Science
Australian National University

November 2006

Typeset in Palatino by TEX and LATEX 2ε.

Hugh Blemings
29 November 2006

This thesis is respectfully dedicated to Dr. Mark Jarnyk (1963-2006). A dear friend, a mentor in all things academic, scientific & technical, a connoisseur of tea and a fine hacker. Greatly missed.

Acknowledgements

I thank Dr Peter Strazdins, my project supervisor for his encouragement, support, guidance and input into this work. My thanks also to Dr Alistair Rendell, the convener of the COMP6702 course for his support and input along the way.

I am fortunate to work with a tremendous group of people at IBM. Thankyou to Ralph Christ, my manager at IBM, for his support in allowing me to undertake the Masters course. Just about everyone at OzLabs fielded a question from me in the last six months in relation to this work, thanks guys. My thanks also to the broader group of colleagues at The Linux Technology Centre, Systems and Technology Group and IBM Research for their support.

While personal acknowledgements are unconventional, I none the less thank my wife Lucy and daughter Rachael for their support throughout my studies. I'm looking forward to a bit more family time in 2007...

Abstract

Contemporary high performance microprocessors are moving beyond faster clocks and wider busses to meet the growing demands for computational power. Techniques such as multiple threads of execution and heterogeneous processing cores are becoming more mainstream bringing with them interesting challenges for operating system and application developers alike.

We sought to quantitatively assess the benefit or otherwise of Chip Multi-Processing (CMP) and Simultaneous Multi-Threading (SMT) using a number of low level and artificial benchmarks, and contrast these results with those for more conventional SMP.

In this Thesis we present the results of our research and experiments on the POWER5 microprocessor.

For the majority of codes enabling SMT was shown to increase performance by 20% or more. Conversely a small number of cases saw performance drop by up to 96%.

We also demonstrate the benefit of CMP for codes such as NAS parallel when benchmarked against an SMP equivalents, improvements of 2.5x being observed in some instances.

Results from measuring low level barrier and lock performance underscores the need for hierarchical algorithms to be refined to be aware of SMT, CMP and SMP characteristics.

Overall our investigations suggest that both SMT and CMP are of benefit, often considerable, but aren't without subtle quirks.

Contents

Acknowledgements **vii**

Abstract **ix**

1 Project Overview **1**
1.1 Basis of this Work . . . 1
1.2 Processor Architectures Examined . . . 1
1.3 Benchmarks . . . 2
1.4 Organisation . . . 2

2 Benchmarks **3**
2.1 STREAM . . . 3
2.1.1 Background . . . 3
2.1.2 Benchmark Internals . . . 3
2.1.2.1 Copy Kernel . . . 4
2.1.2.2 Scale Kernel . . . 4
2.1.2.3 Add Kernel . . . 4
2.1.2.4 Triad Kernel . . . 5
2.1.3 Stream bandwidth calculations . . . 5
2.1.4 Stream on MP systems . . . 5
2.2 NAS Parallel . . . 5
2.3 Other Benchmarks - Apex-Map and perflab . . . 5

3 Processor Architectures **7**
3.1 Introduction . . . 7
3.1.1 Multithreading and Multi-Processing - A timeline . . . 7
3.1.2 Chip Multi-Processing - An Overview . . . 8
3.1.3 Multithreading - An Overview . . . 8
3.2 POWER5 . . . 9
3.2.1 Background . . . 9
3.2.2 Threading Models . . . 10
3.2.3 Instruction Fetching on POWER5 . . . 11
3.2.4 Thread Priority . . . 11
3.2.4.1 Thread Priority in Linux . . . 11
3.2.5 Dynamic Resource Balancing . . . 12
3.2.6 Single-threaded Operation . . . 12
3.2.7 Memory Subsystem on POWER5 . . . 12

3.2.7.1 Cache on POWER5 . . . 12
3.2.7.2 Main Memory on POWER5 . . . 14

4 Experimental Setup **15**
4.1 Systems Used . . . 15
4.2 Processor & Memory Affinity under Linux . . . 15
4.2.1 Processor Affinity . . . 16
4.2.2 Setting Processor Affinity . . . 16
4.2.3 Memory Affinity . . . 17
4.3 Threads, Cores, Dies, Oh My! . . . 17
4.4 Comparing SMT and ST - "SMT vs ST pairs" . . . 19
4.5 Operating System . . . 19
4.6 Compiler . . . 19
4.7 Automation . . . 19
4.8 Thread to CPU affinity . . . 20
4.9 STREAM Benchmark . . . 20
4.9.1 Baseline Setup . . . 20
4.9.2 Details of changes . . . 20
4.9.3 Main Memory & Cache Runs . . . 21
4.10 Barrier & Lock Benchmarks . . . 21
4.10.1 Time Keeping . . . 21
4.10.2 Barrier Codes . . . 22
4.10.3 Round-Robin Lock Codes . . . 23
4.10.4 Overall Benchmark . . . 23
4.11 NAS Parallel Benchmarks . . . 23
4.11.1 Basic Setup . . . 23

5 A Performance Model for Round-Robin Locking **29**
5.1 Background . . . 29
5.2 Description of model . . . 29
5.3 Putting into practice . . . 30
5.4 Atomic update overheads . . . 30

6 Results & Discussion **33**
6.1 STREAM . . . 33
6.1.1 Initial trials . . . 33
6.1.2 Main Memory Bandwidth Measurements . . . 34
6.1.2.1 Main Memory Bandwidth - non-SMT cases . . . 34
6.1.2.2 Main Memory Bandwidth - SMT cases . . . 35
6.1.2.3 Main Memory Bandwidth Per Thread . . . 36
6.1.3 Cache Bandwidth Measurements . . . 37
6.1.3.1 Contrasting Cache Bandwidth for different thread combinations . . . 38
6.1.4 Comparison of Copy with Scale, Add & Triad codes . . . 39

6.1.4.1 Specifics of non-SMT cases . . . 39
6.1.4.2 Specifics of SMT enabled cases . . . 40
6.1.5 Conclusions about STREAM results . . . 41
6.1.5.1 A word from McCalpin . . . 41
6.1.5.2 SMT mostly harmless ? . . . 42
6.2 Barrier & Locking Measurements . . . 43
6.2.1 Barrier Results . . . 43
6.2.1.1 pthread results . . . 44
6.2.1.2 atomic results . . . 44
6.2.1.3 Influence of SMT . . . 44
6.2.1.4 Conclusions for Barriers . . . 44
6.2.2 Round-Robin Locking Results . . . 45
6.2.2.1 Influence of SMT . . . 46
6.2.3 Summary of SMT effects on Barrier and Locking trials . . . 46
6.2.4 Comparison of results against model . . . 47
6.2.5 Additional Barrier and Locking Results . . . 47
6.3 NAS Parallel Benchmarks . . . 48
6.3.1 Starting with an Outlier . . . 48
6.3.2 Analysis of non-SMT cases . . . 49
6.3.3 Analysis of SMT cases . . . 50
6.3.4 Super Linear Speedup . . . 50

7 Conclusions & Future Work **51**
7.1 Conclusions . . . 51
7.1.1 Is SMT worth it ? . . . 51
7.1.2 Suitability of selected benchmarks . . . 51
7.1.3 A Case for Hierarchical Barriers & Locks . . . 52
7.1.4 Disabling CPUs . . . 52
7.1.5 Multicore is here to stay . . . 52
7.2 Future Work . . . 53
7.2.1 Recording Amount of Funny . . . 53
7.2.2 Memory Bandwidth vs Locality Measurements . . . 54
7.2.3 Hardware Assisted Barrier Techniques . . . 54
7.2.4 Physical address chasing Kprobes . . . 54

A Additional Results **57**
A.1 Additional Barrier/Lock Results . . . 57

B Other articles **59**

C Other Benchmarks **61**
C.1 Apex-map . . . 61
C.1.1 Background . . . 61
C.1.2 Benchmark Internals . . . 61
C.2 perflab . . . 62

C.2.1 Background & Benchmark Operation 62
C.2.2 Perflab in Our Work 62

Bibliography **63**

Chapter 1

Project Overview

Simultaneous Multi Threading (SMT) and Chip Multi-Processing (CMP) have become commonplace in the last few years. The Australian National University's Department of Computer Science (DCS) had recently taken delivery of a pair of OpenPOWER 720 systems. The machines came courtesy of the Canberra based "OzLabs" group within IBM's Linux Technology Centre. The desire to make use of the latter to better understand the former was formalised in a proposal for a Masters literature review and subsequent research project.

In this thesis we start with relevant findings in the earlier literature review [Blemings 2006]. From here we explore the POWER5 processor architecture in some detail, conduct a series of experiments to quantify memory bandwidth, barrier/locking primitives and overall system performance using NAS Parallel.

Results are provided and discussed, particularly the contrast of SMP, SMT and CMP. We conclude with some thoughts on potential future work and lessons learned.

1.1 Basis of this Work

Simply put, we took two of the benchmarks identified in the Literature Review, added one of our own and proceeded to run this on the OpenPOWER720 system in a variety of CPU configurations.

From this we were able to determine the differing performance traits of SMP, CMP and SMT, discovering a couple of surprises along the way.

1.2 Processor Architectures Examined

This thesis focuses on IBM's POWER5 architecture which we describe in detail in Section 3.2.

Both the Sun UltraSPARC T1 (aka Niagara) and Sony/Toshiba/IBM Cell Broadband Engine (CBE - aka the Playstation3 chip) were discussed in our earlier literature review [Blemings 2006] and represent two other approaches to multiprocessing.

1.3 Benchmarks

Based on our earlier work we describe two benchmarks in Chapter 2, STREAM §2.1 and NAS Parallel §2.2.

In §4.10 we describe some code written to allow us to quantify low level barrier and locking performance.

1.4 Organisation

This thesis is divided into seven chapters. This chapter provides an overview of the work and it's motivations. Chapter 2 introduces the STREAM and NAS benchmarks and makes note of two other benchmarks that, while not used in this work, are relevant to the general area of study. Chapter 3 discusses microprocessor architectures in particular IBM's POWER5 design which is the centre piece of this work. Chapter 4 details the experimental setup used, including code specifically written for the project and modifications to existing codes. Chapter 5 proposes a simple model for the performance of round-robin locking algorithms. Chapter 6 details our results and Chapter 7 provides some conclusions and direction for future work.

Chapter 2

Benchmarks

In this chapter we describe two benchmarks that are used in our experiments and give pointers to two others that warrant consideration in future work.

2.1 STREAM

2.1.1 Background

The STREAM benchmark is a tool that allows memory bandwidth to be measured. Aimed at high performance systems or supercomputers, it is the work of John D. McCalpin, then of University of Delaware, now at AMD by way of SGI and IBM. To keep STREAM vendor independent it is hosted and overseen by two academics at the University of Virginia.

> The intent of STREAM is not to suggest that "real" applications have no data re-use, but rather to decouple the measurement of the memory subsystem from the hypothetical "peak" performance of the machine. In this respect the test is quite complementary to the LINPACK benchmark test, which is typically optimised to the point that a very large fraction of full speed is obtained on modern machines, independent of the performance of their memory systems. [McCalpin 1995]

On uniprocessor machines, the benchmark can be trivially compiled and run. For use on multiprocessor systems, STREAM is designed to make use of OpenMP or MPI libraries for SMP and cluster machines respectively. As we cover in Section 4.9, we ended up modifying STREAM quite extensively for our work.

Data is available from the University of Virginia web site for a large range of machines. We make use of the data for the OpenPOWER 720 to validate our results.

2.1.2 Benchmark Internals

STREAM does some initial setup to attempt to determine timer accuracy on the host system and work out sane values for the amount of memory to use.

Thus equipped, it repeats four simple kernels ten times, each kernel working on several tens megabytes of data assuming default settings. This data set size ensures that main memory must be used - the benchmark cannot "fit" purely within cache.

The four kernels are Copy, Scale, Add, and Triad. McCalpin describes them thus;

> Each of the four tests adds independent information to the results:
>
> - "Copy" measures transfer rates in the absence of arithmetic.
> - "Scale" adds a simple arithmetic operation.
> - "Sum" adds a third operand to allow multiple load/store ports on vector machines to be tested.
> - "Triad" allows chained/overlapped/fused multiply/add operations.

We now examine each kernel. `N` is the size of the arrays (2,000,000 by default), `a[], b[], c[]` are the three pre-allocated arrays of `doubles` which are used in various ways by each kernel. `scalar` is a `double` set to 3 before the kernels are run, this value ensuring the compiler will not optimise the multiplication to bitwise shifts.

2.1.2.1 Copy Kernel

The Copy kernel has one read and one write operation per iteration for a total of two memory operations per iteration.

```
for (j=0; j<N; j++)
    c[j] = a[j];
```

2.1.2.2 Scale Kernel

The Scale kernel is effectively one read and one write operation per iteration. As McCalpin alludes to, on modern machines we can assume that `scalar` will be held in a register and that the execution time for the multiply operation will be insignificant compared to the main memory accesses. Like Copy this gives a total of two memory operations per iteration.

```
for (j=0; j<N; j++)
    b[j] = scalar * c[j];
```

2.1.2.3 Add Kernel

The Add kernel is two reads, one write and one arithmetic operation per loop iteration - a total of three memory operations.

```
for (j=0; j<N; j++)
    c[j] = a[j] + b[j];
```

2.1.2.4 Triad Kernel

The Triad kernel is effectively two reads and one write, we again assume that the multiply operation will be insignificant relative to the memory accesses and that `scalar` will be held in a register. Like Add this gives three memory operations per cycle.

```
for (j=0; j<N; j++)
    a[j] = b[j] + scalar * c[j];
```

2.1.3 Stream bandwidth calculations

As each benchmark is run the time taken for the specified number of trials is measured and from this the number of iterations per second calculated. This is then multiplied by the number of memory operations per iteration (two or three) and the size of the data structures in use to arrive at the final bandwidth figure.

2.1.4 Stream on MP systems

Stream is designed to make use of OpenMP if it is available and parallelise the kernels appropriately through use of `#pragma omp parallel for`.

If OpenMP is unavailable, the author of stream suggests running one instance per CPU as separate processes with large values of `N` so they keep going for some time. The final instance is then run "normally" and the results manually calculated on the basis of these figures multiplied by the number of instances running. [McCalpin 2006]

As is discussed in Section 4.9 we followed this approach to begin with and then moved to customise STREAM to provide better control over memory and processor affinity.

2.2 NAS Parallel

The NAS Parallel Benchmarks (NPB) are a set of programs that assist in evaluation the performance of parallel supercomputers. The work of NASA, NPB 1.0 was developed in the late 1980's and published shortly thereafter [Bailey et al. 1991] Derived from CFD applications NPB 1.0 consists of five kernels and three pseudo-applications.

The 1991 work is updated in [Bailey et al. 1994] and [Bailey et al. 1995] sees substantial changes to NPB with the release of NPB 2.0.

The NPB have been investigated within DCS before on several occasions including work done by Jean [Jean 2005] and the SPARC-Sulima team[1]

Jean provides a concise description of NPB in his 2005 paper which we commend to the reader for further information.

2.3 Other Benchmarks - Apex-Map and perflab

Two other benchmarks were discussed in the literature review [Blemings 2006] which preceded our work this semester: Apex-Map, a memory benchmark, and perflab, a set of codes for evaluating locking algorithms.

[1] `http://cs.anu.edu.au/Peter.Strazdins/postgrad/completed/NPBArchEvals.html`. This work sought to explore the effects of architectural changes on system performance when running these benchmarks.

We ran Apex-map in a single thread configuration to get some experience with it but ultimately, largely due to time constraints, elected to stick with STREAM.

perflab was not used as we lacked time to fully understand its operation and elected to concentrate on lower level locking codes.

While in we did not make use of either they both have the potential to be useful in future work in this area and so we have included our original observations about them, slightly updated, in Appendix C.1 (Apex-map) and Appendix C.2 (perflab).

Chapter 3

Processor Architectures

3.1 Introduction

In the literature review that preceded this work we examined three process architectures in varying degrees of detail: IBM's POWER5, The Sony/Toshiba/IBM Cell Broadband Engine (CBE) and Sun's UltraSPARC T1 (aka Niagara). All three provide multiple threads on a single core but beyond that they are quite different devices.

The focus on our research was the IBM POWER5 and accordingly it is this architecture we detail in the following sections. First however a general perspective on Multithreading and Multi-processing.

3.1.1 Multithreading and Multi-Processing - A timeline

Symmetric Multi-Threading (SMT) and Chip Multi-Processing (CMP) are not new concepts. In 1995 Tullsen, Eggers and Levy made a case for SMT [Tullsen et al. 1995] and 1996 saw Olukotun et al. advancing an argument for CMP [Olukotun et al. 1996]. The following year Hammond, Nayfeh and Olukotun contributed to a Theme Feature on Billion Transistor Microprocessors in IEEE Computer, once again putting the case for CMP [Hammond et al. 1997].

All three works have a common theme - that conventional uniprocessor models will only get us so far into the future before thread level parallelism of some form or other will be required. They argue that in order to meet the growing demands for computational power, it will no longer be sufficient to rely on faster clock rates and increasingly aggressive out of order execution and pipelining.

While on the basis of various vendors marketing, one might assume otherwise, implementations of SMT and/or CMP are not all that new either. In 2000 IBM's RS64 IV processors provided coarse grained SMT capability [Borkenhagen et al. 2000] and CMP was a centrepiece of the IBM POWER4 line in 2001 [Tendler et al. 2002]. SMT also appeared as in Intel Xeon (HT)[1] offerings in late 2000.[2]

That said, it could be fairly argued that SMT only reached mainstream consciousness with the release of the Intel Pentium4 (HT) microprocessor circa 2004. While CMP has been commonplace in IBM's POWER CPUs for some time now, being server oriented parts they are perhaps rather less visible than Intel's offerings.

In early 2006 we find it starting to sink in within industry that multiple threads of execution, be they SMT, CMP or both is indeed the direction of the future and that accordingly there

[1] Pentium4 CPUs implemented what Intel called Hyper-Threading Technology - using the nomenclature of this paper, HT is SMT as we define below.

[2] Implementations may go back further than this to the mid 1960's - we were reminded that the I/O processors of the CDC6600 system were multithreaded.

is a need to adopt threading in their applications.

In their work on CMP, the authors of [Spracklen and Abraham 2005] present an interesting mental picture: "In our taxonomy, SMT and CMP are two extremes of a continuum characterised by varying degrees of sharing of on-chip resources among the strands." As we shall see, many contemporary microprocessors all but span this continuum on a single die.

3.1.2 Chip Multi-Processing - An Overview

CMP is conceptually straightforward. Take two or more normal processor cores, put a standard SMP interconnect between them, possibly share some cache, shared or separate memory controllers and there's your design. This simplicity is of course attractive to designers as it reduces design complexity and test overhead. There are some variations on this basic approach, for example having a FPU shared between cores as is seen on the UltraSPARC T1, but the essence of the CMP approach remains.

CMP is also attractive from a software standpoint - they appear as a conventional SMP. Depending on the design there may be NUMA effects, the details of the particular design will dictate to what extent this is the case.

3.1.3 Multithreading - An Overview

Multithreading is a more subtle approach to achieving multiple threads of execution. Several different approaches are used, defined in [Sinharoy et al. 2005] as Coarse-grain Multithreading, Fine-grain Multithreading and Simultaneous Multithreading.

The three approaches differ in the detail of how threads are run. For example switching threads every cycle versus switching due to a long latency event and/or whether different execution units on the core allow instructions from different threads to run simultaneously or not.

Armed with these initial remarks, we quote from [Sinharoy et al. 2005, pp 506]

> In coarse-grain multithreading, only one thread executes at any given instant in time. When a thread encounters a long-latency event, such as a cache miss, the hardware swaps in a second thread to use the machine resources rather than letting it idle. By allowing other work to use what otherwise would have been idle cycles, overall system throughput is increased. To conserve chip area, both threads share many of the system resources, such as architected registers. Hence, to swap program control from one thread to another requires several cycles. IBM introduced coarse-grain threading on the IBM pSeries S85 [Borkenhagen et al. 2000].
>
> Fine-grain multithreading switches between threads each cycle. In this class of machines [Alverson et al. 1990], a different thread is executed in a round-robin fashion. As in coarse-grain multithreading, the architected states of multiple threads are all maintained in the processor. Fine-grain multithreading allows overlap of short pipeline latencies by letting another thread fill in execution gaps that would otherwise exist. With a larger number of threads, longer latencies can be successfully overlapped. For long-latency events in a single thread, if the number of threads is less than the number of latency cycles, there will be empty execution cycles for that thread. To accommodate this design, hardware facilities are duplicated. When a thread encounters a long-latency event, its cycles remain unused.

Simultaneous multithreading maintains the architected states of multiple threads. This type of multithreading is distinguished by having the ability to schedule instructions from all threads concurrently [Tullsen et al. 1995]. On any given cycle, instructions from one or more threads may be executing on different execution units. With SMT, the system adjusts dynamically to the environment, allowing instructions to execute from each thread if possible while allowing instructions from one thread to utilise all of the execution units if the other thread(s) cannot make use of them. This allows the system to dynamically adjust to the environment. The POWER5 system implements two threads per processor core. Both threads share execution units if both have work to do. If one thread is waiting for a long-latency event, the other thread can achieve a greater share of execution unit time.

3.2 POWER5

The IBM POWER5 Microprocessor is a two-way simultaneous multithreaded (SMT) dual core (CMP) chip. Each die includes;

- 64k, two way set associative L1 Instruction cache for each core
- 32k, four way set associative L1 Data for each core
- 1.875MB of 10 way set associative L2 cache shared between the two cores
- Directory and interface logic for the external L3 cache (36MB)
- DRAM controller
- Fabric interconnect
- I/O interconnect
- Power management, JTAG interfaces etc.
- Clock frequencies $\simeq$1.65GHz and above (system dependent)

3.2.1 Background

POWER5 is a descendant from the earlier POWER4 and POWER3 architectures which are nicely detailed in [Tendler et al. 2002] and [Papermaster et al. 1998] respectively.[3]

POWER5 is designed to allow effective scalability up to 64 physical processors (128 threads) at one extreme yet be a good fit for one and two way systems. Physically this achieved by having three building blocks.

- A Multi Chip Module (MCM) sees four POWER5 and four cache chips brought together on a single ceramic substrate 95mm on a side. Thus an MCM has eight complete cores (16 threads). MCMs are most commonly used in larger SMP configurations.
- A Quad Chip Module (QCM) sees two POWER5 CPUs and two cache chips on a single ceramic package for a total of four cores/eight threads. QCMs are used in more recent single rack unit designs.

[3] If a longer historical view is sought, a brief but accurate history of POWER, PowerPC and associated technologies appears in "POWER to the people" [Mikes 2004]

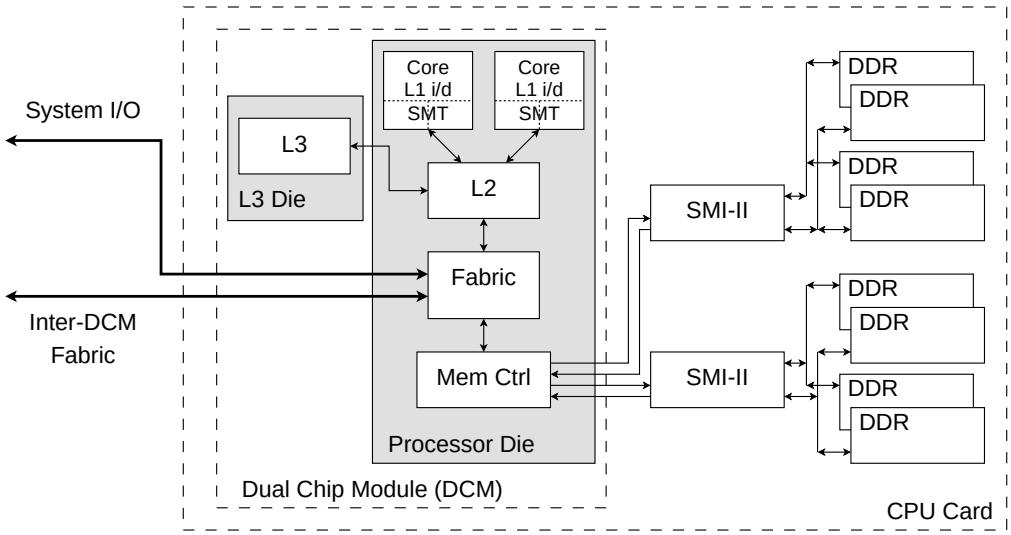

Figure 3.1: POWER5 Block Diagram. Two such cards are used in the 4-Way configuration of the OpenPower 720

- The Dual Chip Module (DCM) is shown with it's associated memory sub-system in Figure 3.1). The DCM comprises a single CPU and cache chip and provides a two core/four thread building block[4]. Two DCMs are used in the OpenPower 720.

The POWER5 is a complex part and a working knowledge will be important to our understanding.[5] Hence we will cover it's internals in more detail in the following sections.

3.2.2 Threading Models

Sections 3.1.3 and 3.1.2 provide general descriptions of threading models. Using those definitions POWER5 is an CMP design where each core also supports SMT. POWER5 is also able to run in single-threaded (ST) mode - SMT disabled - as well as allowing one of the two cores on a chip to be disabled - CMP disabled.

From a software standpoint this allows a single chip to look like;

- A Uni-processor - a single thread of execution - no SMT or CMP
- A 2-Way SMP - two threads of execution, each with an entire core to itself - CMP
- A 2-Way SMP - two threads of execution, both threads residing on a single core - SMT
- A 4-Way SMP - four threads of execution, two threads per core, two cores operating - CMP with SMT

These different modes of operation have some interesting side effects in relation to memory bandwidth and overall computational capability and are a focus of our work.

[4]It is this combination of CPU and cache die that gives the "Dual" in DCM

[5][Kalla et al. 2004] and [Sinharoy et al. 2005] provide a detailed description of POWER5 written by key members of the POWER5 design team.

3.2.3 Instruction Fetching on POWER5

It is useful for us to understand the details of instructions fetch/decode on the microprocessor, again we quote from [Sinharoy et al. 2005, pp 508];

> In SMT mode, two separate program counters are used, one for each thread. Instruction fetches alternate between the two threads. Similarly, branch prediction alternates between threads. In Single Threaded (ST) mode, only one program counter is used, and instructions can be fetched for that thread every cycle.
>
> [...]
>
> Up to eight instructions can be fetched from the instruction cache (IC pipeline stage) every cycle.

Recall that this ability to run separate threads of execution simultaneously on different execution units is what makes POWER5 SMT rather than (say) Fine-grained Multithreaded.

IBM have an article available on instruction fetching and CPI analysis available on their Developer Works website [6] which treats this topic further.

3.2.4 Thread Priority

In a multithreaded system, one thread may use a significant amount of system resources, potentially blocking other threads. [Sinharoy et al. 2005, pp 511] Accordingly the POWER5 design provides various mechanisms to optimise resource usage and balance between threads.

Each thread has eight software controlled priority levels, some levels can only be accessed when the processor is in privileged mode, the balance can be set from user space. When not running, a thread is at Level 0, Levels 1 (the lowest) through to 7 (the highest) apply when the thread is executing. While set by software, these priorities are enforced by hardware.

Electrical power is also conserved automatically by the POWER5 core. When both threads are at Level 1 instruction decode is throttled by hardware.

3.2.4.1 Thread Priority in Linux

The Linux kernel makes use of thread priority, adjusting it up or down slightly when entering certain sections of code. In the kernel code the more generic abbreviation HMT (Hardware Multi Threading) is used. In spin loops for example, `HMT_low()` is used. The kernel idle loop will call to `HMT_very_low()` while doing nothing and `HMT_medium()` during normal operation. Linux does not currently force a priority above `HMT_medium()` - recall these priorities are relative so the core still runs at full speed when it is able.

The macros defined in `./include/asm-powerpc/processor.h` illustrate how special instances of the `or` instruction[7] are used to set thread priority

```
/* Macros for adjusting thread priority (hardware multi-threading) */
#define HMT_very_low()    asm volatile("or 31,31,31 # very low priority")
#define HMT_low()         asm volatile("or 1,1,1    # low priority")
#define HMT_medium_low()  asm volatile("or 6,6,6    # medium low priority")
#define HMT_medium()      asm volatile("or 2,2,2    # medium priority")
#define HMT_medium_high() asm volatile("or 5,5,5    # medium high priority")
#define HMT_high()        asm volatile("or 3,3,3    # high priority")
```

[6] `http://www-128.ibm.com/developerworks/power/library/pa-cpipower1`

[7] as used this is effectively a `nop`

3.2.5 Dynamic Resource Balancing

"Depending on the situation, the POWER5 microprocessor employs one of three mechanisms to throttle threads to perform dynamic resource balancing."[Sinharoy et al. 2005, pp 511]

Firstly, in addition to the software control over priority level mentioned above, the POWER5 core can temporarily adjust the level of a thread to throttle its execution. Such a decision is based on instruction completion statistics gathered from the instruction completion logic on the core.

Secondly, the L2 miss rate for each thread is monitored and when it rises above a specified value, instruction decode for that thread is temporarily disabled.

The final mechanism is employed when an instruction is decoded which will take a long time to complete. For example, sync instructions or an instruction that will cause a stall due to resource contention. Under such situations the core will flush all of the instructions for that thread and inhibit further decodes until the instruction completes.

3.2.6 Single-threaded Operation

While designed to be used in an SMT configuration, the POWER5 cores can be placed in a ST mode. [Sinharoy et al. 2005, pp 514] provides a cogent description of this feature of POWER5;

> Not all applications benefit from SMT. Applications whose performance is execution-unit-limited or which are consuming all of the POWER5 chip memory bandwidth will not see additional performance by having two such threads executing on the same processor. For this reason, POWER5 systems support single-threaded execution mode. In single-threaded mode, a POWER5 system makes all of the rename registers, issue queues, the Load Reorder Queue, and the Store Reorder Queue available to the active thread. This gives the single active thread more rename registers to use, allowing it to achieve higher performance levels than a POWER4 system at equivalent frequencies. Software can dynamically change a processor between single-threaded and SMT modes.

3.2.7 Memory Subsystem on POWER5

The memory subsystem plays a significant role in determining the computers performance characteristics.

In examining the POWER5 memory system, it is useful to contrast it with that of POWER4 (Figure 3.2) that preceded it. Table 3.1 shows this evolution and is drawn from information in [Papermaster et al. 1998], [Tendler et al. 2002], [Sinharoy et al. 2005] and [DeMone 2004].

In examining the information in Table 3.1 and Figures 3.1 and 3.2 it can be seen that in POWER5 cache has been moved closer to the core. Further main memory is now accessed through a separate channel to L3, these two changes along with other tweaks have brought a significant reduction in latency and increase in bandwidth.

We now examine the memory subsystem in a more detail, the reader may wish to refer to Figure 3.1 as we proceed. Note that where bandwidths or latencies are quoted as absolute figures they assume a 1.65GHz core frequency.

3.2.7.1 Cache on POWER5

There are three levels of cache on a POWER5 system, L1, L2 and L3. On systems with multiple DCMs, cross fabric reads of cache are possible which are referred to as L2.75 and L3.75 accesses.

	POWER4	**POWER5**
L1 ICache	64kB Direct Mapped	64kB 2-way associative
L1 DCache	32kB 2-way associative	32kB 4-way associative
L2 Cache	1.44MB 8-way associative 12 cycle latency	1.92MB 10-way associative 13 cycle latency
L3 Cache	32MB 8-way associative 123 cycle latency Clocked at 1/3 CPU frequency	36MB 12-way associative 87 cycle latency Clocked at 1/2 CPU frequency
Memory	$\simeq$ 4GB per die 351 cycle latency	$\simeq$ 16GB per die 220 cycle latency

Table 3.1: POWER4 / POWER5 Memory characteristics

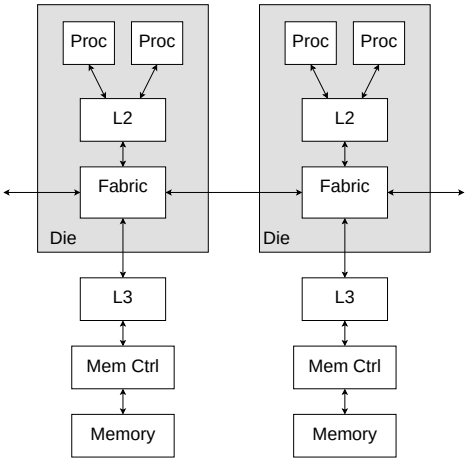

Figure 3.2: POWER4 Block Diagram

Each core has its own L1 cache, hence there are a pair per DCM. They operate at the processor frequency and have hardware coherency. The L1 Instruction cache is 32kB in size, 128 byte lines with two way associativity and a FIFO replacement policy. The L1 Data cache is 32kB in size, 128 byte lines, four way set associative with LRU, store through. Latency is two cycles, bandwidth $\simeq$ 26GB/s.

There is a single L2 cache per DCM of 1.92MB comprised of three slices of 640kB each. L2 is 10 way set-associative with an LRU replacement policy and 128B line size.

Each core on the DCM has a separate port to each of the three L2 slices with a latency of 13 cycles and a theoretical peak bandwidth of $\simeq$ 52GB/s. Additionally each slice has independent access to the fabric, other cores in the system can read/write data to the L2 directly. These L2.75 operations have a latency of around 140 cycles and a peak theoretical bandwidth of $\simeq$ 39GB/s L2 to Fabric and $\simeq$ 79GB/s Fabric to L2.

Each DCM has it's own L3 comprised of three slices of 12MB each for a total of 36MB. The L3 is 12 way set-associative, LRU replacement and has a 256 byte line size. The L3 acts as a victim cache of L2 and has one read and one write port to each L2 slice and a port to the fabric. Latency to the local CPUs is some 87 cycles, 140 cycles for an L3.75 access from another DCM. Peak bandwidth is $\simeq$ 13GB/s read and write.

3.2.7.2 Main Memory on POWER5

Main memory on POWER5 systems is DDR or DDR2 based depending on the model in question. Here we confine ourselves to the arrangement on the OpenPOWER 720.

Each processor card contains half the system memory, thus a 32GB system has 16GB attached to each DCM. Referring to Figure 3.1 it can be seen that each card has eight 266MHz DDR ECC memory modules installed in pairs, two pairs per memory controller.

The SMI-II memory controllers straddle two asynchronous clock domains, the processors' on one side, DDR memory on the other. The interface to the core itself has an eight byte wide read port and a two byte wide write port and operates at half the core frequency. The interface on the memory side is eight bytes wide and operates at the DRAM frequency[8]

On the OpenPOWER720 the SMI-II's are configured to share the physical address space, thus sequential reads and writes are interleaved across the available DIMMs yielding a theoretical per DCM bandwidth of 10.41GB/s.

The memory controllers also perform background operations related to ECC and reliability functions on the system.

[8] the internal logic in this side of the SMI-II operates at twice the DRAM clock

Chapter 4

Experimental Setup

4.1 Systems Used

Except where noted otherwise, all results discussed in the following sections were produced on an IBM OpenPOWER 720 system. While DCS have two such systems of their own our work primarily utilised a system made available at IBM's OzLabs facility. This avoided inconvenience to other students using the DCS machines as well as providing a relatively known "background" workload for our measurements.

The machine in question *jago* has two CPU cards as shown in Figure 3.1, each with a 1.65GHz CPU and 16GB of RAM. This provides a total of four POWER5 CPUs and 32GB of RAM in a single system image[1].

For our barrier/locking experiments a larger machine was used to give a greater range of processor options. This machine, *fandango2*[2], is an eight POWER5 processor IBM pSeries 570 with 16GB RAM.

Architecturally the two systems are all but identical, built as they are from DCM building blocks as shown in Figure 3.1.

4.2 Processor & Memory Affinity under Linux

Linux has well thought out support for both processor and memory affinity both of which we rely on in our work.

SMP:	DCM1				DCM0			
CMT:	Core1		Core0		Core1		Core0	
SMT:	SMT1	SMT0	SMT1	SMT0	SMT1	SMT0	SMT1	SMT0
Bitmask:	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
CPU ID:	7	6	5	4	3	2	1	0

Table 4.1: Layout of CPU bitmask for four way OpenPOWER720

[1]These systems support *partitioning* into multiple system images, something we did not make use of in our work

[2]It is an OzLabs tradition to name PowerPC machines with names ending in "go"

4.2.1 Processor Affinity

From the user or application standpoint Linux takes a simple approach to dealing with different combinations of SMT, CMP and SMP - everything is considered a CPU. Internally of course the scheduler is aware of the subtleties of both SMT and CMP and endeavours to assign work accordingly.

Processors are numbered based on an underlying bitmask structure, that is to say a system can have CPU0, CPU1, CPU4 and CPU5. This bitmask gives some recognition to the underlying layout of CPU cards, dies, cpus and threads. In the case of the OpenPOWER720 however it is pretty straightforward as seen in Table 4.1.

4.2.2 Setting Processor Affinity

There are two approaches to setting processor affinity under Linux that we used in our work.

The first is to use the `taskset` utility from the command line. Part of the sched-utils[3] package, it allows the user to run a command and limit it to a particular set of CPUs. Thus on *jago*, running `taskset 0x55 foo` would invoke the command `foo` and confine it to using the SMT0 threads on each of the four cores.

While `taskset` is useful for quick tests it doesn't implicitly bind a thread of execution within the target application to a particular CPU. The scheduler or internal actions of the called application could cause a thread to migrate to different CPUs within the mask.

The second approach, the use of implicit calls to the `setaffiniy()` and `getaffinity()` functions, mitigates this problem by tying the calling thread to the CPU specified. For our work the procedure in Figure 4.1was used.

```
        /* Master Thread */
        for each thread {
                        /* Create and start threads */
                create_thread(thread_id, cpu_mask);
        }
/* Now wait for worker threads to finish */
join();


        /* Within each thread */
        create_thread(thread_it, cpu_mask) {

                        /* Bind thread to required CPU, quit on failure */
                setaffinity(cpu_mask);

                        /* Allocate and touch memory to ensure affinity */
                allocate_memory();
                touch_memory();

                        /* Run codes */
                do_work();
        }
```

Figure 4.1: Pseudo code for setting processor affinity

[3] http://freshmeat.net/projects/sched-utils

4.2.3 Memory Affinity

By default memory affinity is handled automatically - memory requested by a thread of execution will be allocated from the node closest the CPU on which it is running when the page is first touched. In recent kernels[4] this can be over-ridden but the interfaces to handle this, particularly to user space, are still somewhat in flux.

4.3 Threads, Cores, Dies, Oh My!

As discussed in §1.1 our work focussed on investigating the different combinations of SMP, CMP and SMT. In practice this meant we had one combination of SMP/CMP/SMT when utilising eight processor threads, three combinations of SMP/CMP/SMT when utilising four threads, three when utilising two threads and one for a single processor thread of execution.

We refer to a single core as being either in SMT mode (two threads) or non-SMT mode (one thread). This latter case we will refer to as ST for Single Threaded mode in most cases for grammatical simplicity.

We arrived at the following naming scheme to (hopefully!) make the configuration in use clear. The reader may wish to refer back to Section 3.2.1 and the diagram at Figure 4.2 as they proceed through the following.

The four pieces of information we needed to convey were;

1. Total number of threads - denoted **t** - for the OpenPOWER 720 this is between one and eight
2. Number of SMT threads per CPU - denoted **s** - either one (SMT disabled - ST) or two (SMT enabled) for POWER5
3. Number of CPU cores in use for each die - denoted **c** - either one or two for POWER5
4. Number of dies/DCMs in use - denoted **d** - for the OpenPOWER 720 this is also one or two

Thus for the OpenPOWER 720 we arrived at eight different combinations;

- **t8-s2-c2-d2** - Eight CPU threads active using both DCMs, both CPUs within each DCM enabled and SMT enabled on all four CPUs. This is essentially the "full" configuration for the OpenPOWER 720
- **t4-s1-c2-d2** - Four CPU threads active using both DCMs, both CPUs within each DCM in use, SMT disabled on both CPUs (ST). This is nominally the most balanced four thread configuration.
- **t4-s2-c1-d2** - Four CPU threads active using both DCMs, only one CPU within each DCM in use, SMT enabled on each active CPUs (ST). This is a less balanced four thread configuration than the one above.
- **t4-s2-c2-d1** - Four CPU threads active using one DCM, both CPUs within that DCM enabled, SMT enabled on both CPUs. This is the "pathological" four thread configuration - the threads of one entire DCM remains idle.
- **t2-s1-c1-d2** - Two CPU threads active utilising two DCMs, only one CPUs within each DCM enabled and SMT disabled (ST). This is most balanced two thread configuration.

[4] linux-2.6.17 or so onwards

- **t2-s1-c2-d1** - Two CPU threads active utilising one DCM, both CPUs within that DCM enabled but SMT disabled. This configuration leaves the second DCM largely unused.
- **t2-s2-c1-d1** - Two CPU threads active utilising one DCM, one CPU within the DCM enabled, SMT enabled. This configuration leaves both the second CPU on the active DCM unused as well as the the second DCM largely unused.
- **t1-s1-c1-d1** - One CPU thread active utilising one DCM, one CPU within that DCM enabled, SMT disabled. This configuration leaves both the second CPU on the active DCM unused as well as the the second DCM largely unused.

Figure 4.2 illustrates these different combinations.

The naming scheme assumes that there is no difference between cores and threads on a particular CPU. That is to say that Core 0 and Core 1 will perform identically if run in isolation and similarly that SMT0 and SMT1 within a core will perform identically if run with the other SMT thread inactive.

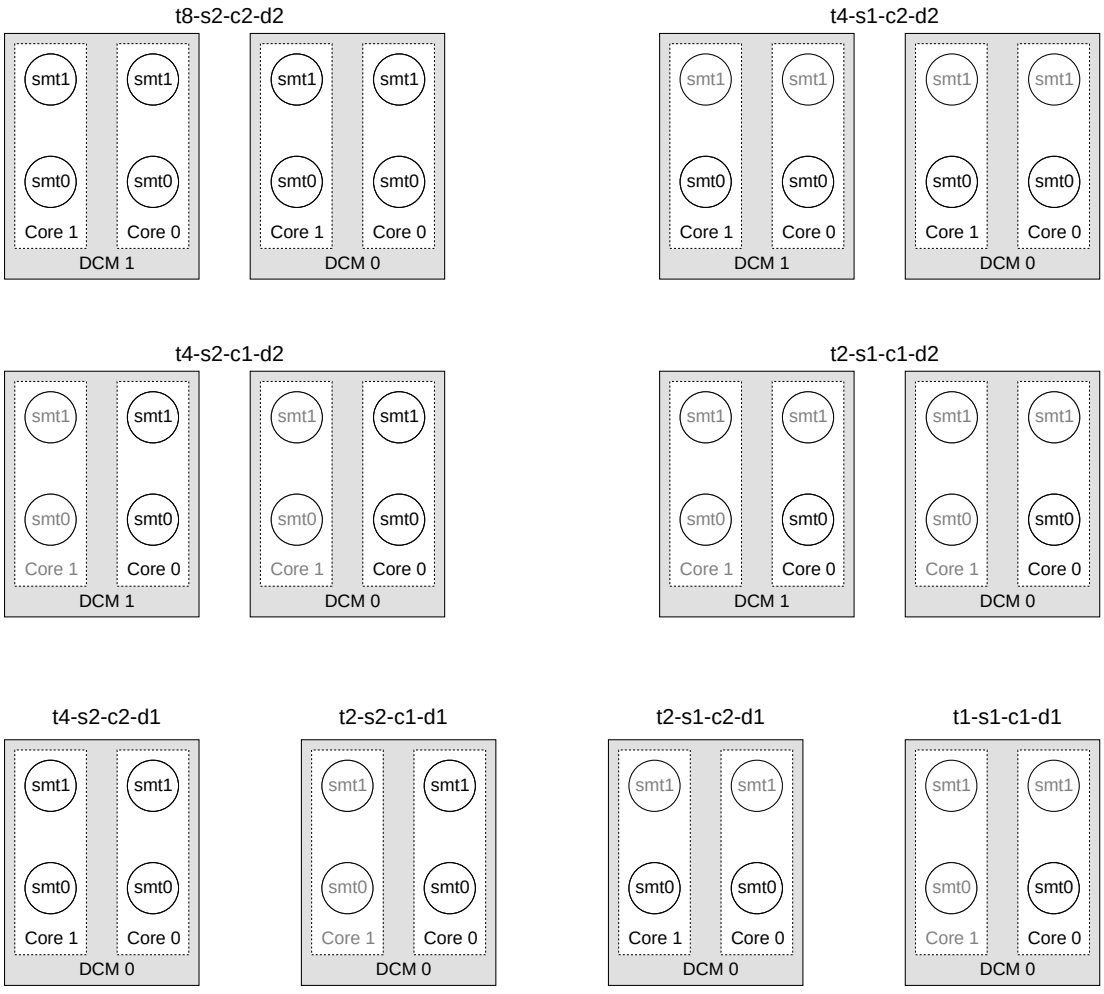

Figure 4.2: The eight different threading combinations described in §4.3 shown diagrammatically

4.4 Comparing SMT and ST - "SMT vs ST pairs"

It should now be evident that on an eight way system like *jago* we have four combinations that will show the difference between SMT and non-SMT (ST) operation.

- t8-s2-c2-d2 *vs* t4-s1-c2-d2
- t4-s2-c1-d2 *vs* t2-s1-c1-d2
- t4-s2-c2-d1 *vs* t2-s1-c2-d1
- t2-s2-c1-d1 *vs* t1-s1-c1-d1

We will refer to these as our "SMT vs ST pairs" in coming sections and use these comparisons to assess the impact, positive or otherwise, of SMT for the trial in question.

4.5 Operating System

The systems in use ran the PowerPC64 "Etch" release of Debian GNU/Linux. Except where noted, we built our own Linux 2.6.18 kernel and made use of it for all runs.

No particular effort was made to quiesce the system in use other than to ensure that no other users were logged in.

4.6 Compiler

In consultation with colleagues it was decided to run a recent though pre-release version of gcc in order to ensure we had the most recent OpenMP support available. In practice this meant all our work with the exception of NAS was built with `gcc (GCC) 4.2.0 20060816 (experimental)` with the NAS work (which required FORTRAN) being compiled with `gcc (GCC) 4.2.0 20061024 (prerelease)`

While both versions were not production versions it was known that they had passed sufficient regression testing for us to proceed with a high degree of confidence.

The only flags used with the compiler of significance were `-O3` and `-Wall` to enable optimisation and show all compilation warnings respectively.

4.7 Automation

Some effort was made to automate the process of running benchmarks, collecting and analysing data and ensuring results were backed up across several locations.

For the most part this consisted of `bash` scripts that would run the jobs required, pipe the resultant output and errors to log files with defined names and generally allow us to do other things while the trials were completed. As some sequences of experiments took over six hours to complete, this was a significant issue.

These scripts will be made available along with the other files from this research on the eScience projects website.

4.8 Thread to CPU affinity

For both the modified STREAM benchmarks (§4.9.2) and our Barrier/Locking tests (§4.10) some effort was put into ensuring that a thread of execution within the benchmark was tied to a particular CPU. Our code was structured such that a particular thread would always map to the same CPU for subsequent trials.

This became particularly important in the context of the round-robin lock benchmarks (§4.10.3) as it meant that a consistent pattern of CPU use was established.

When the benchmark was invoked a bit mask as described in §4.2 was passed that determined the CPU(s) to use and hence the total number of threads.

4.9 STREAM Benchmark

4.9.1 Baseline Setup

Our initial tests with STREAM used a completely unmodified version run multiple times in lieu of using OpenMP to give us a feel for the opreation of the benchmark. The `taskset` utility was used to provide processor affinity for each instance. This method was used to gather the data presented in Table 6.1

STREAM was then built with OpenMP support using the gcc compiler and a similar set of tests run. `taskset` was again used to set overall affinity for the instance of STREAM run, multiple threads being created by the OpenMP `parallel for` directives in the code.

While this approach worked satisfactiorally we were left at the mercy of OpenMP's memory and processor affinity decisions. To give finer grained control over same we elected to modify STREAM as detailed in the following section.

4.9.2 Details of changes

Broadly, our changes to STREAM were as follows

- Addition of command line options to allow processor affinity to be specified, the size of the arrays and the number of repeats.
- Make use of the `pthread` threading library for parallelism.
- Implicit use of Linux's low level `setaffinity()` function. This provided for a consistent match between thread number and the underlying CPU.
- Modifications to the memory allocation code to evenly allocate memory across the processors in use.
- The output from a successful run was expanded to report:
    1. Number of threads and CPU bitmask
    2. Size of array in elements and MB
    3. Number of repeats ('N')
    4. One and Five minute load average for system
    5. Start and End time of runs (seconds since epoch)
    6. For each of the four codes (copy, scalar, add and triad) the maximum, average and minimum bandwidth and times measured

- Synchronisation between threads to ensure that a run was completed by all threads before starting the next. This reduced the "noise" on our data by preventing any single thread to suddenly see more of the available machine capacity because other threads had completed early.

In total this was some 600+ lines of code all told, about 50% more than the standard version. This modified source will be made available on the ANU eScience projects website.

4.9.3 Main Memory & Cache Runs

The scripts used to automate the collection of data from STREAM would perform all the runs of a particular memory size for each thread combination before moving on to the next memory size.

Hence when referring to graphs such as Figure 6.1, time proceeds down the page (most threads to least threads) then across the page left to right (smaller data sets first)

This approach was chosen to minimise the impact any short term loads on the system would have on a the results for a particular thread combination.

4.10 Barrier & Lock Benchmarks

To allow us to benchmark low level lock and barrier primitives we wrote a modest test suite. The basic framework for this code was the modified STREAM source modified as described in Section 4.9). This avoided having to re-write the command line parsing, collection, processor affinity and summary/output codes.

We tested two primitives, an execution barrier and a mutex style round-robin lock. For each we coded up a version using relevant pthreads functions and a version using the low level atomic instructions directly in assembler. We examine these codes in more detail in 4.10.2 and 4.10.3.

4.10.1 Time Keeping

```
volatile inline u64 readtb(void)
{
        u64 tb;

        asm volatile ("mftb %0" : "=r" (tb) : );

        return tb;
}

double mysecond()
{
        /* Constant shown is correct for fandango2 */
        return ((double)readtb() * 5.31793260052222e-09);

}
```

Figure 4.3: Time keeping functions used for barrier and lock experiments

While our modified STREAM source provided an overall starting point we required a very lightweight timekeeping function as the conventional approach of a call to `gettimeofday()` would take longer than the code being measured.

The PowerPC Architecture defines a 64 bit Time Base register available in each CPU which on SMP configurations is guaranteed to be synchronised. In the case of POWER5 these are clocked at around 200MHz giving a resolution of $\simeq$5ns. We made use of it as shown in Figure 4.3.

While crude, the code seemed to work quite well. A further optimisation step would be to simply store direct timebase values and do the final floating point calculation once at the end of the runs.

We note that there has been some recent work on improving the availability of high-resolution timing as part of the standard GNU glibc library. It may be that custom code isn't required in future work which will also reduce platform dependence.

4.10.2 Barrier Codes

Two barrier codes were implemented and benchmarked.

The first, shown in Figure 4.4 used a shared counter locked by `pthread_mutex_lock` and `pthread_mutex_unlock` calls. While more efficient schemes exist, such as the atomic decrement approach, this code was thought to be representative of pthread based barrier implementations in real world applications.

In the closing days of our work it was realised that a `return` statement should have been added after the call to `pthread_mutex_unlock()` in the `if` statement. As shown there is a slim chance that the second call to `pthread_mutex_unlock()` could result in the mutex being unlocked when it was owned by a second thread of execution. We did not have time to repeat our trials but would not expect any great difference, as it will merely mean the counter misses an increment on very rare occasions.

```
void thread_sync(int t_id)
{
        pthread_mutex_lock(&thread_sync_count_mutex);

        thread_sync_counter++;

        if (thread_sync_counter == threads_requested) {
                if (pthread_cond_broadcast(&thread_sync_count_cv) != 0) {
                        fprintf(stderr, "cond_broadcasts failed!!");
                }
                thread_sync_counter = 0;
                pthread_mutex_unlock(&thread_sync_count_mutex);
        }
        else {
                pthread_cond_wait(&thread_sync_count_cv,
                                  &thread_sync_count_mutex);
        }
        pthread_mutex_unlock(&thread_sync_count_mutex);
}
```

Figure 4.4: `thread_sync` function

The second barrier implementation is based on a shared counter scheme. It utilises an atomic decrement function hand coded in assembler as shown in Figure 4.5.

It is based on code developed as part of the ANU SPARC-Sulima project.

4.10.3 Round-Robin Lock Codes

The two locking benchmarks followed the same basic model. A global variable was declared that was incremented by each thread of execution in term - hence the lock was passed round robin style between each thread.

Each thread would spin until the the global variable was equal to its ID whereupon it would try and acquire the lock (waiting if necessary), increment the counter, check for wrapping (via a modulo operation) then free the lock. The two high level functions differed only in which locking function was called and global variables used. They are shown in Figure 4.6.

We reiterate that the round-robin codes were written in a manner that ensured that the lock passed from CPU thread to CPU thread in a consistent pattern.

While the pthread version (obviously) used the pthread libraries `pthread_mutex_lock` and `pthread_mutex_unlock` functions, the assembler version used our own implementations as shown in Figure 4.7. Note that in the interests of clarity this code is edited slightly, mostly to remove the GNU compiler specific semantics for register selection and such like.

4.10.4 Overall Benchmark

The code that was then used to run each of the four tests was straightforward.

Referring to Figure 4.8 we have `t_id` as the thread ID, counting from zero to number of threads. `C` was set to to 0, 1, 2 or 3 depending on which code was being run. `N` was the number of locks or barriers to time, this value is given in the results section. Finally, `NTIMES` was the number of times the codes were run and hence averaged over, the relevant value is also provided with the results. `function_under_test(t_id)` was the lock or benchmark being evaluated.

After the codes were run, the times stored in `start_times[]` and `end_times[]` were processed to arrive at a minimum, maximum and average for each code. The times calculated and output were per thread not the total runtime for that code.

These times were then processed further when plotted, in most cases being divided by the number of threads in operation to arrive at a figure that was normalised by the number of threads in use. This removed the bias that would have otherwise occurred from a barrier or lock naturally taking longer when more threads are in operation.

4.11 NAS Parallel Benchmarks

By comparison to the effort involved with STREAM and our Barrier/Locking benchmarks, running NAS was an exercise in relative simplicity.

4.11.1 Basic Setup

The NAS benchmarks were configured to use a mixture of “Class A” and “Class B” benchmarks, the latter being chosen when the runtime of the “Class A” version was too short to provide reasonable data. All benchmarks were Version 3.2.1

Nine benchmarks were used - ep.A, lu.A, sp.A, bt.A, lu-hp.A, is.B, mg.B, ft.B and cg.B. These were built with gcc as described in §4.6.

Six runs of the nine benchmarks were completed using scripts to automate the process. Figure 4.9 shows the basic arrangement used[5]. This process was then repeated a second time to give a total of twelve runs.

[5]This approach and that used for data parsing inspired by the method used by Tony Breeds [Breeds 2006] in his COMP6702 work

```
#define NUM_PBAR (4)

void pbarrier_init(void)
{
        int i;

        for (i=0; i < NUM_PBAR; i++) {
                pbarrier_at_barrier[i] = threads_requested;
        }

        for (i = 0; i < MAX_CPUS; i++) {
                pbarrier_n_barrier_calls[i][0] = 0;
        }
}

void userspace_atomic_dec(long *var)
{
        asm volatile (

        loop:   ldarx  r1, 0, var    /* Get current value of var  */
                                     /* with address reservation  */

                addic  r1, r1, -1    /* Decrement */

                stdcx. r1, 0, var    /* Attempt to store back
                                        with address reservation  */

                bne-   loop          /* Retry if we lost reservation */
        );
}

void pbarrier(int t_id)
{
        int nbarrier;
        int spin_catch = 100000000;

        nbarrier = pbarrier_n_barrier_calls[t_id][0] % NUM_PBAR;

        /* Only thread 0 gets to set up counter for next barrier */
        if (t_id == 0) {
                pbarrier_at_barrier[(nbarrier + 1) %NUM_PBAR] =
                        threads_requested;
        }

        userspace_atomic_dec(&pbarrier_at_barrier[nbarrier]);

        pbarrier_n_barrier_calls[t_id][0]++;

        /* spinlock */
        while (pbarrier_at_barrier[nbarrier]) {
                if (spin_catch -- == 0) {
                        fprintf(stderr, "thread %d lockedup\n", t_id);
                        pbarrier_at_barrier[nbarrier] = 0;
                        return;
                }
        }
}
```

Figure 4.5: Code snippets for the atomic dec based barrier test

```
void mylock_do_lock_asm(int t_id)
{
        while(mylock_asm_current_thread != t_id) {}

        set_lock(&mylock_atomic_int);

        mylock_asm_current_thread += 1;
        mylock_asm_current_thread %= threads_requested;

        clear_lock(&mylock_atomic_int);
}


void mylock_do_lock_pthread(int t_id)
{
        while(mylock_pthread_current_thread != t_id) {}

        pthread_mutex_lock(&mylock_mutex);

        mylock_pthread_current_thread += 1;
        mylock_pthread_current_thread %= threads_requested;

        pthread_mutex_unlock(&mylock_mutex);
}
```

Figure 4.6: Code snippets for the two lock functions tested

```
void set_lock(int *lock)
{
        asm volatile (
                loop:   lwarx   r1, 0, lock /* Load lock value into r1 */
                                            /* with addr. reservation  */
                        cmpwi   r1, 0       /* See if it's zero        */
                        bne-    loop        /* If not, spin            */
                        li      r1, 1       /* Want to set lock to 1   */
                        stwcx.  r1, 0, lock /* Do store with addr.     */
                                            /* reservation check       */
                        bne-    loop        /* Retry if we lost the    */
                                            /* reservation             */
                        isync               /* Lightweight sync        */
        );
}

void clear_lock(int *lock)
{
        asm volatile (
                        sync                /* Heavyweight memory sync */
                        li      r1, 0       /* Get a zero              */
                        stwx    r1, 0, lock /* And store in the lock   */
        );
}
```

Figure 4.7: Assembler for our low level lock functions `set_lock` and `clear_lock`

```
for (k = 0; k < NTIMES; k++) {
        start_times[0][t_id][k] = mysecond();

        for (i = 0; i < N; i++) {
                function_under_test(t_id);
        }

        end_times[0][t_id][k] = mysecond();
}
```

Figure 4.8: Typical wrapper code used to run each barrier or lock test

```
for run in {1,2,3,4,5,6} ; do
    for thread_mask in {ff,55,33,0f,05,03,11,01} ; do
        for benchmark in {ep.A, lu.A, sp.A, bt.A, lu-hp.A,
                          is.B, mg.B, ft.B, cg.B} ; do

             filename = $date + $benchmark + $threads + $run
             taskset $thread_mask $benchmark > $filename

        done
    done
done
```

Figure 4.9: Pseudo wrapper code used for NAS runs

Chapter 5

A Performance Model for Round-Robin Locking

5.1 Background

Section 4.3 describes the different thread combinations available to us on *jago*. Section 4.10.3 details the benchmark written to allow us to evaluate round-robin locking performance.

The choice of round-robin locking and the known relationship between a thread of execution in the benchmark and a particular thread on the CPU allows us to put model the likely behaviour of the benchmark.

5.2 Description of model

The operation of the round-robin lock is heavily dependent on the efficiency of shared memory operations, in particular the transfer of cache lines containing the locking variable. The characteristic of the lock is each thread updating the counter in turn while the remaining threads read the counter in a spin loop.

We can reasonably argue that the dominant time factor in the process of passing the lock is passing the cache line containing the shared counter (c.f. §4.10.3) between threads of execution. For the system under test, this process of passing the counter can occur over three different paths, each with increasing latency characteristics. In Figure 5.1 the three different paths are shown as a thin green line, a thicker blue line and a still thicker red line.

We assign each of these transfers or latencies a different weight. The thin green line represents the time delay for the lock to be transferred between SMT threads on the same core, l_s. The blue line indicates an inter-core transfer between threads on the same DCM/die, l_c. Finally the red line denotes a transfer between threads on two different DCM/die, l_f.

Figure 5.1 illustrates the seven different thread combinations that can occur (the single thread case is irrelevant in this case) and the "path" taken by the lock from thread to thread. The dashed line indicates the "path" taken by the lock at the start of the next iteration.

Thus at the start of an iteration of the t4-s1-c1-d2 case, the thread on smt0, core0, DCM1 acquires the counter from where it was on DCM0 (smt1, core0). Once acquired is updated and the cache line passes to smt1 on the same code and DCM. It is again updated before transferring across to smt0, core0 on DCM0, updated then transferred to smt1 on the same core which completes an iteration. Again we reiterate that this concept of "transfer" is meant more in a conceptual sense.

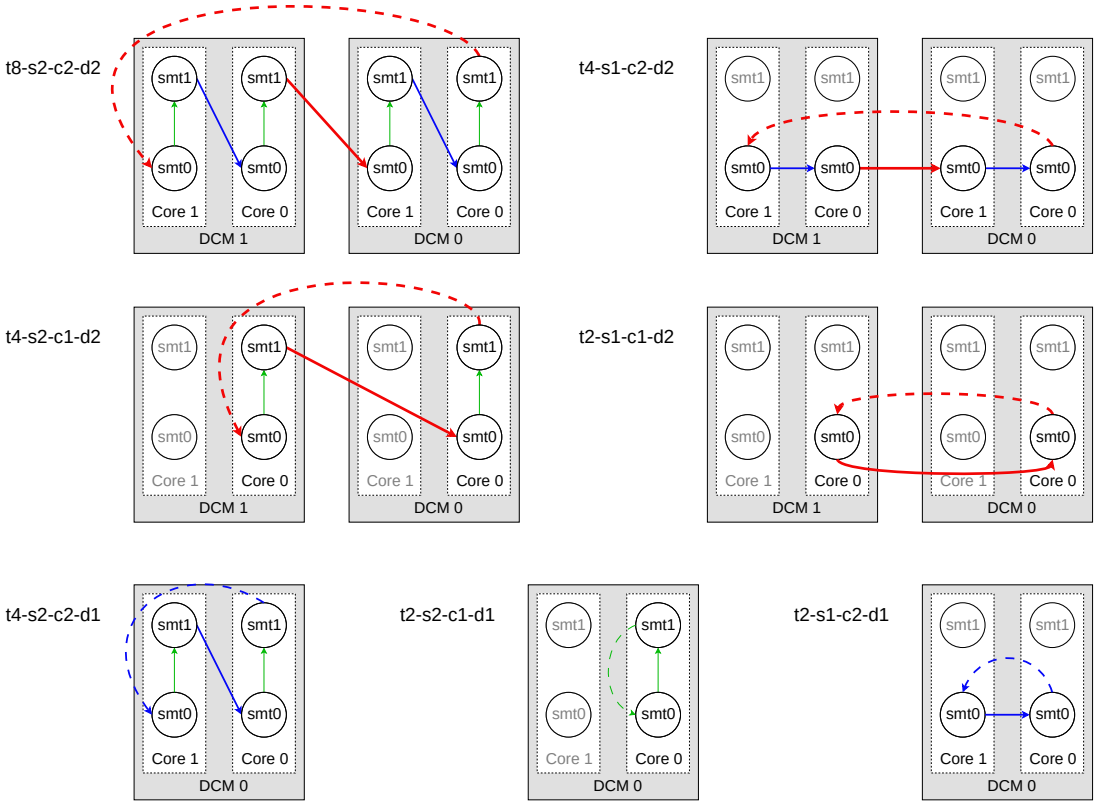

Figure 5.1: Permutations of Round Robin Locking

5.3 Putting into practice

From the above we derive a simple equation for each thread combination to estimate the total latency per iteration, these are shown in Table 5.1

Referring to the documentation for the POWER5 core we can attribute a cycle count and hence time for each latency.

- l_s - transfer between threads is an L1 hit - 2 cycles
- l_c - transfer between cores on same DCM is an L2 hit - 13 cycles
- l_f - transfer between threads on different DCMs is a cross fabric "L2.75" hit - 140 cycles

We took the cycle timing for *jago* of 0.606ns (at 1.65GHz) and arrived at Table 5.2 which we have ordered from lowest cycles per thread to most cycles per thread.

We compare these theoretical figures with those measured experimentally in Section 6.2.4

5.4 Atomic update overheads

The atomic update operation is the second "time sink" in the process, in our model we assume that this operation is constant time for each transaction. In practice however its completion time will vary through coherency operations.

While we have not done so, we would observe that expanding the model to account for this would be a useful area of future work.

Thread arrangement	Estimate of Total Latency
t8-s2-c2-d2	$4l_s + 2l_c + 2l_f$
t4-s1-c2-d2	$2l_c + 2l_f$
t4-s2-c1-d2	$2l_s + 2l_f$
t2-s1-c1-d2	$2l_f$
t4-s2-c2-d1	$2l_s + 2l_c$
t2-s2-c1-d1	$2l_s$
t2-s1-c2-d1	$2l_c$

Table 5.1: Formulae to estimate total latency of round-robin locking for each threading combination

Thread arrangement	Total Cycles (ns)	Cycles Per Thread (ns)
t2-s2-c1-d1	4 (2.42)	2 (1.21)
t4-s2-c2-d1	30 (18.18)	7.5 (4.55)
t2-s1-c2-d1	26 (15.76)	13 (7.88)
t8-s2-c2-d2	314 (190.28)	39.25 (23.79)
t4-s2-c1-d2	284 (172.10)	71 (43.03)
t4-s1-c2-d2	306 (185.44)	76.5 (46.36)
t2-s1-c1-d2	280 (169.68)	140 (84.84)

Table 5.2: Estimated latency in cycles and nanoseconds on basis of figures for POWER5 at 1.65GHz

Chapter 6

Results & Discussion

This chapter describes the results from our experiments with STREAM, our Barrier/Locking Codes and the NAS Parallel Benchmarks. We have aimed to use as consistent terminology as possible in presenting these results, the reader may wish to refer back to Sections 4.3 and 4.4 while reviewing our findings.

6.1 STREAM

Our experimental setup for STREAM is detailed in Section 4.9

We did runs of STREAM in three different ways. Firstly a "sniff test" was performed to allow us to validate our basic setup. Our second and main series of runs stretched from just below 5GB to over 25GB in size. The majority of our focus is on these tests that explore main memory bandwidth. Thirdly we used our modified version of STREAM in a series of runs sized 1.5GB down to 1MB to investigate cache bandwidth.

Published STREAM results tend to focus on peak bandwidth figures. However for our results we have confined our treatment to average[1] bandwidth for the most part as this has more relevance to the real world.

6.1.1 Initial trials

To gain experience with STREAM in practice, we did runs of an unmodified version without utilising OpenMP. Based on suggestions made in [McCalpin 2006], these runs were performed by starting multiple simultaneous instances of STREAM and scaling the results of the final instance.

We achieved some degree of memory and processor locality by using the `taskset` utility to specify a processor for each instance. It is worth reiterating that in this case processor could mean either a physical CPU or SMT thread on a physical CPU.

We completed several runs for each thread combination to ensure there was minimal variance between them. Each run was completed with an array size of 20,000,000 which translated to some 460MB of memory usage. Our results are tabulated in Table 6.1 and were found to be within 8% of the official results (c.f "Standard" figures in Table 6.2). The official results in Table 6.2 made use of a slightly different configuration - utilising the IBM XLF fortran compiler, an earlier Linux kernel and quoting best rather than average figures.

Encouraged that we were on the right track, we proceeded with more involved experiments.

[1] unless otherwise noted our average is an arithmetic mean

jago Threads	Copy avg MB/s	Scale avg MB/s	Add avg MB/s	Triad avg MB/s
t1-s1-c1-d1	2656	2395	3310	3336
t2-s2-c1-d1	2955	2856	3796	3812
t2-s1-c2-d1	3027	2956	3916	3947
t2-s1-c1-d2	5216	4618	6258	6330
t4-s2-c2-d1	3148	3008	4014	3999
t4-s2-c1-d2	5818	5570	7164	7173
t4-s1-c2-d2	5943	5849	7563	7640
t8-s2-c2-d2	6344	5981	7928	7843

Table 6.1: Results for initial "Sniff Test" STREAM Runs

OpenPOWER720 XLF "Official"	Copy best MB/s	Scale best MB/s	Add best MB/s	Triad best MB/s
Eight Thread - Standard	5875	5783	7439	7532
Eight Thread - Tuned	6154	6014	8611	8802

Table 6.2: "Official" figures from STREAM website

6.1.2 Main Memory Bandwidth Measurements

Having established that STREAM was operating as expected we proceeded to do a sequence of runs for memory set sizes from approximately 4.5GB up to just over 25GB. At the low end 4.5GB would ensure that we were well clear of cache effects and at the high end that we did not run the risk of paging to disk (recall *jago* has 32GB of RAM - c.f. §4.1).

6.1.2.1 Main Memory Bandwidth - non-SMT cases

For clarity, Figure 6.1 shows data from a subset of these initial runs - runs without SMT enabled on the CPUs. Recalling the nomenclature explained in Section 4.3 the four thread case utilises both CPU cores on both of the system's DCMs. There are two permutations of two thread operation, one utilising a single CPU from the two DCMs, the second both CPUs on a single DCM only. Finally the single thread case makes use of a single CPU on one DCM.

The single thread case, unsurprisingly, shows the lowest bandwidth with a consistent result of some 2.7GB/s for all the data set sizes tested. A slight downward trend is observed for runs over 16GB where the average bandwidth falls to 2.665GB/s, we conject this is a side effect of an increasing number of accesses coming across the fabric to the core from the second DCMs memory.

The two thread case that utilises a both CPUs on a single DCM achieves just under 3GB/s for runs below the 15GB mark (2.95GB/s) before increasing to 3.05GB/s as both memory arrays are utilised. Compared with the single thread case, this is an increase in bandwidth of between 9% and 12%. This modest increase suggests that we are approaching the single DCM bandwidth limit.

We propose some future work in Section 7.2.2 that will allow these first two conclusions to be investigated further.

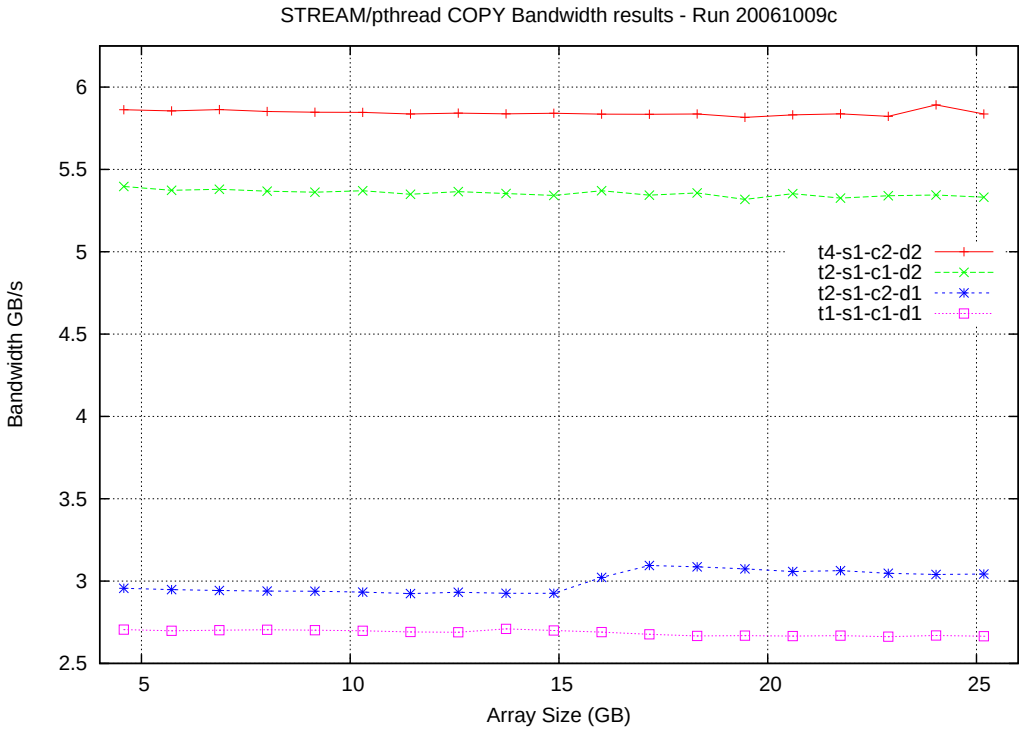

Figure 6.1: STREAM "Copy" Runs showing main memory performance without SMT

The two thread case utilising both DCMs shows a consistent figure of 5.35GB/s. As expected this is approximately double that of the single (one DCM) thread case as we now have both memory subsystems operational.

Finally, the four thread case shows the greatest average bandwidth at 5.86GB/s which is double the two thread, single DCM case until the uptick at 15GB where the ratio drops to a little under double due to the single DCM case getting a little faster at that point.

6.1.2.2 Main Memory Bandwidth - SMT cases

We now refer to Figure 6.2 which shows the four data sets discussed above in addition to the four cases with SMT enabled. This yields us one eight thread case, two more four thread cases and an additional two thread case to those shown in Figure 6.1.

The eight cases shown underscore that perhaps unsurprisingly our bandwidth improves markedly whenever two DCMs are in use. With a single DCM we only just exceed 3GB/s, with two DCMs operational we range from 5.4GB/s up to 5.9GB/s depending on the number of threads.

The eight thread run is about 50MB/s (0.9%) faster than the corresponding four thread run (t4-s1-c2-d2) suggesting that SMT does enable a few more transactions out of the memory subsystem. At 5.9GB/s the eight thread case also represents our fastest figure for the "Copy" benchmark making it some 28% of the theoretical bandwidth for two DCMs (c.f. §3.2.7.2) . At least part of this gap can be attributed to our 1:1 read/write ratio - the theoretical figure apparently assuming a 2:1 ratio.

Contrasting the four thread two DCM SMT based run with four thread two core two DCM

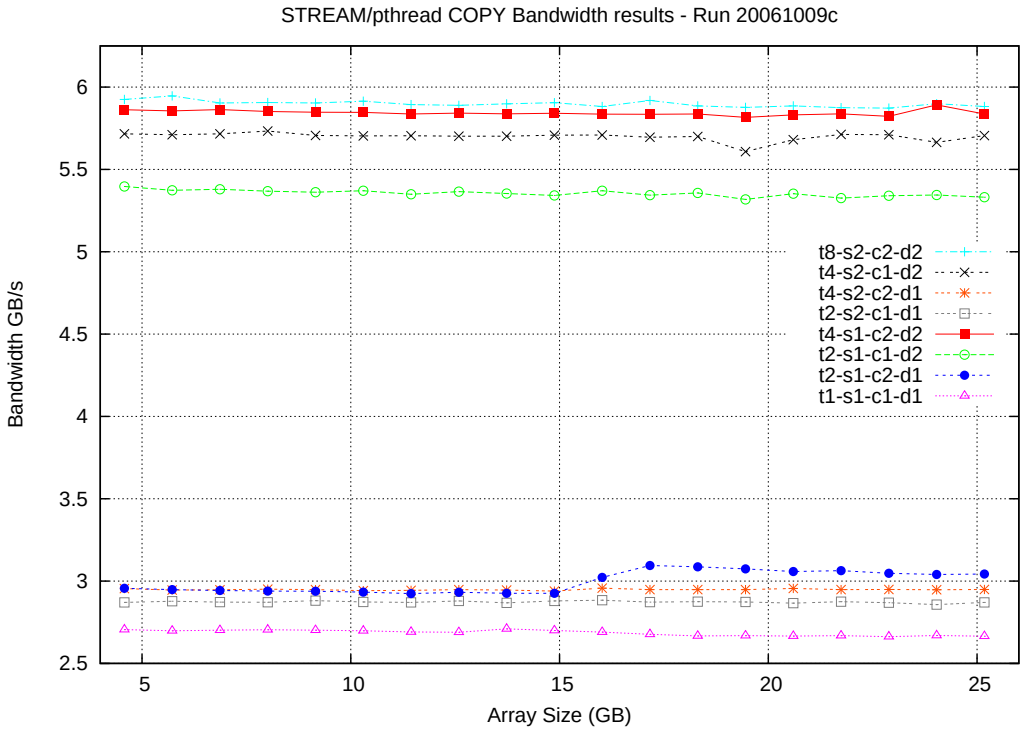

Figure 6.2: STREAM "Copy" Runs showing main memory performance

run (t4-s1-c2-d2 with t4-s2-c1-d2) we see the SMT case producing a bandwidth figure just 2% slower.

A similar comparison between the single DCM two thread cases (t2-s1-c2-d1 and t2-s2-c1-d1) shows the SMT case is 2% slower than the Single Thread (ST) case up to the 16GB mark, where the gap widens to be closer to 6%. We conject that in the SMT case the single core is unable to generate enough additional load/stores to see the full benefit of the second DCM's memory.

Drawing a similar comparison with the single thread case, t2-s2-c1-d1 consistently has 7% higher bandwidth. Thus we infer that the SMT thread is able to generate at least some additional load/stores.

The lower thread count runs that utilise only a single DCM illustrate a drawback of memory affinity. In these instances some benefit could be had by deliberately splitting memory allocations across both DCMs as this would provide more aggregate memory bandwidth. In practice this would likely only be a gain for codes that are not readily parallelised. While we have not investigated it in detail, it is understood that `libnuma` allows some finer grained control over memory allocation with just this in mind. As noted above this is discussed further in Section 7.2.2.

6.1.2.3 Main Memory Bandwidth Per Thread

Figure 6.3 uses the same raw data as that shown in Figure 6.2 however the measured bandwidth has been divided by the number of threads in use yielding a plot of bandwidth per thread.

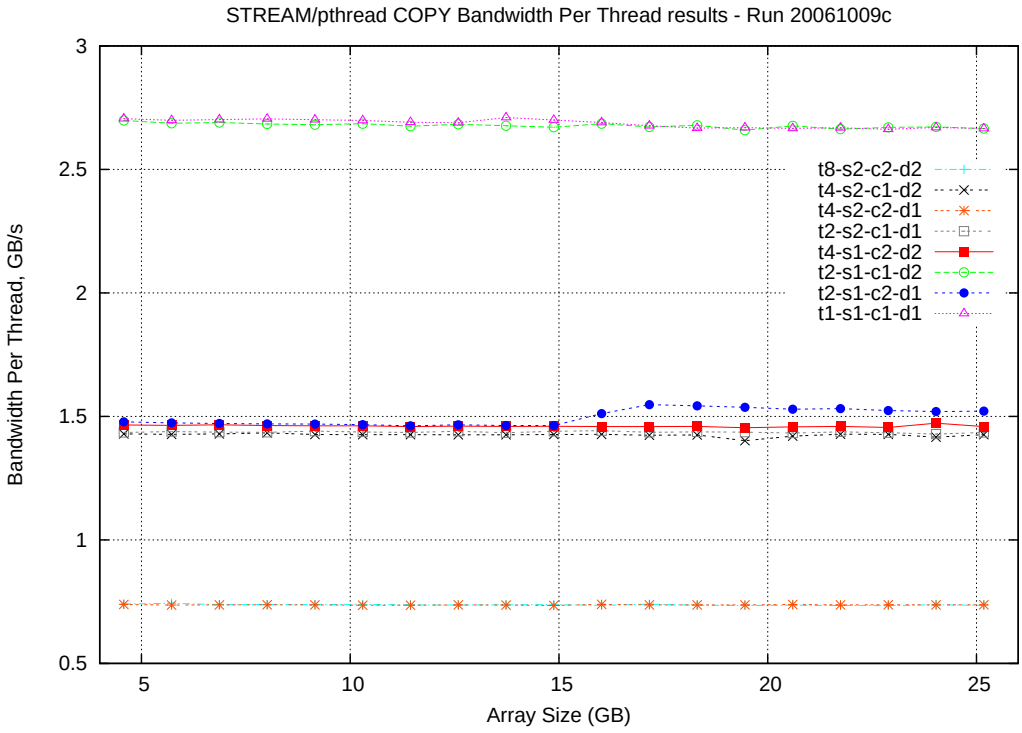

Figure 6.3: STREAM "Copy" Runs showing main memory performance per thread

All cases measured almost identical per-thread bandwidth between the one and two DCM cases indicating, not unexpectedly, little negative interaction between the two DCMs memory systems.

Highest per thread bandwidth is seen in the t1-s1-c1-d1 and t2-s1-c1-d2 cases where a single core has access to the entire resources of its DCM.

Lowest per thread bandwidth occurs as we utilise both cores with SMT enabled, the t8-s2-c2-d2 and t4-s2-c2-d1 cases.

Contrasting the t2-s1-c2-d1/t4-s1-c2-d2 and t2-s2-c1-d1/t4-s2-c2-d2 shows the SMT based cases performing fractionally slower than the CMP equivalent.

6.1.3 Cache Bandwidth Measurements

STREAM is usually run with problem set sizes large enough to ensure it does *not* fit into cache on the system under test making runs of 200MB or over the norm. However it can be run with smaller set sizes and, providing there is sufficient timer resolution available, valid results obtained.

As detailed in Section 3.2.7 we have an L2 of 1.92MB and L3 of 36MB per DCM, two DCMs per system. DCMs snoop across the fabric into the other L2 and L3 which are informally referred to as L2.75 and L3.75 hits. Accordingly we expected to see features in our bandwidth plots around 2, 4, 36 and 72MB depending on the number of threads in operation.

An automated sequence of runs was completed for each thread combination on a closely spaced range of memory sizes ranging from 1MB to 1.5GB. Each run involved between 20 and 100 full iterations through the STREAM benchmark for each size. These results are shown in

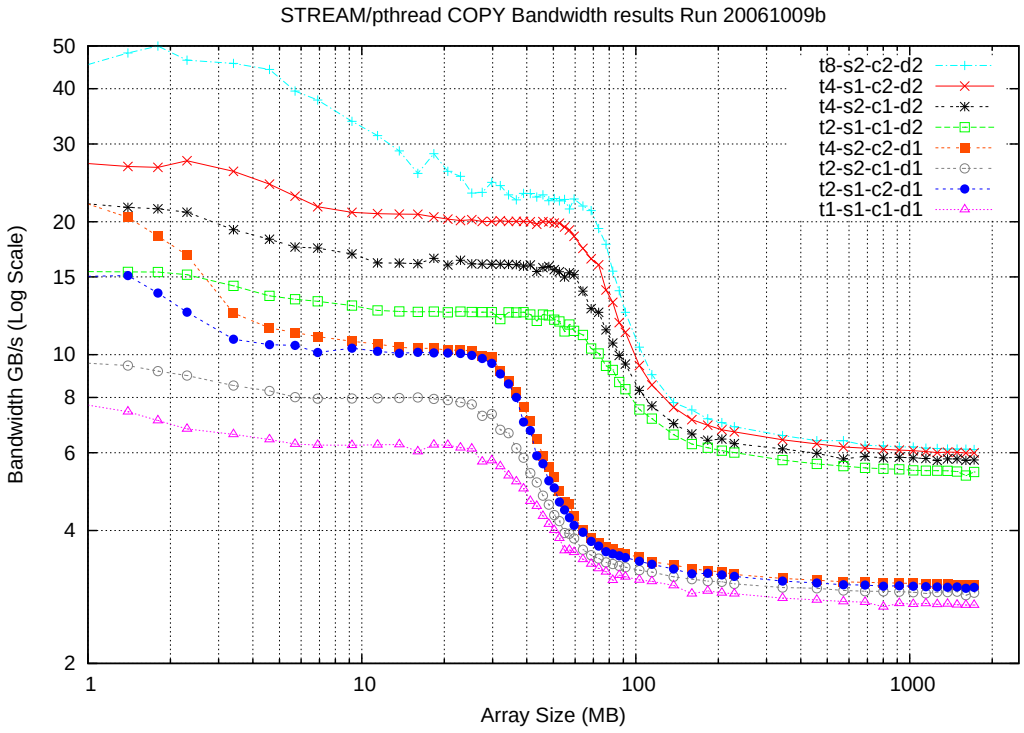

Figure 6.4: STREAM "Copy" Runs exposing cache bandwidths

Figure 6.4.

The resulting graphs show distinct features as we move from operating within L2/L2.75 to within L3 and L3.75 then out to main memory.

Unsurprisingly we consistently see highest bandwidth across the different thread combinations while within the effective area of L2/L2.75 - under ≃4MB (1.92MB x 2). The bandwidth in this region varies from just shy of 8GB/s for the single threaded case to a little under 50GB/s for eight threads.

The single DCM trials drop off slightly quicker than two DCM runs within this region as we see the effect of more accesses going across the fabric (L2.75) than local to the DCM.

Moving out of L2/L2.75 and into L3/L3.75 two distinct "knees" are visible at around 36MB and 72MB. The two DCM cases which make more natural use of the full L3 bandwidth and so sustain bandwidth out toward 70MB in contrast to a rapid fall off around 30MB for the single DCM cases.

We now examine the relative bandwidth figures between different numbers of threads.

6.1.3.1 Contrasting Cache Bandwidth for different thread combinations

Figure 6.5 provides a per-thread view of bandwidth, based on the same raw data as for Figure 6.4

The results obtained within cache are in line with those seen in Section 6.1.2.3 when we looked at a similar plot for main memory (Figure 6.3).

The thread combinations without SMT are again consistently quicker than their SMT-enabled equivalents, an effect more noticeable in cache than main memory.

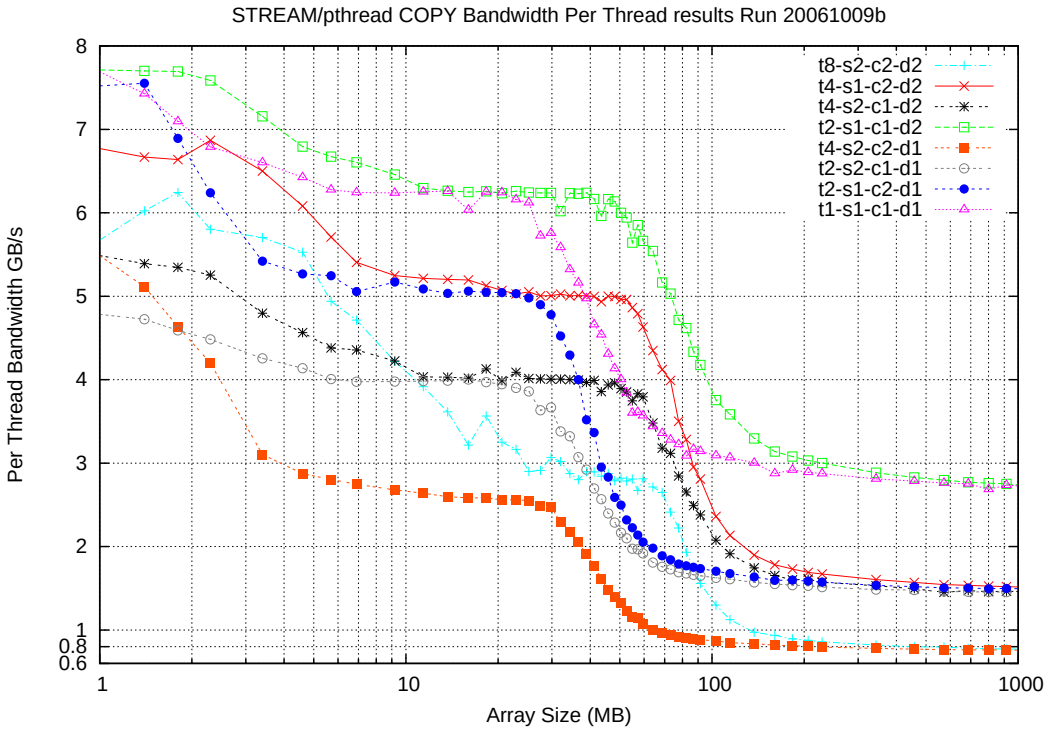

Figure 6.5: STREAM "Copy" Runs exposing relative cache bandwidths

6.1.4 Comparison of Copy with Scale, Add & Triad codes

Returning to our main memory trials, Figures 6.6 and 6.7 show results for the "Scale" (§2.1.2.2), "Add" (§2.1.2.3) and "Triad" (§2.1.2.4) kernels relative to "Copy" (§2.1.2.1) for non-SMT and SMT cases respectively. A "×" symbol is used for "Scale", a "+" for "Add" and a "△" for "Triad" to allow all three to be seen at once. "Copy" itself does not appear on the graph as the other three are shown relative to it and it has been reproduced in earlier graphs - Figure 6.2.

Recall in Section 2.1.3 that STREAM uses a weight of two or three to determine the number of memory operations performed and hence the bandwidth. This infers that our best case figures for "add" and "triad" will be 1.5 times greater than for "copy" (3:2 ratio) and "scalar" being the same as "copy"[2]

In both the SMT and non-SMT cases the "scalar" case runs slightly slower than copy indicating that the additional floating point operation reduces sustained bandwidth by 1-4%. The "scalar" results are otherwise unremarkable.

6.1.4.1 Specifics of non-SMT cases

Examining the non-SMT cases (Figure 6.6) we see an interesting feature in the single thread case (t1-s1-c1-d1) and the two DCM two thread case (t2-s1-c1-d2). The single thread "triad" case oscillates between 1.13 and 1.23 times the "copy" rate, the "add" case between 1.15 and 1.25 times. The period of these oscillations looks to be 8GB for these two cases - 4GB at the higher rate followed by 4GB at the lower.

[2]In this we assume that the CPU won't magically go *faster* if it is doing more processing per iteration...

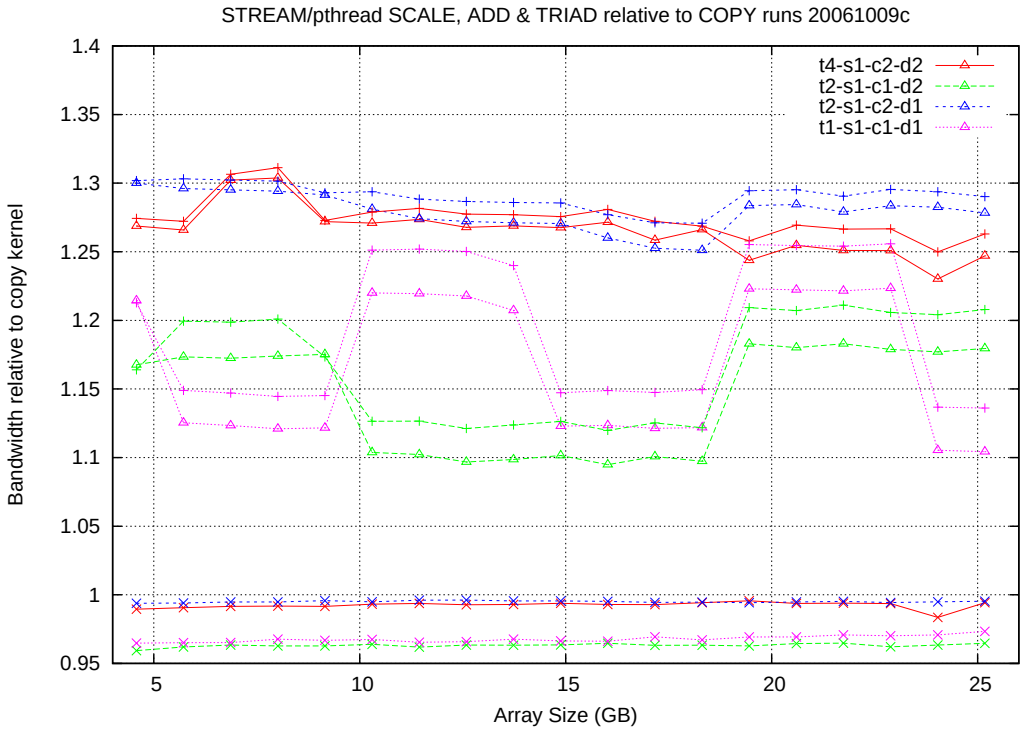

Figure 6.6: STREAM Bandwidth for "Scale"(×), "Add"(+) and "Triad"(△) relative to "Copy" without SMT

The two thread case shows a similar oscillation, the amplitude being somewhat lower and the period apparently double that of the single thread case. We are yet to form a reasoned explanation for this behaviour but would note that it has been seen on a number of different runs. The two configurations exhibiting the behaviour also happen to be the two where a single core on each DCM has no other threads contending for the machine. The fact that these oscillations seem to be linked to 4GB per thread boundaries may indicate some interaction with the MMU address translation logic or the address translation buffers. We discuss this further in §7.2.1

The remaining two cases, t2-s1-c2-d1 and t4-s1-c2-d2 have the use of both cores in common. We see these cases running at 1.25 to 1.31 times "copy". The four thread case perhaps exhibits a slight increase in performance in the same range of cases that the t2-s1-c1-d2 run does, but it is nowhere near as marked. In general we would conclude that these runs merely exhibited the variation one might expect as different threads jockey for access to the CPUs resources.

6.1.4.2 Specifics of SMT enabled cases

Turning now to the four trials with SMT enabled (Figure 6.7) we see some slightly different features.

The eight thread case shows a tendency to oscillate between two values with a period of 2GB or so.

The t4-s2-c1-d2 case shows the greatest consistency, in this respect it is not dissimilar to

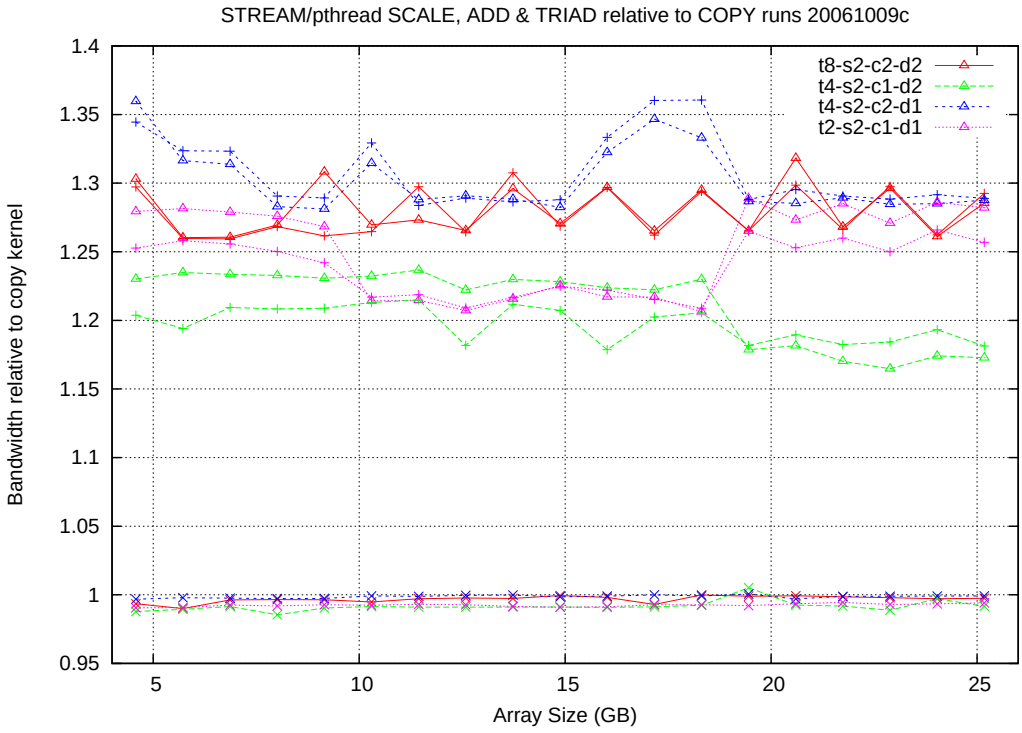

Figure 6.7: STREAM Bandwidth for "Scale"(×), "Add"(+) and "Triad"(△) relative to "Copy" SMT enabled

the non-SMT equivalent case above (t4-s1-c2-d2) though the latter is about 5% faster as one might expect given that more FPUs are available.

The t4-s2-c2-d1 case has highest peak bandwidth relative to the "copy" case however this is somewhat disingeneous as the "copy" figure is also markedly lower (c.f. Figure 6.3)

The two thread SMT case shows somewhat similar features to the two DCM two thread case above - an oscilation effect with a period around 8GB. As noted above we are yet to explain this fully.

The fastest average bandwidth for STREAM is t8-s2-c2-d2 "triad" at 7.55GB/s which is 36% of the theoretical two DCM figure of 20.82GB/s (c.f. §3.2.7.2) This remains quite a large discrepancy, we suspect this may be due to the 20.82GB/s figure not taking into account read to write turnaround and latency.

It is debatable whether the additional data from the "Triad", "Add" and "Scale" aspects of the benchmark are instructive for our work. The "Copy" run gives a pretty accurate picture of bandwidth, the three additional datapoints perhaps not adding much to that picture.

6.1.5 Conclusions about STREAM results

6.1.5.1 A word from McCalpin

McCalpin stated:

> STREAM dates back to a time when floating-point arithmetic was comparable in cost to memory accesses, so that the copy test was significantly faster than the

> others. This is no longer the case on any machines of interest to high performance computing, and the four STREAM bandwidth values are typically quite close to each other. [McCalpin 1995]

Our results appear consistent with his assertion - the four codes are within 35% of each other on the basis of our experiments.

6.1.5.2 SMT mostly harmless ?

Our trials with STREAM suggest that, at worse, SMT has a slight negative effect from an *aggregate* memory bandwidth standpoint - "Triad" and "Add" achieved higher bandwidth than "Copy" alone irrespective of whether the additional parallelism was achieved through CMP/SMP or SMT. The "Scale" results were a few percent *better* for the SMT cases than the corresponding CMP/SMP case; in fairness this sort of improvement would likely dissapear in a real world environment where other threads of execution are vying for cache.

Conversely, the results discussed in §6.1.2.3 suggest that if the codes in question have a need for maximal *per-thread* bandwidth that SMT can in fact work against you if used to achieve eight threads of execution, or four threads using a single DCM. Given that you can achieve four threads of execution without SMT it seems unlikely that this particular configuration would be used in practice unless the other DCM was unavailable (perhaps through being used in another partition). This result is consistent with the observations in [Sinharoy et al. 2005, pp 514] that we discuss at §3.2.6.

Table 6.3 provides a summary of STREAM performance for threads with and without SMT enabled - the SMT vs ST cases. Main memory figures are followed by the equivalent from our 20MB in cache trial in parenthesis.

We have more to say on this in our conclusions (§7.1.1) but would observe that SMT is a win in the majority of cases, the exception being the single DCM, two core case where overall we see a minor degredation in performance. Referring back to Figure 6.2 we see that this reduction in performance only occurs from around 16GB onwards as noted previously.

The figures for the 20MB trial in cache show a markedly higher improvement than the main memory counterparts when SMT is enabled.

Benchmark	t8-s2-c2-d2 vs t4-s1-c2-d2	t4-s2-c1-d2 vs t2-s1-c1-d2	t4-s2-c2-d1 vs t2-s1-c2-d1	t2-s2-c1-d1 vs t1-s1-c1-d1	Average
Copy	0.93 (28.28)	6.43 (27.95)	-1.55 (1.47)	6.96 (26.38)	3.19 (21.02)
Scale	1.38 (14.70)	9.58 (27.98)	-1.18 (10.09)	9.68 (27.49)	4.86 (20.07)
Add	1.19 (12.42)	9.21 (-4.59)	-0.24 (-1.11)	11.04 (61.85)	5.30 (17.14)
Triad	2.17 (11.69)	12.75 (6.44)	0.24 (-1.15)	14.8 (62.99)	7.49 (19.99)
Average	1.42 (16.77)	9.49 (14.44)	-0.68 (2.33)	10.62 (44.68)	**5.21 (23.51)**

Table 6.3: Performance gain (%) for SMT vs ST cases STREAM Benchmarks - Main Memory, Cache in parenthesis

6.2 Barrier & Locking Measurements

The experimental setup used for our investigation into barrier and locking performance is outlined in §4.10

For all experiments the single thread case is included for a reference as it gives some measure of the overhead of the code surrounding the barrier proper.

Conventional shared memory based locking and barrier techniques such as those we studied are heavily influenced by cache and main memory latency. We now examine the results in detail.

6.2.1 Barrier Results

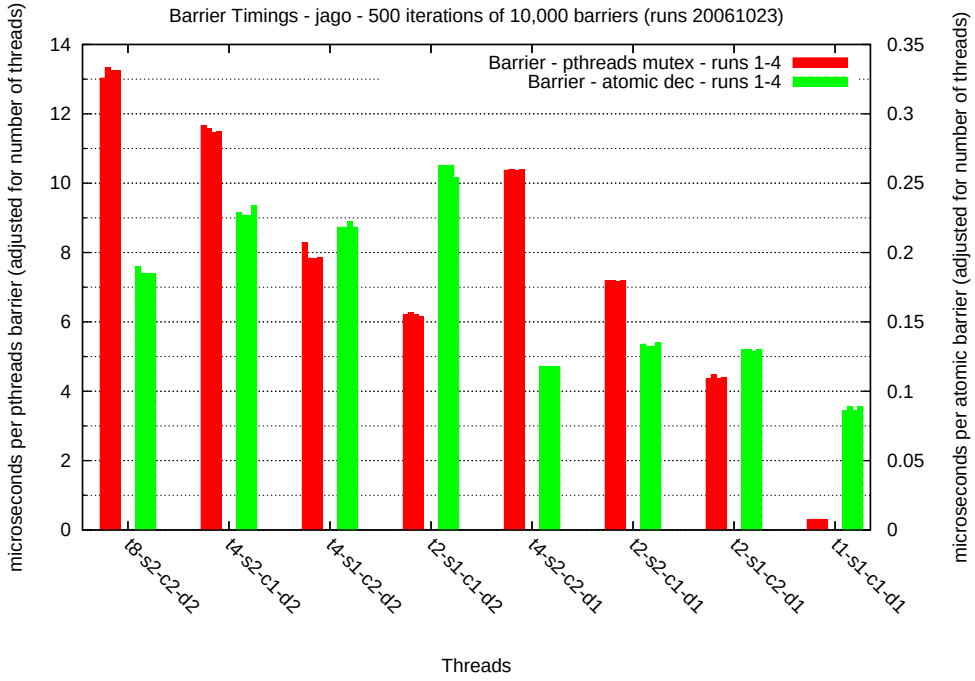

Figure 6.8: Comparison of pthread and atomic based barrier codes for different thread combinations. Note different scales for pthread (red) and atomic-dec (green) results

Figure 6.8 shows the results obtained on *jago* for our two barrier codes. For each thread combination, four runs were performed, results for each run are shown next to each other in the graph. The times shown are per barrier - that is total time normalised by the number of threads being synchronised. Thus, if perfect scalability was achieved, we would expect the results to be flat across the different thread combinations[3]. The two DCM cases are the four to the left, single DCM cases the four on the right hand side. Within this the two SMT cases are on the left, non-SMT on the right.

The pthread based code ran rather slower than the atomic dec version which has a shorter code path and so is presented on a different scale. A linear scale is used for the 'y' axis to allow the relative performance to be gauged.

[3]Assuming the barrier codes are $O(n)$ where n is the number of threads

6.2.1.1 pthread results

The general trend for the pthread results is that two DCM cases are slower than their single DCM counterpart and that the code generally ran slower as the number of threads increased.

Compared against the equivalent atomic case, the pthreads code was slower in all but two instances - those that made use of two cores - t4-s1-c2-d2 and t2-s1-c2-d1.

6.2.1.2 atomic results

The t1-s1-c1-d1 case shows the overhead of the atomic barrier code to be $\simeq$80ns - somewhat higher than expected. A examination of the underlying assembler code would presumably cast some light on this - unfortunately we did not have a chance to perform such analysis.

The single DCM cases are faster overall than any of the two DCM cases - a reflection of not needing any cache line transfers across the fabric. The two thread cases give a clean test case for us to examine the effects of the cacheline being bounced across the fabric. The single DCM case t2-s1-c2-d1 runs at 0.13μs per barrier compared to 0.26μs for the two DCM, single core case t2-s1-c1-d2 representing a difference of 0.13μs or some $\simeq$224 cycles.

The point of coherency for the single DCM case is the L2 cache which can be accessed with a 13 cycle latency ($\simeq$78ns); however reads will of course be serviced from the cores local L1 which has a 2 cycle delay. This contrasts with our two DCM case where the coherency point remains L2, but for the "other" CPU it is L2.75[4]

The one to two DCM difference can be further quantified by examining the t4-s2-c1-d2 (0.23μs) vs t2-s2-c1-d1 (0.13μs) and t4-s1-c2-d2 (0.22μs) vs t2-s1-c2-d1 (0.13μs) cases. The average delta between the two is 0.09μs or, assuming 1.65GHz clock frequency, $\simeq$153 cycles which is within 8% of the figure published in the POWER5 specifications (140 cycles). We compare this combination as the layout of threads within the DCM remains the same, the additional transfers across the fabric being the only change.

The t4-s2-c2-d1 case gives the lowest outright time per barrier figure of 0.12μs. We conject that this occurs as this combination sees all the transfers occuring within the DCM, such transfers being about an order of magnitude faster than across the fabric. The additional pair of threads further "amortising" any overhead compared to the two thread single DCM cases.

6.2.1.3 Influence of SMT

Two pairs of cases lend themselves to direct comparison of the SMT and non-SMT cases while keeping the total number of threads constant; t4-s2-c1-d2 vs t4-s1-c2-d2 and t2-s2-c1-d1 vs t2-s1-c2-d1.

The pthread codes showed the greatest slowdown when SMT was enabled, in the four thread case an increase of 3.6μs (46%) and 2.8μs (64%) for two threads. This we attribute to a more complex code path in the pthread case, something that could be validated with instruction level analysis.

By contrast the atomic versions were just 40ns/67 cycles (2%) slower with SMT enabled and the slowdown was the same for two and four threads.

6.2.1.4 Conclusions for Barriers

Overall there is a large increase in barrier time when the barrier is shared across two DCMs. When Barriers are shared between cores on a single DCM there is a more modest increase than

[4]Recall that data can be read by DCM 0 from DCM 1's L2 cache over the fabric in what is called a L2.75 access

when shared between SMT threads.

6.2.2 Round-Robin Locking Results

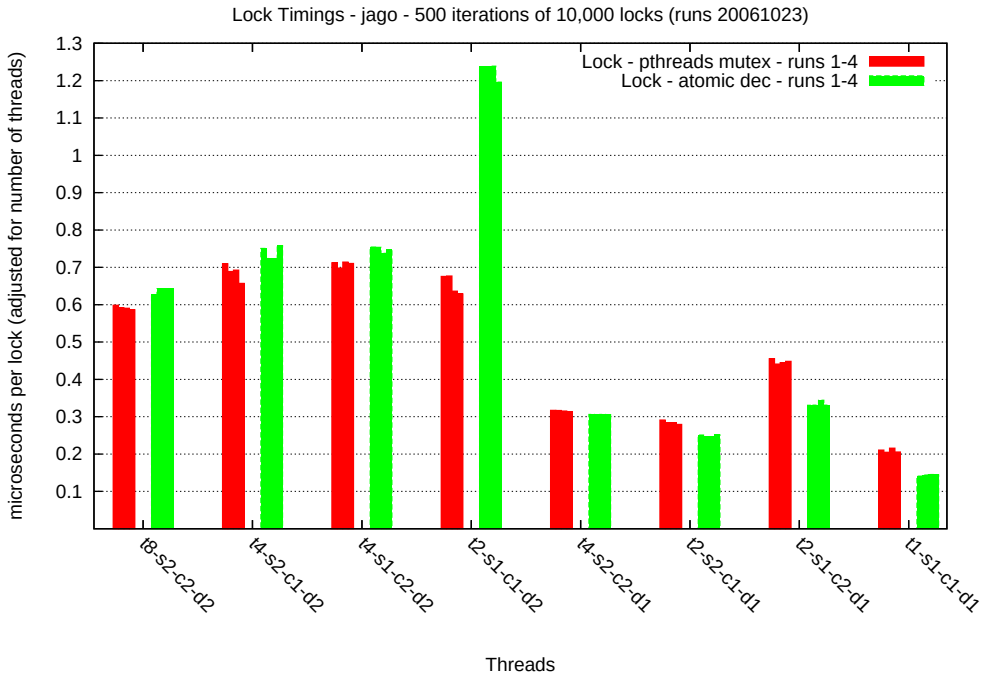

Figure 6.9: Comparison of pthread and low level atomic based locking codes for different thread combinations

Figure 6.9 shows the results obtained on *jago* for our two lock codes and is laid out in a similar fashion to the barrier results at Figure 6.8. Once again the times shown are per lock divided by the total number of threads in use.

For our locking codes, the pthread and atomic versions are, with one exception, much more closely matched than the barrier codes - typically differing by less than 10%.

The exception is the two thread two DCM case where the atomic code is nearly twice as slow as the pthread implementation. The fact that the atomic code is so much slower than its four and eight thread, two DCM equivalents can be attributed to the overhead of the cross fabric transfer of the cache line. This penalty is paid twice per iteration for any of the two, four or eight thread cases, in the latter two it is amortised to a far greater degree[5].

However we are yet to arrive at an explanation for the pthread code exhibiting far less of a penalty than the atomic case.

For the two DCM cases we see the pthread implementation being faster than the atomic code by an average of 5%, the exception as noted being the two thread two DCM case that is closer to 55% faster.

The single DCM cases see this trend reversed with the atomic code being consistently faster than pthreads. The delta is greater than the two DCM cases, ranging from 3.5% for the

[5]Recall that the order in which the locks are taken is deterministic here due to the round robin design unlike the barrier case which are more random

four thread case to 25% for the t2-s1-c2-d1 case.

The single thread case which provides a sense of the base overhead is more closely matched than the Barrier case - the pthread and atomic codes being within 30% of each other. Again an assembler level analysis would show in detail where the differences lie but we conject it is related to the overhead of calling into the pthreads library.

6.2.2.1 Influence of SMT

We now consider the influence of SMT in a similar manner to that used to compare the barrier codes in Section 6.2.1.3 For these comparisons we again look between cases with the same total number of threads.

As noted above, two cases lend themselves to direct comparison of the SMT and non-SMT cases: t4-s2-c1-d2 vs t4-s1-c2-d2 and t2-s2-c1-d1 vs t2-s1-c2-d1.

The SMT enabled cases are consistently quicker than the non-SMT case for both the four and two thread tests which we attribute to better cache behaviour - in the SMT enabled cases there is a greater likelyhood that the operations can be performed with reference to L1 rather than L2 cache.

As noted previously the single DCM cases see the atomic code being quicker than pthreads, a trend that is reveresed for the two DCM case. None the less it is reasonable to conclude that SMT is a win for these codes.

The t8-s2-c2-d2 case is the same as the t4-s1-c2-d2 trial with the addition of SMT. We see a reduction in the per-lock time of 99ns (16%) for the pthread case and 112ns (13%) for the atomic operation, a result of a greater number of lock resolutions being handled within L1 in the eight thread case.

Of the three four thread cases, the t4-s2-c2-d1 case is understandibly the quickest. There are no cross fabric cacheline transfers involved and so delays are minimised.

6.2.3 Summary of SMT effects on Barrier and Locking trials

Benchmark	t8-s2-c2-d2 vs t4-s1-c2-d2	t4-s2-c1-d2 vs t2-s1-c1-d2	t4-s2-c2-d1 vs t2-s1-c2-d1	Average
Barrier - pthread	-69.31	-85.31	-135.11	-96.58
Barrier - atomic	15.863	12.01	9.03	12.30
Round-Robin Lock - pthread	16.65	-4.96	29.09	13.60
Round-Robin Lock - atomic	13.85	39.94	9.75	20.85

Table 6.4: Performance gain (%) for SMT vs ST cases for Barrier and Lock Benchmarks

Table 6.4 shows the data for our SMT vs ST pairs. Comparing Barriers or Locks in a situation where the total thread count varies is somewhat arbitrary as for any $O(n)$ or higher algorithim the time taken per lock will necessarily increase with the number of locks.

None the less the comparison provides an interesting data point alongside the STREAM results presented in Table6.3 and those for NAS that appear in Table 6.6 in the next section.

The data presented shows a dramatic reduction in performance with the addition of SMT for the pthread barrier codes and a slight degradation for the t2-s1-c1-d2 vs t4-s2-c1-d2 trial of the pthread locking code. Otherwise SMT once again provides an overall gain, something that can be attributed to a greater number of lock "transfers" occuring within the same core.

6.2.4 Comparison of results against model

Thread arrangement	Model Total	Model Per Thread	Measured pthread	Measured atomic
t2-s2-c1-d1	2.42	1.21	283	246
t4-s2-c2-d1	18.18	4.55	317	306
t2-s1-c2-d1	15.76	7.88	441	331
t8-s2-c2-d2	190.28	23.79	599	642
t4-s2-c1-d2	172.10	43.03	689	724
t4-s1-c2-d2	185.44	46.36	698	754
t2-s1-c1-d2	169.68	84.84	677	1237
t1-s1-c1-d1	n/a	n/a	205	144

Table 6.5: Estimated and measured latency in nanoseconds. Estimates assume POWER5 at 1.65GHz

In Chapter 5 we proposed a model to estimate the performance of the round-robin locking codes for the different thread combinations.

As Table 6.5 illustrates we were, unfortunately, way off in the predicted figures compared to those measured.

However the model did accurately determine the cycles per thread ranking of the trials including the somewhat subtle distinctions between SMT and ST cases with the same number of threads.

We conject that the model inadequately took into account the overhead associated with the coherency traffic and execution time of the `lwarx` and `stwcx` instructions (c.f. Figure 4.7) The `sync` and `isync` instructions are also known to be expensive, though their effect is somewhat accounted for in the inclusion of terms for the cacheline transfer itself.

By way of example, when a `stwcx` write originating on smt0, Core1, DCM1 completes it must be propagated to the other caches which may include inter-DCM and/or cross fabric updates depending on the case. Hence our model needs some additional terms that are dependent on the other cores active in each case.

While really a matter for future work, we speculate that the following equation would get us closer to the observed data:

$$t_i = a_0 + a_1 \times l_t$$

Where t_i is the total time per iteration, a_0 and a_1 are system dependent constants, and l_t is the latency from the original model ($4l_s + 2l_c + 2l_f$ for the t8-s2-c2-d2 case).

6.2.5 Additional Barrier and Locking Results

We made use of a larger POWER5 machine *fandango2* to collect some results for our barrier and locking codes.

fandango2 has four DCMs - double the number used in *jago*, this necessitated adding a suffix to our thread labelling scheme. By way of example there are now at least two t4-h2-c1-d2 combinations so we have labelled them t4-h2-c1-d2 (a) and t4-h2-c1-d2 (b). The symmetric nature of the design means these combinations should perform identically, this is borne out by the results obtained.

Regrettably we did not have time to properly integrate these into the forgoing analysis or draw any deep conclusions. We include them in Appendix (§**??**) for completeness and make the following quick observations;

1. The atomic t8-h1-c2-d4 barrier and lock trials are substantially slower than the other runs. While the additional cross fabric traffic accounts for this to some degree, the t4-h1-c1-d4 case isn't such an outlier.

2. For the most part the results for identical thread combinations between the two machines are consistent. The underlying hardware is very similar so this is not unexpected.

3. Doubling the number of DCMs in use does not degrade the round-robin locking by as greater a degree as would have been expected given the additional cross fabric transfers.

6.3 NAS Parallel Benchmarks

The setup used for our investigation into NAS is outlined in §4.11 As noted in that section we ran each benchmark of interest in turn six times and averaged[6] the results. This process was in turn repeated twice, the graphs in Figures 6.10 and 6.11 showing these two trials adjacent to each other in each colour bar.

6.3.1 Starting with an Outlier

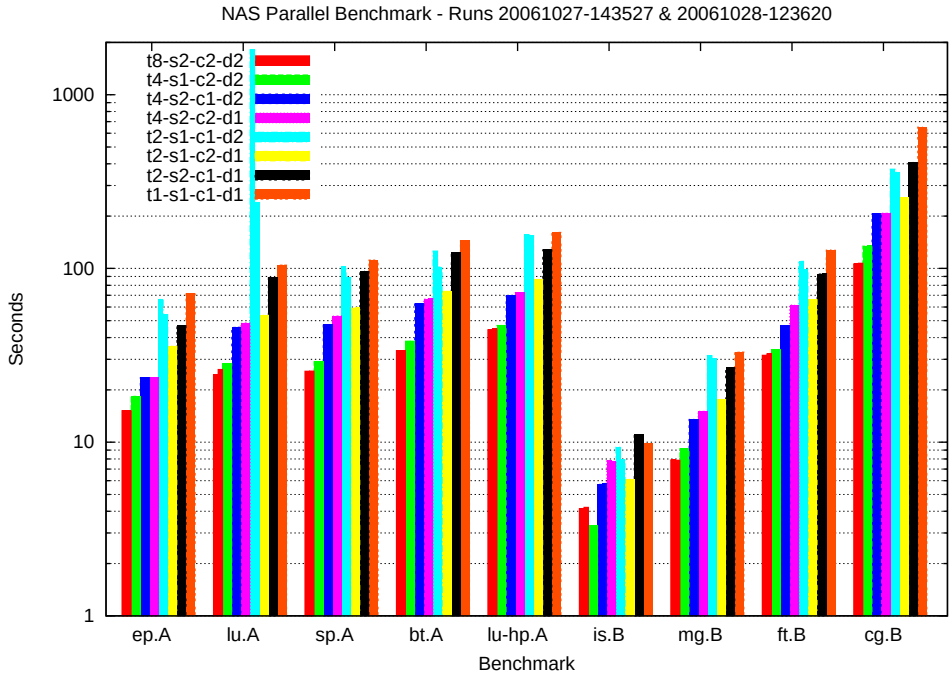

Figure 6.10: Results for selected NAS benchmarks on *jago* showing absolute timings

[6] arithmetic mean

Figure 6.10 shows the results for nine of the NAS benchmarks against the eight thread combinations available on *jago*.

The main datapoint evident is an outlier in the t2-s1-c1-d2 trial of the lu.A benchmark (blue bar). An examination of the underlying data shows run times of 52.43s, 2747.78s, 2748.50s, 2927.90s, 1300.67s & 1118.26s for the first trial and 151.00s, 150.61s, 52.37s, 52.47s, 52.50s & 981.27s. This produces the two averages plotted of 1815s and 239s both well outside the region expected on the basis of the other thread combinations tested. We do not have a concrete explanation for this but time permitting would re-run these benchmarks under closer scrutiny to see what can be concluded. Examining the raw data suggests that the actual figure is around 53s.

6.3.2 Analysis of non-SMT cases

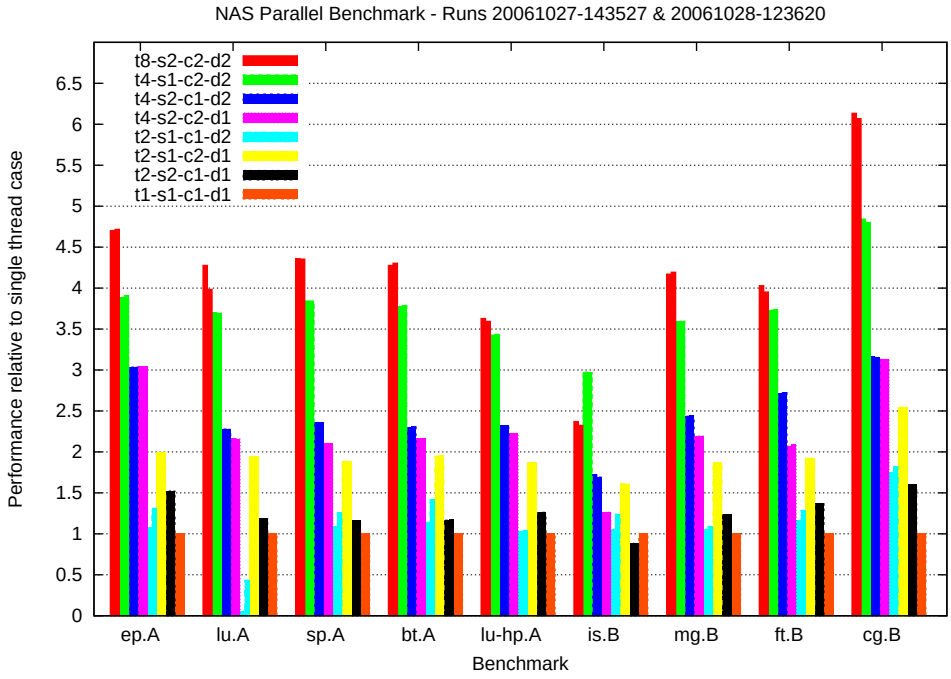

Figure 6.11: Results for selected NAS benchmarks on *jago* showing ratio to single thread case

We now refer to Figure 6.11 which shows the results obtained relative to the single thread case.

Looking at the two thread cases we observe that the single DCM run (t2-s1-c2-d1) is consistently faster than the two DCM case (t2-s1-c1-d2). The former achieves anything from 1.6 to 2.5 times the single thread case whereas the latter manages 1.5x for the cg.B trial and otherwise is scarecely faster than the single thread case.

This result is noteworthy in that it is rather at odds with our experience with STREAM. This suggests that any improvement in bandwidth that the two DCM case may provide is outweighed by the additional coherency traffic generated for NAS style workloads.

Benchmark	t8-s2-c2-d2 vs t4-s1-c2-d2	t4-s2-c1-d2 vs t2-s1-c1-d2	t4-s2-c2-d1 vs t2-s1-c2-d1	t2-s2-c1-d1 vs t1-s1-c1-d1	Average
ep.A	17.25	61.05	34.30	34.32	36.73
lu.A (corrected)	10.43	13.52	10.03	15.47	12.36
sp.A	11.82	50.49	10.28	13.83	21.60
bt.A	11.89	45.15	9.72	14.42	20.29
lu-hp.A	4.98	55.34	16.04	20.71	24.27
is.B	-26.58	33.33	-27.38	-12.58	-8.30
mg.B	14.11	56.16	14.71	18.80	25.95
ft.B	6.40	55.20	7.58	26.99	24.05
cg.B	20.94	43.60	18.67	37.44	30.16
Overall Average:	**7.92**	**45.98**	**10.44**	**18.82**	**20.79**

Table 6.6: Performance gain (%) for SMT vs ST cases NAS Parallel Benchmarks

6.3.3 Analysis of SMT cases

Once again we examine the results for our usual SMT vs ST combinations (c.f. §4.4) Table 6.6 shows a summary in tabular form. In the table the "lu.A (corrected)" entry assumes a run time of 53s for the anomalous t2-s1-c1-d2 case described in §6.3.1

SMT provides improved benchmark results for all the NAS codes tested with the exception of is.B - the Integer Sort where it results in a slight degredation of performance in all but one case.

The most dramatic improvement is in the ep.A run when we contrast the results for t4-s2-c1-d2 and t2-s1-c1-d2, here the gain is around 61%. The average improvement from enabling SMT across all the benchmarks with the exception of is.B is of the order of 20%.

Enabling SMT saw a degradation in most cases for is.B, however, comparing t2-s1-c1-d2 and t4-s2-c1-d2 saw the time for the run reduce from 8.5s to 5.7s - an improvement of about 33%

We can account for this by considering that in this case SMT can more effectively cover up the cache misses (and related coherency traffic) without the core becoming saturated overall. This latter point also explains the negative figures generally, in these cases the proportion of cache traffic outweighs the ability SMT to cover up same due to execution unit limitations.

Overall we see NAS benefitting from SMT as it has a more balanced instruction mix and is not so bandwidth limited. Comparing the improvements here to those seen for STREAM and the barrier/locking codes supports this conclusion.

6.3.4 Super Linear Speedup

Super Linear speedup is observed on the cg.B benchmark for the t4-h1-c2-d2 and t2-h1-c2-d1 thread arrangements which achieve 4.7x and 2.5x speedup respectively.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

Chapter 6 presented our results the associated discussion of their implications. From these we feel there are a number of broader points that can be drawn which we present in this section.

7.1.1 Is SMT worth it ?

Our results show SMT to be a useful gain in the substantial majority of cases. The cases where SMT works against performance are sufficiently specialised that they should not be a cause for dismissing it.

- STREAM improved by between 3.5% and 7.5% - an encouraging picture for bandwidth sensitive applications, or at least one that suggests it does no harm!
- Our barrier/locking codes were more of a mixed bag. The worst case being an average degradation of 97% - possibly more to do with the way our code was written given the other three benchmarks showed an improvement of between 12% and 21%.
- NAS averaged between -8% and 37%, the average being nearly 21%. Given NAS is a more real world measure of system performance, this speaks well of the utility of SMT.

If we ignore the two outlier cases we see that SMT provides a performance gain of anywhere from 3% to 37% depending on the workload.

Figures vary but most literature on the subject suggests that the implementation of SMT on a contemporary processor adds about 10% to the die area (c.f. [Mathis et al. 2005] and [Sinharoy et al. 2005]).

It would seem that this minimal increase in die area is worthwhile for the performance gained, doubly so when one considers that a CMP implementation adds closer to 80% die area for the second core.

Thus we conclude SMT is indeed worth it.

7.1.2 Suitability of selected benchmarks

Both STREAM and NAS give realistic and reproducible measures of processor performance. The results correlate well in general - a thread combination that was better on one was usually better on the other.

NAS however did exhibit consistently higher gains for the addition of SMT. With SMT enabled, the processor is better able to hide cache and memory latency of the system - the

instruction mix in NAS lends itself to this far more so than the simpler codes of STREAM which are by design bandwidth limited.

This suggests that some caution should be exercised in concluding too much from very simple benchmarks for SMT based threading - it may be that more complex codes actually do better.

Our modest barrier and lock benchmark shows some utility in evaluating this aspect of the system. Clearly a matter for future work but we feel it could be improved and optimised to give a more consistent sense of processor performance.

7.1.3 A Case for Hierarchical Barriers & Locks

Our measurements on Barriers and Locks demonstrated better performance when dealing with SMT threads or a threads within a single DCM.

As the number of cores/threads increase, the overhead of thread "unaware" barriers and locks will become increasingly significant. We believe there is merits in developing hierarchical locks that are written to be aware of the underlying thread placement, particularly in relation to SMT and CMP versus SMP threads.

Such algorithms will require intimate knowledge of the underlying hardware platform. Accordingly there is a case for providing such primitives to user space from the operating system kernel or a library rather than relying on the application.

With the increasing number of cores on a "typical" system this would be a timely area of future research. In §7.2.3 we describe some related work.

7.1.4 Disabling CPUs

As described in §4.2.2 we made use of processor affinity to create the various thread combinations desired. This meant that the unused CPUs were left idle or possibly running other "light" tasks on what was otherwise a quiesced system.

Through the Linux kernel's "hotplug" infrastructure it is possible to literally take a CPU offline completely making it unavailable to kernel or user space.

If a POWER5 core is run in ST rather than SMT mode it can make certain resources that would have been used by the second SMT thread available to the running thread. These include additional rename registers, issue queues, the LRQ and SRQ[1].

The availability of these additional resources in ST mode may have increased the performance seen our benchmarks somewhat. This would have the effect of reducing the apparent gain of enabling SMT.

Alternatively, the fact that other processes on the system would now be scheduled over a smaller number of threads may have offset any apparent gain.

Some simple experiments could be performed to compare (say) four thread results for the various benchmarks when using processor affinity versus actually offlining the CPUs.

7.1.5 Multicore is here to stay

In the "Conclusions" section of the literature review that preceded this work we wrote, in part;

> In locating, reading and analysing the literature on the three areas of interest it has become clear that they document areas of study that are both active and important.

[1] LRQ - Load Reorder Queue, SRQ - Store Reorder Queue

> We are also struck by the timeliness of such research. While the consideration of multiprocessing has a rich, and by computer science standards, relatively long history, it is only in the last few years that devices that implement such techniques have become commonplace.
>
> We have been fortunate during this process to have had the opportunity to informally discuss future directions with microprocessor designers at a number of vendors. From these conversations it is clear that multiple threads of execution are here to stay and that indeed Moore's law really is slowing.[Blemings 2006]

This was some six months ago now and in that time we have seen all major vendors announce product lines that have multiple cores on a single die. The extent of this multi threading is all but astonishing in some cases - the potential for hundreds of cores on a single die brings with it the potential of systems with thousands of threads in a single box.

Clearly the rapid rise of multiple cores will bring with it an increasing expectation that the codes we write can exploit them, this irrespective of whether the application is computational chemistry, virtual reality or the latest media engine.

Our results suggest that this is likely to require changes in the way algorithms are designed and, potentially, a far greater awareness of the underlying hardware than is the norm at present. Scope exists for the Operating System to abstract this to some degree, but particularly for performance critical codes, the metaphor that it's all "just a thread" may no longer be sufficient.

7.2 Future Work

There are several matters that in our view warrant further research.

7.2.1 Recording Amount of Funny

> "When you are taking data, if you see something funny, Record Amount of Funny."[2]

We saw a number of anomalous results during the course of our experiments. Most were traced to a programming error or other human factor in short order, others were not.

We are particuarly curious about the "oscillations" observed in the STREAM "Triad" and "Add" results (Figures 6.6 and 6.7).

They only seem to occur on 4GB boundaries and only in the triad/add cases, where there are two reads per write. The single thread case has a period of 8GB, the two thread case potentially of 16GB suggesting some relation to the memory footprint of a single thread.

The eight thread case seems to exhibit a similar oscillation albiet at a shorter period - 2GB in this case. We assume these two results are somehow related.

We speculate some interaction with the address translation process either in the CPU itself or perhaps within the Linux kernel.

To investigate this further we would look at such things as;

- Ensure result is reproduceable. While we did several trials in the same one-two week period all of which showed the same pattern, we'd naturally confirm it was still present to get a baseline.

[2] "Milligan's Law" - attributed to Tom Milligan by Bob Pease in [Pease 1991]

- Find the shortest trial that produces the effect. The runs used in our experiments take tens of minutes each. We would hope that a similar pattern would emerge in far shorter trials.
- Identify the exact transition points - they look to be at 2GB or 4GB boundaries - are they really ?
- Work out if the effect relates to the fact that "triad" and "add" have floating point instructions, or just an additional write. This could be as simple as modifying "copy" to do two writes of the same data.
- See if similar effects are visible on other POWER5 systems and/or different architectures entirely.
- Consider whether effects observed relate to underlying DIMM size (2GB on *jago*).

More generally it would be instructive to use performance counter analysis for some of our codes to better understand what is occuring. For example counting cache misses would allow us to see deeper into the behaviour of the locking and barrier codes.

7.2.2 Memory Bandwidth vs Locality Measurements

In §6.1.2.1 we observed an increase in performance in the two thread case as we passed the half way point in system memory. We believe this is related to a greater proportion of accesses coming from the second DCM's memory.

We envisage a micro benchmark or set of modifications to STREAM could allow the user to specify different memory access patterns in a similar manner to the way we have trialed different thread arrangements. This could be achieved through the manipulation of pointers after memory allocation or through the use kernel affinity interfaces.

7.2.3 Hardware Assisted Barrier Techniques

The POWER5 microprocessor provides a set of special purpose registers that are designed to allow the implementation of hardware assisted barrier routines. At present this facility is only exposed to the user through library calls unique to the AIX operating system.

In essence these Barrier Synchronisation Registers (BSRs) are a small region of memory that is shared among multiple CPUs. What makes them unusual is that they have an update and coherency mechanism that is faster than conventional cache line updates. To achieve this the size of the region is very small - of the order of a few words.

We have had some informal discussions with IBM's HPC group who have done some trials under AIX of these hardware assisted barriers. The results are most encouraging on small systems (2-4 way) and little short of impressive for larger (32 way and up) machines.

We plan to implement and benchmark BSR support under Linux and compare this against the best conventional, software based barriers we can find.

7.2.4 Physical address chasing Kprobes

At one point in our work we were faced with the question of how we find out if memory really is being allocated in the right place. Put another way, are we getting memory on the DCM on which the thread is running or the other one ? The question arose as we were seeing truly

bizzare results from STREAM - it turned out to be a problem with our synchronisation codes but this wasn't discovered until a novel solution to finding physical addresses was found[3].

Use was made of the "Kprobes" facility in the Linux kernel to attach a software breakpoint to the kernel routine responsible for the low level allocation of pages. Thus whenever a new page was touched our test code would be called.

The test code would output the current process ID and the physical address of the page allocated. We were able to limit the number of log messages to a managable level by only inserting the probe immediately prior to running our benchmark - hence we didn't fill the log with tens of thousands of extra entries that from background operation of the system. It was this requirement that motivated our used of the "kprobes" interface[4].

This data could then be post processed by matching the captured process ID with the known ID of the threads of our benchmark - these latter being recorded along with the requested CPU affinity information.

In the end we only spent an afternoon playing with this before realising that the problem we were chasing was in fact elsewhere. However we did get far enough along to have some confidence that this technique could usefully be applied in the future.

[3]Credit for this goes to Chris Yeoh from OzLabs who, prompted by a casual discussion in the morning put together a rough implementation before afternoon tea...

[4]An excellent tutorial on kprobes was presented by Dave Boutcher at the 2006 Ottawa Linux Symposium - [Boutcher 2006]

Additional Results

A.1 Additional Barrier/Lock Results

These results are provided as discussed in §6.2.5

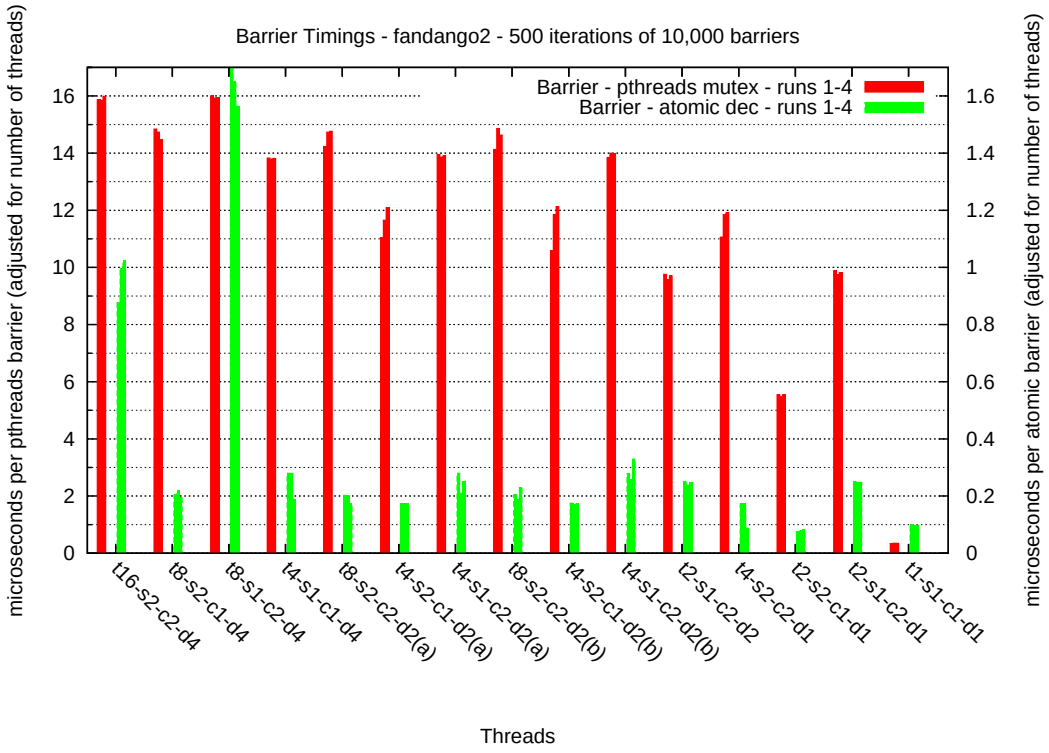

Figure A.1: Barrier results from *fandango2*

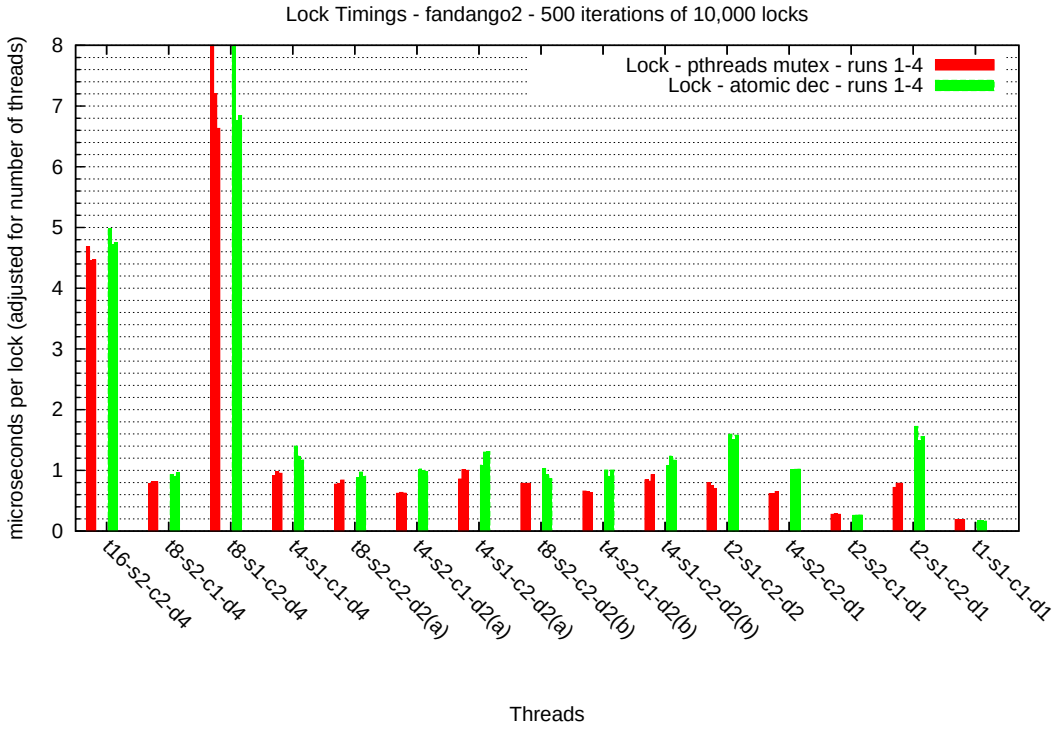

Figure A.2: Barrier results from *fandango2*

Appendix B

Other articles

The following references were read during the course of our research and the preparation of the Literature Review that preceeded it. While they are not directly quoted or cited, we would none the less like to acknowledge their influence.

- NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport [Antony et al. 2006]
- Performance evaluation of SMP architectures using the OpenMP NAS parallel benchmarks [Jean 2005]
- Superspeculative Microarchitecture for Beyond AD 2000 [Lipasti and Shen 1997]
- RCU vs. Locking Performance on Different CPUs [McKenney 2004b]
- Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels [McKenney 2004a]
- Towards Hard Realtime Response from the Linux Kernel on SMP Hardware [McKenney and Sarma 2005]
- Extending RCU for Realtime and Embedded Workloads [McKenney et al. 2006]
- One Billion Transistors, One Uniprocessor, One Chip [Patt et al. 1997]
- Superspeculative Microarchitecture for Beyond AD 2000 [Lipasti and Shen 1997]
- Trace Processors: Moving to Fourth-Generation Microarchitecture [Smith and Vajapeyam 1997]
- Microarchitecture Optimisations for Exploiting Memory-Level Parallelism [Chou et al. 2004]
- Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications [Spracklen et al. 2005]
- Store Memory-Level Parallelism Optimisations for Commercial Applications [Chou et al. 2004]
- Dynamic-sized Lockfree Data Structures [Herlihy et al. 2002]
- Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects [Michael 2004]
- Chip Multithreading Systems Need a New Operating System Scheduler [Fedorova et al. 2004]

Appendix C

Other Benchmarks

As noted in §2.3, we did not make use of Apex-map or perflab in our work this semester. They are included here largely for completeness.

C.1 Apex-map

C.1.1 Background

As introduced by Strohmaier and Shan [Strohmaier and Shan 2005] Apex-map is "...a novel synthetic memory access probe ... to measure global data access performance. Apex-map is designed based on parameterised concepts for temporal and spatial locality and generates a global data access stream according to specified levels of these measures of locality."

Apex-map allows fine grained and highly automated analysis and profiling of a memory system. It is able to run a series of tests each time changing the temporal or spacial locality of the data accesses, when plotted it yields a striking picture of memory throughput of the system in question.

The benchmark is available in both sequential and parallel versions, the latter intended to be used with `mpirun` to achieve its parallelism.

C.1.2 Benchmark Internals

The internal operation of apex-map is well described in [Strohmaier and Shan 2005]

> The synthetic memory access probe Apex-Map is designed based on parameterised concepts for temporal and spatial locality. It uses a blocked data access to a global array of size M to simulate the effects of spatial locality. The block length L is used as measure for spatial locality and L can take any value between 1 (single word access) and M. A non-uniform random selection of starting addresses for these blocks is used to simulate the effects of temporal locality. A power function distribution is selected as non-uniform random distribution and non-uniform random numbers X are generated based on uniform random numbers r with the generating function $X = r^{1/\alpha}$. The characteristic parameter α of the generating function is used as measure for temporal locality and can take values between 0 and 1. A value of $\alpha = 1$ generates uniform random numbers while small values of α generate random numbers centred towards the starting address of the global data array.

C.2 perflab

C.2.1 Background & Benchmark Operation

The "perflab" benchmarks are a suite of tools written by Tom Hart and Paul McKenney to allow the analysis of different locking methods. [Hart 2005] [Hart et al. 2006]

In their work, Hart, McKenney and Demke Brown were particularly focussed on memory reclamation performance. Accordingly the benchmark runs tests based on "Quiescent-state-based reclamation (QSBR), epoch-based reclamation (EBR), hazard pointer-based reclamation (HPBR) and reference counting"[Hart et al. 2006]. They provide user space implementations of each in the codes.

The general operation of the benchmark is described thus;

> In our tests, a parent thread creates N child threads, starts a timer, and stops the threads upon timer expiry. Child threads count the number of operations they perform, and the parent then calculates the average execution time per operation by dividing the duration of the test by the total number of operations. The CPU time is the execution time divided by the minimum of the number of threads and the number of processors. CPU time compensates for increasing numbers of CPUs, allowing us to focus on synchronisation overhead. Our tests report the average of five trials. [Hart et al. 2006]

C.2.2 Perflab in Our Work

Informal discussions with Paul McKenney suggest that the perflab tools will provide an ideal platform to understand performance of different locking regimes.

They are designed to be extensively scripted making automated testing practical. This ought to allow us to readily compare operation of different algorithms under different configurations of the POWER5 system.

Unfortunately there is essentially no documentation of the benchmarks and their use beyond references in Hart et. al's two papers. It was suggested to the author that documenting their use would be a useful contribution to the endeavour...

Also requiring some attention will be the source code which currently only has architecture specific atomic primitives defined for PPC32 and x86-32. Versions will need to be written to suit PPC64/CBE and UltraSPARC T1 should we elect to run on that platform.

In examining the perflab source code it is clear that the tools lend themselves to modification and extension. We may look at adding locking based on standard `pthread` primitives to provide a further reference point.

Bibliography

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The tera computer system. In *ACM International Conference on Supercomputing* (Amsterdam, Netherlands, June 1990). (p. 8)

ANTONY, J., JANES, P. P., AND RENDELL, A. P. 2006. NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. *2006 IEEE International Conference on High Performance Computing (to appear)*. (p. 59)

BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, L., FATOOHI, R., FINEBERG, S., FREDERICKSON, P., LASINSKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. 1994. The NAS parallel benchmarks. Technical Report RNR-94-007 (March), NASA Ames Research Center. (p. 5)

BAILEY, D., HARRIS, T., SAPHIR, W., VAN DER WIJNGAART, R., WOO, A., AND YARROW, M. 1995. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020 (Dec.), NASA Ames Research Center. (p. 5)

BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAM, V., AND WEERATUNGA, S. K. 1991. The NAS parallel benchmarks. *International Journal of Supercomputer Applications 5*, 3, 63 – 73. (p. 5)

BLEMINGS, H. 2006. A COMP6720 project: Multithreading issues on contemporary powerpc microprocessors. Available: `http://escience.anu.edu.au/project/06S1/HughBlemings/HughBlemingsFinalReport.pdf`. (pp. 1, 5, 53)

BORKENHAGEN, J. M., EICKMEYER, R. J., KALLA, R. N., AND KUNKEL, S. R. 2000. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development 44*, 6 (November). (pp. 7, 8)

BOUTCHER, D. 2006. Practical kernel debugging with kprobes. In *Ottawa Linux Symposium 2006* (Ottawa, Canada, July 2006). http://www-users.cs.umn.edu/ boutcher/kprobes/. (p. 55)

BREEDS, T. 2006. Methodologies for network-level optimisation of cluster computers. Master's thesis, Australian National University. (p. 24)

CHOU, Y., FAHS, B., AND ABRAHAM, S. 2004. Microarchitecutre optimizations for exploiting memory-level parallelism. (p. 59)

CHOU, Y., SPRACKLEN, L., AND ABRAHAM, S. 2004. Store memory-level parallelism optimizations for commercial applications. (p. 59)

DEMONE, P. 2004. Sizing up the super heavyweights. Available: `http://www.realworldtech.com/page.cfm?ArticleID=RWT100404214638&p=1`. (p. 12)

FEDOROVA, A., SMALL, C., NUSSBAUM, D., AND SELTZER, M. 2004. Chip multithreading systems need a new operating system scheduler. In *11th ACM SIGOPS European Workshop* (Sept. 2004). ACM. (p. 59)

HAMMOND, L., NAYFEH, B. A., AND OLUKOTUN, K. 1997. A single-chip multiprocessor. (p. 7)

HART, T. E. 2005. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto. (p. 62)

HART, T. E., MCKENNEY, P. E., AND DEMKE BROWN, A. 2006. Making lockless synchronization fast: Performance implications of memory reclamation. In *2006 International Parallel and Distributed Processing Symposium (IPDPS 2006)* (March 2006). (p. 62)

HERLIHY, M., LUCHANGCO, V., MARTIN, P., AND MOIR, M. 2002. Dynamic-sized lockfree data structures. (p. 59)

JEAN, N. 2005. Performance evaluation of SMP architectures using the OpenMP NAS parallel benchmarks. Master's thesis, Australian National University. (pp. 5, 59)

KALLA, R., SINHAROY, B., AND TENDLER, J. 2004. IBM POWER5 chip: a dual-core multithreaded processor. *IEEE Micro 24*, 2. (p. 10)

LIPASTI, M. H. AND SHEN, J. P. 1997. Superspeculative microarchitecture for beyond ad 2000. (p. 59)

MATHIS, H. M., MERICAS, A. E., MCCALPIN, J. D., EICKEMEYER, R. J., AND KUNKEL, S. R. 2005. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM Journal of Research and Development 49*, 4. (p. 51)

MCCALPIN, J. D. 1995. Sustainable memory bandwidth in high performance computers. Available: `http://home.austin.rr.com/mccalpin/papers/bandwidth/bandwidth.html`. (pp. 3, 42)

MCCALPIN, J. D. 2006. STREAM: Sustainable memory bandwidth in high performance computers. Available: `http://www.cs.virginia.edu/stream`. (pp. 5, 33)

MCKENNEY, P. E. 2004a. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University. (p. 59)

MCKENNEY, P. E. 2004b. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004). Available: `http://www.linux.org.au/conf/2004/abstracts.html\#90` `http://www.rdrop.com/users/paulmck/rclock/lockperf.2004.01.17a.pdf`. (p. 59)

MCKENNEY, P. E. AND SARMA, D. 2005. Towards hard realtime response from the linux kernel on SMP hardware. In *linux.conf.au 2005* (Canberra, Australia, April 2005). Available: `http://www.rdrop.com/users/paulmck/RCU/realtimeRCU.2005.04.23a.pdf`. (p. 59)

MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. 2006. Extending rcu for realtime and embedded workloads. In *Ottawa Linux Symposium 2006* (Ottawa, Canada, July 2006). To appear: `http://www.rdrop.com/users/paulmck/RCU/`. (p. 59)

MICHAEL, M. M. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *Transactions on Parallel and Distributed Systems 15*, 6 (June). (p. 59)

MIKES, N. 2004. Power to the people. Available: `http://www-128.ibm.com/developerworks/library/l-powhist` [Viewed March 28, 2006]. (p. 9)

OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K.-Y. 1996. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Parallel Languages and Operating Systems* (Oct. 1996). (p. 7)

PAPERMASTER, M., DINKJIAN, R., MAYFIELD, M., LENK, P., CIARFELLA, B., O'CONNELL, F., AND DUPONT, R. 1998. Power3: Next generation 64-bit powerpc processor design. Available: `http://www-03.ibm.com/servers/eserver/pseries/hardware/whitepapers/power3wp.html` [Viewed May 7, 2006]. (pp. 9, 12)

PATT, Y. N., PATEL, S. J., EVERS, M., FRIENDLY, D. H., AND STARK, J. 1997. One billion transistors, one uniprocessor, one chip. (p. 59)

PEASE, R. A. 1991. *Troubleshooting Analog Circuits*. The EDN series for design engineers. Butterworth-Heinemann. (p. 53)

SINHAROY, B., KALLA, R. N., TENDLER, J. M., EICKEMEYER, R. J., AND JOYNER, J. B. 2005. POWER5 system microarchitecture. *IBM Journal of Research and Development 49*, 4. Available: `http://www.research.ibm.com/journal/rd/494/sinharoy.pdf`[Viewed March 30, 2006]. (pp. 8, 10, 11, 12, 42, 51)

SMITH, J. E. AND VAJAPEYAM, S. 1997. Trace processors: Moving to fourth-generation microarchitectures. (p. 59)

SPRACKLEN, L. AND ABRAHAM, S. G. 2005. Chip multithreading: Opportunities and challenges. *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*. (p. 8)

SPRACKLEN, L., CHOU, Y., AND ABRAHAM, S. G. 2005. Effective instruction prefetching in chip multiprocessors for modern commercial applications. *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*. (p. 59)

STROHMAIER, E. AND SHAN, H. 2005. Apex-map: A global data access benchmar to analyze HPC systems and parallel programming paradigms. (p. 61)

TENDLER, J. M., DOBSON, S., FIELDS, S., LE, H., AND SINHAROY, B. 2002. Power4 system microarchitecture. *IBM Journal of Research and Development 46*, 1. (pp. 7, 9, 12)

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Amsterdam, Netherlands, June 1995). (pp. 7, 9)