Performance Models for Electronic Structure Methods on Modern Computer Architectures

Joseph Antony

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

May 2009

Typeset in Times by $\TeX$ and $\LaTeX\,2_\varepsilon$.

Except where otherwise indicated, this thesis is my own original work.

Joseph Antony
29 May 2009

Acknowledgements

I am indebted to my supervisor Alistair Rendell for his encouragement and support all throughout my candidature. My thanks also goes to Peter Strazdins and Andrey Bliznyuk, my co-supervisors. A warm thanks goes out to Ben Evans and the NCI National Facility team both as a PhD candidate and as a staff member. I'd like to take this opportunity to thank Mike Frisch and team Gaussian in Wallingford CT for not only sponsoring the ARC linkage grant that led to this thesis, but for all their help, advise and encouragement over the years.

I've been enriched by my interactions with past and present students, staff at ANU Computer Science; I'd like to especially thank:
Rui Yang, Pete Perrapong Janes, Jin Wong, Warren Armstrong, Jie Cai, V. Ganesh, Josh Milthorpe, Yang Xi, Arrin Daley. A special mention and thanks goes to Andrew Over and Bill Clarke. Bill had informed me about the state-of-the-art in full machine simulation, C++ run-time intricacies and Roman-era archaeological digs in Syria. Discussions with Andrew often revolved around SG1 and the UltraSPARC IIICu while Armin van Buuren was in 'A State of Trance'. Joint work with Pete Janes enabled the development of the LPM model in Chapter 3, often sketched out on Saturdays after Pete's fishing trips from Lake Burley Griffin. Rui Yang deserves an special mention for all his help throughout my thesis years, both as a joint collaborator (Chapter 5) and for his feedback and input often over coffee in his office.

A big vote of thanks goes to:
SGI (Todd Churchwood, Anthony David, Roberto Gomperts);
IBM (Hugh Blemmings, James Kelly);
IBM academic program (for access to compilers and BLAS libraries);
Apple ADC (Tess Collins);
ANU DCS (Pascal Vuylsteker, Steve Blackburne, Eric McCreath, Bob Edwards, James Fellows, Steve Hanley, Hugh Fisher, Robin Garner, Chris Johnson, Henry Gardner);
ANUSF (Jason Ozolins, Rika Kobayashi, Robert Davy, Jonathan McCabe, David Singleton, Judy Jenkinson, Margaret Kahn, Robin Humble, Bob Gingold);
ANU DoI (Allan Williams, Markus Buuchorn, Andrew Wellington, Doug Moncur);
Cray Australia (Lindsay Hood);
Stuart Watson and Richard Walker.

On a personal note: a special thanks to ‘F & M Inc.’

This work was made possible as a result of funding from the Australian Research Council in conjunction with Gaussian Inc. and Sun Microsystems Inc. under ARC Linkage Grant LP0347178.

Abstract

Electronic structure codes are computationally intensive scientific applications used to probe and elucidate chemical processes at an atomic level. Maximizing the performance of these applications on any given hardware platform is vital in order to facilitate larger and more accurate computations. An important part of this endeavor is the development of protocols for measuring performance, and models to describe that performance as a function of system architecture. This thesis makes contributions in both areas, with a focus on shared memory parallel computer architectures and the Gaussian electronic structure code.

Shared memory parallel computer systems are increasingly important as hardware manufacturers are unable to extract performance improvements by increasing clock frequencies. Instead the emphasis is on using multi-core processors to provide higher performance. These processor chips generally have complex cache hierarchies, and may be coupled together in multi-socket systems which exhibit highly non-uniform memory access (NUMA) characteristics. This work seeks to understand how cache characteristics and memory/thread placement affects the performance of electronic structure codes, and to develop performance models that can be used to describe and predict code performance by accounting for these effects.

A protocol for performing memory and thread placement experiments on NUMA systems is presented and its implementation under both the Solaris and Linux operating systems is discussed. A placement distribution model is proposed and subsequently used to guide both memory/thread placement experiments and as an aid in the analysis of results obtained from experiments.

In order to describe single threaded performance as a function of cache blocking a simple linear performance model is investigated for use when computing the electron repulsion integrals that lie at the heart of virtually all electronic structure methods. A parametric cache variation study is performed. This is achieved by combining parameters obtained for the linear performance model on existing hardware, with instruction and cache miss counts obtained by simulation, and predictions are made of performance as a function of cache architecture.

Extension of the linear performance model to describe multi-threaded performance on complex NUMA architectures is discussed and investigated experimentally. Use of dynamic page migration to improve locality is also considered.

Finally the use of large scale electronic structure calculations is demonstrated in a series of calculations aiming to study the charge distribution for a single positive ion solvated within a shell of water molecules of increasing size.

Contents

List of Papers

1. J. ANTONY, M. J. FRISCH, AND A. P. RENDELL
   Modelling the Performance of Gaussian Chemistry Code on x86 Architectures
   Proceedings of HPSC, 6 – 10 March 2006, Hanoi, Vietnam
   `http://www.springer.com/math/cse/book/978-3-540-79408-0`

2. J. ANTONY, P. P. JANES, AND A. P. RENDELL
   Exploring Thread and Memory Placement on NUMA Architectures:
   Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport
   Proceedings of IEEE HiPC 2006, 18 – 21 December 2006, Bangalore, India
   `http://dx.doi.org/10.1007/11945918_35`

3. R. YANG, J. ANTONY, P. P. JANES, AND A. P. RENDELL
   Memory and Thread Placement Effects as a Function of Cache Usage:
   A Study of the Gaussian Chemistry Code on the SunFire X4600 M2
   Proceedings of IEEE I-SPAN 2008, 7 – 9 May 2008, Sydney, Australia
   `http://doi.ieeecomputersociety.org/10.1109/I-SPAN.2008.13`

4. A. P. RENDELL, J. ANTONY, W. ARMSTRONG, P. P. JANES, AND R. YANG
   Building Fast, Reliable, and Adaptive Software for Computational science
   Proceedings of SciDAC 2008, 13 – 17 July 2008, Seattle, Washington, USA
   `http://stacks.iop.org/1742-6596/125/012015`

5. R. YANG, J. ANTONY, AND A. P. RENDELL
   A Simple Performance Model for Multithreaded Applications
   Executing on Non-Uniform Memory Access Computers
   Proceedings of IEEE HPCC 2009, 25 – 27 June 2009, Seoul, Korea
   `http://dx.doi.org/10.1109/HPCC.2009.39`

6. R. YANG, J. ANTONY, AND A. P. RENDELL
   Effective Use of Dynamic Page Migration on NUMA Platforms
   Proceedings of IEEE I-SPAN 2009, 14 – 16 December 2009, Kaohsiung, Taiwan
   `http://doi.ieeecomputersociety.org/10.1109/I-SPAN.2009.127`

List of Presentations

1. *Linear Scaling Algorithms and OpenMP*
   October 2003, Student Forum, APAC Conference, Gold Coast, Australia
2. *Development of Efficient Multi-Threaded Applications for Modern Shared Memory Architectures*
   February 2005, Seminar, Department of Computer Science, ANU, Canberra, Australia
3. *Performance modelling of cache blocking on integral evaluation in Gaussian*
   August 2005, Seminar, Gaussian Inc., Wallingford, CT, U.S.A
4. *Some aspects of Gaussian's performance on the Intel Itanium Platform*
   August 2005, Seminar, Intel Compiler Group, Santa Clara, CA, U.S.A
5. *Performance Modelling of Scientific Applications on HPC Platforms*
   October 2005, Student Forum, APAC Conference, Gold Coast, Australia
6. *Gaussian Performance Modelling and NUMA Thread & Memory Placement*
   4 March 2006, Seminar, Sun Asia Pacific Science and Technology Center, Singapore
7. *Modelling the Performance of the Gaussian Computational Chemistry Code on the x86 Architecture*
   6 – 10 March 2006, International Conference on High Performance Scientific Computing, Hanoi, Vietnam
8. *An update on: Gaussian Performance Modelling*
   7 June 2006, Seminar, Department of Computer Science, ANU, Canberra, Australia
9. *Exploring Thread and Memory Placement on NUMA Architectures*
   18 – 21 December 2006, IEEE International Conference on High Performance Computing, Bangalore, India
10. *NUMALink, HyperTransport and Gaussian*
    8 April 2008, Seminar, ANU Supercomputer Facility, ANU, Canberra, Australia
11. *Effective Use Of Dynamic Page Migration on NUMA Platforms*
    14 – 16 December 2009, IEEE I-SPAN Conference, Kaohsiung, Taiwan, R.O.C.

List of Tables

List of Figures

List of Abbreviations

CGTO	Contracted GTO
DCache	Data Cache
DFT	Density Functional Theory
DLP	Data Level Parallelism
ERI	Electron Repulsion Integrals
FIFO	First-In First-Out
FMM	Fast Multipole Method
GTO	Gaussian Type Orbital
HF	Hartree-Fock
HWPC	Hardware Performance Counters
ICache	Instruction Cache
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
K.E.	Kinetic Energy
LRU	Least Recently Used
MD	McMurchie-Davidson
MO	Molecular Orbital
MPA	Mulliken Population Analysis
NPA	Natural Population Analysis
NUMA	Non-Uniform Memory Access
PDM	Placement Distribution Model
PGTO	Primitive GTO
RDF	Radial Distribution Function
SCF	Self-Consistent Field
TLB	Translation-Lookaside Buffer
TLP	Thread Level Parallelism
UMA	Uniforma Memory Access

Chapter 1

Introduction

Obtaining good performance from scientific applications which execute on parallel, shared memory, multi-core systems is a non-trivial exercise in multivariate optimization arising from the complexities of on-chip memory hierarchies, influence of architectural techniques designed to boost on-chip performance and inherent latencies imposed by underlying hardware interconnect. Further, the commoditization of NUMA architectures coupled with multi-core processors have now enabled end-users access to systems with high core counts and large shared memory systems, previously accessible only at dedicated supercomputing facilities. The confluence of multi-core chip architectures and NUMA interconnect technologies exacerbate the already complex task of single threaded performance tuning. Thus, the development of NUMA performance analysis methodologies and allied performance models which can predict application performance as a function of system architecture is timely.

Electronic structure codes are computationally intensive scientific applications routinely used by chemists, biologists, physicists and others for the ab-initio (i.e. first-principles) study of matter at the atomic level. In this work we use the Gaussian electronic structure code [89], which is currently one of the most widely used shared-memory parallel electronic structure codes. We focus on its implementation on current commodity, multi-core, NUMA hardware platforms. Often the execution times, for these codes, may take in the order of weeks on a single processor system. Moreover, grand challenge problems in the computational chemistry area requires the development of linear scaling algorithms that map efficiently onto the underlying parallel hardware. In both cases it is important that the underlying code performs efficiently on the processor architecture being used.

1.1 Original Contributions

This thesis makes the following original contributions –

- Performance characterization of commodity NUMA hardware platforms using latency (lmbench), bandwidth (Stream) and BLAS kernels (Level 1, 2, 3) as a function of thread and memory placement.

- Performance characterisation of the PRISM Electron Repulsion Integral (ERI) evaluation algorithm used in the Gaussian electronic structure code in terms of platform independent and dependent components.
- Use of a linear performance model for ERI evaluation. Validation of this model across four microprocessor architectures and seven individual microprocessors.
- Use of the linear performance model with functional cache simulation to predict the performance of the PRISM ERI algorithm as a function of cache configuration.
- Identification of key cache parameters which influences the performance of ERI evaluation on cache based architectures. Characterisation of the cache miss components which limit PRISM's performance and proposals to improve performance.
- The extension of the single-threaded LPM to account for NUMA and thread, memory placement. The extension is validated and used to assess the performance of parallel Gaussian.
- An assessment of the use of dynamic page migration and node interleaving to improve data locality and reduce contention in the Gaussian code.
- Characterisation and analysis of the atomic charges obtained from density functional methods for a solvated potassium ion in a large water complex.
- Demonstration of the basis set sensitivity if atomic charges are computed using density functional wavefunctions.

1.2 Organization of Thesis

The structure of the thesis is as follows: Chapter 2 is background material relevant to the chapters that follow. It presents an overview of microprocessors and interconnects; performance evaluation and electronic structure methods. Chapter 3 focuses on thread and memory placement techniques on Non-Uniform Memory Access Systems (NUMA) and presents a placement distribution model (PDM) to aid performance characterisation of NUMA architectures. Chapter 4 introduces a simple linear performance model (LPM) to describe the cost of computation for electron repulsion integrals. It outlines the PRISM ERI algorithm, evaluates the use of the LPM for ERI evaluation and combines use of the LPM with functional cache simulation in order to predict performance on non-existent cache architectures. Chapter 5 uses the PDM and the LPM to model the effects of thread and memory placement, cache effects on the Gaussian code. It also extends the LPM to account for multi-core and NUMA systems and considers the use of dynamic page migration and node interleaving within the Gaussian code. Chapter 6 is an applications chapter which examines chemical charges obtained from a set of water cluster complexes. Chapter 7 concludes the thesis and presents directions for future work.

Chapter 2

Background

Modern microprocessors are complex marvels of silicon engineering. In 2009 the most complex processor contains two billion transistors [257], operates at 4Ghz [60, 155] and incorporates large on-chip caches [233]. These microprocessors can execute on the order of several billion instructions in one second [163], they employ multiple levels of cache memory and a host of micro-architectural techniques to ensure the processor does not stall. In turn this complexity makes the analysis and modelling of application performance very difficult.

This thesis aims to construct performance models for electronic structure codes. In this Chapter a review of microprocessors technologies, and performance evaluation techniques are given. The chapter then introduces the electronic structure methods used in this thesis namely the Hartree-Fock method and Density Functional theory methods.

2.1 Microprocessors, Caches, Interconnects

This section reviews microprocessor concepts – the CPU, cache memory, multiprocessor systems and hardware performance counters. For in-depth discussions of computer organization and architecture, Hennessy and Patterson's texts [113, 209] are recommended.

A microprocessor is the physical embodiment of a Turing machine [283] which has the features of a Central Processing Unit (CPU) on an integrated circuit. The CPU operates on instructions defined by an Instruction Set Architecture (ISA). It works in a loop where instructions and data undergo the following four steps: fetch, decode, execute, retire/write-back. Instructions are fetched from main memory then decoded, leading to data operands of the instruction being fetched. Once the required data operands are on-chip, the instruction is executed by the CPU and the instruction is retired i.e. the resulting data from the execution is stored back into main-memory. If data and instructions are both stored in main memory, the system is referred to as being a von Neumann architecture [42]. Whereas if data and instructions are stored separately from each other, it is referred to as a Harvard architecture [219].

The von Neumann architecture (Figure 2.1) imposes a separation between the CPU and

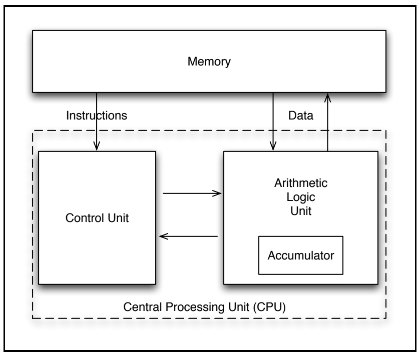


Figure 2.1: The von Neumann Architecture

memory i.e. data needs to be explicitly moved from main memory into the on-chip registers for use. This leads to the von Neumann bottleneck where the limiting factor becomes how fast data can be moved to and from memory. This limitation led to the use of interposing cache memory [107] between main memory and the CPU. In effect, caches help circumvent the von Neumann bottleneck as the vast majority of memory and instructions have locality [66] both in space and time. This locality means a given working set could reside in fast cache memory and thus mitigating the time taken to fetch data from main memory. The use of cache is also one of the means of shielding the CPU from what is termed the "memory wall" [181], where the rate of increase in microprocessor speed/innovation outstrips improvements made in Dynamic Random Access Memory (DRAM) technology. This leads to DRAM speeds impacting CPU performance as the faster CPU (operating on the order of Ghz) is stalled waiting for memory requests to be satisfied from the slower DRAM (operating in the order of hundreds of MHz). In addition to employing caches, processors also use a host of other micro-architectural features to extract instruction and data parallelism as well as exploiting thread-level parallelism. All these concepts, which are critical to the efficient operation of a modern microprocessor are briefly outlined in the following sections.

2.1.1 On-Chip Parallelism

In order to increase performance modern microprocessors extract parallelism from the instruction stream using a range of techniques which can be classified as, Data Level Parallelism (DLP) , Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP).

2.1.1.1 Data Level Parallelism

Data level parallelism [114] aims to extract parallelism from the data quantities being operated on by have a single operation manipulate a set of data items in parallel. Single-Instruction, Multiple Data (SIMD) operations are an example of this. Microprocessors are able to operate on data items in parallel using extensions to their ISA and two examples are Intel's SSE [259] for the x86 ISA and IBM's AltiVec [193] for the PowerPC ISA.

2.1.1.2 Instruction Level Parallelism

Instruction Level Parallelism (ILP) are a host of techniques which attempt to increase the number of instructions that can be in flight at any given time so as to increase the utilization of on-chip hardware resources. Instructions can either be retired 'in-order' or 'out-of-order' and most processors use the former. These techniques include:

Super scalar execution Multiple execution units execute instructions in parallel, subject to data dependencies [313]. Multiple instructions are decoded and sent to a dispatcher, which queues instructions and releases groups of instructions for execution once it has determined there are no data dependencies in the instructions being dispatched to on-chip execution units. This in effect allows a microprocessor to execute more than one instruction per cycle.

Instruction Pipelining Instruction pipelining divides the processing of an instruction into multiple independent stages [222]. Once a task completes its initial step, it gets fed into the next stage of the pipeline. This process allows multiple instructions to be overlapped.

Out-of-order Execution Out-of-order execution [251] is a technique which allows a processor to continue executing instructions without data dependencies, out-of-order from their original program order. This in turn allows the processor to: (a) permit better use of multiple functional units i.e. increasing throughput; and (b) avoiding expensive stalls. The processor fetches instructions, dispatches these to an instruction queue. Instructions are ready for issue once their inputs are ready. At this point, instructions which do not have dependencies can be issued out-of-order with respect to the instruction stream. Once instructions have been issued, their results are retired.

Branch prediction and Speculative execution The execution of a branch in a pipelined microprocessor leads to a pipeline bubble, as the branch requires new instructions to be fetched by the instruction stage. To avoid a pipeline bubble, the microprocessor executes one of the paths of a branch, based on branch prediction [180, 314] information without prior knowledge of the outcome of the branch, this is known as speculative execution [151, 269]. If the branch was guessed correctly, there is no delay in the instruction fetch stage. If it was incorrect, the pipeline needs to be flushed and the branch re-executed. By using branch prediction, the number of instructions available for scheduling by the dispatcher increase and allows for useful work to be carried out prior to the branch being resolved.

2.1.1.3 Thread Level Parallelism

Thread Level Parallelism (TLP) techniques refer to the use of multiple threads of execution to better facilitate on-chip resource utilization. One TLP approach is to have a microprocessor execute a set of multiple threads and when any one thread experiences a long latency event like a stall when data is retrieved from main memory, the microprocessor performs a context switch to another thread which is ready to execute [149, 202].

HyperThreading [150, 173] is a special case of TLP implemented per core in a microprocessor. A processor's on-chip hardware resources which stores architectural state (i.e. the current program counter and set of registers) is divided amongst two or more threads of execution. Execution resources (e.g. Floating Point Units) are not divided amongst the threads. When an executing thread experiences a stall, the processor core switches to executing the other thread.

The UltraSPARC T1 Niagara [149] chip uses a different strategy to exploit TLP. The T1 has eight cores each of which can support eight threads of execution, allowing for a total of 64 threads per CPU. Each core acts as a barrel processor which selects instructions to execute amongst the eight threads it has assigned to it. If a thread experiences a stall event it is made unavailable for execution which the stall is resolved, execution continues using instructions from other threads.

2.1.2 Cache Memory Structure

Cache memory [107, 250] is fast, intermediate memory interposed between the microprocessor and main memory. Caches are designed to be faster than the backing storage it fronts which leads to a higher cost per bit. There is no functional difference between a CPU directly accessing memory and going via a cache. This requires the cache to implement some form of consistency model to ensure that the results obtained are identical without the presence of cache memory. Cache consistency is discussed in detail in Section 2.1.4.

The working set is defined as the subset of pages from main memory that is mapped into cache for use by the application executing on the microprocessor. A given cache needs to be large enough to encompass the working set of both instructions and data being used by an application, whereby there is an increased likelihood of both the CPU's reads and writes being satisfied from cache.

Figure 2.2 represents the memory hierarchy of most general purpose microprocessors. Main memory is composed of banks of DRAM (Dynamic Random Access Memory) [59],

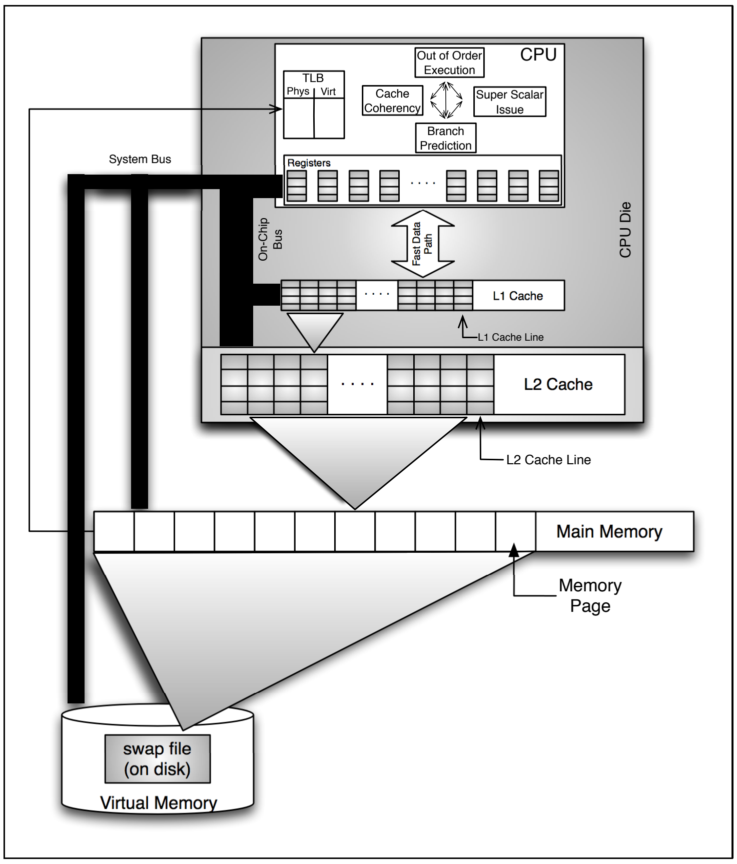


Figure 2.2: The Memory Hierarchy Architecture. Adapted from [32].

which take on the order of 300 CPU cycles to read. All loads and stores issued by the CPU transit via the cache. Caches are part of a storage hierarchy for CPUs, as seen in Figure 2.2. Here, it can be seen that caches are organized in tiers i.e. L2 (Level 2) cache and L1 (Level 1) cache. Pages in main memory (pages are typically 4K or 8K in size and main memory is in the order of Gigabytes of DRAM) are mapped onto locations within the highest level of cache (in this case L2), which can range from 1MB to 30MB [183, 257]. Data from L2 then gets mapped into L1 (typically on the order of 32Kb to 64Kb), which then get forwarded into registers on the CPU for execution. This spread of cache structures is done to speed-up access to data, exploit locality and importantly to optimize most of the on-chip memory as caches occupy up to half of the surface area in a CPU die.

2.1.2.1 Capturing Locality

Caches are beneficial if access to program data exhibits ”locality” [66]. Three different types of locality are often defined [131],

Temporal Locality Temporal locality refers frequent use of the same data elements and the accesses are close together in time e.g. a sub-block of an array that is operated on in each iteration of a loop.

Spatial Locality Spatial locality refers to detecting and responding to streams of data and/or instructions which are spatially close in memory to each other, e.g. linear data stored in an array being operated on within a loop nest.

Algorithmic Locality Algorithmic locality [131] is more subtle. It occurs when an algorithm repeatedly executes specific code or repeatedly accesses specific data blocks and these blocks are distributed widely throughout main memory. While the overall result is program behavior which is predictable, its repeatability cannot be detected by on-chip locality detection mechanisms [1]. It is often difficult to detect using current cache controlling hardware which usually relies on detecting regular strided access patterns. Also, increasing the size of on-chip caches by a factor of two or more does not lend to capturing algorithmic locality, as the code and data blocks involved often exceeds the capacity of the on-chip cache. Some examples of applications which perform repeated operations on large datasets are computer graphics applications, computer simulators and quantum chemistry applications. In order to exploit this type of locality compiler and user-level intervention is needed, usually by carefully placed pre-fetch instructions. Pre-fetch instructions are those which request the memory subsystem to initiate a fetch of cachelines prior to their use. If timed correctly, it can result in algorithms that can tolerate long latencies.

2.1.3 Primary Cache Characteristics

As mentioned caches store copies of data from a higher level of the memory hierarchy. This data is arranged in cachelines or blocks representing consecutive main memory addresses. The defining features of any cache organization are [32, 107],

Cache size Cache capacity is the maximum usable capacity of the cache.

Line size Line size is the smallest unit of data transferred either between caches or into a cache.

Associativity As cache memories are two orders of magnitude smaller than main memory, mapping schemes are used to map main memory locations into cache. Cache associativity refers to the various options a replacement scheme has for locating a cacheline in the cache. This is further detailed below.

[1] On-chip logic detects strides in data or instruction streams and initiates prefetching as per the nature of the stride [61].

2.1.3.1 Cache Associativity

As mentioned above cache associativity refers to the various options a replacement scheme has in replacing cachelines in cache with lines from main memory. These can be broadly categorized under the following three headings, **Direct mapped**: A Direct mapped cache is one where a main memory address maps to only one cacheline within the cache. This implies multiple locations in main memory can map onto the same cacheline.

Fully associative: A Fully associative cache is one where a main memory address can map onto any cacheline within the cache. When an address is requested from the cache, there is a directory look-up operation to find the required line. Thus a fully associative cache behaves as a hardware database [131] i.e. a cacheline can be placed at any cacheline storage location. Searching the cache for a given cacheline requires checking each cacheline being stored in cache. As in the case of the direct mapped caches, cache lines from memory can get mapped onto the same location, but conflicts are much less likely.

Set associative: Set associate caches behave as fully associative caches with the modification that a cacheline can reside in one of a fixed set of cacheline locations. A set of bits from a cacheline's address are used to index a cache directory, and for each address there are N sub-caches into which the cacheline could potentially reside. Thus a 4-way set associative cache can place a cacheline in one of 4 different locations in cache.

2.1.3.2 Types of Caches

On modern microprocessors there three independent caches – Instruction cache (ICache), the Data cache (DCache) and the Translation-Lookaside Buffer (TLB) cache. Separate L1 ICache and L1 DCaches is reminiscent of the Harvard architecture, where instructions to be fed to the CPU core are store in the L1 ICache and data in the L1 DCache. The L1 ICache is read-only, whereas the L1 DCache is read-write. The L2 cache is a unified cache which stores both data and instructions. Its function is similar to memory in the von Neumann architecture. The TLB cache is used in caching translations between real and virtual addresses; this is used to speed up translations of mappings between memory locations referenced in a program versus its actual physical location.

2.1.3.3 Cache Replacement Policies

Caches implement some form of replacement policy to determine where newly fetched data from memory will reside in cache. Some of these policies are Least Recently Used (LRU), First-In First-Out (FIFO) and Random replacement [107]. LRU has the advantage that the most frequently used data will remain in cache; however it is more complex to implement than

Random replacement as the cache's associativity increases. FIFO caches are often used in embedded and resource constrained microprocessors [249].

2.1.3.4 Cache misses and Cacheline eviction

If a cacheline is required but is not present in the cache, a cache miss is said to occur. Two types of cache misses are usually identified and occur when cachelines need to be read or written to i.e read misses and write misses respectively. On a read miss, the L1 and L2 caches are searched for the cacheline. A cache hit results if it is found, else the cache miss is then forwarded to the memory sub-system to fetch the required cacheline. On a write-miss the cache has two options depending on its implementation, either it allocates a cacheline (write allocate) or it by-passes the cache and goes to main-memory (no write allocate). Usually write back caches implement a write allocate for write-misses and write through caches will be no write allocate. Cache misses can be classified [72, 133] as

Cold misses Cold misses or compulsory misses occur the first time data is referenced by the CPU and needs to be fetched from main memory.

Capacity misses Capacity misses occur due to the limited size of the cache i.e. data being referenced by the CPU exceeds the capacity of the cache resulting in lines being evicted from cache.

Conflict misses Conflict misses occur when references are made to cachelines which map onto the same set (i.e. in set-associative caches) and causes cachelines already resident in the cache to be evicted. If the evicted line is re-referenced within some short period of time, it constitutes a conflict miss.

The use and re-use of cachelines is driven by load and store instructions issued and executed by the CPU. This leads to cachelines being selected for eviction from cache based on the eviction policy e.g. LRU. Once a cacheline has been selected for eviction, it needs to be migrated from cache into main memory. Two policies are used to determine how this migration occurs to main memory, either a write-through or a write-back policy. Once data is moved from registers into the L1 data cache, a write-through policy will push the cacheline back into main memory, whereas a write-back policy will commit the cacheline into L1 or L2 and only evicts the line from cache if there is a new cacheline to be brought into the cache. L1 DCaches are designed to be inclusive or exclusive of the L2 cache, where an inclusive L1 DCache will have cache lines that can also exist in the L2. An exclusive L1 DCache will not have cache lines which are resident in the L2 cache.

2.1.4 Cache Coherency and Memory Consistency in Multiprocessor Systems

Shared-memory processing environments are created by using groups of microprocessors which are able to access global shared memory via some interconnect. A key driving factor in the evolution of shared-memory multiprocessors was the observation that multiple levels of cache reduced the memory bandwidth requirements of each processor [99], allowing for multiple processors to use the same memory bus [113]. Introduction of caches into multiprocessor systems introduces two problems namely, (a) how does a processor know when a cacheline of interest to that processor is being used by another processor and (b) what a processor can infer about operations of a remote processor by observing the ordering of data reads and writes of the remote processor to main memory.

Problem (a) is addressed by a cache coherency scheme and problem (b) relates to the particular memory consistency model which has been implemented.

Cache coherency is defined by Jacob [131] "In the presence of a cache, reads and writes behave (to a first order) no differently that if the cache were not present". As a cache is interposed between a CPU and memory it needs to ensure the processors' view of memory is consistent with respect to data being cached i.e. data written to in a cache is guaranteed to be committed to memory and thus visible to all other processors. To enforce cache consistency, a cache coherency policy is defined which ensures that there is one globally unique, well-defined value associated with any given memory location.

A memory consistency model for a shared-memory multiprocessing system, "is a formal specification of how the memory system will appear to the programmer" [6]. The memory consistency model determines how a store operation in one thread of execution is made visible to the load operation of another thread, as well as implications about other load and store operations in both threads. All modern microprocessors implement some variation of a relaxed consistency model which allows for reads and writes to complete out-of-order, but also provides synchronization primitives to enforce ordering between multiple threads of execution [113]. This involves the use of hardware memory control and atomicity preserving operations such as: `fence` on the x86 ISA, `sync` on the POWER ISA, and `membar` on the SPARC ISA.

2.1.4.1 Cache Coherency Protocols

A cache coherency protocol enables for the physical implementation of cache consistency. Consistency needs to be maintained with the backing store as well as with other clients in a multi-processor setup. In single processor environments the write-back and write-through policies ensure the cache is consistent with its backing store, but in a multi-processor setup a

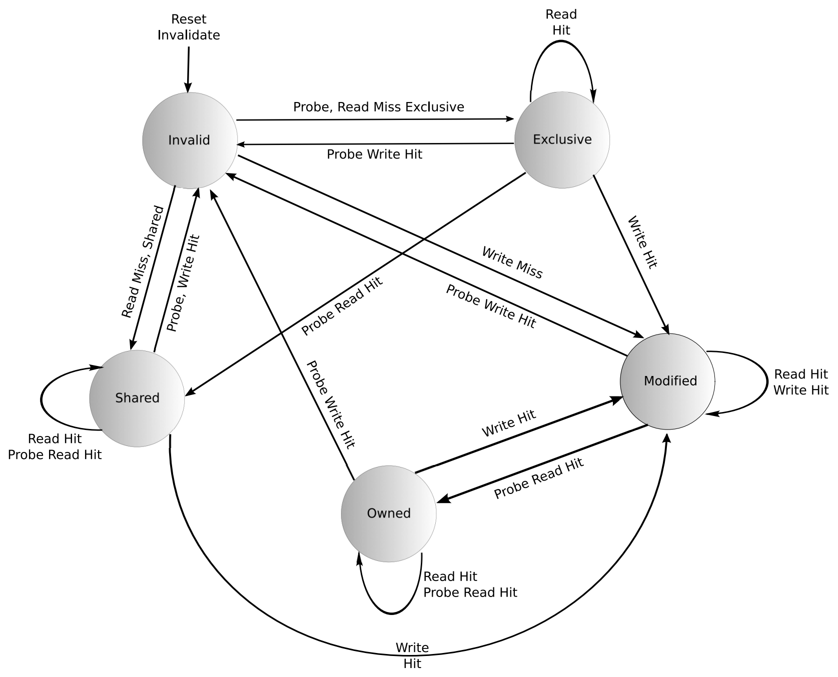


Figure 2.3: The MOESI Protocol states and transition conditions. If, for a given cacheline, a processor experiences a cache-miss a probe message is sent to other caches to obtain the required cacheline. From [5].

hardware coherence mechanism must be used. The MESI [124] and MOESI protocols are two examples of widely used coherency mechanisms. The MOESI protocol [270] is of interest in this thesis as it is implemented by two of the platforms used in this thesis, for which more details are presented in Section 2.1.7. Every cacheline is tagged using state bits to represent its most current state as part of the coherency protocol, in effect the coherency protocol is a per cacheline state-machine allowing the processor to implement cache consistency.

The MOESI protocol, shown in Figure 2.3, has the following states: **M**odified, **O**wned, **E**xclusive, **S**hared and **I**nvalid –

Invalid Cachelines marked as invalid do not hold valid copies of data. All cacheline entries in the cache start off in the invalid state. Cachelines can also enter an invalid state once invalidated i.e. it transitions from one of the other states. Valid data for a cacheline resides either in main memory or in another processor's cache.

Exclusive A cacheline in the exclusive state is exclusive to that particular cache i.e. it is the most current copy of the cacheline and is not in any other cache. Read misses, if serviced from memory cause cache lines to be marked as Exclusive as these cachelines are newly fetched directly from memory.

Modified A cacheline transitions into the a Modified state from Exclusive if the processor's made modifications to that line i.e. if the cache gets a write hit. Future write hits will set the cache line in the Modified state. If the cacheline needs to be evicted, it is transitioned from the Modified state into Invalid.

Shared A Shared cacheline is one which has not been modified and is in cache, but could exist in another processor's cache. A read-miss on a Modified cacheline will lead to a transition into the Shared state. A read-miss response from another cache will also cause the cacheline state to transition to Shared.

Owned A Modified cacheline can transition to the Owned state when the cache forwards the cacheline to another requesting cache, thus by-passing the backing store. The Owned state allows for a cacheline to be replicated to other caches (which hold it in the Shared state). The cache that owns the cacheline is permitted to modify Owned lines only.

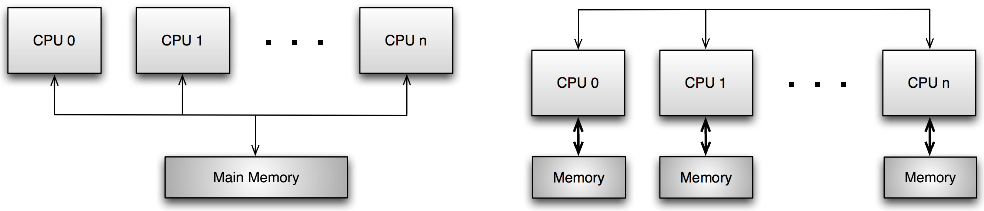
2.1.5 Memory Structures

There are two broad classifications for cache coherent multiprocessor systems – UMA and NUMA. Uniform memory access (UMA) and Non-Uniform Memory Access (NUMA) are two techniques used to create shared memory systems. In this thesis both types of shared memory systems are used.

UMA refers to a shared memory system where each processor can access main memory with the same fixed latency. UMA machines are also referred to as Symmetric Multi-Processors (SMP). UMA systems (Figure 2.4 (a)) either use a bus based [217] or cross-bar [13] to connect the processors. In bus based systems cache coherency is implemented by having individual processors monitor or snoop the bus [102]. This allows processors to make state transitions for its cachelines in accordance with the hardware coherency protocol as shown in Figure 2.3. Bus based approaches [159] are inherently unscalable as coherency traffic limits the number of shared processors that can effectively use the shared bus.

In NUMA systems the time taken to access main memory is non-uniform i.e. cache misses for data that is not local to a processor have higher latencies than those which are in remote memory (Figure 2.4 (b)). While coherency traffic in a NUMA system can also be implemented by using a broadcast across a shared bus, it can be also be implemented using a more scalable directory based approach [58]. In directory based approaches there is an in-memory directory (implemented as a bitmap) representing cachelines and at which processor these are being cached in. Although this can be implemented as one central directory, it can quickly become a bottleneck and thus most modern directory based cc-NUMA[2] (e.g. the SGI Altix [166, 234]) use a distributed directory approach. Here each processor is responsible for maintaining the

[2]Cache Coherent NUMA



(a) UMA (b) NUMA

Figure 2.4: UMA and NUMA organization of shared memory systems [131]

coherency state of its local memory. If a processor requires a cacheline and it is not to be found in its local caches or memory, the processor (or its proxy[3]) directly contacts the foreign processor which manages the memory in which the required cacheline resides. Writes to the cacheline by a remote processor are forwarded to the owning processor, which in turn sends invalidation messages to other processors that might be having copies of the cacheline. This decentralization of the cache directory results in less overhead compared to a centralized directory.

2.1.6 Hardware Performance Counters

As evident from preceding sections modern microprocessors are extremely complex systems. To aid both end-users and engineers in understanding the performance of applications executing on these systems all modern microprocessors have a set of dedicated on-chip and off-chip registers for performance monitoring [3,125,126,128,187,267]. These registers, which are referred to as hardware performance counters (HWPC), are initialized to record a pre-defined set of hardware performance events [79,255]. Performance events, like Cycle counts and total L1 data cache misses, are sometimes counter specific and often some events cannot be measured in conjunction with others [199]. This arises from decisions made at the design stage of the microprocessor in order to (a) reduce the overhead of implementing the performance counter infrastructure in silicon and (b) conserving the on-chip transistor budget for other features.

On configuring the on-chip counters the microprocessor increments the HWPC(s) for the specific set of event(s) of interest. All microprocessors support delivery of signals either to an operating system handler or user space handler in the event any register overflows. Hardware counters can measure performance events for code which is either in user-space, in the kernel or both. Performance counters can be used to either obtain absolute event counts [79] or sample for events based on overflow of the underlying counter or to generate an interrupt if there has been a fixed number of events that have occurred [39].

[3]In the SGI Altix system this is the SHUB [166]

Often hardware performance counters are exposed via libraries to shield end users from the complexity of managing the processes by which HWPCs need to be initialized and accessed. This process is often different between processors from the same vendor. PAPI [37] is cross platform performance counter library which exposes a uniform API and runs across multiple microprocessor types (Intel, AMD and PowerPC). PAPI also supports hardware performance counter event multiplexing [7,69]. This uses an event sampling approach to enable more events to be counted than there are available hardware registers. Events counted using multiplexing will therefore have some statistical uncertainty associated with them. Within discrete periods of time a set of events are monitored, recorded, the counters are reset for the next set of events and the process continues. Care must be taken as this process can introduce errors if the short code segments are being measured or a large number of events are sought [19, 176].

2.1.7 Microprocessors Used in This Thesis

Table 2.1, lists processor and cache characteristics for the microprocessors used in this thesis. The memory latencies given in the table were measured quantities, using the `lmbench` pointer chasing benchmark [184]. Five different processors were chosen the Intel Pentium M, P4; AMD Opteron; IBM G5/PPC970Fx; and the Sun UltraSPARC IIICu (USIIICu). All processors use out-of-order execution except the USIIICu, which is an in-order processor. The processors have a range of clock frequencies ranging from 900Mhz to 3Ghz. All processors are superscalar: the x86 processors can issue 3 instructions per clock cycle, whereas the G5 and USIIICu can issue 4 instructions per cycle.

The Intel x86 processors and the G5 system have a UMA architecture and use a northbridge [217] chipset to link memory and processors. The AMD Opteron system has a NUMA architecture that also uses coherent HyperTransport [56, 218, 282]. The USIIICu is used in the Sun V1280 [266], a twelve processor system and it uses the FirePlane protocol [50] to create a shared memory system. The V1280 exhibits slight NUMA characteristics.

The microprocessors had a varying number of on-chip performance counters (2 to 18). The sizes and organization of their cache hierarchies were also different. Measured `lmbench` latencies for L1, L2 and main memory latencies show an expected factor of 10 difference between the various levels of the memory hierarchy. In the Appendix, Table A.1 lists the PAPI native hardware performance counter events used in this thesis. In Section A.2 of the Appendix, Tables A.2.1 – A.2.7 present `lmbench` plots for linear and random pointer chasing for all the processors.

		Pentium M	Intel P4	AMD Opteron	G5/PPC970Fx	UltraSparc IIICu
Clock Rate	(Ghz)	1.4	3.0	2.2	2.0	0.9
Ops. per Cycle	(Cy)	3	3	3	$4^{\dagger}$	4
Memory Subsystem		NtBr	NtBr	cHT	U3H	FirePlane
Perf. Counters	(No.)	2	18	4	8	2
L1 DCache	Size (KB)	32	16	64	32	64
	Associativity (Ways)	8	8	2	2	4
	Line size (Bytes)	64	64	64	128	32
	Cache Policies	LRU, WB	P-LRU	LRU, WB, WA	LRU, NWA, WT	P-LRU, NWA
L2 Unified	Size (MB)	1	1	1	0.5	8
	Associativity (Ways)	8	8	16	8	2
	Line size (Bytes)	64	64	64	128	64
	Relation to L1	Inclusive	Inclusive	Exclusive	Inclusive	Exclusive
	Cache Policies	LRU	P-LRU	P-LRU	LRU	AM, P-LRU
`lmbench` Latencies for						
L1 DCache	Latency (Cy)	3	4	3	3	3
L2 Unified	Latency (Cy)	10	28	20	16	16
Main Memory	Latency (Cy, $\approx$)	201	285	260 – 300	330	360 – 380

† = 4 ops + 1 branch
NtBr = Northbridge, HT = HyperTransport, U3H = Apple's NtBr U3H bus
LRU = Least Recently Used, P-LRU = Pseudo-LRU,
WB = Write Back, WA = Allocate on Write, NWA = No allocate on Write, AM = Allocate on Miss

Table 2.1: Processor characteristics of clock rate, cache sizes and measured latencies for L1 DCache, L2 cache and main memory latencies for the microprocessors used in this thesis.

2.2 Performance Evaluation

John [199] classifies performance evaluation into two major categories; performance measurement and performance modelling. Performance measurement is the use of hardware or software techniques which allow for measurement of either time taken to completion for a given task or event counts of interest within some period of time. Performance modelling refers to the creation and validation of a model which accounts for observations made from performance measurement of a given system. A modified version of a table presented in [199] detailing various performance evaluation options is given in Table 2.2. The various sub-sections of this table are expanded below. Following this is a brief introduction to dynamic binary translation (DBT) is presented prior to an introduction to the Valgrind DBT tool.

Performance Measurement	Software assisted profiling	
	On-chip hardware performance counters	
Performance Modelling	Analytic Modelling	Parametric models
		Probabilistic models
		Queuing theory
		Neural net, Markov models
	Simulation	Statistical, Monte-Carlo simulation
		Trace driven
		Execution driven
		Full system simulation

Table 2.2: Classification of performance evaluation techniques

2.2.1 Performance Measurement using Software Assisted Profiling

The `gprof` [101] profiling tool is an example of a software assisted profiling tool. One of its features is the ability to generate an execution profile which can be used to attribute an applications' execution time on a per subroutine basis. This is often achieved using a sampling approach, wherein at specific time intervals an application's program counter is sampled for the lifetime of the application. Intel's VTune [127] tool, Sun's Performance Analyzer [268] and the Tau [244] performance tuning and analysis tools can also be used for profiling applications. Both VTune and Tau can either use interrupt based sampling or can use hardware performance counters to sample events of interest and attribute these to sections of executed code. The OProfile [160] tool also uses a sampling approach to allow both user and kernel space profiling of an application under Linux. It can apportion execution time either based on walltime or performance counter events corresponding to code segments that ran. DTrace [45] for Solaris and systemtap for Linux [214] are two OS instrumentation tools which allow for the creation of arbitrary user programs which can be used to instrument running production systems. Both consume OS specific probes which are exposed via kernel and application interfaces, to instrument a running OS i.e it is possible to trace an application running in user-space and its subsequent interactions all the way down to the kernel. JIFL [201], is a just-in-time fine-grained dynamic instrumentation framework for the Intel x86 architecture. Unlike DTrace and systemtap, which inserts trap instructions, JIFL is able to patch in jump instructions into a running OS' code image thus allowing for finer instrumentation. By using an API, it is possible to profile system execution time in both user and kernel space with minimal overhead. PinOS [40], extends the Pin [165] instrumentation framework to allow whole-system instrumentation. This is achieved by using vitalization provided by the Xen [21] hypervisor. The application and OS of interest is run within a Xen DomU [4]. PinOS can then either attach to the running instance of the OS and effectively instruments the DomU using Pin's instrumentation API. Also, these tools have been used in identifying and modelling cache performance problems [170, 171].

2.2.2 Performance Measurement using On-Chip Counters

As discussed in Section 2.1.6 HWPCs can be used to measure performance events of interest. Access to hardware performance counters can either be via compiler instrumentation, use of tools described above or it can be inserted directly into areas of user code of interest, either by using APIs from tools like Dyninst [118] and Pin [165] or manually patching in calls via libraries like PAPI [37]. The advantage of using tools like Dyninst and Pin is that source code for the original binary is not required as these tools can operate on binaries. Platform specific definitions for performance counters used in this thesis are given in Appendix A, Table A.1.

2.2.3 Performance Modelling using Analytic Techniques

Analytic modelling is usually employed for large computer systems i.e. aggregate models of the microprocessor, memory and interconnect. It attempts to build a system specific mathematical model and is high level, often using characteristics of input event distributions. Computer systems are aggregations of hardware and software resources and a set of tasks or jobs compete for these resources [199]. Analytic models attempt to weight various system parameters and present empirical equations of system performance. This results in formulations which are based on probabilistic models [53, 122, 195, 241, 253], statistical models [145], non-linear

[4]Xen acts as a hypervisor effectively virtualizing underlying hardware. The DomU (guest/user domain) are OS instances that run atop the hypervisor and are controlled from the Dom0 (controller/host domain).

regression models [141], stochastic models [120], queuing theory [67], Markov models [254] and neural-net models [129]. Explicit parametrisation of scientific codes based on inputs are also another type of analytical model [175]. Analytic models have quick turn around times and can model very large systems [241]. Yet the models are dependent of underlying simplifications of the real system and depend on input parameters which characterize the processor, interconnect and workloads being studied. An example of analytic modelling is a performance model for a Particle Transport Code (PTC) given in Mattis and Kerbyson [175]. The PTC's input is parametrized along with platform specific parameters like communication latencies and cache specific parameters. The model is validated and allows for quick performance estimates on future computing platforms.

2.2.4 Performance modelling using Simulation

Simulation techniques use either a software model, which abstracts underlying hardware or employs statistical techniques to simulate components or subsystems of interest. The hardware platform being studied is abstracted and modelled in software. It have now become the defacto means of performance evaluation in computer architecture studies [248]. Simulators are constructed to be functional or timing accurate simulators. A functional simulator can simulate the functionality of target hardware i.e. runtime register values of a simulated application can be retrieved from simulated registers. To create a timing accurate simulation, the simulation needs to model both functionality and per cycle latencies of processor events.

Statistical, Monte-Carlo Simulation

Statistical simulation [24, 75] and Monte-Carlo [256] simulation, collect a series of microarchitecture specific and microarchitecture-independent characteristics. This collected profile is then used as input for simulating a trace-driven statistical simulator.

Trace Driven and Execution Driven Simulation

Simulations can either be trace or execution driven. Trace driven simulation [285] uses a trace as its input. The trace can be address values, instruction streams or processor state. The trace itself is obtained prior to executing the simulation. This trace is then used as input either for functional or timing accurate simulation [295]. A draw-back of trace driven simulation is that traces are specific to a particular instance of a running application and cannot model non-deterministic and timing dependent effects [97]. Execution driven simulators interpret instructions from a binary source to perform its simulation rather than using a pre-obtained trace input. Dynamic binary translation is often used in implementing execution driven simulators, where the input binary is read-in and used to progress the simulation.

Over [204] classifies simulation tools by their level of detail – component-level, user-level and full machine simulation. Each of these levels of detail are implemented using techniques give under the Simulation heading of Table 2.2.

Component-level simulation is used when a particular sub-system of interest needs to be modelled i.e. branch predication, out-of-order retirement. This is done when the behavior of the component in isolated is required, rather than its incorporation in a larger framework i.e. branch prediction within a microprocessor.

User-level simulation aims to model the microarchitecture of a target processor and its execution pipelines. Typically these simulators are unable to simulate the effect of OS and I/O interactions as BIOS and other hardware sub-systems required for system boot have not been modelled. RSIM [121, 208], SimpleScalar [18, 41] are examples of this type of simulation. These execute user-level workloads and assume I/O subsystems do not impact overall performance of the simulation.

Full-machine simulation , on the other hand, models hardware in sufficient detail to permit simulations of entire operating systems, I/O subsystems and network interface cards. SimICS [168], SimOS [230, 231] and M5 [27] are examples of such simulators.

2.2.5 Dynamic Binary Translation and the Callgrind/Valgrind tool

Dynamic binary translation is a technique used for performance modelling simulation. Dynamic binary translation (DBT) uses basic blocks [5] from an input binary, which is annotated and re-compiled to the target hosts' instruction set. Typically a cache simulation model is fed these basic blocks in order to obtain cache specific information e.g. the total number of L1, L2 cache misses. In this thesis, the Valgrind [190, 192] dynamic binary translation [44] framework is used in conjunction with the Callgrind [140, 191, 274, 296, 297] functional cache simulation tool. The Valgrind dynamic binary instrumentation framework provides basic blocks to the Callgrind tool which in turn performs dynamic, execution driven cache simulation Valgrind, is a program supervision framework which permits the writing of tools or plug-ins which supervise the execution of an underlying executable. In this work version 3.2.1 of Valgrind and the Callgrind tool. Cache parameters for the L1 Instruction, L1 Data and L2 Unified cache sizes are specified prior to running an executable via Valgrind entirely in user-space. An executable and its command-line options are passed onto Valgrind which in turn sets up an environment for callgrind to perform the cache simulation.

[5] A basis block is a distinct section of object code which has one entry point, one exit point and does not contain any jump instructions within it [14, 302].

Valgrind decodes a native executable and converts the native ISA's op-codes into a platform independent representation called VEX. This representation or instruction stream is then fed into the Callgrind tool which in turn performs its cache simulation using these VEX basic blocks. On completing its cache simulation, Callgrind hands the basic block back to Valgrind. At this point Valgrind becomes a just-in-time compiler and outputs native machine code from the basic block. It is this very process which allows Valgrind/Callgrind to supervise the program's execution and monitor all of the user-space instruction stream of an executable.

The cache simulation allows for the recording of more events than is possible using on-chip performance counters. Hardware designers are firstly, limited by the amount of on-chip silicon that can be devoted to performance monitoring constructs and need to ensure its operation does not in any way impede the execution speed of the processor. Simulation takes longer to run (Valgrind simulations can experience a 10x slowdown), but is able to record and annotate events which cannot be obtained using on-chip performance counters alone.

2.3 Electronic Structure Methods

This section gives an overview of electronic structure methods used in this thesis. It includes background material relating to Schrödinger's wave equation, the Hartree-Fock method, chemical basis sets, two-electron integral evaluation and Density Functional Theory.

Computational quantum chemistry encompasses a body of theory called electronic structure theory which has been developing over the course of the last century [77, 210]. Central to it is the Schrödinger wave-equation,

$$\widehat{H}\psi = E\psi \tag{2.1}$$

Quantum mechanics postulates that for any system there exists a wavefunction ψ that fully describes the system under consideration. In Equation 2.1 $\widehat{H}$ is the operator applied to ψ yielding the scalar E, multiplied by ψ [55]. In this case the operator $\widehat{H}$ is the Hamiltonian operator and E is the system's energy. In the context of quantum chemistry the Hamiltonian operator $\widehat{H}$ includes terms for the kinetic energies of the electrons and nuclei and the coulombic interactions between them.

A two atom system is illustrated in Figure 2.5. In this Figure a right handed coordinate system with the origin at O has atom A and B at a distance R_A and R_B from the origin, and two electrons i, j at a distance r_i and r_j from the origin. The distance between the two nuclei is R_{AB} whereas the distance between the electrons is r_{ij}. In this system there are interactions between nuclei, between electrons and between electrons and nuclei. The Hamiltonian for a

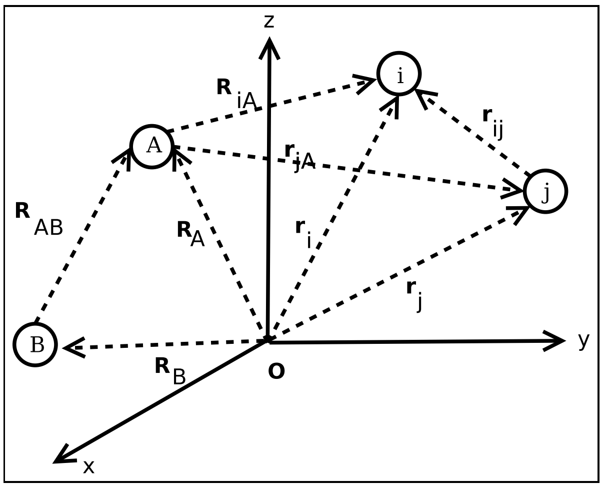


Figure 2.5: A molecular coordinate system where i, j denote electrons and A, B denote atoms. From [271].

generalized system of M point nuclei and N point electrons is given by,

$$\begin{aligned}\widehat{H} = &-\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2 - \sum_{A=1}^{M}\frac{1}{2M_A}\nabla_A^2 \\ &-\sum_{i=1}^{N}\sum_{A=1}^{M}\frac{Z_A}{r_{iA}} + \sum_{A=1}^{M}\sum_{B>A}^{M}\frac{Z_A Z_B}{R_{AB}} \\ &+\sum_{i=1}^{N}\sum_{j>i}^{N}\frac{1}{r_{ij}}\end{aligned} \tag{2.2}$$

where M_A is the ratio of mass of nucleus A to that of the mass of an electron, Z_A is the atomic number of nucleus A, ∇_i^2 and ∇_A^2 are the Laplacian for the i^{th} electron and the A^{th} nucleus. The first two terms are associated with the kinetic energy (K.E.) of the electrons and nuclei respectively; the third term represents coulombic attraction between the electrons and nuclei; the fourth term represents coulombic repulsion between nuclei and the fifth term represents coulombic repulsion between electrons.

Schrödinger's wave-equation only has closed-form solutions for relatively trivial systems, and in general can only be solved using various numerical or approximate methods. A central simplification is the Born-Oppenheimer approximation. This comes about from the observation that nuclei are much heavier than electrons (1800 times heavier) so electronic relaxation is near instantaneous and electrons can be considered to move in a field of fixed or clamped nuclei. This gives rise to the electronic Hamiltonian which describes the motion of electrons in a field of fixed nuclei,

$$\widehat{H}_{elec} = -\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2 - \sum_{i=1}^{N}\sum_{A=1}^{M}\frac{Z_A}{r_{iA}} + \sum_{i=1}^{N}\sum_{j>i}^{N}\frac{1}{r_{ij}} \tag{2.3}$$

Solution to the electronic Schrödinger equation for different nuclear coordinates gives rise to the concept of a potential energy surface [83], where minimums on the surface indicate stable nuclear geometries. Thus, the electronic energy of the system depends parametrically on nuclei locations and explicitly on electron locations,

$$\begin{aligned} E_{elec} &= E_{elec}(R_A) \\ E_{total} &= E_{elec} + \sum_{A=1}^{M}\sum_{B>A}^{M}\frac{Z_A Z_B}{R_{AB}} \end{aligned} \tag{2.4}$$

and the total electronic energy includes the constant of nuclear repulsion. Taken together both Equations 2.3 and 2.4 represents the electronic structure problem. In subsequent discussions we will consider the electronic energy only and hence not use the $total$ and $elec$ subscripts. Even with the Born-Oppenheimer approximation, it is only possible to solve the Schrödinger equation for trivial systems and its formulation requires further simplification.

2.3.1 Hartree-Fock approximation

The Hartree-Fock (HF) approximation is a simplification of Schrödinger's equation where the wavefunction is constructed as an anti-symmetric product of one-electron functions or Molecular Orbitals (MOs) ϕ,

$$\psi = |\phi_1(\mathbf{x}_1)\,\phi_2(\mathbf{x}_2)\,\ldots\,\phi_N(\mathbf{x}_N)| \tag{2.5}$$

Here each MO ϕ_i describes the motion of one electron in the system. The MOs are expanded in terms of N basis functions χ

$$\phi_i = \sum_{\mu=1}^{N} c_{\mu i}\,\chi_\mu \tag{2.6}$$

where, $c_{\mu i}$ are molecular orbital coefficients. The challenge for HF theory is to determine the set of orbitals ϕ which gives the best approximation to the exact wavefunction, ψ_{exact}. This is achieved using the Variational Principle which states that an approximate wavefunction has an energy greater than or equal to the exact energy [271] i.e. the energy of the exact wavefunction is a lower bound to energies obtained from approximate wavefunctions. This renders the problem to one where the set of coefficients which minimizes the energy of the approximate wavefunction needs to be determined. By using a trial wavefunction, the energy is evaluated for a set of MOs which are guessed initially and then these MOs are systematically varied until the evaluated energy does not change. Thus the Variational Principle permits the construction of a trial wavefunction and affords the means to test its quality. The HF method is the simplest trial wavefunction that can be created which is a single determinant where N orbitals are occupied by N electrons; Equation 2.5. The variational optimization of the HF wavefunction gives rise to the canonical Hartree-Fock equations,

$$\sum_{\nu=1}^{N}(F_{\mu\nu}-\varepsilon_i\ S_{\mu\nu})\,c_{\nu i}=0 \quad \mu=1,2,\ldots,N. \tag{2.7}$$

which can be represented in its matrix form as,

$$F\,C=S\,C\,\varepsilon \tag{2.8}$$

where, F is called the Fock matrix; C is a matrix of the molecular orbital expansion coefficients; S is the overlap matrix representing the overlap between basis functions and ε is a diagonal matrix where ε_i is the orbital energy of each MO χ_i.

In essence, Equation 2.8, allows one to determine the N MO basis functions by solving the secular equation,

$$\begin{vmatrix} F_{11}-E_1S_{11} & F_{12}-E_2S_{12} & \ldots & F_{1N}-E_NS_{1N} \\ F_{21}-E_1S_{21} & F_{22}-E_2S_{22} & \ldots & F_{2N}-E_NS_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ F_{N1}-E_1S_{N1} & F_{N2}-E_2S_{N2} & \ldots & F_{NN}-E_NS_{NN} \end{vmatrix} = 0 \tag{2.9}$$

to obtain solutions E_j.

Equation 2.8 for the Fock matrix F, represents the interaction of an average field of electrons acting on each MO. For a closed shell system this is given by,

$$\begin{aligned} F_{\mu\nu} &= \left\langle \phi_\mu \middle| -\frac{1}{2}\nabla^2 \middle| \phi_\nu \right\rangle - \sum_k^{nuclei} Z_k \left\langle \phi_\mu \middle| \frac{1}{r_k} \middle| \phi_\nu \right\rangle \\ &\quad + \sum_a^{\frac{N}{2}}\sum_\lambda\sum_\sigma C_{\lambda a}C_{\sigma a}\left[2(\phi_\mu\ \phi_\nu|\phi_\sigma\ \phi_\lambda)-(\phi_\mu\ \phi_\lambda|\phi_\sigma\ \phi_\nu)\right] \\ &= H_{\mu\nu}^{core}+\sum_\lambda\sum_\sigma P_{\lambda\sigma}\left[(\phi_\mu\ \phi_\nu|\phi_\sigma\ \phi_\lambda)-\frac{1}{2}(\phi_\mu\ \phi_\lambda|\phi_\sigma\ \phi_\nu)\right] \\ &= H_{\mu\nu}^{core}+\sum_\lambda\sum_\sigma P_{\lambda\sigma}\left[J_{\lambda\sigma}-\frac{1}{2}K_{\lambda\sigma}\right] \\ &= H_{\mu\nu}^{core}+G_{\mu\nu} \end{aligned} \tag{2.10}$$

where, $H_{\mu\nu}^{core}$ is referred as the core Hamiltonian matrix that combines K.E. and nuclear attraction terms, $G_{\mu\nu}$ is the two-electron integral contribution to Fock matrix, P is called the Density matrix and is defined over the MOs coefficients,

$$P_{\lambda\sigma} = \sum_{i=1}^{occupied} c_{\lambda i} c_{\sigma i} \tag{2.11}$$

and the four index quantities $(\phi_\mu\ \phi_\nu | \phi_\lambda\ \phi_\sigma)$ represent electron-repulsion integrals (ERI) [92, 229, 272]. The ERI integrals are further categorized depending on the order of subscripts, although the form of the basic integral is the same; $(\phi_\mu\ \phi_\nu | \phi_\sigma\ \phi_\lambda)$ are categorized as coulomb integrals (J) and $(\phi_\mu\ \phi_\lambda | \phi_\sigma\ \phi_\mu)$ are exchange integrals (K). Coulomb integrals arise from the classical repulsion between electron distributions, whereas exchange integrals arise from non-classical interactions between electron distributions. The HF equations are solved iteratively in a self-consistent field (SCF). Figure 2.6 presents key steps in the SCF. Once a molecular system has been specified by its nuclear coordinates, number of electrons, atomic numbers and a basis set, the overlap matrix S and core Hamiltonian $H_{\mu\nu}^{core}$ are computed. The density matrix P is guessed. This then leads to the iterative steps of the SCF . To form the Fock matrix F, the two-electron integrals are computed. Fock formation is a $O(N^4)$ process since there are N^4 two-electron integrals. After formation, the Fock matrix is diagonalized to obtain coefficients C and energies ε. A new density matrix is formed using the obtained C and a convergence criterion is evaluated e.g. if the new density matrix differs from the old density matrix, subject to a cut-off, the solution has converged, else the new density matrix is used to form a new Fock matrix. If the solution converges other quantities of interest can be evaluated e.g. dipole moments and charge distribution amongst the atoms in a molecular system. SCF convergence problems do arise and some of the most commonly used techniques are extrapolation of the Fock matrix, damping of the density matrix, level shifting of MOs, the use of direct inversion of iterative subspace (DIIS) [55, 83].

2.3.2 Atomic Basis Functions

MOs are constructed using a linear combination of basis functions for which the coefficients are obtained using the SCF procedure. These basis functions (χ) are generally located at the various atomic nuclei, and represented as the product of a radial function and an angular function [112,271]. The preferred radial function is a Gaussian (G_{nl}) and the preferred angular function is a spherical harmonic (Y_{lm}),

$$\chi_\alpha = G_{nl} Y_{lm} \tag{2.12}$$

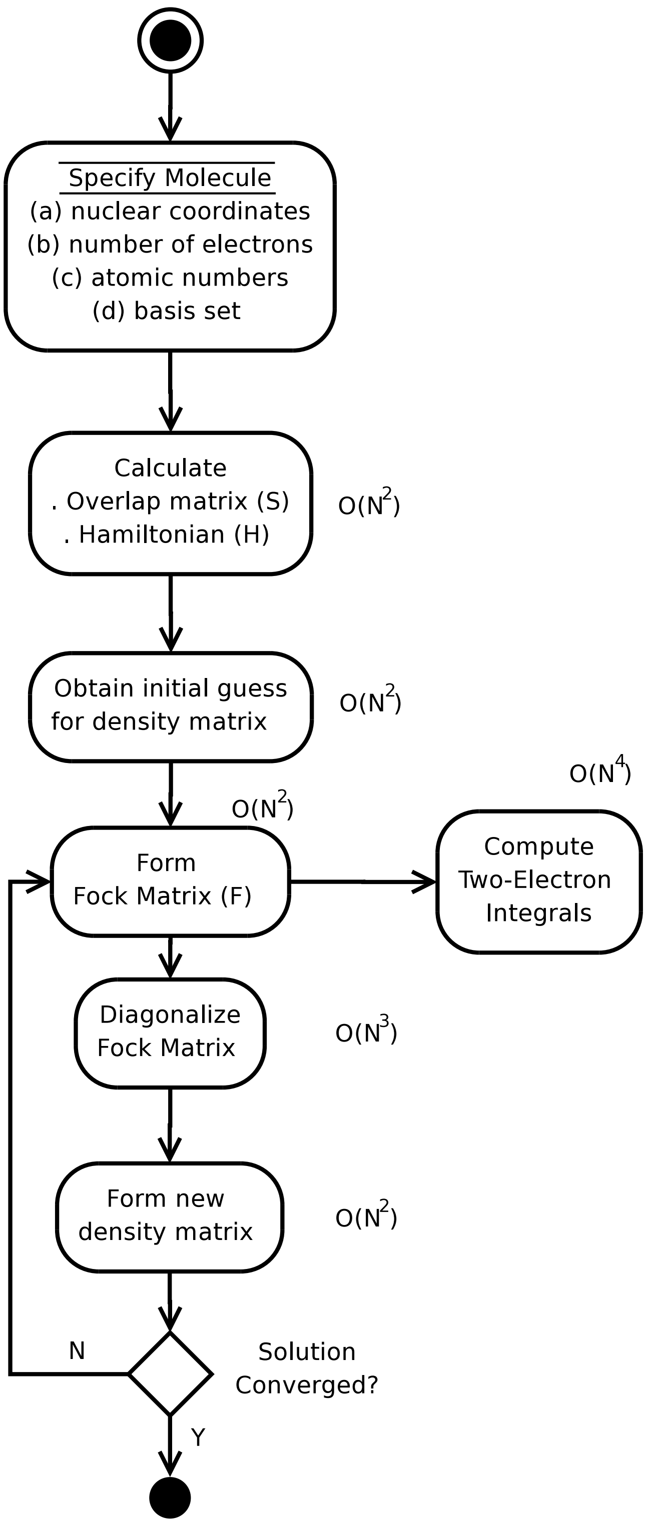


Figure 2.6: Activity diagram for the SCF procedure and computational complexity for key steps.

The subscripts n, l and m are referred to as the principal, angular and magnetic quantum numbers respectively reflecting their role in atomic orbital calculations [169]. The Spherical harmonic is given as,

$$Y_{lm}(\theta,\varphi) \quad = \quad \sqrt{\tfrac{2l+1}{4\pi}\tfrac{(l-m)!}{(l+m)!}}\; P_l^m(cos\,\theta)\, e^{im\varphi} \tag{2.13}$$

where P_l^m is a Legendre polynomial ($l \geq 0$ and $|m| \leq l$). When considering spherical harmonic functions of rank $l \geq 1$, all the orders of spherical harmonic functions within that rank i.e. $\forall m : -l \leq 0 \geq +l$ are treated together. For example, when considering a basis function with a spherical harmonic of rank l = 3, all of its 7 components (i.e $m : -3 \leq 0 \geq 3$) are also included as basis functions. All functions for a given value of l are referred to as a *shell.* Shells with l = 0,1,2,... are referred to as s, p, d, f, g shells respectively.

The Gaussian G_{nl}, in Equation 2.12 is located at coordinates $A = (A_x,A_y,A_z)$ with an angular momentum $a = (a_x,a_y,a_z)$ and an exponent α,

$$G_{nl}(\alpha,\mathbf{A}) \equiv G_{nl}^{PGTO}(\alpha,\mathbf{A}) \quad = \quad (x-A_x)^{a_x}(y-A_y)^{a_y}(z-A_z)^{a_z}\; e^{-\alpha\,|\mathbf{r}-\mathbf{A}|^2} \tag{2.14}$$

G_{nl} is referred to as a primitive Gaussian-type orbital (PGTO). A set of primitive basis functions which share the same center A, exponent α and angular quantum number l is called a primitive shell e.g. the set of p functions on a nuclei p_x, p_y, p_z is a primitive p shell.

PGTOs are usually combined in a fixed linear combination to create a contracted Gaussian function (CGTOs). GTOs have a r^2 exponential term which causes them to decay rapidly and this does not mimic the central cusp which is observed for electron distribution around an atom [271]. In order to get the correct behaviour a set of GTOs are multiplied by contraction coefficients to produce the requisite cusp-like behaviour at the nuclear center [246],

$$\phi_\kappa^{CGTO}(\mathbf{A}) = \sum_{p=1}^{L} d_{p\kappa}\, G_{nl}^{p}(\alpha_{p\kappa},\mathbf{A}) \tag{2.15}$$

where L is the length of the contraction, $d_{p\kappa}$ is a contraction coefficient and $\alpha_{p\kappa}$ is the exponent on each constituent PGTO. A set of CGTOs with the same center and set of angular momentum exponents is called a contracted shell. CGTOs are the de-facto basis function of choice in most electronic structure calculations as multi-center Gaussian integrals that arise from their use in Equation 2.7 can be efficiently evaluated. It is this reason that has led to the widespread use of Gaussians as to represent basis sets.

Over the years, chemists have defined libraries of basis sets which: yield chemically accurate energies, are cost effective computationally, and transferable so that a given description would be useful in different chemical bonding scenarios. Some examples of commonly used basis sets are 6-31G*, aug-cc-pCVTZ, def2-TZVP and so on [65, 80, 81, 111, 237, 277]. To summarize, a basis function is characterized by (a) its location, (b) its degree of contraction and (c) the rank of its angular components.

2.3.3 Two-Electron Integral Evaluation

Evaluation and processing of the two-electrons dominate the HF method. A Simple ERI for four primitive s functions[6] i.e. (ss|ss) is,

$$I^{ERI} \equiv \int\int e^{-\alpha|r_1-A|^2} e^{-\beta|r_1-B|^2} \frac{1}{r_1-r_2} e^{-\gamma|r_1-C|^2} e^{-\delta|r_1-D|^2} \tag{2.16}$$

where A, B, C, D are centers of the four functions; $\alpha, \beta, \gamma, \delta$ their exponents. Using the Gaussian product rule [33, 34, 271] the four center problem can be combined into a two center problem,

$$I^{ERI} = G_{AB}G_{CD} \int\int e^{-\zeta|r_1-P|^2} \frac{1}{r_1-r_2} e^{-\eta|r_2-Q|^2} \tag{2.17}$$

where

$$\begin{aligned} G_{AB} &= exp[\tfrac{-\alpha\beta}{\alpha+\beta}|A-B|^2] & G_{CD} &= exp[\tfrac{-\gamma\delta}{\gamma+\delta}|C-D|^2] \\ P &= \tfrac{\alpha A+\beta B}{\alpha+\beta} & Q &= \tfrac{\gamma C+\delta D}{\gamma+\delta} \\ \zeta &= \alpha+\beta & \eta &= \gamma+\delta \end{aligned} \tag{2.18}$$

The final ERI expression for an s function is,

$$I^{ERI} = G_{AB}G_{CD} \frac{2\pi^{\frac{5}{2}}}{\zeta\eta(\zeta+\eta)^{\frac{1}{2}}} F_m(T) \tag{2.19}$$

where $F_0^{EE}(T)$ is the incomplete Gamma function,

$$F_m(T) = \int_0^1 e^{-Tu^2} du \tag{2.20}$$

Boys [33] showed how higher angular momentum functions can be obtained from lower ones by partial differentiation with respect a given coordinate. Functions with higher angular momentum values are computed using a set of recursive relations which successively build up angular momentum from primitive s functions. There are several schemes for evaluating ERIs [73, 109, 112, 161, 197, 198, 229, 272]. In this thesis the McMurchie-Davidson scheme is of interest and is outlined in the following sub-section.

[6]An s function arises when a given orbital has $l = 0$

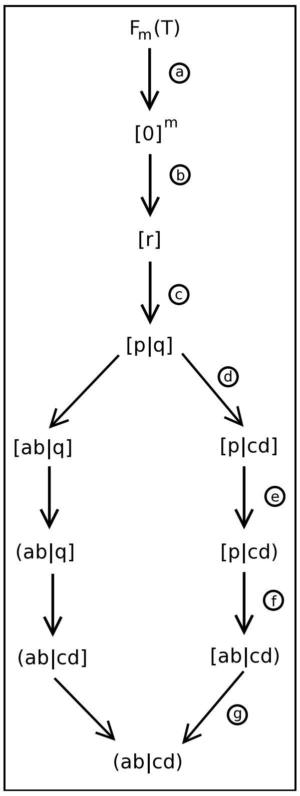


Figure 2.7: The McMurchie-Davidson ERI scheme. From [229].

2.3.3.1 The McMurchie-Davidson scheme for ERIs

McMurchie and Davidson (MD) computes ERIs based on Hermite Gaussians [182, 308]. Hermite Gaussians offer a compact means of representing one and two electron integrals. Using Boys' observation [34] on partial differentiation w.r.t. a center, MD derived recurrence-/recursion relations using Hermite Gaussians to allow calculation of ERIs for CGTOs. Referring back to Equation 2.10, pairs of basis functions $(\phi_\mu\ \phi_\lambda|$ and $|\phi_\nu\ \phi_\sigma)$ are grouped together when computing the Fock matrix. This grouping referred to as a shell-pair.

The MD ERI evaluation procedure is shown in Figure 2.7. The process begins through the evaluation of a quantity labelled as $[0]^m$, which is related to $F_m(T)$. The $[\mathbf{0}]^m$ quantities are formed using $F_m(T)$ (i.e. Step (a) in Figure 2.7),

$$[\mathbf{0}]^m = D_A D_B D_C D_D\, G_{AB} G_{CD} \sqrt{\frac{2\pi^5}{(\zeta\eta)^3}}\,(2\vartheta^2)^{m+\frac{1}{2}}\, F_m(T) \tag{2.21}$$

where D_A, D_B, D_C, D_D are contraction coefficients of the Gaussians; m is determined by the sum of the angular momentums of the constituent basis functions; $\vartheta = \sqrt{\frac{\zeta\eta}{\zeta+\eta}}$; and $0 \le m \le L$. $[0]^m$ are single scalar values.

The first step in the MD process constructs the $[\mathbf{r}]^{(m)}$ vector from the $[0]^m$ scalar quantity. MD showed that $[\mathbf{r}] \equiv [\mathbf{r}]^{(0)}$ can be generated using a two-term recurrence relation (RR) (Step (b)),

$$[\mathbf{r}]^{(m)} = R_i[\mathbf{r} - \mathbf{1_i}]^{(m+1)} - (\mathbf{r_i} - 1)[\mathbf{r} - \mathbf{2_i}]^{(m+1)} \tag{2.22}$$

where i is the Cartesian axis direction; $\mathbf{1}_i$ and $\mathbf{2}_i$ represents a unit vector with a value of 1 or 2 in direction $\mathbf{i}$. As Equation 2.22 is a recursive relation which shows that any given $[\mathbf{r}]^0$ integral can be assembled from an elementary set of $[0]^m$ integrals i.e. any given $[\mathbf{r}]^{(m)}$ can be generated using lower angular momentum terms in upto three different ways, each of which has different number of terms and as a results differing costs in accessing data operands from memory. Finding the most optimal path from the elementary $[\mathbf{0}]^m$ to $[\mathbf{r}]^m$ involves a tree search procedure, which for $L \geq 5$ is non-trivial and for $L \geq 8$ is unsolved [135].

Transferring angular momentum from $[\mathbf{r}]$ to $[\mathbf{p}|\mathbf{q}]$ proceeds by using the following relation to shift angular momentum from one $[\mathbf{r}]$ to two centers $\mathbf{p}$ and $\mathbf{q}$ $([\mathbf{p}|\mathbf{q}] \equiv [00p|q00])$,

$$[\mathbf{p}|\mathbf{q}] = (-1)^q[\mathbf{p}+\mathbf{q}] \tag{2.23}$$

where $\mathbf{p}$ and $\mathbf{q}$ are products of functions $\mathbf{a},\mathbf{b}$ and $\mathbf{c},\mathbf{d}$ respectively. Here $\mathbf{p}$ and $\mathbf{q}$ are Hermite functions [92].

Step (d) proceed by converting a Hermite Gaussian into two Cartesian Gaussians by using the following RR, which shifts angular momentum from the one Hermite $\mathbf{q}$ onto a Cartesian Gaussian $\mathbf{c}$,

$$|\mathbf{q}\ \mathbf{cd}] = Q_i|(\mathbf{c}-\mathbf{1_i})\mathbf{d}(\mathbf{q}-\mathbf{1_i})] + (Q_i - C_i)|(\mathbf{c}-\mathbf{1_i})\mathbf{dq}] + (2\eta)^{-1}|(\mathbf{c}-\mathbf{1_i})\mathbf{d}(\mathbf{q}+\mathbf{1_i})] \tag{2.24}$$

where the angular momentum is shifted from Hermite $\mathbf{q}$ onto $\mathbf{c},\mathbf{d}$.

Step (e) involves carrying out a contraction step amongst the $|\mathbf{cd}]$ integrals (denoted by the change from square brackets to round brackets) where , the constituent GTOs that make up a CGTO are combined.

$$[\mathbf{p}|\mathbf{cd}) = \sum_{i=1}^{K_A}\sum_{j=1}^{K_B}[\mathbf{p}|\mathbf{c}_i\mathbf{d}_j] \tag{2.25}$$

Steps (f) and (g) are carried out using similar transfer and contraction relations to Equations 2.24, 2.25. As a pedagogical exercise, a basic MD integral evaluation algorithm and SCF code was developed in C++ and parallelized using OpenMP.

2.3.3.2 The PRISM algorithm

The efficacy of an ERI algorithm is dependent on when PGTOs are combined to form fully contracted CGTOs. Algorithms like MD choose to carry out integral contraction i.e. the

$$\begin{tikzcd}
F_m(T) \arrow[d, "T_0"] & & \\
{[0]^m} \arrow[r, "C_1"] \arrow[d, "T_1"] & {}_{abp}(0]^m \arrow[r, "C_2"] \arrow[d, "T_2"] & {}_{abp}(0)^m_{cdq} \arrow[d, "T_3"] \\
{[r]} \arrow[r, "C_3"] \arrow[d, "T_4"] & {}_{abp}(r] \arrow[r, "C_4"] \arrow[d, "T_5"] & {}_{abp}(r)_{cdq} \arrow[d, "T_6"] \\
{[ab|q]} \arrow[r, "C_5"] \arrow[d, "T_7"] & (ab|q] \arrow[r, "C_6"] \arrow[d, "T_8"] & (ab|q)_{cdq} \arrow[d, "T_9"] \\
{[ab|cd]} \arrow[r, "C_7"] & (ab|cd] \arrow[r, "C_8"] & (ab|cd)
\end{tikzcd}$$

Figure 2.8: The MD-PRISM ERI scheme. From [229], [92]

contraction step, $(\mathbf{ab}|\mathbf{cd})$, is the last to be performed. Gill et al. [95] realized that the efficiency of various integrals algorithms is tied not only to the nature of the integrals being computed, but also to when the contraction steps are performed. In recognition of this, they created the PRISM algorithm which dynamically chooses when contractions are performed. Figure 2.7, is the front face of MD PRISM. There are at most three transformation (T) steps and two contraction (C) steps to generate a given $(\mathbf{ab}|\mathbf{cd})$ shell-quartet. The MD algorithm corresponds to the $T_1T_4C_5T_8C_8$ path in PRISM. PRISM further recognizes that shell-quartets that have identical angular momentum types and contraction lengths can be treated together. This allows for vectorization of these shell-quartets into batches on vector machines or these batches are further cache-blocked for operation on cache based architectures.

PRISM's mode of operation is as follows – Once the shell-pair data is collected, integral screening is performed to reduce the total number of significant shell-pairs that need to be considered. The list of remaining shell-pairs are sorted and pairing of shell pairs is done. At this stage, there is batching of similar shell-quartets of the same type to increase sharing of intermediate quantities. After this step, the $[0]^m$ integrals are computed. Each of the transformation steps involves the use of driver routines which take pre-compute solutions to the tree search problem (i.e. the best way of computing an integral intelligently without resorting to recursion) and execute a series of operations array locations in order to generate the required integral [92].

We note that the implementation of PRISM, in the Gaussian code, has additional paths which correspond to the generalisation of the Obara-Saika (OS) integral algorithm [197, 198], in addition to MD-PRISM. Reference [92] casts OS recurrence relations into a form similar to MD-PRISM. These OS paths are subsequently used by PRISM for certain contracted integral as its implementation yields lower run-times.

2.3.4 Density Functional Theory

Density Functional Theory (DFT), is an alternate approach to solving Schrödinger's wave-equation using the density of the system rather than a wavefunction. DFT is very commonly used as it is able to give chemical accuracy, while having similar costs to HF theory. It has its origins in the Hohnberg-Kohn existence theorem which states there exists a one-to-one mapping between the ground state electronic energy of a molecular system and its observed electron density $\rho(\mathbf{r})$,

$$\rho(\mathbf{r}) = \int \psi^2(\mathbf{r}) \tag{2.26}$$

As it is an existence theorem, there was no prescription given for the construction of the functional that computes the electronic energy.

In the first of two crucial steps, Hohnberg and Kohn showed electronic density follows a variational principle [116]. To achieve this we assume there exists a well-behaved system which integrates to the actual number of electrons. The system's electron density will determine a candidate Hamiltonian and wavefunction. From the variational principle for MOs, it can be shown that the energy for the system being considered is always greater than or equal to the ground state energy E$_0$,

$$\int \psi_{candidate}(\mathbf{r}) \; \widehat{H}_{candidate} \; \psi_{candidate}(\mathbf{r}) d\mathbf{r} = E_{candidate} \geq E_0 \tag{2.27}$$

The second step, is the Kohn-Sham (KS) SCF method, which creates a framework to allow systematic improvement of the electronic energy as a function of density for a given functional. The KS SCF procedure, first assumes there exists a fictitious system of non-interacting electrons which has the same overall electronic density as the real system of interest. The energy functional, is created so as to divide up energy into the following components,

$$E_{Total}[\rho(\mathbf{r})] = E_T[\rho(\mathbf{r})] + E_V[\rho(\mathbf{r})] + E_J[\rho(\mathbf{r})] + \{E_X[\rho(\mathbf{r})] + E_C[\rho(\mathbf{r})]\} \tag{2.28}$$

where, E_T is the K.E. of the non-interacting electrons, E_V is the coulomb energy from electron-nuclei interaction, E_J is the coulomb energy of electron-electron repulsion, E_X is the exchange energy and E_C is correlation energy. In Equation 2.28, the $E_T + E_V + E_J$ terms represent the classical energy of $\rho(\mathbf{r})$. Whereas, the E_X and E_C terms represent the remaining energy terms i.e. E_X is the exchange energy arising from the antisymmetry of ψ and E_C is the correlation energy arising from the motions of individual electrons. The main thrust of research work into functionals, over the last 50 years, has been in the creation of accurate functionals for E_T, E_X and E_C. Subsequent sections sketch out sub-components of the BLYP and B3LYP functionals, following the presentation given in [93], as these are used in experimental studies later in the thesis. The Hartree K.E. functional is given by,

$$E_T^{H28} = -\frac{1}{2}\sum_i^n \int \psi_i(\mathbf{r})\nabla^2\psi_i(\mathbf{r})d\mathbf{r} \tag{2.29}$$

this is used by HF theory as seen in Equation 2.8. Equation 2.29 was modified by Fock, in 1930, to account for the anti-symmetric nature of the wavefunction and the Fermi correlation or exchange term,

$$E_X^{F30} = -\frac{1}{2}\sum_i^n\sum_j^n \int\int \frac{\psi_i(\mathbf{r_1})\psi_j(\mathbf{r_2})}{|\mathbf{r_1}-\mathbf{r_2}|}d\mathbf{r_1}d\mathbf{r_2} \tag{2.30}$$

The original reference system in DFT [147] was chosen to the Jellium, an idealized system of N positive electrons which are uniformly distributed in a box of volume V so as to have a net neutral charge. Dirac showed the exchange energy for Jellium was,

$$E_X^{D30} = -\frac{3}{2}\left(\frac{3}{4\pi}\right)^{\frac{1}{3}} \int \rho^{4/3}(\mathbf{r})d\mathbf{r} \tag{2.31}$$

This functional was in error by 10% in comparison to energies predicted by $F30$ (Eqn. 2.30). To address this, Becke [22] in 1998 created an effective functional using $D30$ and a semi-empirical fitting parameter $b = 0.0042$, which was determined using $F30$ exchange energies for the first six Nobel atom gases [2],

$$E_X^{B88} = E_X^{D30} - b\int \rho^{4/3}(\mathbf{r})\frac{x^2}{1 + 6\,b\,x^2\sinh^{-1}x}d\mathbf{r} \tag{2.32}$$

In 1988 Lee, Parr and Yang [157] abandoned Jellium as a reference system and used the isolated Helium atom instead. This led to the LYP correlation functional which is combined with Becke's exchange functional and refined to BLYP,

$$E^{BLYP} = E_T^{H28} + E_V + E_J + E_X^{B88} + E_C^{LYP} \tag{2.33}$$

The B3LYP functional which is based on E^{BLYP} but includes HF exchange and thus is referred to as a hybrid-functional, uses $F30$ in the Kohn-Sham SCF procedure,

$$\begin{aligned} E^{B3LYP} = \; & E_T^{H28} + E_V + E_J \\ & + (1-c_1)E_X^{D30} + (c_1)E_X^{F30} \\ & + (c_2)E_X^{B88} + (1-c_3)E_C^{VWN} + (c_3)E_C^{LYP} \end{aligned} \tag{2.34}$$

where coefficients c_1, c_2, c_3 are obtained from fitting experimental data and E_C^{VWN} is a correlation functional developed by Vosko, Wilk and Nusair [292]. B3LYP is routinely used functional by quantum chemists owing to its lower runtime cost, and the ability to yield chemically meaningful results [137].

2.3.4.1 Kohn-Sham Equations

The Kohn-Sham (KS) equations for DFT allow for a variational approach [116] to obtaining electronic energies using a given systems electron density. The KS equations are given as,

$$\hat{f}^{KS}\varphi_i = \varepsilon_i \; \varphi_i \tag{2.35}$$

where, $\hat{f}^{KS}$ is the KS Fock operator; φ_i are KS orbitals and ε_i is a diagonal matrix of orbital energies. The KS Fock operator is defined as,

$$\hat{f}^{KS} = -\frac{1}{2}\nabla^2 - \sum_k^{nuclei} \frac{Z_k}{|\mathbf{r}-\mathbf{r}_k|} + \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}' + V_{XC} \tag{2.36}$$

where, the first two terms denote the kinetic energy and nuclear attraction; the second last term refers electron-electron coulomb terms and the last V_{XC} is a functional derivative of E_{XC} w.r.t density i.e. $V_{XC} = \frac{\delta E_{XC}}{\delta \rho}$. If the KS orbitals φ are expanded using a set of N basis functions ϕ,

$$\varphi_i = \sum_{\mu=1}^{L} c_{\mu i}\phi_\mu \tag{2.37}$$

in effect, this renders the problem similar to the HF method (Section 2.3.1). Thus the KS matrix equations [147] are,

$$F^{KS}C = S\,C\,\varepsilon \tag{2.38}$$

where, F^{KS} is the KS Fock matrix; S is the overlap matrix; C is the matrix of MO coefficients and ε is a matrix of MO energies. Here, F^{KS} is defined as,

$$\begin{aligned} F^{KS} &= H^{core} \\ &+ \int\int \phi_\mu(\mathbf{r})_1 \ \frac{\rho(\mathbf{r}_2)}{r_{12}} \phi_\nu(\mathbf{r}_1) d\mathbf{r}_1\mathbf{r}_2 \\ &+ \int \phi_\nu(\mathbf{r}_1)\ V_{XC}(\mathbf{r}_1)\ \phi_\nu(\mathbf{r}_1) d\mathbf{r}_1 \end{aligned} \tag{2.39}$$

$$= H^{core} + J_{\mu\nu} + V_{\mu\nu}^{XC}$$

where, $\rho(\mathbf{r}) = \sum_i^N |\varphi_i(\mathbf{r})|^2$ is the density; $J_{\mu\nu}$ is the coulomb term and $V_{\mu\nu}^{XC}$ is the exchange-correlation term.

2.3.4.2 Evaluation of the Coulomb and Exchange Terms

DFT handles both the coulomb and XC terms differently to HF. In HF, the coulomb terms are evaluated as four two-center electron repulsion integrals (ERI). These ERI are required in HF for both coulomb energy and for evaluation the exchange energy. In contrast, in DFT, these are de-coupled by the use of approximate functionals. Coulomb terms in DFT are handled by re-writing $J_{\mu\nu}$ as,

$$J_{\mu\nu} = \int\int \phi_\mu(\mathbf{r}_1)\phi_{nu}(\mathbf{r}_1)\ \frac{\rho(\mathbf{r}_2)}{r_{12}} d\mathbf{r}_1 d\mathbf{r}_2 \tag{2.40}$$

Furthermore if the density $\rho(\mathbf{r}_2)$ is expanded in terms of an auxiliary basis set ω such that,

$$\rho(\mathbf{r}) \approx \tilde{\rho}(\mathbf{r}) = \sum_\kappa^K c_\kappa \omega_\kappa(\mathbf{r}) \tag{2.41}$$

This expansion reduces the computational cost of evaluating J from $O(N^4)$ to $O(N^2K)$ [147], where N is the number of AO functions. Standard ERI techniques are then used to evaluate the associated three center integrals.

HF theory has exchange integrals but does not include electron correlation. In DFT both the exchange and correlation terms are given by,

$$\int \phi_\nu(\mathbf{r}_1)\ V_{XC}(\mathbf{r}_1)\ \phi_\nu(\mathbf{r}_1) d\mathbf{r}_1 \tag{2.42}$$

is evaluated entirely using numerical quadrature. Standard meshes have been defined for DFT calculations [94], which are then used to evaluate V_{XC} at each point on the grid. The following discretization is used in evaluating the exchange and correlation terms,

$$V_{\mu\nu}^{XC} \approx \tilde{V}_{\mu\nu}^{XC} = \sum_{p}^{P} \phi_{\mu}(\mathbf{r}_p)\ V_{XC}(\mathbf{r}_p)\ \phi_{\nu}(\mathbf{r}_p)\ W_p \tag{2.43}$$

which approximates V_{XC} into a series of P terms. Each term is then computed as a product of numerical values of $\phi_{\mu}(\mathbf{r})$, $\phi_{\nu}(\mathbf{r})$ for the exchange-correlation potential V_{XC} at each point $\mathbf{r}$ on the grid. These are then multiplied by weights W_p. We note that equations 2.42 and 2.43 represent a simplification of DFT known as the local density approximation (LDA) [93, 147], whereas the BLYP and B3LYP functionals are gradient corrected functionals i.e.

these functionals include terms that are dependent on the gradient of the system's electron density and are referred to as generalized gradient approximation (GGA) [212] functionals. Consequently the equations for the GGA involve additional terms requiring basis function derivatives.

Most DFT codes use the prescription given by Becke [22], where the volume occupied by all the atoms are divided into separate, but overlapping regions using Voronoi polyhedra. This is done so that the sum of individual integrals gives the total integral being sought,

$$V_{\mu\nu}^{XC} = \sum_{A} \int F_A(\mathbf{r}) dr \tag{2.44}$$

where the functional F_A is evaluated for all atom fragments volumes. Each of the individual integrals are then computed using spherical polar quadrature [2],

$$\begin{aligned} V_{\mu\nu}^{XC} &= \int_0^{\infty} \int_0^{\pi} \int_0^{2\pi} F_A(\mathbf{r}, \theta, \phi)\ r^2\ sin\theta\ d\mathbf{r}\ d\theta\ d\phi \\ &\approx \sum_{rad} \sum_{ang} w_{rad} w_{ang} F_A(\mathbf{r}_i) \\ &\approx \sum_{p}^{P} w_p^{rad} \sum_{q}^{Q} w_q^{ang}\ F_A(\mathbf{r}_p, \theta_q, \phi_q) \end{aligned} \tag{2.45}$$

where, there are P radial and Q angular points, corresponding to radial and angular weights w^{rad}, w^{ang}. The integration grid for each atom is obtained as a direct product grid of both the radial and angular quadrature grids; radial quadrature is done on a sphere using Lebedev grids [2, 147], whereas angular quadrature can be done using various prescriptions, of which the Gaussian quantum chemistry code uses the Euler-MacLaurin scheme [275, 304].

To conclude this section on DFT, there are no analytic expressions for DFT functionals rather numerical quadrature over three dimensional grids is used to obtain the electronic energy of the system under consideration,

$$E \approx \sum_{i=1}^{N_{grid}} w_i f(\rho(\mathbf{r}_i), x(\mathbf{r}_i)) \tag{2.46}$$

where $f(\rho(\mathbf{r}_i), x(\mathbf{r}_i))$ is a density functional and w_i are weights used for numerical quadrature.

Chapter 3

Thread and Memory Placement on Non-Uniform Memory Access Systems

3.1 Introduction

Creation of scalable shared memory multiprocessor systems has essentially been made possible by cache-coherent Non-Uniform Memory Access (cc-NUMA) hardware [154, 159]. This approach uses a basic building block comprising one or more processors with local memory and an interlinking cache coherent interconnect [58]. Unlike Uniform Memory Access (UMA) systems which are composed of processors with identical cache and memory latency characteristics, NUMA systems exhibit asymmetric memory latency and possibly asymmetric bandwidths between its building blocks. On such platforms the operating system should consider physical processor and memory locations when allocating resources (i.e. memory allocation and CPU scheduling) to processes [291]. To accommodate these characteristics, operating systems, such as Solaris and Linux, have been extended to be "NUMA-aware" and to provide application programmer interfaces that allow the user to perform specific thread and memory placement. Pthreads [64] and OpenMP [49, 228] are two widely used programming models that target shared memory parallel computers. Both, however, were developed for UMA platforms and make no assumptions about the physical location of memory or where a thread is executing. Although there has been debate about the merit of adding NUMA extensions to these programming models [48, 57, 194], there has been no officially accepted extensions to

A portion of work reported in this Chapter was carried out in collaboration with Pete Peerapong Janes (Department of Computer Science, ANU). Material from this chapter first appeared in: Proceedings of HiPC 2006, EXPLORING THREAD AND MEMORY PLACEMENT ON NUMA ARCHITECTURES: SOLARIS AND LINUX, ULTRASPARC/FIREPLANE AND OPTERON/HYPERTRANSPORT, `http://dx.doi.org/10.1007/11945918_35` either Pthreads or OpenMP to support this. Though we do not explicity consider the use of codes that use the Message Passing Interface (MPI)[1], the work presented here is of use to such codes as the MPI API does not expose locality of execution and memory to applications that use it, and hence MPI code performance at a per node level is subject to the OS' memory and thread placement decisions.

The aims of this chapter are two fold: first, to develop a framework in terms of tools and protocols to facilitate memory and thread placement experiments on NUMA systems; second, to propose and test a placement distribution model (PDM) which attempts to classify observed performance.

In the context of the thread and memory placement framework, the following are discussed with reference to both the Solaris and Linux operating systems,

(a) the specifics of how a user-space thread can be bound to a specific processor;

(b) how to affect the allocation of memory onto specific memory banks[2];

(c) verification that the thread is running on a specific processor and memory has been allocated at the requested location.

The PDM is a means of classifying observed performance results, from experiments that use specific thread and memory placement. It uses directed graphs representing processor, memory and processor interconnect layout to categorize results into groups of "contention classes". Contention classes denotes the degree of contention over interconnect links which exist between processor nodes. Using the tools and protocols framework and the PDM, performance characteristics of two contemporary NUMA architectures – the UltraSPARC [267] using the FirePlane interconnect [50] and the Opteron [146] using HyperTransport [56, 218, 282] - are explored through a series of latency, bandwidth and basic linear algebra (BLAS) experiments.

The Chapter is structured into the following sections – thread, memory placement and its verification on Solaris and Linux is discussed in Section 3.2. The experimental hardware and software platforms used are described in section 3.3, while Section 3.4 outlines the basic latency and bandwidth experiments. Section 3.5 outlines and evaluates the Placement Distribution Model using `Stream` benchmark [177, 178] and BLAS [10]. The Chapter concludes with Section 3.6 covering related work and section 3.7 presenting conclusions.

[1]MPI is the de-facto API standard used to encode message passing parallel applications.

[2]A memory bank is a set of DIMMs, which are accessible to a processor. Specifically, the interest here is to allocate memory in specific memory locations which are local to a given thread of execution.

3.2 Thread and Memory Placement

Conceptually, both Solaris and Linux are similar in their approach to abstracting underlying groupings of processors and memory based on latency. Yet, the mechanics of using the two NUMA APIs are quite different. Below we provide a brief review of Solaris thread and memory placement APIs, before contrasting this with the Linux NUMA support. We then consider placement verification for both Solaris and Linux.

3.2.1 Solaris NUMA Support

Solaris represents processor and memory resources as locality groups [138, 179]. A locality group (`lgrp`) is a hierarchical DAG (Directed Acyclic Graph) representing processor-like and memory-like devices, which are separated from each other by some access-latency upper bound. A node in this graph contains at least one processor and its associated local memory. All the `lgrps` in the system are enumerated with respect to the root node of the DAG, which is called the root `lgrp`. The root `lgrp` is special, in that it contains all other `lgrps`, i.e all the memory and processors in the system. `lgrps` are used by Solaris to make scheduling, load balancing and memory bandwidth optimisation decisions on a per page basis. Two modes of memory placement are available apart from the default first-touch policy, these are next-touch[3] and random[4]. The former is the default for thread private data, while the latter is useful for shared memory regions accessed by multiple threads as it can reduce contention. A collection of APIs for user applications wanting to use `lgrp` information or provide memory management hints to the operating system is available through `liblgrp` [262]. Memory placement is achieved using `madvise()`, which provides advice to the kernel's virtual memory manager. The `meminfo()` call provides virtual to physical memory mapping information. We also note that memory management hints are acted upon by Solaris subject to resources and system load at runtime. Threads have three levels of binding or affinity – *strong*, *weak* or *none* which are set or obtained using `lgrp_affinity_set()` or `lgrp_affinity_get()` respectively. The operating system will try to avoid moving threads with *strong* affinity to other `lgrps`, while those with *weak* affinity will be moved for load balancing purposes. Threads with an affinity of *none* will automatically be assigned a home `lgrp` by the operating system. Solaris' memory placement is determined firstly by the allocation policy and then with respect to threads accessing it. Thus there is no direct API for allocating memory to a specific `lgrp`, rather a first touch memory policy must be in place and then memory allocated by a thread that is bound to that specific `lgrp`. Within an `lgrp` it is possible to bind a specific thread to a

[3]The next thread which touches a specific block of memory will possibly have access to it locally i.e. if remote memory is accessed it will possibly be migrated.

[4]Memory is placed randomly amongst the `lgrps`.

specific processor by using the `processor_bind()` system call.

3.2.2 Linux NUMA Support

NUMA scheduling and memory management became part of the mainstream Linux kernel as of version 2.6. Linux uses NUMA distances (i.e. number of hops from either CPUs or memory) obtained from the ACPI's (Advanced Configuration and Power Interface) SLIT (System Locality Information Table). Using this, it assigns NUMA policies in its scheduling and memory management subsystems. Memory is managed per NUMA node using pools of pages which are again per node [54]. Within the kernel, each NUMA node has a swapper thread, which is responsible for memory allocation on that node.

Memory management policies include *strict*[5] allocation to a node, *round-robin*[6], and *non-strict preferred* binding to a node (meaning that allocation is to be *preferred* on the specified node, but should fall back to a default policy if this proves to be impossible).

In contrast, Solaris specifies policies via `madvise()` for shared and thread local data i.e its API is descriptive in informing the kernel about possible access patterns by threads and as a result may migrate pages [7].

The default Linux NUMA policy is to map pages on to the physical node which faulted them in, and in many cases maximises data locality. A number of system calls are also available to implement different NUMA policies. These system calls modify scheduling (`struct task_struct`) and virtual memory (`struct vm_area_struct`) related variables structures within the kernel. Example system calls include `mbind()`, which sets the NUMA policy for a specific memory area; `set_mempolicy()`, which sets the NUMA policy for a specific process; and `sched_setaffinity()`, which sets a process' CPU affinity. Several arguments for these system calls are supplied in the form of bit masks, and macros, which makes them relatively difficult to use.

For the application programmer a more attractive alternative is provided by the `libnuma` API. Alternatively, `numactl` is a command line utility that allows the user to control the NUMA policy and CPU placement of a entire executable (and can also be used to display NUMA related hardware configuration and status). Within `libnuma`, useful functions include the `numa_run_on_node()` call to bind the calling process to a given node and `numa_alloc_`
`onnode()` to allocate memory on a specific node. Similar calls are also available to allo-

[5]Memory allocation is to occur at a given node. It will fail if there is not enough memory.

[6]Memory is dispersed equally amongst the nodes.

[7]Refer to manpages for `madvice(3C)` and `lgrp_affinity_set(3LGRP)`. Solaris uses this information to make placement decisions but may not act on this subject to system load and memory pressure at the time the request was made

cate interleaved memory, or memory local to the caller's CPU. In contrast to Solaris' memory allocation procedure, `numa_alloc` calls result in modifications to variables within the process' `struct vm_area_struct` in the Linux kernel. Thus, the physical location of the CPU/core that performs the memory allocation is irrelevant i.e. any given thread can allocate memory to memory banks that are not local to it. The `libnuma` API can also be used to obtain the current NUMA policy and CPU affinity. To identify NUMA related characteristics `libnuma` accesses entries in `/proc` and `/sys/devices`. This makes applications using `libnuma` more portable those that use the lower level system calls directly.

3.2.3 Placement verification in Solaris and Linux

Solaris provides a variety of tools[8] to monitor process and thread lgroup mappings – `lgrp info`, `pmadvise`, `plgrp` and `pmap` on the Opteron and V1280 systems. The `lgrpinfo` tool displays the lgroup hierarchy for a given machine. The `pmadvise` tool can be used to apply memory advice to a running process. The `plgrp` tool can observe and affect a running thread's lgroup, it can also diagrammatically represent the system affinities for the underlying hardware platform. The `pmap` tool permits display of lgroups and physical memory mapping for all virtual address associated with a running process.

On Linux, `libnuma` provides a means for controlling memory and process placement on Linux systems, it does not provide a means for determining where a given area of memory is physically located. A kernel patch that attempts to addresses this issue is provided by Per Ekman [211]. The patched kernel creates per-PID `/proc` entries that include, among other things, information about which node a process is running on, and a breakdown of the locations of each virtual memory region belonging to that process. While we found that this package was generally sufficient as a verification tool it involved having to check quickly the `/proc` entries while the program was running. We also found that under some circumstance the modified kernel failed to free memory after a process had terminated. Based on the work of Ekman [211] we designed an alternative kernel patch that provides a system call and user level function to return the memory locations for each page in a given virtual memory range. This utility proved considerably more convenient as it could be called from within a running application. Recent Linux kernels have better NUMA visibility allowing user-space access to both NUMA specific allocation information [286], obviating the need for the patch. There is also the ability to find out where individual pages reside and if required affect their migration [9].

[8]`http://opensolaris.org/os/community/performance/numa/observability`

[9]Man page for `migrate_pages()`

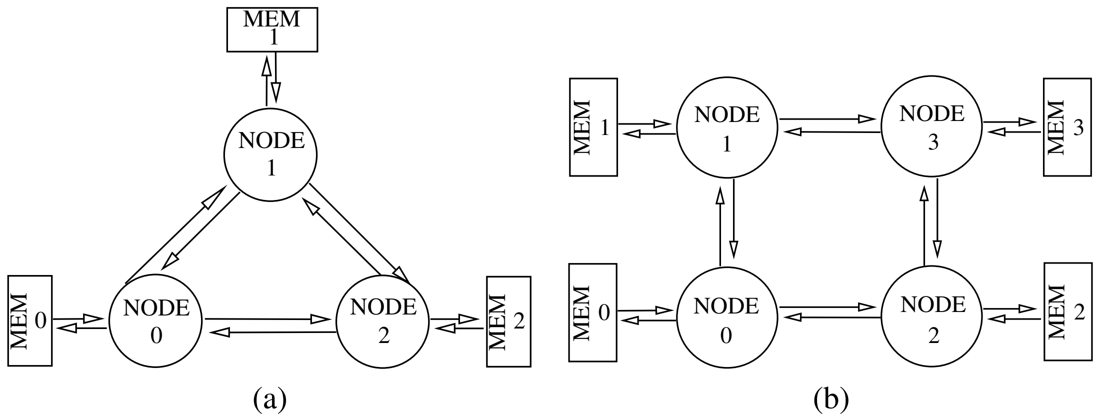


Figure 3.1: (a) Schematic diagram of the V1280 UltraSPARC platform and (b) Celestica Opteron platform

3.3 Experimental platforms

Two NUMA platforms were used in this work: a twelve processor Sun UltraSPARC V1280 [266] and a four processor AMD848 Opteron system based on the Celestica A8448 [47] motherboard. Schematic illustrations of the V1280 is given in Figure 3.1 (a) and the Opteron system is given in Figure 3.1 (b). The V1280 has twelve USIIICu CPUs clocked at 900Mhz, whereas the A8448 has four AMD848 processors clocked at 2.2Ghz.

The V1280 has, as a result of its topology, two NUMA domains[10]. The first NUMA domain arises from fixed latencies experienced by CPUs in a given node accessing memory local to it. The second NUMA domain denotes CPU accesses to memory which is not local to it. Another way of describing this is that any CPU in the V1280 experiences one hop to local memory or two hops to non-local memory.

There are three NUMA domains for the A8448; the first domain, is to memory which is local to a node (i.e. Node 0 accessing MEM 0, which is one hop away); the second domain is to memory which is two hops away (i.e. Node 0 accessing MEM 1 or MEM 2); the third domain is to memory which is more than one hop away (i.e. Node 0 accessing MEM3, which is three hops away).

3.3.1 Software Environment

While Solaris 10 was used on the V1280 system, the Opteron platform was configured to boot either Solaris 10 or OpenSuSE 10.

The Sun Studio 11 compilers [263] were used on Solaris platforms, while version 6.0 of the

[10]For clarity, a NUMA domain designates those CPUs which when grouped together, exhibit uniform latencies in accessing a set or sets of memory banks

Portland Group [276] compilers were used under Linux[11]. Flags for the highest optimisation levels were used on both compilers.

To obtain accurate performance data, the PAPI library [37] was used to access hardware performance counters under Linux, while the `libcpc` [261] infrastructure was used under Solaris.

Numeric libraries used under Linux are the ACML (version 3.0) from AMD [185], ATLAS (version 3.6) [299] and GOTO BLAS [100] (version 1.00), while Sunperf (Sun Studio 11) [264] was used under Solaris.

3.4 Basic Latency and Memory Bandwidth Characterization

This section discusses observed memory latency, serial memory bandwidth and parallel memory bandwidth for the two NUMA platforms.

3.4.1 Latency Characterisation

To determine the memory latency characteristics of the two platforms the `lmbench` [184] memory latency benchmark was modified to accept memory and thread placement parameters. Latencies to get data from level-one cache (L1) on the Opteron and UltraSPARC were measured as 3 and 2 cycles respectively, while accessing level-two cache (L2) took 20 cycles on both platforms. The latencies recorded for a thread bound to a particular node accessing memory at a specific location are given in Table 3.1. From these, the NUMA ratio[12] of the Opteron system is found to be 1.11 for one hop and 1.53 for two hops from any given processor, while on the V1280 there is only one NUMA level with a ratio of 1.2.

3.4.2 Bandwidth Characterisation

To determine the memory bandwidth characteristics of the two platforms the `Stream` benchmark (cf. Table 3.4.1) was modified to accept memory and thread placement parameters. This benchmark performs four different vector operations, corresponding to vector copy, scale, add, and triad. On the Opteron there are four nodes and four physically distinct memory locations, while on the UltraSPARC there are three nodes and three memory banks. For a single thread one would expect that the "best" possible `Stream` performance would be obtained when a

[11]At the time when the experiments were performed, Sun Studio compilers for Linux were available as an alpha release and hence we decided to use the Gaussian recommeded PGI compilers instead.

[12]NUMA ratio = $\frac{RemoteLatency}{LocalLatency}$

Table 3.1: Main Memory latencies (Cycles) from `lmbench`. The pointer chasing benchmark from `lmbench` is used to determine memory latencies. Results were obtained for the Opteron and V1280 platforms by pinning a thread on a given node and placing memory on different nodes.

Thread	Memory Location						
	Opteron				V1280		
Location	0	1	2	3	0	1	2
0	225	250	250	345	220	265	265
1	250	225	345	250	265	220	265
2	250	345	225	250	265	265	220
3	345	250	250	225	–	–	–

Table 3.2: Stream benchmarks

Copy	$C(:) = A(:)$
Scale	$B(:) = s * C(:)$
Add	$C(:) = A(:) + B(:)$
Triad	$A(:) = B(:) + s * C(:)$

thread is accessing vectors that are stored entirely in local memory. Conversely the "worst" possible performance would correspond to a thread accessing data stored in memory located as far away as possible. Results for these two scenarios are given in Table 3.3. For the Opteron system running Solaris we find performance differences between best and worst memory placement vary from a factor of 1.4 to 1.6. For Linux on the same platform we find a somewhat larger variation with factors between 1.09 to 2.35. On the V1280 system the effect is considerably less indicating relatively mild NUMA characteristics. (We note that the superior performance of the copy operation on the Opteron using Linux reflects the use of specialised instructions by the PGI compiler to perform the memory moves).

Table 3.3: Serial `Stream` bandwidths (GB/s) for the Opteron and V1280 systems. A single thread was pinned to a given node and had its memory placed on different nodes. Best and Worst refer to thread and memory placements which are expected to give the best and worst possible performance (See text for details).

	Opteron				V1280	
	Solaris		Linux		Solaris	
Test	Best	Worst	Best	Worst	Best	Worst
Copy	2.17	1.99	4.68	3.14	0.72	0.71
Scale	2.50	1.58	2.35	1.47	0.79	0.74
Add	2.75	1.17	2.55	1.54	0.83	0.81
Triad	2.24	1.51	2.44	1.52	0.85	0.79

Table 3.4: Parallel `Stream` bandwidths (GB/s). Threads were pinned to various nodes and had its memory placed locally ("Best") or remotely ("worst"). Four threads were run concurrently for the Opteron while twelve threads were run concurrently for the V1280 system.

	Opteron				V1280	
	Solaris		Linux		Solaris	
Test	Best	Worst	Best	Worst	Best	Worst
Copy	8.98	2.53	16.55	4.22	4.89	3.56
Scale	9.98	2.67	9.60	2.67	4.91	3.46
Add	10.85	2.85	10.33	2.94	5.22	3.57
Triad	9.17	2.68	9.87	2.96	5.14	3.71

A threaded version of the `Stream` benchmark was run using all available processors and for all possible thread and memory placements on both the Opteron and V1280 systems. Results for this are presented in Table 3.4, in terms of "Best" and "Worst" results. It was found that the worst case scenario on the Opteron corresponds to all of a given node's data being allocated the maximum number of hops away from it i.e. all of node 1's data resides on node 3. Not surprisingly on both the Opteron and V1280 system the difference between good and bad memory placement has increased significantly over that observed for the serial benchmark.

3.5 A Placement Distribution Model

In the above we considered "best" and "worst" case scenarios for the various `Stream` benchmarks. In the general case, on the Opteron system, each vector or data quantity used in a `Stream` benchmark could be located in the memory associated with any one of the four available nodes. For the parallel add and triad benchmarks, on the Opteron system, this means that there are a total of 4^{16} possible thread/memory combinations[13] while $4^8 * 4!$ copy and scale benchmarks are possible (add and triad use 3 data quantities while copy and scale use 2 data quantities). Evaluating the performance characteristics of each of these cases quickly becomes impossible for large NUMA systems. Hence, it would be useful to develop a model that can categorize different thread, memory placements and subsequently be used as a set/pool of configurations to sample from.

With this aim, a Placement Distribution Model (PDM) was developed that attempts to categorize the occurrence and type of possible thread and memory placements. The inputs to the model are a directed graph of the NUMA system, which describes its processor and

The PDM was co-developed with Pete Peerapong Janes (Department of Computer Science, ANU)

[13]A given data quantity could reside in 4 possible memory locations and each thread could run on 4 possible processors i.e. there are a total of 4^3 experiments for one thread and three data quantities. For all the 4 threads in the system there are $4^3 * 4 * 4^3 * 3 * 4^3 * 2 * 4^3 * 1 = 4^{12} * 4!$ possible combinations.

memory layout, and the data quantities used per thread. Thus, Figures 3.1 (a) and (b) can be re-interpreted as graphs, where links entering and exiting nodes are arcs. Traffic associated with each link is modeled as weights along the links between nodes. To simplify cache coherent interactions for the NUMA system, we assume that nodes will route cache traffic to their local on-chip memory controller or if the data quantity of interest is not local to the node, the request is forwarded by the memory controller along the most direct path. This model is a simplification of the coherent HyperTransport (cHT) protocol [56, 218] where cHT's Request, Probe, Response message sequence is replaced by a discrete message to the home node owing the data quantity (instead of the cache line), via links between the nodes. The modelling does not account for coherency traffic which is broadcast amongst various nodes, rather link and node contention are accounted for.

An algorithm to enumerate the number of thread and memory configurations, as well as categorizing them into contention classes is presented in Algorithm 1. Inputs to the PDM are $\mathbb{N}$, the set of all processor nodes; $\mathbb{M}$, the set of all memory nodes; $\mathbb{L}$, the set of all links between both memory and processor nodes; $\mathbb{T}$, the set of all data quantities in use by the threads. From this, the set $\mathbb{E}$, is formed which denotes all possible thread and memory placements for data movement. A graph $\mathbb{G}$, represents processors and memory layout of the NUMA platform. The set $\mathbb{D}$ denotes a given data quantity which resides in a particular memory node, and the set of inputs (i.e. processor and memory and data quantities are represented by the set $\mathbb{I}$). These inputs are used to traverse over all possible thread and memory placement configurations for each data quantity. A traversal implies data quantities are moved over a link and this entails a cost $W(l)$ per link l. Each traversal contributes to a cumulative entry in the cost matrix C. Three procedures are defined in Algorithm 1 namely *OptPath*, *FlowSize* and *ComputeDistribution*. Procedure *OptPath* returns the optimal path i.e a set of ordered pairs of $<x,y>$ between two end points $<n,m>$ while procedure *FlowSize* computes a cost associated with moving data quantities contained in set $\mathbb{Q}$ over links contained in set $\mathbb{P}$ and procedure *ComputeDistribution* uses a set of data quantities as used per thread for all threads in set $\mathbb{Q}'$ and computes the cost for these data quantities for an ordered set of inputs $\mathbb{I}$.

A state machine was coded to perform walks along the links of graph $\mathbb{G}$, for all possible thread and memory placements given a specific processor/memory topology and data quantities. These walks model link traffic moving from a source node to a target node, traffic moving from a node to its local memory and traffic moving from one memory bank to another. In the event that there are two paths to the required destination of equal length, the traffic is split equally along each path. This assumption is made as a simplification to avoid complex specification of the underlying coherency protocol. For example, if a placement dictates that Node 1 will be continuously accessing memory from Node 0, we increment variables belonging to each connector along the route to record the quantity of the data movement. This results in

Algorithm 1 Algorithm for the Placement Distribution Model

```
1: $\mathbb{N} \leftarrow \{node_1,\ node_2,\ \dots\ , node_i\}$        ▷ The set of all processor nodes
2: $\mathbb{M} \leftarrow \{mem_1, mem_2,\ \dots\ , mem_j\}$        ▷ The set of memory nodes
3: $\mathbb{L} \leftarrow \{\ link_1,\ link_2,\ \dots,\ link_k\}$        ▷ The set of all links between nodes
4: $\mathbb{T} \leftarrow \{\ data_1,\ data_2,\ \dots,\ data_l\}$        ▷ The set of data quantities
5: $\mathbb{E} \leftarrow \mathbb{N}\ \mathrm{x}\ \mathbb{M}$        ▷ Cartesian product denoting data movement
6: $\mathbb{G} \leftarrow <\mathbb{E}, \mathbb{L}>$        ▷ Graph $\mathbb{G}$ representing memory and processor layout
7: $\mathbb{D} \leftarrow \{<x,y>\ | x \in \mathbb{T},\ y \in \mathbb{M}\}$        ▷ A data quantity x resides in memory location y
8: $\mathbb{I} \leftarrow \mathbb{E}\ \mathrm{x}\ \mathbb{D}$        ▷ Set of inputs for thread, memory placement
9: $\mathbb{I} \equiv \{<e, f>\ | e = <n,m> \in \mathbb{E}, f = <x,y> \in \mathbb{D}\}$
10: $W(l) \mid l \in \mathbb{L}$        ▷ Weight matrix W
11: $C(x,y)$        ▷ Cost matrix C

Require: $<n,m> \in \mathbb{E}$
12: procedure OPTPATH($<n,m>$)        ▷ Optimal path from n to m where $n,m \in \mathbb{E}$
13:                                    ▷ Use appropriate alogrithm or heuristic
14:     return $\{<x,y>\ | x,y \in \mathbb{L}\}$        ▷ to get path between $<n,m>$
15: end procedure

Require: $x \in \mathbb{D}\ \forall x \in \mathbb{Q}$
Require: $<x,y> \in \mathbb{L}\ \forall\ <x,y> \in \mathbb{P}$
16: procedure FLOWSIZE($\mathbb{Q}, \mathbb{P}$)        ▷ Compute cost of moving data items across link $\mathbb{P}$
17:     cost $\leftarrow 0$
18:     for all (link $\in \mathbb{P}$) do
19:         for all (qty $\in \mathbb{Q}$) do
20:             cost $\leftarrow$ cost + $|$ qty $| * W(link)$
21:         end for
22:     end for
23:     return cost
24: end procedure

25: procedure COMPUTEDISTRIBUTION
26:     $\mathbb{Q}' \leftarrow \{x \mid x \in \mathbb{D}\}$        ▷ Set of data quantities of interest
27:     for all ($i \in \mathbb{I}$) do        ▷ Loop over input $\mathbb{I}$ ($i \equiv <e,f>$)
28:         links $\leftarrow OptPath(e)$ where $e \in i$        ▷ Get the optimal path for a given e
29:         for all (($j \leftarrow$ links) $\wedge$ ($f \in i$)) do        ▷ Loop over links and use $f \in <e,f>$
30:             $C(i,j) = C(i,j) + FlowSize(\mathbb{Q}', j)$
31:         end for
32:     end for
33: end procedure
```

Table 3.5: Copy and Scale (GB/s) `Stream` benchmark results for the placement distribution model. Contention classes denote the ranges of link contention for all the nodes in the system. %Fr gives the frequency of occurrence of a given class in percent. The standard deviation (σ) for Copy, Scale are for twenty random samples from each contention class. Each thread and memory configuration was run ten times.

Contention		Solaris				Linux			
Class	%Fr	Copy	σ	Scale	σ	Copy	σ	Scale	σ
Opteron									
2-3	2.6	5.7	0.6	6.0	0.9	7.4	1.1	5.6	0.5
3-4	51.9	5.0	0.6	5.1	0.8	6.7	1.1	4.7	0.6
4-5	34.6	4.5	0.4	4.8	0.5	6.4	0.4	4.2	0.5
5-6	9.2	3.9	0.3	4.6	0.3	5.5	0.6	3.6	0.4
6-7	1.5	3.3	0.2	3.4	0.3	4.4	0.3	3.0	0.4
7-8	0.1	3.0	0.6	3.0	0.6	3.3	0.9	2.7	0.6
V1280									
08-12	10.3	4.0	0.4	4.0	0.4				
12-16	59.7	3.8	0.4	3.9	0.4				
16-20	24.8	3.7	0.5	3.8	0.3				
20-24	5.0	3.6	0.8	3.6	0.8				

a tuple holding values for link contention and node contention. Using the PDM, for a given processor and memory layout, we can obtain costs for thread and memory placement which are distributed in ranges which we term as *link contention classes*. Here link contention classes specifically refers to tuples of contention classes pertaining to links between processors. The range of any given tuple gives the degree of contention upon links which originate from a given node. By enumerating all possible thread and memory placements and grouping these into contention classes, performance experiments can be run by sampling potential placements from a given contention class. Hence, given a particular NUMA topology, the PDM is able to create contention class classifications for various thread and memory placements.

3.5.1 Stream Experiments

Table 3.5 characterises the copy and scale `Stream` benchmarks according to the maximum level of contention on any given link. This table shows, for example, that on the Opteron system 51.9% of all possible memory placement configurations have link contentions greater than or equal to 3 but less than 4, while 0.1% have a link contention of between 7 and 8. The ranges 3 – 4 and 7 – 8 are the *link contention classes*. For the V1280, 59.7% of all possible memory placement configurations have link contention between 12 and 16, whereas 5.0% of configurations have link contention between 20 and 24.

To test the PDM's usefulness, for each contention class obtained from the PDM, twenty random configurations were generated i.e. thread and memory placement for all threads and data quantities which yields a link contention that lies in the range of all observed link contention classes in Table 3.5. These placement configurations are subsequently used to perform Copy and Scale `Stream` measurements. All experiments were run ten times for each of the twenty random configurations.

On the Opteron, `Stream` Copy results have a standard deviation (σ) between 0.2 to 0.6 GB/sec (for Solaris) i.e. 0.6% to 10% variation. The results are between 0.3 to 1.1GB/sec (for Linux) i.e. 7% to 15% variation. For `Stream` Scale, the results have a standard deviation between 0.3 to 0.9 GB/sec (for Solaris) and from 0.4 to 0.6 GB/sec (for Linux); i.e. 9% to 15% (Solaris) and 11% to 13% (Linux). `Stream` Copy results for the V1280 have a standard deviation between 0.4 to 0.8 GB/sec (11% to 22% variation) and the Scale results have a standard deviation between 0.3 to 0.8 GB/sec (10% to 22% variation).

The results also show, that on the Opteron system given random vector placement the probability of landing in a 3 – 4 link contention class is the highest, and within this class one would expect to see a performance degradation of about 20% from the optimal result. Whereas on the V1280 the effect is much less. We also note that the link contention range for the V1280 is much greater, than that of the Opteron system, as there are a greater number of links between processor and memory nodes.

The PDM's contention classes indicate that thread and memory configurations for the lowest contention class gives better results than those of a larger contention class. For the Opteron, the largest standard deviations are observed for the 2 – 3, 3 – 4 and 7 –8 contention classes. For 2 – 3 and 3 – 4, these are 12% (Copy), 15% (Scale) for Solaris and 16% (Copy), 12% (Scale) for Linux. For 7 – 8, the standard deviations are 20% (Copy, Scale) for Solaris and 27% (Copy) and 22% (Scale). On the V1280, the largest standard deviation is observed for the 20 – 24 contention class and this equates to a 22% deviation.

The larger standard deviations arise as the PDM does not completely model the coherency transactions for either the Opteron or V1280 systems.

Overall, by using the standard deviation obtained for each contention class, the PDM is able to capture the nature of each contention class. For the lowest contention class, the standard deviations for the Opteron are between 12% to 15% on Solaris and 12% to 16% on Linux. While on the V1280 it is 10%. These results shows the PDM gives reasonable results and can be used to classify observed thread and memory placement performance experiments, without resorting to running experiments involving all possible thread and memory placements.

Table 3.6: BLAS `Stream` Triad, Level 2 BLAS, Level 3 BLAS (GigaFlops) results for the placement distribution model. Results are averages for twenty random generated configurations per contention class. Each configuration was run twenty times. Tr = Triad; B2 = BLAS Level 2; B3 = BLAS Level 3.

Contention		Solaris						Linux					
Class	%Fr	Tr	σ	B2	σ	B3	σ	Tr	σ	B2	σ	B3	σ
Opteron													
3 – 4	1.9	0.5	0.03	1.6	0.3	15.3	0.1	0.5	0.04	1.5	0.3	15.6	0.1
4 – 5	38.1	0.4	0.04	1.4	0.1	15.2	0.1	0.4	0.05	1.4	0.3	15.6	0.1
5 – 6	38.2	0.4	0.05	1.5	0.1	15.2	0.1	0.4	0.04	1.4	0.3	15.6	0.1
6 – 7	16.0	0.4	0.03	1.4	0.2	15.2	0.1	0.4	0.04	1.3	0.3	15.6	0.1
7 – 8	5.0	0.3	0.02	1.3	0.3	15.2	0.1	0.3	0.05	1.3	0.3	15.6	0.1
8 – 12	3.4	0.3	0.02	1.1	0.3	14.9	0.1	0.3	0.05	0.8	0.3	15.6	0.1
V1280													
12 – 16	8.3	0.4	0.05	1.0	0.1	17.4	0.7						
16 – 20	48.3	0.3	0.05	1.0	0.1	15.8	3.2						
20 – 24	30.7	0.3	0.05	1.0	0.1	16.2	2.9						
24 – 28	10.2	0.3	0.05	1.0	0.1	17.4	0.7						
28 – 40	2.3	0.3	0.05	1.0	0.1	17.5	0.6						

3.5.2 BLAS experiments

Using the memory placement framework developed above, experiments were conducted for level 2 (DGEMV – matrix vector) and level 3 (DGEMM – matrix multiply) BLAS operations. Results obtained for square matrices of dimension 1600 using `ACML` on the Opteron and `sunperf` on the V1280 are given in Table 3.6. In addition we also include results obtained from the triad `Stream` benchmark as these are representative of level 1 BLAS operations.

The contention class distributions for Triad, Level 2 BLAS (B2) and Level 3 BLAS (B3) vary from those given in Table 3.5, as there are three data quantities involved. There is a 38% chance for both the 4 – 5 and 5 – 6 contention classes on the Opteron and a 48.3% chance on the V1280.

The largest standard deviations on the Opteron are 12% (Triad/Solaris), 18% (L2 BLAS/-Solaris), 0.6% (L3 BLAS/Solaris), 16% (Triad/Linux), 37% (L2 BLAS/Linux), 0.6% (L3 BLAS/ Linux). For the V1280 these are 17% (Triad/Solaris), 10% (L2 BLAS/Solaris) and 20% (L3 BLAS/Solaris).

The results show greatest NUMA effects on the Opteron system, where, as expected the variation is largest for triad, less for Level 2 BLAS and almost constant for Level 3 BLAS. This reflects the fact that a well written DGEMM will spend most of its time working on data that is resident in the level 2 cache, but this is not possible for Level 1 or Level 2 BLAS where data must be streamed from memory to processor.

3.6 Related Work

Brecht [35] evaluates the importance of placement decisions on NUMA machines with different NUMA ratios. It was found that application placement which mirrored hardware was beneficial to application performance and its importance increased with the NUMA ratio.

Robertson and Rendell [226] quantify the effects of memory bandwidth and latency on the SGI Origin 3000 using `lmbench` and `Stream`. Using a 2D heat diffusion application, they stress the importance of good thread and memory placement and show that relying on the operating system for thread and memory placement is not always optimal.

Tikir and Hollingsworth [279] use link counters and a bus analyzer, on the SunFire 6800 system to effect transparent page migration, without modification to the operating system or application code. They are able to improve execution time of benchmarked applications by 16%. This is achieved by using a combination of hardware counters, runtime instrumentation and `madvise()`.

Tao et al. [134] describe an application tuning framework for NUMA architectures by using hardware monitoring tools capable of capturing inter-node coherency traffic. This in turn can be visualized offline to study the impact of specific memory placement on application performance.

Prestor and Davis [284] develop a device driver on the SGI Origin 2000 allowing user-level access to various hardware performance counters on the Origin's ASICs. A memory profiling tool which uses this driver allows for periodic profiling of both memory and coherency performance events. They are able to derive various performance metrics for a parallel application and show how various code modifications affect the performance of a parallel FFT kernel.

Löf and Holmgren [164] use the `madvise()` call and larger page sizes to reduce the execution time of a commercial PDE solver on a SunFire 15000. It is shown that the overhead associated with memory affinity changes could be fully attributed to the invalidation of DTLB entries.

SGI provides two user level data placement and verification tools – dplace, dlook [239]. The dplace tool allows a user to specify specific thread placement for the life of an application while dlook permits verification of placement. Both tools are available under SGI systems only.

3.7 Conclusions

The support for thread binding and memory placement provided by Solaris and Linux has been outlined and contrasted. For Linux, the kernel was modified in order to provide a user API that could be used to verify binding and determine physical memory placement from a user supplied virtual address. Using the various thread and memory placement APIs, a framework was outlined for performing NUMA performance experiments.

Detailed measurements of the latency, bandwidth and BLAS performance characteristics of two different hardware platforms were undertaken. These showed the Opteron system to be ”more NUMA” than the Sun V1280 system, despite the fact that it had only 4 processors. To assist with the analysis of performance data, a simple placement distribution model for both platforms was outlined. The PDM uses directed graphs to represent processor, memory and interconnect layout.

It was found that if multiple level 1 or level 2 BLAS operations are run in parallel on the Opteron system performance differences of up to a factor of two were observed depending on memory and thread placement. For level 3 BLAS, differences are much smaller as there is much better re-use of data from level 2 cache.

The use of the PDM and subsequent experiments show that memory placement is important in achieving good performance on NUMA platforms. The PDM categorizes performance results in terms of contention classes and for the lowest contention class, is able to do so with standard deviations for `Stream` Copy and Scale between 12% to 15% (Opteron/Solaris), 12% to 16% (Opteron/Linux), 10% (V1280/Solaris). The PDM results for `Stream` Triad, L2 BLAS and L3 BLAS indicate that a cache blocked computations are affected less by memory and thread placement than Triad and L2 BLAS, which require data to be streamed from memory into the processor. The PDM errors ranged from 0.6% to 18% for Solaris/Opteron, 0.6% to 37% for Linux/Opteron and 10% to 20% for the Solaris/V1280.

It would be beneficial for an application to be able to discover, at runtime, the processor and memory topology, and subsequently be able to use this information within the application to effect thread and memory placement, which is specific to its needs.

Results obtained in this Chapter should the importance of both thread and memory placement. Both the Solaris and Linux operating systems utilize NUMA specific information to affect thread scheduling and memory management decisions. It would be beneficial for a user space application to be able to discover, at runtime, both the processor and memory topology and subsequently use this information to effect thread and memory placement.

Chapter 4

Use of a Simple Linear Performance Model for Electron Repulsion Integral Evaluation

4.1 Introduction

All modern microprocessors utilize a cache memory hierarchy to ameliorate latencies associated with accessing main memory. In recognition of this much effort has been devoted to designing algorithms that carefully orchestrate computation in synchrony with data movement [91, 100, 153, 306, 307]. Almost always these algorithms involve a variety of trade-offs, such as the size of a cache blocking factor, or whether to recompute an intermediate quantity on the fly or pay the penalty of storing and retrieving the data from some distant memory location. In this respect developing models that can be used to describe performance at various levels in the cache hierarchy as algorithmic or system hardware parameters are changed is important [78, 290].

As discussed in Chapter 2, traditional cache performance models are either analytical or simulation based. Analytic models parametrize various aspects of the system to give an empirical performance estimate, while simulation based techniques predict performance based on the sequence of executable instructions. Simulation based techniques can be functional or cycle accurate, using inputs that are either execution driven (i.e. generated by interpreting instruc-

Material from this Chapter was published in: (a) Proceedings of HPSC Vietnam 2006, MODELLING THE PERFORMANCE OF GAUSSIAN CHEMISTRY CODE ON X86 ARCHITECTURES, in Modeling, Simulation and Optimization of Complex Processes, Editor: Hans Georg Bock. Springer 2008, (b) BUILDING FAST, RELIABLE, AND ADAPTIVE SOFTWARE FOR COMPUTATIONAL SCIENCE, Journal of Physics: Conference Series, 2008 `http://stacks.iop.org/1742-6596/125/012015`.

tions from the binary being simulated) or trace driven (i.e. the streams of loads and stores for the simulation are intercepted and saved to disk for offline use). Cache behaviour is then simulated by supplying the instruction sequences to the cache simulator which in turn models the cache hierarchy. Although trace and execution driven methods are 100 to 1000 times slower than execution on native hardware, they capture dynamic aspects of code execution which occur at run-time (i.e. side-effects arising from interactions between the application, operating system and hardware), that analytical cache models are unable to capture.

In this Chapter the utility of a simple Linear Performance Model (LPM) is investigated to determine if it can provide sufficiently accurate predictive information (that can be used to guide algorithmic decisions or model the effects of cache blocking changes) for quantum chemistry calculations. In this model [191] the overall performance is given as a simple linear combination of instructions issued and cache misses,

$$Cycles = \alpha * (I_{Count}) + \beta * (L1_{Misses}) + \gamma * (L2_{Misses}) \tag{4.1}$$

where I_{Count} is the instruction count, $L1_{Misses}$ the total number of Level 1 cache misses, $L2_{Misses}$ the total number of Level 2 cache misses, and coefficients α, β, γ are penalty factors. The value of α reflects the ability of the code to exploit the underlying super-scalar architecture, β is the average cost of an L1 cache miss, and γ is the average cost of an L2 cache miss. We will refer to the coefficients α, β and γ as the Processor and Platform specific Coefficients (PPCoeffs).

The LPM differs from other cache performance models in that it ignores the intricacies of program execution and assumes platform and processor specific factors that influence application performance can be averaged out and captured by the values of the PPCoeffs. For a candidate algorithm PPCoeffs could be obtained by running the code on similar processor family revisions which have different cache sizes or by varying a given fundamental cache blocking factor within the algorithm. Either method will yield different instruction, L1 and L2 cache miss counts allowing the PPCoeffs to be obtained using a least squares fit of the observed counts to Equation 4.1.

The aims of this chapter are three fold – (i) to study the effects of cache blocking on the PRISM electron repulsion integral (ERI) algorithm within the Gaussian code; (ii) to assess the ability of the LPM to describe the execution time of quantum chemistry calculations that use PRISM across a variety of hardware platforms; (iii) to combine parameters obtained for the LPM on existing hardware with functional cache simulation to predict the effect of architectural changes on the total runtime.

The structure and layout of this chapter is as follows. Section 4.2, gives an overview of ERI evaluation and in particular focuses on the PRISM algorithm and its use of cache blocking. Section 4.3 covers methodology, software and hardware platforms and benchmark systems used in this chapter. Sections 4.4, 4.5 and 4.6 consider aims (i), (ii) and (iii) respectively. A review of related work is given in Section 4.7, while Section 4.8 concludes the chapter.

4.2 ERI Evaluation: the PRISM algorithm

4.2.1 Two-Electron Repulsion Integrals

As detailed in Chapter 2 the evaluation of two-electron repulsion integrals (ERIs) lies at the core of nearly all calculations performed using the Gaussian code. A two electron integral $(\phi_\mu\ \phi_\nu|\phi_\lambda\ \phi_\sigma)$ is given by,

$$(\phi_\mu \phi_\nu \,|\, \phi_\lambda \phi_\sigma) = \iint \chi_\mu(r_1)\,\chi_\nu(r_1)\,\frac{1}{|r_1 - r_2|}\,\chi_\lambda(r_2)\,\chi_\sigma(r_2)\; dr_1\, dr_2 \tag{4.2}$$

where r_1 and r_2 are the coordinates for two electrons (e_1 and e_2); χ_μ, χ_ν and χ_λ, χ_σ are pairs of basis functions used to represent e_1 and e_2. ERIs describe the various interactions between and among the electrons and nuclei in the system being studied. Since many electronic structure methods are iterative, and the number of integrals too numerous to store in memory, they are usually re-computed several times during the course of a typical calculation [132]. For this reason algorithms that compute electronic structure integrals fast and on-demand are extremely important to the computational chemistry community.

Gaussian uses an efficient implementation of the PRISM algorithm [85, 95, 96, 136] to compute ERIs. For clarity, the following terms are re-referenced from Chapter 2,

- **GTO** Gaussian type orbital. Ref: Eqn 2.12.
- **PGTO** Primitive GTO: $G_{nl}(\alpha, \mathbf{A})$. Ref: Eqn 2.14.
- **CGTO** Contracted GTO: $\phi_\kappa^{CGTO}(\mathbf{A})$. Ref: Eqn 2.15.
- **Primitive/Contracted Shell** For a given PGTO or CGTO, a shell contains all functions of that PGTO or CGTO that have the same total angular momentum.
- **Shell-pair** A pair of shells.
- **Shell-quartet** All ERIs that are defined by four shells.

The PRISM ERI algorithm tailors the evaluation of ERIs according to the shell-quartet type being computed; essentially it dynamically chooses the most optimal way to compute and contract the constituent PGTOs [92]. PRISM computes ERIs in batches, where a batch is a set of shell-quartets having the same four angular momentum values as well as the same degree of contraction for each of the two shell-pairs in the quartet.

Using the example from Gill [92], the formation of batches from (ss|ss) quartets starts with all four s functions having a contraction length of one. When this has been computed, the batch of (ss|ss) quartets for which the first function on the left has a contraction length of 2 and all others are singly contracted is computed. This process continues until all possible contractions for (ss|ss) integrals complete, wherein the set of (ps|ss) integrals are considered for evaluation and so on.

ERI evaluation using GTOs requires the use of recursive relationships that build up the target angular momentum using ERI values of lower angular momentum [34, 243]. This process inherently leads to intermediates and other quantities that are shared amongst the ERIs [106, 161]. Similar shell quartets are often evaluated using the same ERI evaluation algorithm. The use of batching within PRISM aids the reuse of shared intermediate quantities but more importantly it is able to use the same evaluation logic across a batch of integrals in the form of vector operations that facilitate pipelining of memory requests and floating-point operations. The size of these batches can rapidly become very large as the same basis set is generally applied to all atoms of the same type within the system being studied, e.g. all oxygen atoms in a large water cluster system will have the same basis set. Computing integrals in batches leads to large inner-loop lengths in PRISM, as numerous integrals of the same type are evaluated. While this gives rise to good on-chip pipelining of operations and exposes opportunities to exploit Instruction Level Parallelism (ILP), the number and size of data quantities required to compute a given batch can exceed the total amount of available on-chip cache memory, resulting in expensive cache misses and poor performance. To address this the implementation of PRISM in Gaussian, imposes cache blocking of these data quantities. This ensures data references are serviced from the on-chip cache, but at the cost of recomputing some shared intermediate data quantities.

To illustrate the composition of functions χ used in ERI evaluation, Table 4.1 presents a detailed break-down of basis functions on each atom of a water molecule, for a 6-31G* basis set. The table has three columns which are 'Atomic Center', 'Atomic Orbital' and 'Gaussian Functions' respectively. The first column gives the atom and its Cartesian coordinates; the second describes the name of the atomic orbital and the number of GTOs associated with it; and the third column gives the composition of the GTOs that make up the atomic orbital.

For clarity Equations 2.14 and 2.15 from Chapter 2 are reproduced,

$$\begin{aligned} G_{nl}(\alpha,\mathbf{A}) \equiv G_{nl}^{PGTO}(\alpha,\mathbf{A}) &= (x-A_x)^{a_x}(y-A_y)^{a_y}(z-A_z)^{a_z}\; e^{-\alpha|\mathbf{r}-\mathbf{A}|^2} \\ \phi_{\kappa}^{CGTO}(\mathbf{A}) &= \sum_{p=1}^{L} d_{p\kappa}\, G_{nl}^{p}(\alpha_{p\kappa},\mathbf{A}) \end{aligned} \tag{4.3}$$

In the above expressions, a PGTO is described by its exponent α and its center $\mathbf{A}$, whereas

Table 4.1: Decomposition of 6-31G* basis functions on a Water molecule

Atomic Center				Atomic Orbital		Gaussian Functions			
Atom	X	Y	Z	Shell Type	Function Number	Exponent (α)	Coefficient ($d_{p\kappa}$) S	P	D
O	0.00	0.22	0.00	S	1				
						0.55D+04	0.18D-02	0.00	0.00
						0.83D+03	0.14D-01	0.00	0.00
						0.19D+03	0.69D-01	0.00	0.00
						0.53D+02	0.23D+00	0.00	0.00
						0.17D+02	0.47D+00	0.00	0.00
						0.58D+01	0.36D+00	0.00	0.00
				SP	2–5				
						0.16D+02	-0.11D+00	0.71D-01	0.00
						0.36D+01	-0.15D+00	0.34D+00	0.00
						0.10D+01	0.11D+01	0.73D+00	0.00
				SP	6–9				
						0.27D+00	0.10D+01	0.10D+01	0.00
				D	10–15				
						0.80D+00	0.00D+00	0.00	0.10D+01
H	1.43	-0.89	0.00	S	16				
						0.19D+02	0.33D-01	0.00	0.00
						0.28D+01	0.24D+00	0.00	0.00
						0.64D+00	0.81D+00	0.00	0.00
				S	17				
						0.16D+00	0.10D+01	0.00	0.00
H	-1.43	-0.89	0.00	S	18				
						0.19D+02	0.33D-01	0.00	0.00
						0.28D+01	0.24D+00	0.00	0.00
						0.64D+00	0.81D+00	0.00	0.00
				S	19				
						0.16D+00	0.10D+01	0.00	0.00

a CGTO is described by its center **A**, its contraction length 'L', the contraction coefficients '$d_{p\kappa}$' and the constituent PGTO exponents. The 'Exponent' column in Table 4.1 corresponds to individual values of 'α' for each PGTO (cf. Equation 4.3) and the 'Coefficient' column corresponds to $d_{p\kappa}$ for the CGTO.

The oxygen atom in Table 4.1 is located at Cartesian coordinates (0.00, 0.22, 0.00). It has a total of 15 functions defined at this center. There are 6 PGTOs that are contracted together to describe the inner 'core' orbital (Table 4.1 – Function Number: 1, Shell Type: S). The outer 'valence' orbitals are described by two groups of functions: a CGTO containing a linear combination of 3 PGTOs (Function Numbers: 2-5) and a single uncontracted PGTO (Functions: 6-9). In the previous two cases the functions are of type SP, which is defined as a combination of both an s function and three p functions where the exponents are the same but the contraction coefficients can be different. SP functions are used as its combined treatment is computationally advantageous for ERI computation. The last function is a D function (Function Numbers: 10-15, Shell Type: D) that is termed a polarization function. The use of polarized functions affords flexibility in describing electron density, during the SCF optimization, as it effectively allows molecular orbitals to be more asymmetric about a given nuclear center.

The ‘s’ functions are spherically symmetric around their centers [169], ‘p’ functions are axially symmetric about the x, y or z axis and are labelled ‘p_x’, ‘p_y’ and ‘p_z’ accordingly. Higher order angular momentum functions have other symmetry properties, such as ‘d’ functions which have 6 Cartesian symmetries – ‘d_{xy}’, ‘d_{yz}’, ‘d_{xz}’, ‘d_{x^2}’, ‘d_{y^2}’ and ‘d_{z^2}’. Due to these symmetry components, the SP function has a total of 4 entries in Table 4.1 while the D function has 6 entries.

The 6-31G* basis set [110, 205] is referred to as a split-valence basis set, reflecting the fact one CGTO is used to describe core electrons while pairs of CGTOs are used for valence electrons. Furthermore the core CGTO comprises of 6 PGTOs, while the one of the two valence CGTOs has 3 PGTOs. The second valence CGTO is a singly contracted PGTO. This composition is reflected in the 6-31G name. The asterisk in 6-31G* denotes the presence of polarization functions.

Having considered an example of basis set composition, the next section looks at the PRISM algorithm for evaluating ERIs and its use of cache blocking.

4.2.2 PRISM and Cache Blocking

Algorithm 2 is an outline of the PRISM ERI algorithm. The algorithm begins by generating a shell-pair list for the entire molecular system and classifying these as being significant or negligible [85]. If two shells are so far apart that their overlap is negligible, it is discarded[1]. The shell-pair list is then sorted by the angular momentum of each CGTO. This sorted list is then subjected to a screening test [108,112,206] which culls more shell-pairs by only accepting those which make a significant contribution to the Fock matrix. A cache blocking factor, *MDC*, which will be used later in the ERI evaluation is then set to some value. This defaults to a value that corresponds to half the size of the highest level of cache, but can be varied as a runtime parameter. This sets the stage for the actual ERI evaluation to begin.

PRISM is implemented as a six-fold loop, these are labeled ① to ⑥ in Algorithm 2. In the following description, a shell-quartet $(\chi_\mu\, \chi_\nu|\chi_\lambda\, \chi_\sigma)$ is denoted by its subscripts $(\mu\nu|\lambda\sigma)$. The first loop is over the range ‘0 to LTot’, where the value of this loop index reflects the sum of the angular momentums of the four functions that will be involved in a given integral quartet. Thus LTot is determined by the highest angular momentum function used in the basis set, which for

[1]GTOs basis decay exponentially. Gill [92] notes that most shell-pairs in a system are negligible and as a result, when system size increases the number of significant shell-pairs grows linearly. This leads to a quadratic growth in the number of significant shell-quartets.

Algorithm 2 An outline of the PRISM algorithm

```
Generate significant shell-pair list for molecular system      i.e. charge interactions between
                                                               basis functions $\chi_\mu(r_1)\,\chi_\nu(r_1)$

Sort shell-pair list and cull those                            sort shell-pairs by angular momentum type
which are not significant                                      and then by degree of contraction L
                                                               for each component function

$MDC \leftarrow$ Cache blocking size
LoopOver ① all types of $(\mu\,\nu|\,\lambda\,\sigma)$ i.e. LTot do
    Compute base quantities for recursion
    Create driver for loop ①
    LoopOver ② Total Angular Momentum type for $(\mu\,\nu|$ do
        Create driver for loop ②
        LoopOver ③ Total Angular Momentum type for $|\,\lambda\,\sigma)$ do
            Create driver for loop ③
            LoopOver ④ Contraction types for $(\mu\,\nu|$ do
                LoopOver ⑤ Contraction types for $|\,\lambda\,\sigma)$ do        Begin new quartet
                    Determine memory optimal path for
                    evaluating this particular $(\mu\nu|\lambda\sigma)$

                    $NPerS4 \leftarrow$ Words per Quartet required
                              to store this particular $(\mu\nu|\lambda\sigma)$

                    $NLeft \leftarrow$ Available working memory in words
                    $MaxCom \leftarrow \frac{NLeft}{NPerS4}$
                    if $(MaxCom > MDC)$ then
                        Reduce $MaxCom$ to hold as many $NPerS4$ that
                        can be held in a cache of size $MDC$
                    end if
                    iCount $\leftarrow 0$
                    LoopOver ⑥ all batches of type $(\mu\nu|\lambda\sigma)$ do
                        Screen shell-pair for possible inclusion

                        if (iCount $\leq$ MaxCom) then
                            Add shell-quartet to current cache block
                            iCount $\leftarrow$ iCount + 1
                        else
                            Compute $(\mu\nu|\lambda\sigma)$ ERI for cache blocked batch
                            Incorporate contributions into Fock matrix
                        end if

                    end LoopOver ⑥
                end LoopOver ⑤
            end LoopOver ④
        end LoopOver ③
    end LoopOver ②
end LoopOver ①
```

the water molecule given in Table 4.1 is the d function on the oxygen atom, giving an LTot value of 8 that will be associated with the (dd|dd) quartet. Within the first loop, shell-pair data is used to compute base quantities which will be used to build recursively the required integral. The PRISM implementation in Gaussian achieves this recursive build through the use of drivers [92]. A driver is a simple table driven description of how the requisite ERI is to be constructed and corresponds to one of the paths in the PRISM shown in Figure 2.7. In the case of Loop ① the driver corresponds to Equation 2.21, and is constructed while paying attention to minimising both FLOP and MOP costs [85].

The next two loops i.e. **LoopOver** ② and **LoopOver** ③ represent the total angular moment for the $(\mu\nu|$ and $|\lambda\sigma)$ sides of the ERI. Another two sets of drivers are created that correspond to Equations 2.22 and 2.24.

Loops ④ and ⑤ iterate over the degrees of contraction for the $(\mu\nu|$ and $|\lambda\sigma)$ parts of the ERI. This completes the definition of an ERI quartet. At this stage PRISM determines a memory optimal [85] path for constructing this quartet type and drivers are created that correspond to Equations 2.24 and 2.25. The number of memory words required to store the quartet (NPerS4) is determined. The amount of working memory available to PRISM is set to NLeft.

A quantity called MaxCom is then defined as the number of shell-quartets that can fit into the free working memory available to Gaussian, this is MaxCom $\leftarrow \frac{\text{NLeft}}{\text{NPerS4}}$. To ensure cache residency of data quantities, the value of MaxCom may then be reduced so that only as many NPerS4 quartets as can be held in a cache of size MDC will be computed at one time. We also note that currently in Gaussian the value of MDC is determined at compile time. This value is choosen so as to both minimize pipeline stalls and ensure data quantites are cache blocked.

Loop ⑥, which is the inner loop, now iterates over quartets of the current shell type. A batch of quartets of size (MaxCom * NPerS4), which is small enough to be cache resident, is defined for evaluation. Additional screening of the shell-pairs is performed, and the batch of ERIs is computed. After evaluation the ERIs are multiplied with the density matrix to give contributions to the Fock matrix (cf. Equation 2.8). The process continues, with Loop ⑥ iterating until all quartets of the current shell type have been evaluated.

4.2.3 Shell-Quartets for the Water molecule

Having discussed the PRISM algorithm, Table 4.2 presents the number of quartets, batches and integrals generated per LTot for the water molecule and 6-31G* basis set given in Table 4.1. It should be noted that the *Total # of ERI* column corresponds to the number of integrals that make it all the way to the Fock matrix formation stage i.e the stage where computed ERIs

Table 4.2: The number of shell-quartets, batches and total ERIs generated per LTot for a water molecule using the 6-31G* basis set. *MDC* is a compile time value of 32Kw. The *Total # of ERI* column corresponds to the number of integrals that are available at the Fock matrix formation stage.

H_2O, 6-31G*				
LTot	# of Quartets	# of Batches	Total # of ERI	Avg. ERIs per Quartet
0	120	28	120	1
1	150	28	442	3
2	175	52	1237	7
3	110	38	1976	18
4	71	36	2390	34
5	26	16	1552	60
6	11	9	1084	99
7	2	2	120	60
8	1	1	336	336

are incorporated into the Fock matrix[2]. As noted previously, the maximum angular quantum number for the three atoms in the water molecule is 2 (for a 'd' function), hence the largest value of LTot is 8. Finally, the average 'ERI per Quartet' column is derived from the 'Total # of ERI' and '# of Quartets' columns.

For LTot = 0, there are 120 significant shell-quartets from which 28 batches were formed and this results in 120 unique ERIs being computed. As the value of LTot increases, there is a steady increase in the number of quartets (upto 175 for LTot = 2), and a corresponding increase in the number of batches generated. The 'Total # of ERI per LTot' increases from a minimum of 120 upto a maximum of 2390 then reduces to 336. The largest number of integrals occurs for LTot = 4, which represents all integrals of the type – (d d | s s), (p p | p p), (d p | p s), (d s | d s), (d s | p p).

The number of average 'ERIs per Quartet' increases with LTot. This, plus the fact that higher angular momentum integrals are composed of lower angular momentum ones, means that more FLOPs are usually needed to construct higher angular momentum quartets[3] than smaller ones.

The number of quartets and total number of ERIs per LTot are determined by the system under study and the nature of the basis set being used, while the number of batches is a function of the size of *MDC*. The variation of batches with respect to *MDC* will be covered in a later section.

[2]These results were obtained by running Gaussian with the `nosymm` and `nofmm` options.

[3]The FLOP count will also depend on the degree of contraction

Table 4.3: Characteristics of microprocessors used in this study

Micro-Architecture	Processor	Ghz	L1 Size	L2 Size	Peak FPC^	IPC*
AMD64	Opteron AMD848	2.2	64Kb	1MB	2	3
AMD64	Athlon64 X2 4200+	2.2	64Kb	512Kb	2	3
NetBurst	Pentium 4	3.0	16Kb	1MB	2	3
NetBurst	Pentium 4 EM64T	3.0	16Kb	2MB	2	3
P6	Intel Pentium M	1.4	32Kb	1MB	2	3
PowerPC	G5 (PPC970Fx)	1.8	32Kb	512Kb	4	4
PowerPC	G5-XServe (PPC970Fx)	2.0	32Kb	512Kb	4	4

^ Floating-point Instructions per Cycle (FPC) * Theoretical Instructions Per Cycle (IPC)

Summary: PRISM and ERI Evaluation

The PRISM ERI algorithm generates shell-pair information for the molecular system under study. From these a set of basic quantities are computed, which are then used to recursively build up angular momentum upto the required target angular momentum of the shell-quartet being computed. The process of recursively building angular momentum is encapsulated as drivers, which are table driven. Once drivers have been constructed, integrals are computed in batches. PRISM determines a memory optimal path for computing the ERIs for a given shell-quartet.

4.3 Methodology

This section discusses the various hardware and software resources, the functional cache simulator, molecular benchmark systems and methodology used in this chapter.

4.3.1 Microprocessors and Compilers

Four distinct microprocessor families: AMD64 [146], NetBurst [87], P6 [245] and PowerPC [16, 247] were used as part of this study. Table 4.3 lists the clock speed, Level one (L1) and Level two (L2) cache sizes, peak floating-point performance and the maximum instructions per cycle. Clock speeds for the processors vary from 1.4Ghz to 3.0Ghz. The NetBurst processors have a 16Kb L1 instruction[4] cache and a 16Kb L1 write-through data cache. While the AMD processors, the P6 and PowerPC processors use a 64Kb L1 instruction cache and a 64Kb write-back data cache. Both the AMD and Intel processors have the same peak Floating-point Instructions per Cycle (FPC) and theoretical Instructions Per Cycle (IPC) of 2 and 3

[4]Strictly speaking these are trace caches [87].

respectively. For studies involving PowerPC, two different hardware platforms were used – an Apple G5 server [15] and an Apple G5 rack-mounted XServe [16]. The PowerPC processors can issue 4 instructions per cycle and have a peak FPC of 4.

On the AMD64 and Intel platforms the PGI Fortran and C compilers (version 6.1-1) were used, while on the PowerPC IBM's xlf and xlc compilers (version 10.01) were used.

4.3.2 Hardware Performance Counters

Hardware performance counter values were read by inserting instrumentation code into G03 source and recompiling (version G03D01 [86] was used). The instrumentation code in Gaussian also allowed internal program state to be captured, while an external shared library was written to abstract away tasks involved with reading of performance counters. This external library uses the PAPI cross-platform hardware performance counter infrastructure [37] to specify native processor events. On platforms which did not have the capability of recording a specific set of multiple events (ref. Appendix A.1), PAPI's hardware counter overflow and counter multiplexing features were used. PAPI version 3.2.1 was used in conjunction with Mikael Peterson's perfctrs [186] Linux kernel patch (version 2.6.19) on a Linux 2.6.11.4 kernel with the SuSE 9.2 distribution across all hardware platforms. The native processor events which were used for obtaining instruction counts, total L1 and L2 cache misses are described in Table A.1 of the Appendix.

4.3.3 Functional Cache Simulation

Functional cache simulation was performed using the Callgrind simulator [140, 191, 274, 296, 297] along with the Valgrind [190, 192] dynamic binary translation tool. Callgrind's cache simulation implements a write-allocate cache and has options for either a write-through or write-back policy for all data caches. The cache simulation is synchronous, implements two levels of cache and is inclusive. Read and write references are not distinguished for the write-through simulation, whereas the write-back simulation implements an L1 write-through and L2 write-back and distinguishes between read and write references to cachelines. Thus, simulations for the Opteron and Athlon64 processors were configured using write-back caches; the Intel processors used write-through. The PowerPC cache configurations were configured using a write-back simulation.

To facilitate the Callgrind simulation of Gaussian, a shell-script wrapper for the executable was created which intercepted command-line arguments and then forwarded them to Valgrind/-Callgrind augmented with the appropriate cache simulation parameters i.e. cache size, associativity and linesize for L1 and L2 caches. The experiments used Valgrind version 3.2.2 with Callgrind.

Table 4.4: Test molecular systems

No.	System name	No. of Atoms	Basis sets used	# Basis	# PGTOs
1	k300a-04	34	6-31G*	232	472
			6-31++G(3df,3pd)	860	1136
2	k300a-08	241	6-31G*	1543	2956
			6-31++G(3df,3pd)	5966	7622
3	test397	168	3-21G	882	1440
4	α - Al_2O_3†	10	3-21G*	130	222

† Periodic Boundary Conditions (PBC) calculation

4.3.4 Data Collection and Analysis

Each of the benchmark systems were run five times. The data from these were processed using a series of Python scripts which subsequently aggregated performance counter data. The least-squares fit for the LPM was performed using NumPy [63]. Blocking factors ranging from 2 Kw (KiloWords) up to 1024 Kw were used with the PRISM algorithm; where a KiloWord is equal to 8 Kilobytes.

4.3.5 Molecular Systems and Benchmarks

Four molecular systems were chosen for performance experiments, these are enumerated in Table 4.4. For each of these systems, both the HF and B3LYP methods were used. Systems 1 and 2, are water cluster complexes that relate to the application study in Chapter 6. The geometries for these systems were obtained from molecular dynamics (MD) simulations of a potassium ion surrounded by a cluster of water molecules at 300 Kelvin [28]. System 1 contains all water molecules within a radius of 4 Å from a central potassium ion, of which there are 11 (ref. Appendix Table A.9). System 2 has a radius of 8 Å and contains 40 water molecules (ref. Appendix Table A.9) From here on, the two systems are referred to as *k300a-04* for the 4 Å case and *k300a-08* for the 8 Å case. The two systems were chosen to test the effect of increasing the number of molecules. For both systems two basis sets are used, a moderate 6-31G* set and an extended 6-31++G(3df,3pd) set. Larger basis sets give better results but lead to longer execution times and greater memory use. System 1 generates 232 basis functions with the 6-31G* basis set and 860 basis functions with the 6-31++G(3df,3pd) basis set. For System 2 the equivalent number of functions are 1543 and 5966.

Benchmark system 3 is based on the Valinomycin molecule. It is derived from the Gaussian G03 quality assurance (QA) suite. It has 168 atoms consisting of nitrogen, carbon, oxygen and hydrogen. It is referred to as *test397*, reflecting its number in the QA suite. The calculation uses a 3-21G basis set and gives rise to 882 basis functions. It was chosen as it is a widely used

Table 4.5: Execution times of two test systems for one SCF cycle on the AMD848 Opteron, using the HF method

System	Basis set	# of Functions	Time taken	
k300a-04	6-31G*	232	12.57	sec
	6-31++G(3df,3pd)	860	21.37	min
k300a-08	6-31G*	1543	19.65	min
	6-31++G(3df,3pd)	5966	3.40	days

benchmark.

Benchmark system 4 is based on a Periodic Boundary Conditions (PBC) [148] calculation and consists of three Aluminum trioxide molecules which form an infinite sheet in the (0001) orientation [232]. The system has 130 basis functions and was chosen as it exercises a different path through the Gaussian code.

In summary, a total of four distinct molecular systems were chosen. For the k300a-04 and k300a-08 systems two different basis sets were used, 6-31G* and 6-31++G(3df,3pd), while for test397 and α - Al_2O_3† the 3-21G and 3-21G* basis sets were used respectively. This gives rise to the six benchmark systems that will be used in this chapter.

4.4 Investigating the PRISM Integral Evaluation Algorithm and Cache Blocking

In this section the effects of cache blocking on the performance of the PRISM algorithm is studied. The objectives are to (a) explore the effect of cache blocking in PRISM, on the overall Cycle and Instruction counts, and total L1 and L2 misses; (b) investigate the effects of cache blocking on the batch sizes of different quartet types; and determine if cache blocking should be tailored according to (c) quartet type; or (d) the nature of the molecular system and basis set.

In what follows we use the k300a-04 system, the HF method, and the Opteron to study objectives (a) to (c). For objective (c), the larger k300a-08 system is also included, while for (d) all the benchmark systems are used.

These four objectives are addressed in sub-sections 4.4.1 – 4.4.6. First, however, we consider the overall runtime for one SCF cycle on the AMD platform and how it varies as a function of system size.

4.4.1 SCF Execution Time as a Function of System Size and Basis Set for an SCF Cycle

To explore the effect of increasing the molecular system size and basis set size, Table 4.5 presents execution times for the k300a-04 and k300a-08 systems using two different basis sets obtained on the AMD848 Opteron platform. Traditional integral evaluation scales as $O(N^4)$. The application of basis function thresholding reduces this to $O(N^2)$ for conventional SCF implementations [112, 132]. This further reduces to O(N) if the Fast Multipole Method (FMM) [130,258,300] is used. Another effect is the fact that the per unit cost for the evaluation of basis functions of higher angular momentum is lower as many ERIs are now being computed as part of each shell quartet.

As a consequence of the above we find that in the case of k300a-04, the use of a larger basis set results in a 102x increase in runtime rather than the 185x that would be predicted purely by the $O(N^4)$ scaling associated with integral evaluation; this is predominantly due to the more efficient evaluation of higher angular momentum integrals. Comparing the larger k300a-08 system and the 6-31G* basis set with the k300a-04 system and the same basis, there is a 7 times increase in basis functions, but only a 93x increase in execution time; this is largely due to screening and use of FMM integral evaluation technology. Contrasting the execution time for k300a-08 using the 6-31G* basis set with that for k300a-04 using the 6-31++G(3df,3pd) basis set it is found to be much faster even though it contains many more basis functions; this indicates that the benefits of better screening and use of FMM in the large system are significantly greater than the benefits of reduced unit cost per integral associated with the use of higher angular momentum functions in the larger basis.

A result in Table 4.5 that may appear to be somewhat unexpected is the timing difference between the use of the 6-31G* and 6-31++G(3df,3pd) basis sets with the k300a-08 system; where there is an observed 249x difference in timings, but $O(N^4)$ scaling would suggest that this should be a smaller difference of 223x. This indicates that there are some other factors apart from the number of basis functions that affect the scaling. The problem here is with the presence of diffuse functions[5] in the 6-31++G(3df,3pd) basis set. Specifically, integral screening of shell pairs and shell quartets is less effective in the presence of diffuse functions as there are interactions with a large number of other basis functions (i.e these interactions give rise to shell pairs and quartets that can no longer be screened out for a given overall level of accuracy). In Gaussian, the default for systems with greater than 80 atoms is to have both FMM and linear scaling exchange technologies turned on as these significantly speed up the

[5]The two plus signs in 6-31++G(3df,3pd) indicate the presence of diffuse functions. 'These are very shallow Gaussian basis functions, which more accurately represent the "tail" portion of the atomic orbitals, which are distant from the atomic nuclei' [303].

Table 4.6: Execution characteristics for PRISM with the k300a-04 water-cluster system using HF/6-31G* on a 2.2Ghz AMD848 Opteron.

Blocking (Kw)	Number of Batches	Cycles ($x10^{10}$)	Instr Count ($x10^{10}$)	FP Count ($x10^{9}$)	L1 Misses ($x10^{8}$)	L2 Misses ($x10^{7}$)
4	226922	3.85	5.38	8.42	3.86	0.16
16	75729	3.00	4.11	8.37	3.91	0.19
32	**39347**	**2.69**	**2.84**	**6.21**	**4.47**	**0.27**
64	20049	2.75	3.18	6.18	5.43	1.39
256	5624	3.58	2.91	6.14	7.50	4.89
1024	2174	4.64	2.84	6.08	8.82	6.51

calculation for systems with and without diffuse functions.

From the table it can be seen that increasing either the basis set or molecular system size results in increasing execution time. Moreover execution time can quickly become very large taking days or more to complete a computation on a single processor. For this reason ensuring that the underlying code performs efficiently on any given processor architecture is extremely important.

4.4.2 Effect of Cache Blocking on Cycle, Instruction Counts and Cache Misses

To explore the effect of cache blocking on PRISM's performance the k300a-04 system using the HF method and a 6-31G* basis set was run using various cache blocking sizes from 4 – 1024Kw on the AMD848 Opteron. The results are given in Table 4.6, where we also report the number of batches.

Cache blocking factors are given on the left hand side of Table 4.6. Corresponding to each blocking factor the associated number of batches generated and hardware event counts for cycles, instruction executed, FLOP count, L1 and L2 misses are given the table's columns.

Consider first the effect of cache blocking on the number of batches. Increasing the batch size from 4 Kw to 1024 Kw results in a large decrease in the number of batches. This behaviour is to be expected since, as discussed in section 4.2.2, the number of integrals that are processed per batch (in loop ⑥ of Algorithm 2) increases.

Execution time, which is implied by the cycle count, is 3.85 x 10^{10} cycles for 4Kw, decreasing to a minima of 2.69 x 10^{10} cycles for 32Kw before increasing to 4.6 x 10^{10} cycles for 1024Kw. Depending on the blocking factor used, a 40x variation in execution time is observed.

Table 4.7: Variation in the number of batches with varying blocking factors for the k300a-04 system using a 6-31G* basis set with the HF method. Data obtained from one SCF cycle.

	Total	Number of Batches						
LTot[1]	NoQrt[2]	4 Kw	16 Kw	32 Kw	64 Kw	256 Kw	1024 Kw	AsyLim[3]
0	352176	13190	3727	1894	963	268	100	66
1	708932	38584	10672	5411	2761	770	295	187
2	806845	56118	16245	8313	4248	1224	503	362
3	589046	57697	17251	8849	4528	1300	531	378
4	310929	39138	14386	7378	3770	1068	419	286
5	116550	16696	9152	4874	2451	652	223	112
6	32097	4601	3398	2057	1025	266	82	34
7	5651	809	809	508	271	68	19	5
8	623	89	89	63	32	8	2	1

1) LTot	Total Angular Momentum
2) NoQrt	Number of Quartets
3) AsyLim	Asymptotic Batch Limit

As the blocking factor increases the number of instructions decreases. This arises due to the following: first, increasing the blocking factor leads to a reduction in the number of batches. Second, the work required to re-compute shared intermediate quantities amongst shell-quartets in a batch reduces as the batch size increases. As the work done reduces with increasing batch size, so does the number of instructions being executed. A similar reduction is also seen for the FLOP count.

While reducing re-computation by increasing the blocking factor is good, it also gives rise to an increase in cache misses, since batches now start to overflow cache. This behavior is evident from the L1 and L2 miss counts, which increase as the blocking factor gets larger. Also, the L1 misses are an order of magnitude larger than L2 misses, suggesting that PRISM's memory access patterns are cache blocked for the L2 cache, and not for the L1 cache.

4.4.3 Effect of Cache Blocking on ERI Batching

The previous sub-section considered the effects of cache blocking on measured cycle counts for the k300a-04 system with a 6-31G* basis set and the HF method. The total number of batches generated as a function of cache blocking, was also reported. It was found that increasing the blocking factor reduced the total number of batches being processed. Thus it is of interest to examine the effect of cache blocking at a finer level. In Table 4.7 we present a detailed breakdown of batch size according to the quartet LTot value for blocking factors ranging from 4Kw to 1024Kw. Also included is the asymptotic batch size limit ('AsyLim'). The 'AsyLim' results correspond to the number of batches each LTot would generate if an infinite sized cache

Table 4.8: Cycle count per LTot for k300a-04 using HF/6-31G* on a 2.2Ghz AMD848 Opteron

	Total	Asy.	Qrt.	Cycle count (x10^9)					
LTot	NoQrt	Lim	PAB.	4 Kw	16 Kw	32 Kw	64 Kw	256 Kw	1024 Kw
0	352176	66	5336	1.01	0.89	**0.84**	0.98	0.97	1.30
1	708932	187	3791	3.07	2.54	**2.36**	2.69	3.36	4.02
2	806845	362	2229	5.96	4.66	**4.37**	4.70	6.37	7.72
3	589046	378	1558	8.51	6.17	**5.72**	6.21	8.22	10.4
4	310929	286	1087	9.12	6.53	**5.70**	5.88	7.86	10.4
5	116550	112	1041	6.48	5.24	4.53	**4.20**	5.52	7.72
6	32097	34	944	3.24	2.92	2.35	**2.02**	2.67	3.67
7	5651	5	1130	0.95	0.94	0.84	**0.67**	**0.67**	1.01
8	623	1	623	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**
Total Cycles (x10^{10})				3.85	3.00	**2.69**	2.75	3.58	4.64
% Increase from 32Kw				+30.17	+10.56	0.00	+2.32	+24.94	+42.10

Qrt. PAB. – Average number of Quartets per batch, without the use of cache blocking

was used.

Table 4.7 is divided into two sections. The first section gives the total number of quartets generated for each LTot (in the 'Total NoQrt' column). The second section gives the 'Number of Batches' generated as a function of cache blocking and the asymptotic batch size limit for each LTot. Values for 'Total NoQrt' were obtained by setting a counter to 0, at the start of loop ① in Algorithm 2. This counter was then incremented by the number of shell quartets that were eventually chosen by PRISM for computation, within loop ⑥. The values for 'AsyLim' were obtained by counting the number of times loop ⑤ in Algorithm 2 was entered.

From the table it is seen that as the value of LTot increases, the total number of shell-quartets increases from 352,176 to 806,845 and then drops sharply to 623 for LTot = 8. As mentioned before, the distribution of shell-quartets results from the underlying construction of the 6-31G* basis set and the nature of the system under study.

Starting from the 4Kw blocking factor, as the value of LTot increases, the number of batches rises to a maximum of 57,697 and reduces sharply there after. Increasing the blocking factor by four to 16Kw results in approximately a 3.3 fold reduction in the number of batches. This reduction drops to 2.7 fold at LTot = 4, while for LTot $\geq$ 7 there is no reduction at all. The latter is a consequence of batching being ignored if the value of 'NPerS4', in Algorithm 2, is greater than the available cache size.

From 32Kw to 1024Kw, increasing the blocking factor leads to a proportional reduction in the number of batches. Surprisingly the use of a 1024Kw blocking factor, which corresponds to a cache of size 16MB, would still result in batches being cache blocked.

4.4.4 Cache Blocking as a Function of Quartet Type

In the previous section, it was seen that the effect of cache blocking on the number of batches generated varied according to LTot value. This raises the question whether the cache blocking factor should vary according to the quartet type. To address this, we present in Table 4.8 the cycle count for each LTot as a function of each cache blocking factor.

There are four major columns in Table 4.8 – ‘Total NoQrt’, ‘AsyLim’, ‘Qrt. PAB.’ and ‘Cycle count’. The ‘Total NoQrt’ and ‘AsyLim’ columns are reproduced from Table 4.7 to aid discussion. The ‘Qrt. PAB.’ column is the average number of quartets per batch, assuming there is no cache blocking. Values in this column are derived from the previous two columns. The ‘Cycle count’ row presents measured cycle counts for each LTot as a function of the cache blocking factor. Minimum cycle counts entries for each LTot have been highlighted in bold font. At the bottom of the table, the ‘Total Cycles’ column corresponds to the sum of cycle counts for each blocking factor. The ‘% Increase from 32Kw’ gives the percentage difference between cycle counts for the 32Kw blocking factor and other blocking factors.

For each value of LTot, there is a corresponding value for the total number of quartets and the asymptotic batch limit. Trends for these two columns were discussed previously. Values for the ‘Qrt PAB.’ column decrease from LTot = 0 to 6. After this there is an increase, followed by a slight decrease for LTot = 8.

The expectation for measured Cycle counts will depend on the number of quartets for a given LTot, the number of asymptotic batches and the FLOP cost associated with computing a given quartet. For 4Kw, we see that as LTot increases, the cycle count gradually increases, peaks for LTot = 4 and reduces. Moving to 16Kw, the cycle count peaks again at LTot = 4. For 32Kw, the cycle count peaks at LTot = 3 and this trend holds upto 256Kw. For 1024Kw, shell-quartets with LTot = 3 and 4 take the same amount of time to compute. It was mentioned earlier, in section 4.2.3, that higher angular momentum functions cost more in FLOPs to assemble than those of lower angular momentum. If we consider the variation of the ‘Qrt. PAB’ column, it is seen that lower angular momentum integrals, though numerous are computed rapidly. As the value of LTot increases, the cost of assembling higher angular momentum integrals increases and if these are numerous, it would dominate the overall cost. However the number of these integrals will decrease after some LTot value. This explains the presence of a peak followed by a decrease of the observed cycle counts. Aggregate cycle counts for each blocking factor in the ‘Total Cycles’ row, corresponds to the Cycle counts presented in Table 4.6.

Observing the progression of bold values from the top of the table to the bottom, shows that there are two blocking factors which give the lowest cycle count per LTot. From LTot = 0 to 4 it is the 32Kw blocking factor. Following this there is a switch to the 64Kw factor. This transition indicates that it could be beneficial to switch blocking factors at runtime.

To assess further, the usefulness of switching blocking factors, we expand the scope of systems examined to include larger system sizes and bigger basis sets. Thus, we include the k300a-04 system with the larger 6-31++G(3df,3pd) basis set, the k300a-08 system with both 6-31G* and 6-31++G(3df,3pd) basis sets.

To aid presentation of data and facilitate comparison, we now switch to using plots for cycle count. We also include total L1 and L2 misses per LTot as a function of the cache blocking factor. These plots are given in Figures 4.1 and 4.2.

Data for the 6-31G* and 6-31++G(3df,3pd) basis sets are given in the first and second columns respectively. For each column there are three sub-plots which correspond to cycles, L1 and L2 misses. For each sub-plot the x-axis denotes LTot and the y-axis corresponds to the units for cycles, L1 and L2 misses.

For the k300a-04 system, cycle counts are reproduced from Table 4.8. As before, as LTot increases (from 0 to 8) the cycle count initially peaks and then gradually decreases; with a 32Kw blocking factor giving the lowest cycle count. Moving onto L1 misses for k300a-04 using the 6-31G* basis set, the curve has peaks which correspond to the peaks seen for Cycle counts. In terms of the ordering of misses, the L1 misses for the 32Kw blocking factor is in between the 16Kw and 64Kw curves. L2 misses are an order of magnitude less than the L1 misses. The L2 misses also peak at the same values of LTot as L1 misses, and ordering of the miss curves are the same as L1 misses. Unlike L1 misses, there is a large separation between the 256Kw and 1024Kw cases. This miss behaviour is to be expected. The Opteron has a 1MB on-chip L2 cache, thus blocking for 256Kw (i.e. a blocking factor which corresponds to a 4MB L2 data cache) and 1024 Kw (i.e. a blocking which corresponds to a 16MB L2 data cache) will lead to greatly increased cache misses. A 64 Kw (512Kb) blocking factor is the upper bound on the Opteron, after which the L2 miss penalty becomes much larger and detrimental to performance. Further, we also note, that there are cases (e.g. 6-31G*) where the blocking size which results in the lowest cycle count (32Kw) does not always have the lowest L2 miss rate (2Kw). This indicates that there are potential processor pipelining issues which need to be further investigated.

Consider the plots for k300a-04 that use the larger basis set shown in Figure 4.1. For these plots, the value of LTot now varies from 0 to 12 due to higher angular momentum functions in the basis set. As LTot increases, the cycle count curves gradually increase and peak at LTot = 5, 6 and reduces there after. Though cycle counts vary by two orders of magnitude between 6-31G* and 6-31++G(3df,3pd), the curves for cycle count have the same features albeit with peaks shifted. This shift arises from the use of the larger basis set. Unlike the k300a-04 with 6-31G* basis set, the lowest cycle counts are obtained for a 64Kw blocking factor rather than 32Kw. L1 miss counts for the larger basis set have the same ordering of curves as the 6-31G*

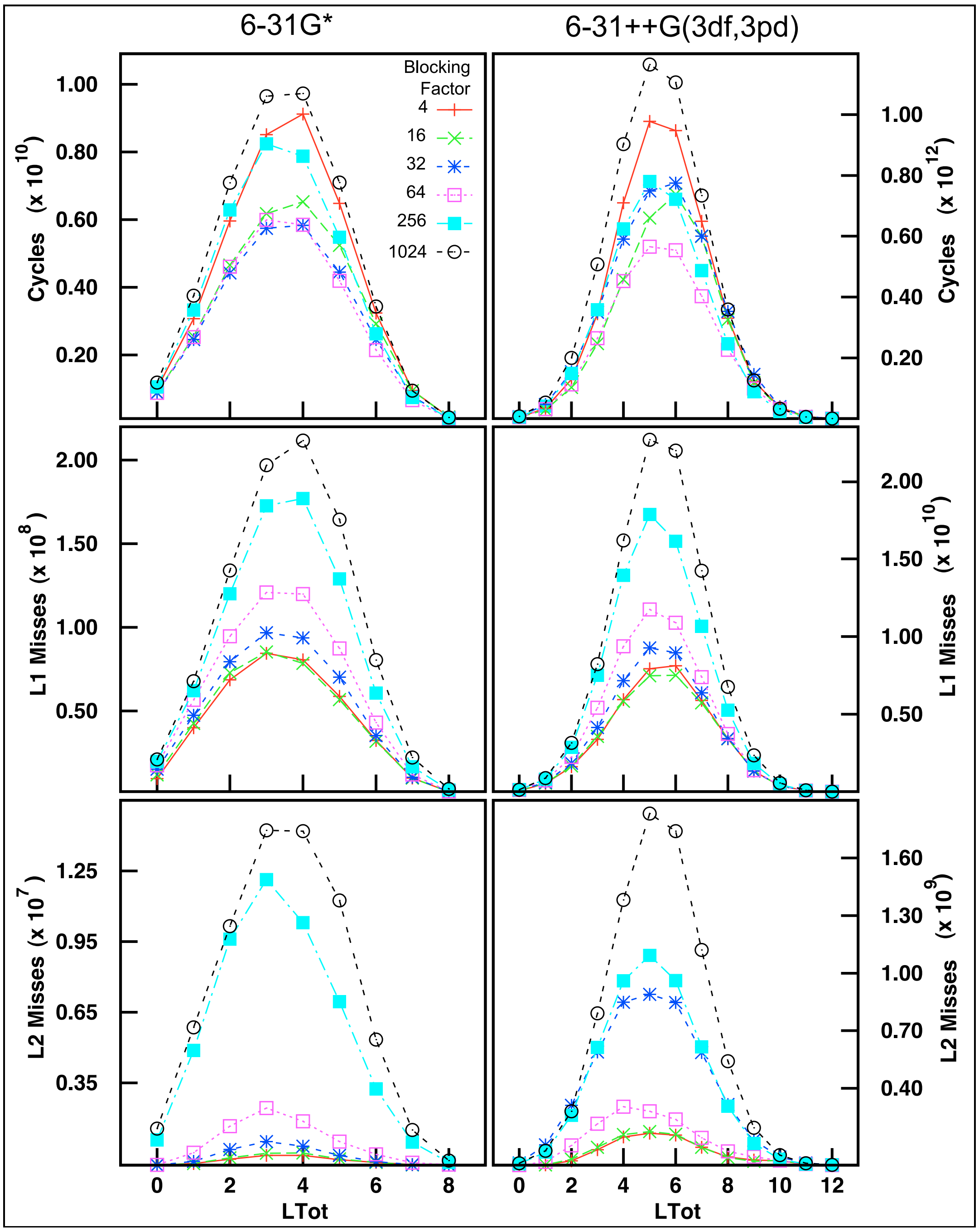


Figure 4.1: Per LTot breakdown of cycle counts and total cache misses (L1 and L2) for the k300a-04 water cluster system using a 6-31G* basis set and the HF method on an AMD848 Opteron system.

case. L2 misses for the larger basis set have the same ordering as L1 misses except for 32Kw. Interestingly use of 32Kw with the larger basis set results in L2 misses which are almost comparable to using a blocking factor that overflows cache. Presumably this pathological behaviour is due to subtle interplay between the system, basis set and input geometry.

Results for the k300a-08 system using the 6-31G* and 6-31++G(3df,3pd) basis sets, which are given in Figure 4.2. For k300a-08 using the 6-31G* basis set, trends for cycle counts are similar to those seen for k300a-04 with 6-31G*. Cycle counts peak at LTot = 3, 2. The lowest cycle count was obtained for a 32Kw blocking factor. The ordering of miss curves and trends for L1, L2 misses using 6-31G* are similar to those seen in k300a-04 with 6-31G*.

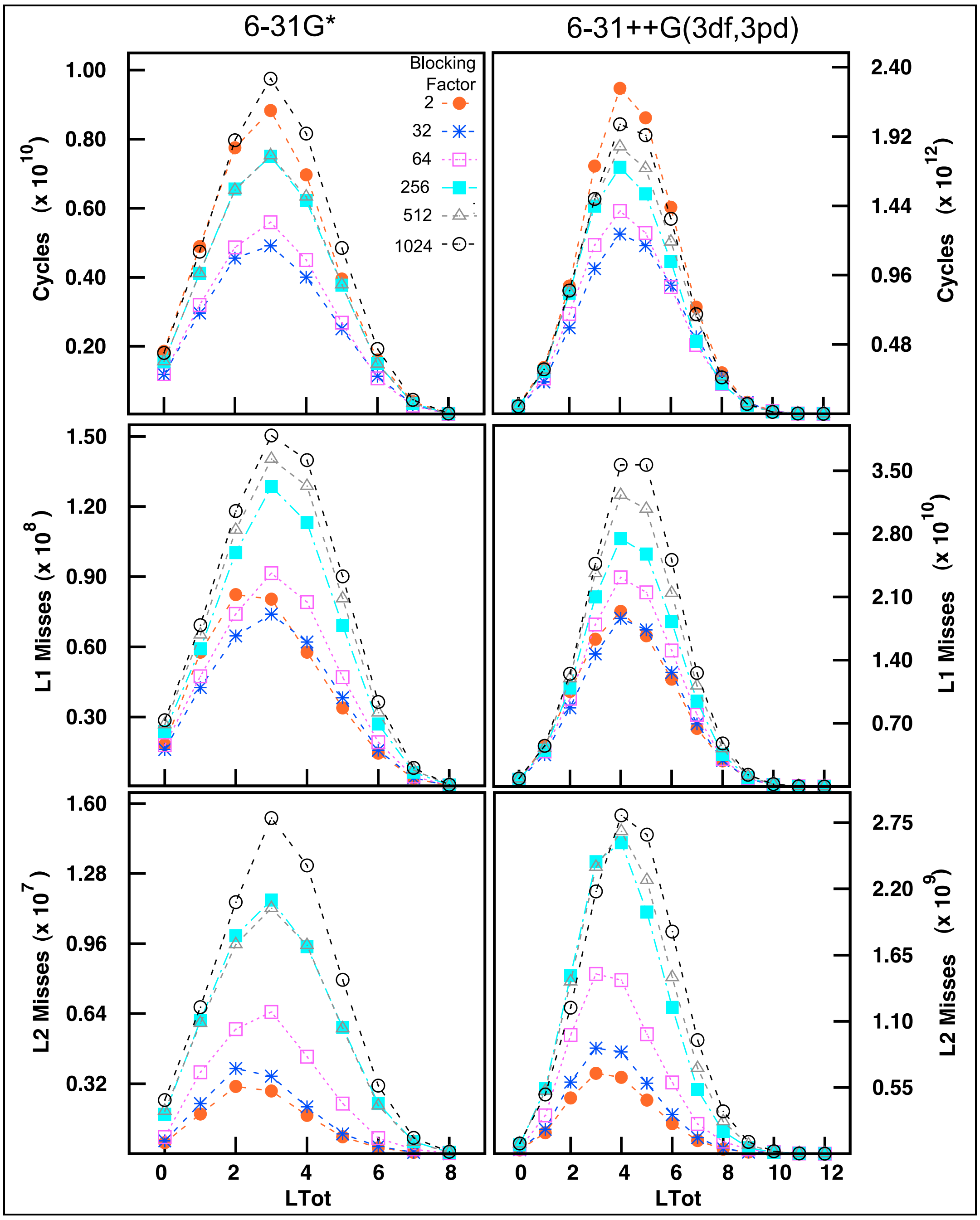
In the case of the larger basis set for k300a-08, the ordering of cycle count curves are similar to k300a-08 with the 6-31G* basis set. But, the cycle count peaks occur at LTot = 4, 5. Cycle counts are two orders of magnitude larger than those obtained for 6-31G*. Unlike the k300a-04 system using the larger basis set, here the 32Kw blocking factor gives the lowest measured cycle counts per LTot. Ordering of L1 misses are identical across the k300a-04 and k300a-08 systems, while the ordering of L2 misses are almost similar to those for k300a-08 with the 6-31G* basis set. Unlike k300a-04 and the larger basis set, the use of a 32Kw blocking factor is not pathological. It is interesting to note that as system and basis set size have increased, the 256Kw – 1024Kw blocking factors are now well clustered indicating its higher L1 miss count.

From the plots given in Figures 4.1 and 4.2 there are two observations on blocking factors which gave the lowest cycle counts: (a) the blocking factor that provided the lowest cycle count for the value of LTot is the one that gives the largest number of batches; (b) in all the plots there were at most two blocking factors that gave the lowest cycle counts per LTot (as in Table 4.6), but one blocking factor always out performed the other in terms of obtaining the lowest cycle counts. Thus, it would be better to tailor the blocking factor to the entire calculation rather than to the LTot value.

Figures 4.1 and 4.2 considered cycle count and total L1, L2 miss curves for a set of cache blocking factors, as a function of LTot. To augment these plots we now consider the FLOPs per LTot, with a view of using it to measure how well PRISM is able to use floating point hardware as a function of cache blocking.

From earlier discussions it was noted that FLOP costs increase as a function of the angular momentum type and contraction length of the underlying basis set. Thus the expectation is that FLOPs should be greatest for those values of LTot which have the largest cycle count per LTot. Plots for the variation in floating point performance per LTot i.e. FLOP count per cycle are shown in Figure 4.3. FLOPs for each system are categorized by the basis set and molecular system. Hence on the left hand side of the figure, the first plot is for k300a-04 using a 6-31G* basis set, below which is the k300a-08 system using a 6-31G* basis set. The FLOP count

Figure 4.2: Per LTot breakdown of cycle counts and total cache misses (L1 and L2) for the k300a-08 water cluster system using a 6-31G* basis set and the HF method on an AMD848 Opteron system.



per cycle is a derived quantity which is indicative of the average rate at which floating point operations are executed[6].

Considering the k300a-04 system with the 6-31G* basis set the FLOPs decrease initially as LTot increases and then rises to a maximum before decreasing again. There is a slight increase at LTot = 8, which is to be expected as higher angular momentum quantities require more FLOPs. For k300a-04 and the 6-31++G(3df,3pd) basis set, there is a drop in the FLOP count at LTot = 1, followed by an increase to a maximal value and then a trailing decrease. With the k300a-08 system, the peak FLOPs occurs around LTot = 5 for the 6-31G* basis set. It peaks at LTot = 4 for the 6-31++G(3df,3pd) basis set.

Cycle counts peak at LTot = 3 for 6-31G* (for both k300a-04 and k300a-08) and for 6-31++G(3df,3pd) its LTot = 5 (k300a-04) and LTot = 4 (k300a-08). However, as measured FLOPs peaks do not correlate with these cycle count peaks. Thus, the initial expectation that FLOPs would peak for peak values of cycle count is not valid. This observation is possibly the side-effect of the processor being stalled, waiting for appropriate cachelines to be streamed from the L2 cache, and hence cannot saturate the floating point functional units. Comparing the overall FLOPs per LTot, between k300a-04 and k300a-08 for both basis sets, it is seen that the smaller k300a-04 achieves better use of the on-chip floating point units than the larger k300a-08 system.

To summarize the discussion for this sub-section on the effects of cache blocking as a function of quartet type, (a) it was seen that there are two blocking factors which yield the lowest cycle count per LTot for the k300a-04 system using a 6-31G* basis set and the HF method. By use of an expanded set of systems and basis sets, it was seen that (b) the blocking factor which gave the lowest cycle counts per LTot depends both on the molecular system and basis set being used. For all the expanded systems, (c) the use of a single blocking factor was a better option as execution time would be skewed towards those values of LTot with large number of batches and larger angular momentum quantum number shell-quartets. Also, (d) the total FLOPs per cycle for a quartet type was not directly related to those values of LTot with the largest number of batches. This possibly indicates that PRISM's use of floating point hardware is being hampered by excessive L1 cache misses.

4.4.5 Cache Blocking as a Function of Benchmark Systems and Architecture

In previous sub-sections, we considered the effects of cache blocking on the cycle, instruction and L1, L2 miss counts. We also considered if it was appropriate to use different blocking

[6]On the AMD848 Opteron, the hardware performance counter event used is the PAPI native event FP_MULT_AND_ADD i.e. the sum of events from both the FADD and FMUL pipelines

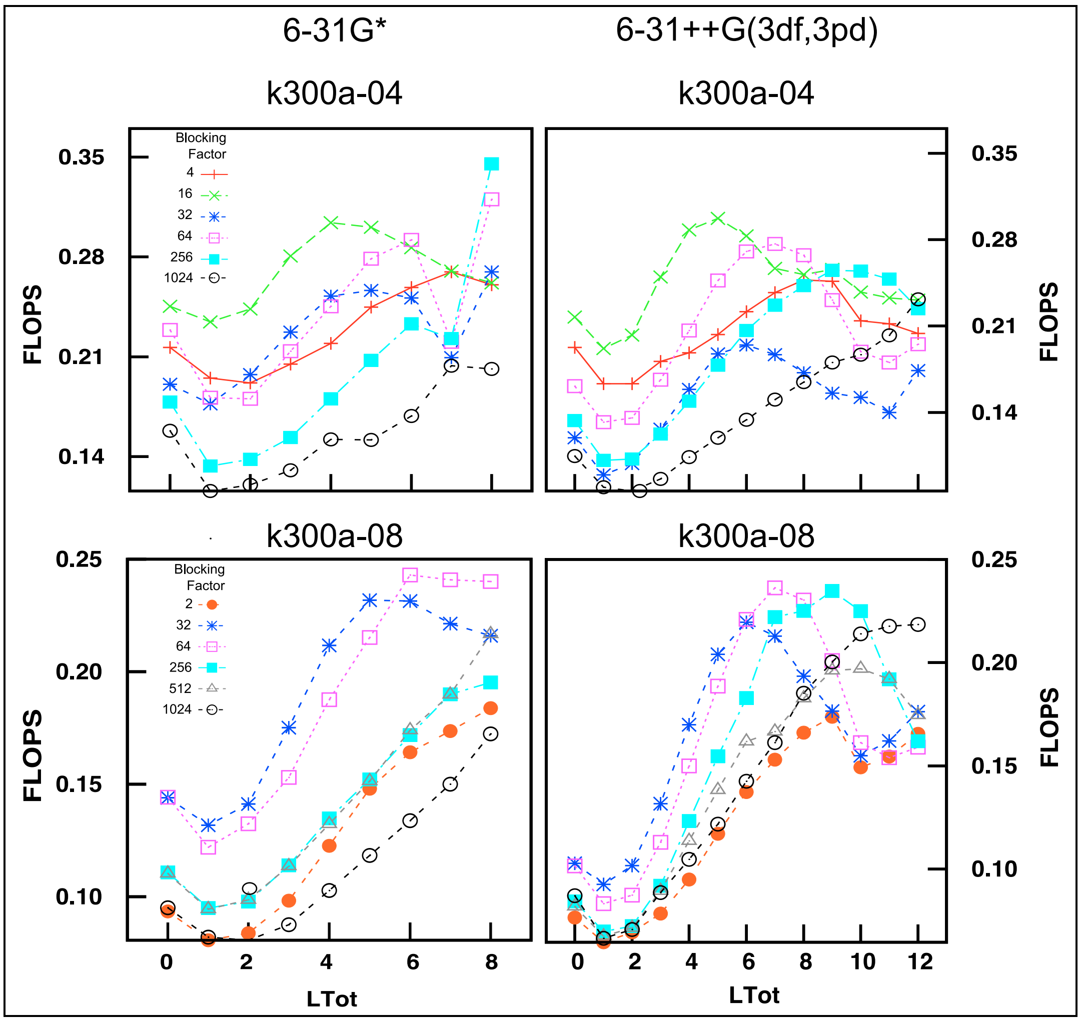


Figure 4.3: FLOP count per LTot for the k300a-04 and k300a-04 water cluster systems using the HF method on a 2.2Ghz AMD848 Opteron processor

factors on a per LTot basis. It was found that while the optimal blocking factor does vary for different values of LTot, there was not a strong case to do this as the total execution time was dominated by those values of LTot that gave rise to the largest number of batches. In this sub-section, we examine the sensitivity of blocking factor to the molecular system, calculation type and hardware architecture.

In Table 4.9 the blocking factor that gave the lowest execution time is reported for 5 benchmark systems, 7 hardware platforms and 2 calculation types. Also given are the default cache blocking factors. (Results for the k300a-08 system using the 6-31++G(3df,3pd) basis set are excluded as it required more physical memory than was available on all machines, except for the AMD848 system).

The table is divided into two sections, corresponding to results for the HF and B3LYP

Table 4.9: Optimal cache blocking factor observed for the five test systems, across six processor platforms

		Default Blocking	k300a-04		k300a-08	test397	α-Al_2O_3
		Size (Kw)	A*	B†	A*	3-21G	3-21G*
HF	Opteron	64	32	64	32	32	64
	Athlon64	32	64	64	64	32	64
	EM64T	128	64	64	64	64	64
	Pentium 4	64	64	64	64	32	64
	Pentium M	64	64	64	64	64	64
	G5	32	32	64	32	32	64
	G5-XServe	32	32	32	32	32	64
B3LYP	Opteron	64	64	64	64	32	64
	Athlon64	32	64	64	64	32	64
	EM64T	128	64	64	64	64	64
	Pentium 4	64	64	64	64	32	64
	Pentium M	64	64	64	64	64	64
	G5	32	32	64	32	32	64
	G5-XServe	32	32	64	32	32	64

A* – 6-31G* basis set B† – 6-31++G(3df,3pd) basis set

methods. The hardware architecture type is given on the left hand side of the table. The default blocking factor for each hardware architecture is given in the 'Default Blocking Size' column. Finally, the blocking factor which gave the lowest cycle counts is given under each molecular system and basis set columns.

For k300a-04 using the 6-31G* basis set for HF, the optimal blocking factor is 64Kw for the Opteron and 32Kw for the Athlon64. The 64 Kw blocking factor is the dominant blocking factor for all Intel processors. While the optimal factor for the PowerPC processors is 32Kw.

Moving to the larger basis set with k300a-04, it is seen that all x86 processors do better with the 64Kw blocking factor, whereas 32Kw can be better for PowerPC processors. For the larger k300a-08 system, the trend is similar to the k300a-04 system using 6-31G*. With test397, the results are a mix of 32Kw and 64Kw blocking factors. For the α - Al_2O_3 calculation, on all hardware platforms, a blocking factor of 64Kw gave the lowest cycle counts for HF. If we now look across all HF calculations, a default blocking factor that always gives the lowest cycle count is true only for the Pentium M. For the EM64T, a optimal blocking factor of 64Kw was observed across all benchmarks, though the default blocking factor is a 128Kw.

Considering the B3LYP results, for k300a-04 and the 6-31G* basis set, the 64Kw blocking factor is preferred on x86 processors and 32Kw on PowerPC. For k300a-04 and the larger basis

Table 4.10: Timing differences for a 32Kw blocking factor versus a 64Kw blocking factor, expressed as a percentage.

		k300a-04		k300a-08	test397	α-Al_2O_3	Sum of	Preferred
		A*	B†	A*	3-21G	3-21G*	Percentages	Factor
HF	Opteron	−2.2	+6.0	−0.5	−2.3	+3.4	+4.3	64
	Athlon64	+1.2	+5.4	+0.2	−1.8	+3.6	+8.6	64
	EM64T	+7.5	+10.6	+5.0	+3.8	+6.8	+33.7	64
	Pentium 4	+4.6	+8.8	+0.8	−1.2	+5.5	+18.5	64
	Pentium M	+2.8	+7.8	+2.2	+0.5	+5.4	+18.7	64
	G5	−6.0	+2.2	−5.1	−6.4	+0.8	−14.5	32
	G5-XServe	−7.8	+0.9	−5.3	−7.2	+0.7	−18.6	32
B3LYP	Opteron	+0.6	+3.1	+0.9	−0.3	+2.5	+6.8	64
	Athlon64	+0.5	+5.1	+0.1	−1.2	+3.2	+7.7	64
	EM64T	+3.6	+10.8	+2.9	+2.4	+5.5	+25.2	64
	Pentium 4	+2.0	+7.7	+0.7	−0.6	+4.4	+14.1	64
	Pentium M	+1.5	+7.2	+1.9	+0.8	+4.2	+15.5	64
	G5	−3.0	+2.4	−3.9	−5.0	+1.3	−8.2	32
	G5-XServe	−4.0	+1.9	−3.8	−6.0	+0.6	−11.5	32

A* – 6-31G* basis set B† – 6-31++G(3df,3pd) basis set

set, it is entirely 64Kw. For k300a-08, all x86 processors performed better with the 64Kw blocking factor, whereas it was 32Kw for PowerPC. The results for test397 are mixed as in the HF case and the α - Al_2O_3 calculation is similar to the HF case in that the 64Kw blocking factor is preferred.

As Table 4.9 contains a mix of 32Kw and 64Kw factors, it is useful to determine how each performs relative to the other if only one blocking factor is used for all the benchmark systems. In order to do this we now consider the relative difference in timing between the two cache blocking factors across all systems.

Table 4.10 presents the relative difference in times between 32Kw and 64Kw. The first part of the table corresponds to the HF method and the second to the B3LYP method. The hardware platforms and methods are given on the left hand side. For each platform, the difference in times between 32Kw and 64Kw blocking factors expressed as a percentage[7] is given for each benchmark system. A positive entry in the table indicates that a 32Kw blocking factor is preferred, whereas a negative entry indicates that 64Kw is preferred. The cumulative percentages for a given hardware platform is given in the 'Sum of Percentages' column. Based on the sum of percentages, a preferred blocking factor is determined in the 'Preferred Factor' column.

[7]i.e. $100*\frac{(32KwTime-64KwTime)}{32KwTime}$

For the Opteron it is seen that the k300a-04 system with 6-31G* runs 2.2% faster if a 32Kw blocking factor is used. Whereas, for the k300a-04 system with the larger basis set, it runs 6% slower with the 32Kw blocking factor than a 64Kw blocking factor. For k300a-08 and 6-31G*, there is a 0.5% increase in time if 32Kw is used. The test397 system runs 2.3% faster with a 32Kw blocking factor and the α - Al_2O_3 system runs 3.4% slower with a 32Kw blocking factor. Overall, the sum of these percentages is +4.3%. This indicates that overall use of a 32Kw blocking factor results in times that are 4.3% slower compared to a 64Kw blocking factor, hence the preferred blocking factor for the Opteron, is 64Kw.

If we consider the Athlon64 processor, the absolute difference in timings for individual benchmarks is similar to those obtained for the Opteron, and a 64Kw blocking factor is also preferred.

For the EM64T system all benchmarks perform better using a 64Kw blocking factor. It is interesting to note that the sum of differences shows that the use of a 32Kw blocking factor leads to a 33% increase in overall execution time.

For the Pentium 4 system, apart from test397, all other benchmarks perform better with 64Kw. Overall an 18.5% increase in execution time results from the use of a 32Kw blocking factor, thus a 64Kw blocking factor is preferred. It is of interest to compare the difference in times for the EM64T and Pentium 4 systems, as both use the NetBurst microarchitecture. There is a 15.2% difference in overall timings between the two processors. This indicates that the EM64T is more sensitive to the use of an appropriate cache blocking factor, than the Pentium 4 system.

The Pentium M performs better with a 64Kw blocking factor, across all molecular systems. Use of a 32Kw blocking factor would result in times that are 18.7% slower.

For the G5 system, use of a 32Kw blocking factor is beneficial except for k300a-08 and α - Al_2O_3 systems. Overall, use of a 32Kw blocking factor results in a 14.5% reduction in runtime. Similar trends are seen for the G5-XServe system, where there is a 18.6% reduction in runtime when 32Kw is used compared to 64Kw.

Moving to the B3LYP section of the table, the Opteron and Athlon64 processors perform better with a 64Kw blocking factor, there is a 6.8% and 7.7% increase in runtime if a 32Kw blocking factor is used. If we compare B3LYP results with HF, there is very slight variation between the two.

For the EM64T processor, a net 25.2% increase in runtime results for a 32Kw blocking factor. This increase is not as large as what is seen for HF. The Pentium 4 system has an overall increase of 14.1% in its runtime for using a 32Kw factor.

The Pentium M shows a 15.5% increase in runtime for the 32Kw blocking factor.

The G5 and G5-XServe have a 8.2% and 11.5% reduction in runtime for the use of a 32Kw blocking factor.

To summarize, it would be beneficial to use a 64Kw blocking factor on x86 machines and a 32Kw blocking factor for PowerPC processors, for the set of benchmarks used here.

4.4.6 Summary: PRISM and Cache Blocking

To conclude this section, the PRISM algorithm limits the number of shell-quartets processed to cache blocked quantities, this influences the batches processed by the inner loop of PRISM. An appropriate blocking factor produces an optimal run-time and this is subject to both the input molecular system and basis set used. The optimal blocking factor is shown to influence the L2 and L1 miss rates, as its use leads to the lowest cycle count for an ERI class (i.e. all the integrals for a given LTot) with the most number of integrals. It is better to use one fixed cache blocking parameter for an entire SCF cycle than to dynamically vary the cache blocking according to the quartet angular momentum type. For the set of benchmark systems which were assessed, a 64Kw blocking factor was found to be best for x86 processors, while a 32Kw blocking factor gave best performance for the PowerPC processors.

4.5 A Linear Performance Model

In the previous section it was shown that the instruction count, cache miss counts and overall execution time of the Gaussian code varies significantly according to the size of the cache blocking parameter used by the PRISM ERI evaluation algorithm. This section seeks to exploit those variations in order to derive the α, β, and γ fitting coefficients that are part of the LPM. To achieve this objective, three issues need to be considered.

The first issue is to assess whether the fit required by the LPM is numerically stable and whether the PPCoeffs are reasonable given their physical interpretation (average use of the underlying superscalar architecture and the latency of a cache miss). This issue is addressed in Section 4.5.1 where the validity of the LPM on the Opteron, for a range of different test systems is investigated.

Having established the in principle validity of the LPM, the next issue to address is the accuracy of the LPM and the stability of the fitted parameters with respect to changes in calculation type. For example, is the LPM accurate enough to be useful, or are the fitting parameters so highly dependent on the calculation type as to devalue its use? This issue is addressed in Section 4.5.2.

The final issue is the applicability and accuracy of the LPM across a range of different hardware architecture types; while the LPM may successfully describe performance on one hardware architecture type, it is not clear that it will work equally well if the hardware architecture and calculation type is changed. This issue is addressed in Section 4.5.3, where the

Table 4.11: Measured PPCoeffs for the AMD848 Opteron using the LPM using the HF method

PPCoeff	k300a-04		k300a-08	test397	α-Al_2O_3
	A*	B†	A*	3-21G	3-21G*
α	0.7	0.6	0.7	0.7	0.6
β	5.1	2.0	-0.4	-4.5	4.5
γ	320.3	403.1	411.0	367.4	452.6
Fitting Error	2.4%	0.8%	2.6%	1.8%	0.7%

A* – 6-31G* basis set B† – 6-31++(3df,3pd) basis set

applicability of the LPM is considered for the different hardware systems listed in Table 4.3 and for both the HF and B3LYP methods.

4.5.1 Stability of PPCoeffs on the Opteron

Using the five benchmark molecular systems, hardware performance counter results for cycle count, L1 and L2 misses and instruction counts were measured on the AMD848 Opteron system. PPCoeffs were then obtained for Equation 4.1 by use of a least squares fit. The results are presented in Table 4.11 where the PPCoeffs (α, β and γ) are presented column-wise for each system.

The values for α given in Table 4.11 are between 0.6 to 0.7. Values for β range between -4.5 to 5.1, while values of γ range from 320.3 to 452.6. The fitting errors for cycle count, for all systems range from 0.7% to 2.6%. This indicates the LPM is able to fit cycle times well.

Even though the PPCoeffs are fitting coefficients it is useful to attribute physical meaning to these. Thus α can be considered to be the CPI (Cycles per Instruction); β the L1 miss penalty; γ the L2 miss penalty. Using this interpretation, values of α and γ seem reasonable and concur with values given for IPC and measured L2 miss latencies as presented in Table 2.1, Chapter 2.

The values of β on the other hand, are negative and in some cases, implying that a cache miss penalty is beneficial! Clearly negative values for β are 'unphysical'. Possible strategies to remedy this include –

(a) use of an averaged value for β obtained over all systems after setting to zero, unphysical values ;

(b) use the value of β as measured for an L1 miss using the `lmbench` benchmark;

(c) take 50% of the measured L1 latency from `lmbench` on the grounds that a real application code may not experience the worst case latency owing to out-of-order execution and timely prefetching ;

Table 4.12: Cycle count fitting errors for the LPM on the Opteron, when β is ignored, for the HF method

		% Fitting Error						
		I	II	III	IV	V	%AE	σ
System	I	**3.0**	5.3	3.4	6.1	3.6	4.6	1.3
	II	6.4	**1.0**	4.2	4.6	2.1	4.3	1.8
	III	3.5	3.9	**2.3**	4.4	3.6	3.8	0.4
	IV	7.2	5.4	4.0	**1.9**	3.4	5.0	1.7
	V	5.8	1.1	3.7	3.9	**1.3**	3.6	1.9
	Avg. PPCoeffs	4.8	2.2	3.0	3.5	1.9	3.1	1.2

%AE – Percentage Average Error σ – Standard Deviation

No.	*System*	*Basis set*
I	k300a-04	6-31G*
II	k300a-04	6-31++(3df,3pd)
III	k300a-08	6-31G*
IV	test397	3-21G
V	α-Al_2O_3	3-21G*

(d) ignore β in the LPM and fit only for α and γ instead.

(e) use Cycles Per Instruction (CPI) from the microprocessor vendor's literature and measured L1, L2 miss penalties from `lmbench` i.e. entirely replacing α, β and γ;

To address the effectiveness of each of the above options a Percentage Average Error (%AE) metric was used. The %AE is the average value of cycle count fit errors for the five benchmark molecular systems expressed as a percentage. The %AE for the five strategies are, **(a)** 3.0%, **(b)** 28%, **(c)** 12.1%, **(d)** 3.9% and **(e)** 14.3%. Standard deviations for the five measurements range from 0.44% to 1.79%.

Although strategy (a) gives the lowest error, whether β is zero or not is somewhat random. Hence, we have chosen strategy (d) which removes β from the LPM. If this is done, values of α and γ are positive and have physical meaning. In comparison to strategy (a), (d) is in error by an additional 0.9%. Thus in the rest of the chapter, the LPM will be modified to include only instruction counts and L2 cache misses.

4.5.2 Accuracy of the LPM for the AMD848 Opteron

Table 4.12 contains the fitting errors for the various benchmark systems and the HF method when using different PPCoeffs on the AMD848 Opteron platform. The various benchmark systems used to derive the PPCoeffs are listed down the left hand side. These are then used to fit the execution times of the other systems, represented by the different columns. Thus the diagonal elements, given in bold, represents the self-fit error. Off-diagonal elements correspond to the transferability of PPCoeffs from one system to another. Also, given are the percentage average errors for each system and its standard deviation. The final row in the table corresponds to the use of averaged PPCoeffs i.e using the average value of each α and γ from all benchmarks for the Opteron. Each of these benchmarks are encoded as a Roman numeral from I to V.

The diagonal elements vary from 1.0% to 3.0%, indicating the LPM is able to give good self-fits even though β is now ignored. Errors when using the PPCoeffs for system I, with other test systems range from 3.0% to 6.1% (i.e. for Row I). Whereas, the use of PPCoeffs from systems II to V to fit the cycle count for system I results in errors ranging from 3.5% to 7.2% (i.e. Column I).

If we consider the transferability of the LPM from a smaller basis set to a larger one, then in going from I $\rightarrow$ II results in 5.3% error. Going from a larger basis set to a smaller one i.e II $\rightarrow$ I, there is a 6.4% error. If molecular system size is increased i.e. I $\rightarrow$ III, there is a 3.4% error and for II $\rightarrow$ III it is 4.2%. Going from the larger system to a smaller one i.e. III $\rightarrow$ I or II the errors are 3.5% and 3.9%. This shows a smaller molecular system with a smaller basis set gives reasonable results when scaled to the larger system sizes and basis sets considered here.

To estimate the LPM's overall transferability to other systems, the Percentage Average Error (%AE) metric is also given in Table 4.12, along with its standard deviation. The %AE was computed for a given row, excluding the error for the self-fit. The %AE varies from 3.6% to 4.6% (with a standard deviation of 0.9% to 1.7%). This indicates the LPM is reasonably transferable across calculations on the Opteron.

For most systems the use of average PPCoeffs give lower errors, compared to off-diagonal errors. The %AE for using average PPCoeffs is 3.1%, with a standard deviation of 1.2. This is the lowest measured value among the %AE values.

From the above discussion, it can been seen that the LPM can fit all systems well (1.0% – 3.0% error). It is reasonably transferable across systems (3.0% – 4.6% error) and the use of averaged PPCoeffs across a range of systems shows good transferability (%AE of 3.1).

4.5.3 Stability of PPCoeffs Across Different Hardware Architectures

In previous sub-sections the numerical stability, accuracy and transferability of the LPM across a range of molecular systems was considered. In this sub-section, we consider the stability of PPCoeffs on different hardware architectures and between the HF and B3LYP methods.

Table 4.13 presents α and γ PPCoeffs obtained across the various processor platforms and

Table 4.13: Measured LPM PPCoeffs, CPI and L2 latency characteristics across processors and test systems for the HF method

			k300a-04		k300a-08	test397	α-Al_2O_3	Avg.	Std.
		*CPI**	A*	B†	A*	3-21G	3-21G*		Dev.
	Opteron	$0.3\dot{3}$	0.69	0.63	0.68	0.66	0.64	0.66	0.03
	Athlon64	$0.3\dot{3}$	0.68	0.59	0.65	0.64	0.63	0.64	0.03
	EM64T	$0.3\dot{3}$	1.20	1.12	1.28	1.25	1.07	1.18	0.08
α	P4	$0.3\dot{3}$	1.16	1.08	1.20	1.18	1.05	1.13	0.06
	PM	$0.3\dot{3}$	0.75	0.67	0.75	0.72	0.67	0.71	0.04
	G5	0.25	0.77	0.74	0.74	0.73	0.67	0.73	0.04
	G5-XServe	0.25	0.76	0.73	0.74	0.70	0.65	0.71	0.04
		L2Lat^{+}							
	Opteron	300	361	426	390	454	422	411	35.8
	Athlon64	238	599	625	459	505	529	543	68.2
	EM64T	499	190	199	180	187	182	187	19.0
γ	P4	430	236	227	244	235	181	224	25.2
	PM	204	76	81	79	79	89	81	4.9
	G5	692	439	415	466	452	463	447	20.8
	G5-XServe	614	467	445	459	490	492	474	20.1

A* – 6-31G* basis set B† – 6-31++(3df,3pd) basis set

L2Lat^{+} – Measured using `lmbench` CPI* – From vendor literature

benchmark systems. The table is divided into two sections, corresponding to the two PPCoeff components namely α and γ. The first column in the section for measured α is the theoretical Cycles per Instruction (CPI) for each processor as obtained from vendor literature. For γ the first column gives the `lmbench` measured latency to access the L2 cache (ref. Appendix, Section A.2). The final two columns 'Avg.' and 'Std. Dev.' are the average value for α and β and the standard deviation across each row.

If we consider k300a-04 with a 6-31G* basis set, the values of α, going down the column, vary from 0.68 to 1.20. There are two distinct groups of values, those which are less than 0.75 and those which are greater than 1.

Using vendor provided numbers, the Opteron upto the Pentium M have the same values of α. The PowerPC processors have a value of 0.25. If we use the physical definition of α, the suggestion is that the closer the value of α is to the CPI, the better a given code is at utilizing on-chip resources. Thus the ordering is Athlon64, Opteron, PM, P4 and EM64T for a CPI $0.3\dot{3}$.

Values of α for the PowerPC processors indicate that x86 based processors perform better, though the PowerPC processors are rated for a CPI of $0.2\dot{5}$. The large values of α (> 1) for the

P4 and EM64T is most likely explained by the NetBurst architecture used for these chips. The architectural emphasis for these processors was towards the use of deep pipelines to ensure high clock rates [87]. The downsides to this approach are that pipeline flushes are expensive and cache misses re-circulate until data references have been satisfied.

Moving down the column to the L2 latencies for the k300a-04 and 6-31G* basis set, we see very different results for the various processors. Note that the LPM ignores β and, hence, L1 latencies would be absorbed into the values of α and/or γ. This would be reflected in an increased value for both quantities. An increased value of α would indicate larger CPI values, whereas a large γ value would indicate that the L2 miss latency potentially encompasses L1 miss latencies also.

It is interesting to note that except for the Opteron and Athlon64 systems, all other systems have values of γ that are less than the expected worst case latencies, with the Pentium M being the lowest. In the case of the AMD processors, it is posited that the interplay between three other microarchitectural features is responsible – (a) out-of-order execution (b) hardware prefetching of cachelines [43, 76, 289] and (c) L1 to L2 intra-cache bandwidth. Compared to the other microprocessors, the AMD processors do not have deep out-of-order execution resources i.e. the AMD processors do not have as many instructions in-flight compared to the Intel and PowerPC processors. Hence it may not be able to mask the effects of long latency loads/stores as effectively. Hardware prefetching also plays a role in reducing the L2 penalty for all of the microprocessor systems used here. While the Opteron and Athlon use hardware prefetching, it is likely that prefetches were not timely enough. Once data for this prefetch makes its way to the L1 cache, it would cause lines to be evicted from L1 and would cause conflict misses for other cachelines. The AMD processors' L1/L2 data cache have cachelines which are exclusive to either the L1 or L2 caches, it performs an allocate-on-write for write misses and implements a write-back policy. All of these features create coherency traffic and thus affects intra-cache bandwidth. In synchrony with these cache features, prefetch requests and legitimate read or write-miss cacheline requests need to be handled. The interplay between these factors could potentially reduce intra-cache bandwidth [200, 305], and lead to an increased value of γ.

The Pentium M has lowest values of γ and we posit that it might be a direct result of its micro-architectural features which involve macro-ops fusion, aggressive pre-fetch hardware and memory reference disambiguation [8] techniques [90, 123].

If we refer to Table 4.14 and consider, the values of α across all benchmarks, we find that these are remarkably stable for any given processor. There are some variations depending on

[8]Memory disambiguation is a hardware technique used to speed-up out-of-order execution by identifying and hoisting-out loads in a memory reference stream which are not dependent on prior loads and stores.

Table 4.14: Measured LPM PPCoeffs, CPI and L2 Latency characteristics across processors and test systems for the B3LYP method.

		k300a-04		k300a-08	test397	α-Al_2O_3	Avg.	Std.
		A*	B†	A*	3-21G	3-21G*		Dev.
	Opteron	0.59	0.60	0.63	0.60	0.60	0.61	0.02
	Athlon64	0.63	0.58	0.60	0.62	0.60	0.61	0.02
	EM64T	1.20	1.12	1.28	1.25	1.07	1.18	0.07
α	Pentium 4	1.12	1.07	1.17	1.12	1.02	1.10	0.06
	Pentium M	0.72	0.66	0.73	0.69	0.66	0.69	0.03
	G5	0.77	0.73	0.65	0.72	0.66	0.71	0.05
	G5-XServe	0.74	0.72	0.74	0.70	0.65	0.71	0.04
	Opteron	346	469	369	390	381	391	46.8
	Athlon64	493	542	403	394	478	462	62.6
	EM64T	169	198	144	177	177	173	20.0
γ	Pentium 4	227	224	236	221	173	216	24.5
	Pentium M	73	78	78	78	86	78	4.6
	G5	436	414	519	443	463	455	40.1
	G5-XServe	457	422	459	490	492	474	22.1

A* – 6-31G* basis set B† – 6-31++(3df,3pd) basis set

L2Lat+ – Measured using `lmbench` CPI* – From vendor literature

the molecular system and basis set; a standard deviation between 0.03 to 0.08 is observed, across all the different processor types. For γ the earlier observation that the Opteron and Athlon64 having higher γ values still holds. The lowest standard deviation for average values of γ in descending order are for the EM64T, PowerPC processors, Pentium M, Opteron and Athlon64. The Athlon64 has the largest observed deviation.

We now consider the B3LYP method and PPCoeffs obtained for it. The B3LYP method is a hybrid method which combines HF and numerical quadrature (cf. Section 2.3.4, Chapter 2). The implementation of the B3LYP method uses additional algorithms to the HF method, exercising a different path through the Gaussian code. Thus evaluating PPCoeffs for B3LYP provide another test of the effectiveness of the LPM.

The layout of Table 4.14 is similar to Table 4.13, with hardware systems on the left and benchmark systems across the columns.

If we consider the k300a-04 system with a 6-31G* basis set, values of α get clustered into two different groups, similar to that seen in the table for HF. The values for γ are slightly lower than those of obtained using the HF method, but otherwise the threads seen are similar.

Table 4.15: Self-fitting errors for the HF and B3LYP methods using the LPM

		% Self-Fitting Error				
		k300a-04		k300a-08	test397	α-Al_2O_3
		A*	B†	A*	3-21G	3-21G*
HF	Opteron	3.0	1.0	2.3	1.9	1.3
	Athlon64	4.5	5.8	5.3	6.0	1.8
	EM64T	2.7	1.4	2.0	2.7	0.8
	Pentium 4	2.3	1.8	2.3	2.2	0.5
	Pentium M	2.0	0.8	1.9	2.5	1.0
	G5	1.8	1.5	2.3	2.7	2.2
	G5-XServe	2.4	1.7	2.0	2.9	2.4
B3LYP	Opteron	1.2	2.4	1.8	1.3	6.8
	Athlon64	3.3	6.5	4.8	4.5	5.4
	EM64T	1.0	1.3	5.3	0.8	4.3
	Pentium 4	0.9	1.4	1.5	1.3	3.7
	Pentium M	2.0	0.8	1.2	1.3	2.1
	G5	1.1	1.4	2.8	2.1	2.0
	G5-XServe	1.3	1.6	3.1	2.5	2.8

A* – 6-31G* basis set B† – 6-31++G(3df,3pd) basis set

4.5.4 LPM Summary

In previous sub-sections, it was shown that the α and γ PPCoeffs were able to give good fits for measured cycle counts according to Equation 4.1. The numerical stability of the LPM, with respect to variation in hardware was assessed. The accuracy of the LPM was evaluated on the AMD848 Opteron platform, using the HF method and the various benchmark systems. Fits for cycle counts were in error by 1.3% to 3.0%. The numerical stability with respect to variation of hardware platform and calculation method was performed, and the PPCoeffs were found to be stable.

In Table 4.15 we summarize the observed self-fitting errors for HF and B3LYP across all hardware platforms. Table 4.16 gives the corresponding data for the %AE.

Table 4.15 is divided into two sections corresponding to the 'HF' and 'B3LYP' methods. In each section, the processor is on the left hand side followed by the self-fitting errors for each system. (The row for the Opteron in '% Fitting Error for HF', in Table 4.15, corresponds to the diagonal in Table 4.12).

For Table 4.15, if we now consider the error distribution for any given system across the set of processors for HF, i.e. going down the columns, then the average error ranges from 1.4% to 3.0% (σ ranges from 0.70 to 1.70). The error distribution for all processors, apart from the Athlon64, range from 1.6% to 2.3% (σ ranges from 0.45 to 0.80). For the Athlon64 it is upto

Table 4.16: Percentage average error (%AE) for Transferability and for using Averaged PPCoeffs

	Transferability Error				Avg. PPCoeffs			
	HF		B3LYP		HF		B3LYP	
	%AE	σ	%AE	σ	%AE	σ	%AE	σ
Opteron	4.3	0.6	5.9	1.7	3.1	1.2	3.3	2.8
Athlon64	8.8	2.1	7.0	2.8	6.7	2.6	4.7	2.1
EM64T	6.9	1.8	6.0	1.7	4.8	2.7	4.5	2.9
Pentium 4	7.6	1.8	7.4	3.1	4.7	2.9	4.7	2.6
Pentium M	5.0	0.6	4.3	0.4	3.6	0.8	3.2	1.2
G5	4.5	0.5	4.4	0.7	3.3	1.4	3.3	1.5
G5-XServe	4.3	0.6	3.6	0.5	3.3	1.4	2.8	1.4

4.7% (σ = 1.71). For B3LYP the fitting errors range from 1.5% to 2.9% (σ ranges from 0.9 to 2.0). The errors for a given processor across all systems range from 1.5% to 4.9% (σ ranges from 0.6 to 2.3). Thus, overall, the self-fitting errors obtained across all processors have an upper bound error of approximately 5%.

We now consider Table 4.16. This table gives both the 'Transferability Error' and errors obtained using averaged PPCoeffs ('Avg. PPCoeffs') for a given processor. Transferability errors are used to assess how well the PPCoeffs from one system fit cycle counts for another. The use of averaged PPCoeffs indicates how well fits are on average across all systems. Both errors are presented in terms of the %AE and standard deviation. Results for 'Transferability Error' do not include the self-error when the %AE was calculated.

For Table 4.16 the errors range from 4.3% to 8.8% (σ ranging from 0.5 to 2.1). For B3LYP, the range is 4.3% to 7.0% (σ ranging from 0.5 to 2.8). These ranges indicate the LPM has increased errors, compared to self-fits, arising from the use of PPCoeffs from one system to another.

If we now consider the %AE errors for using averaged PPCoeffs, these show that for HF, there is an error range between 3.1% to 6.7% (σ ranging from 0.8 to 1.2) and B3LYP has an error range between 2.8% to 4.7% (σ ranging from 1.2 to 2.9). These results indicates that the use of average PPCoeffs gives better results for assessing other types of calculations than a single PPCoeff being used to fit another system and basis set, for the same type of processor.

To summarize, the self-fitting errors and %AE results presented in this section for the HF and B3LYP methods show that the LPM can be considered to be reasonably accurate in modelling molecular systems using both HF and B3LYP methods as a function of instruction counts and L2 miss counts, across a range a processors and benchmark systems. It has been shown that the LPM can obtain good cycle count fits of between 4.3% and 8.8% for the HF method and between 3.1% to 6.7% for the B3LYP method.

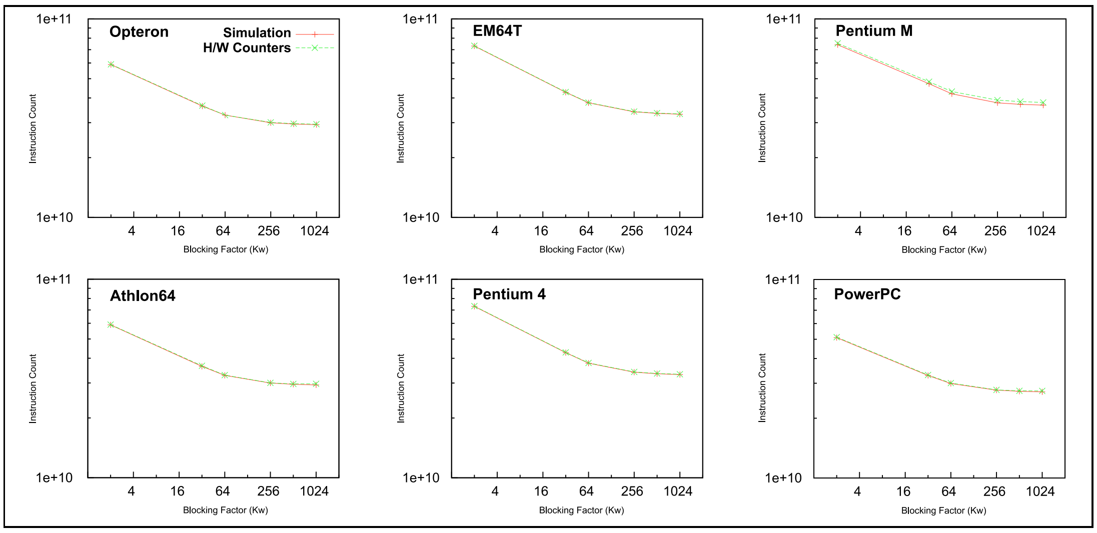


Figure 4.4: Plots for instruction counts obtained from simulation and hardware performance counters for k300a-04/6-31G*. Simulation results were obtained using Valgrind/Callgrind.

4.6 Performance Estimation using Functional Cache Simulation and the LPM

In Section 4.5 it was shown that the LPM can be used to obtain cycle counts, if the instruction count, total L2 misses and PPCoeffs are known. In this section we combine the PPCoeffs obtained by fitting measured results on existing architectures (from Section 4.5) with instruction counts and L2 misses obtained from functional cache simulation, in order to predict performance as a function of cache design.

To perform the functional cache simulation the Callgrind tool, which is part of the Valgrind program supervision framework, is used to perform execution driven functional cache simulation. We consider all six hardware architecture types, but present results only for the k300a-04 benchmark with the 6-31G* basis set and the HF method.

Prior to exploring the effect of architectural changes on performance, it is important to validate the Callgrind functional cache simulator. This is done in section 4.6.1 where the instruction count and cache misses obtained from functional cache simulation are compared with those from hardware performance counters. Following this, we perform a detailed parametric study to identify parameters of interest. This is performed using a cache configuration similar to an AMD Opteron. A detailed study of the effects of changing these parameters across all validated systems is presented in section 4.6.3.

4.6.1 Validation of the Callgrind Functional Cache Simulator

In this sub-section we consider the validation of the Callgrind functional cache simulator. This process is achieved by comparing event counts for instructions executed, total L1 and L2 misses and cycles counts from both functional simulation and hardware performance counters. Using a series of plots, each of these counts is examined to see how well simulation can reproduce observed event counts for the k300a-03 system, using a 6-31G* basis set.

The cache configurations for the simulations presented here are given in Table 2.1, Chapter 2. For the simulations, the L1 organisation of the Opteron and Athlon64 are identical. This is also the case for the EM64T and Pentium 4 processors. The NetBurst architectures use a trace cache for instructions, which Callgrind does not explicitly simulate and hence instruction counts could be different.

Figure 4.4 presents plots for instruction count across all the six processors obtained from simulation and hardware performance counters. In the figure, data from simulation (colored red) and those obtained using hardware performance counters (colored green) are presented in the same sub-plot. The x-axis denotes the PRISM blocking factor used (in Kilowords), the y-axis uses a log scale. The AMD processors (Opteron, Athlon64) are grouped in column 1, the two Intel NetBurst processors (EM64T, Pentium 4) are in column 2, the Pentium M and PowerPC-like cache configurations are in column 3.

Hardware instruction counts for the Opteron and Athlon64 are similar to those obtained from simulation. The EM64T and Pentium 4 simulation results are remarkably similar to those obtained from hardware performance counters. In the case of the Pentium M, results reported from simulation are slightly lower than those obtained from hardware counters. From the figure it is seen that for all six processors, the instruction count obtained from simulation are near identical to those obtained using hardware performance counters.

The LPM ignores L1 misses but for completeness, results for the total L1 misses obtained from simulation and hardware counters are presented in Figure 4.5. For the AMD processors, there is an initial divergence at 2Kw, but simulation is in agreement with measured L1 miss count after 32Kw. For the Intel NetBurst processors, the overall trends are followed but the values from the simulations are less than those obtained using hardware counters. Pentium M results show an initial discrepancy in L1 miss counts, but good agreement is achieved thereafter. Values for the PowerPC shows that the simulation is not able to reproduce either the trends or the absolute counts. The L1 miss counts for the Opteron, Athlon64 and Pentium M processors appear to be satisfactory. L1 miss trends for the EM64T and Pentium 4 are offset by an almost constant amount from measured L1 misses. For the PowerPC, overall L1 miss trends are not reproduced correctly. Discrepancies between simulation and experiment arise as a result of the simplified cache implementation in Callgrind for the L1 cache.

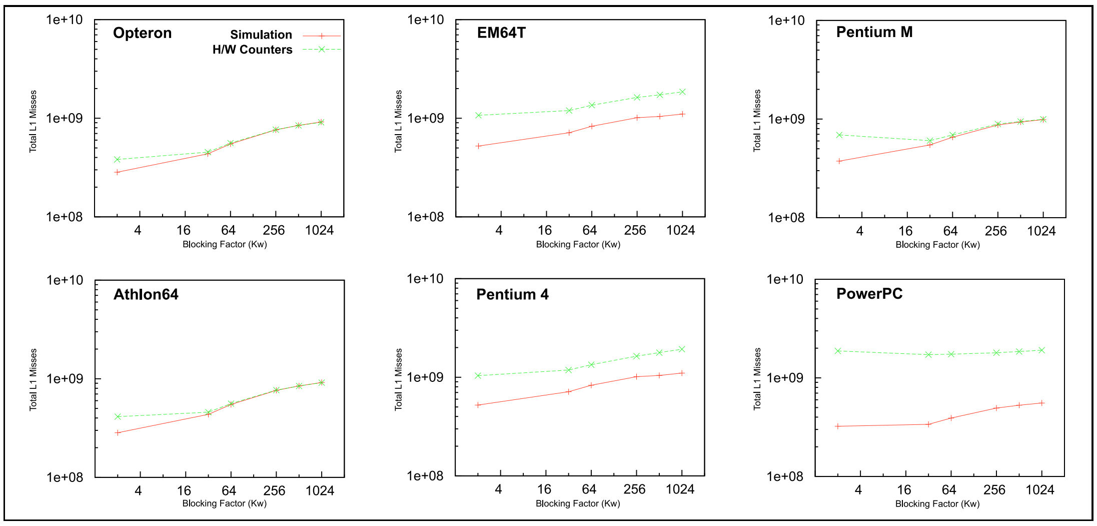


Figure 4.5: Plots for total L1 Misses obtained from simulation and hardware performance counters for k300a-04/6-31G*. Simulation results were obtained using Valgrind/Callgrind.

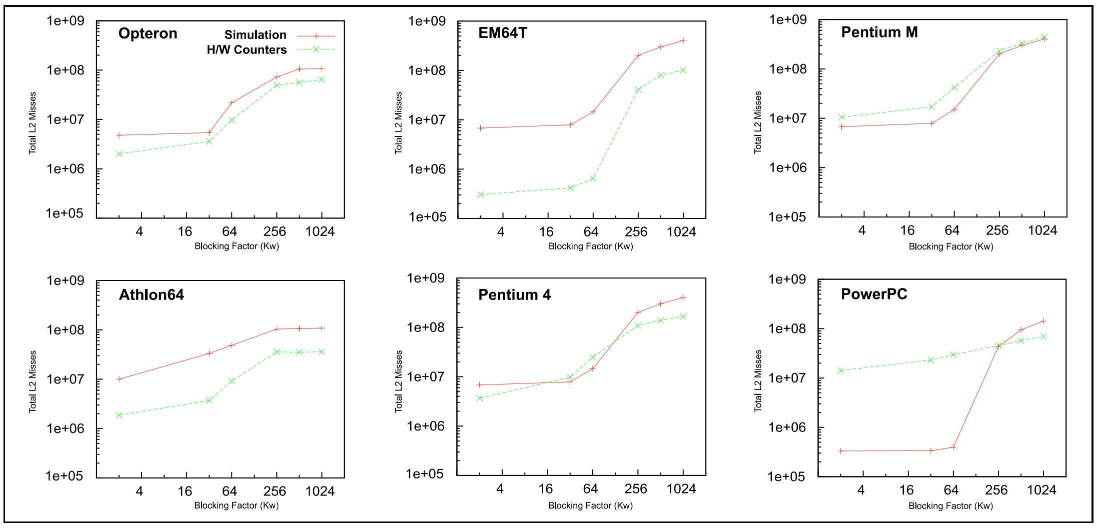


Figure 4.6: Plots for total L2 Misses obtained from simulation and hardware performance counters for k300a-04/6-31G*. Simulation results were obtained using Valgrind/Callgrind.

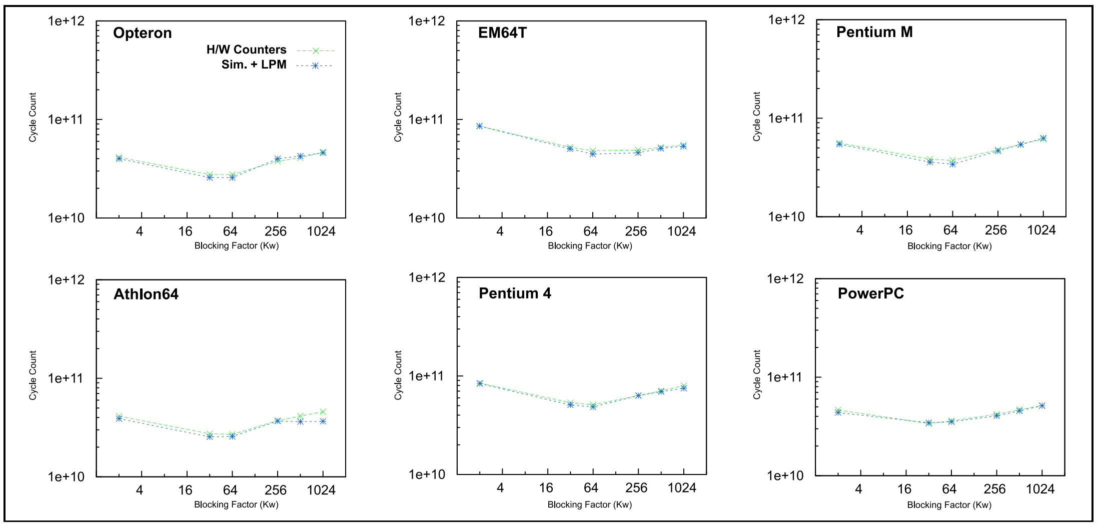


Figure 4.7: Plots for Cycle counts obtained from simulation and hardware performance counters for k300a-04/6-31G*. Simulation results were obtained using Valgrind/Callgrind.

The total L2 misses obtained from simulation and hardware performance counters are shown in Figure 4.6. For the Opteron, values produced by simulation are slightly greater than those obtained from hardware counters, yet simulation is able to closely reproduce the trends observed in hardware counter results. In the case of the Athlon64, a large increase is seen before the L2 miss curve levels off at 256Kw and beyond. Simulation is unable to capture the initial slope from 2Kw to 32Kw and the increase from 32Kw to 256Kw. For the EM64T, simulation over-estimates L2 misses upto 64Kw by two orders of magnitude, yet it is able to reproduce the overall trend. For the Pentium 4, simulation results are well correlated with measured values. In the case of the Pentium M there is good agreement between simulation and hardware counter values. For the PowerPC, simulation is unable to reproduce the overall trend upto 64Kw. After this point a steep jump in simulated L2 misses arises and this is not in agreement with measured hardware counter values. In summary, simulation is able to reproduce the L2 misses satisfactorily for the Opteron, Pentium 4 and Pentium M systems.

The final validation plot is for cycle counts obtained from simulation and hardware counters. In Figure 4.7 for each processor there is an additional curve which corresponds to measured hardware counter cycle counts and is colored blue. The 'H/W Counters' curve corresponds to hardware performance counter results and the 'Sim. + LPM' curve corresponds to the use of PPCoeffs and the LPM (Equation 4.1) for each platform using I_{Count}, $L1_{Misses}$, $L2_{Misses}$ which were obtained from Callgrind.

For the Opteron, simulation results are slightly larger than those obtained using hardware counters. The use of the LPM and PPCoeffs with values for instruction count, total L1 and L2 misses from simulation produces a result which is in very good agreement with hardware

Table 4.17: Valgrind/Callgrind hardware cache configuration

	L1 ICache			L1 DCache			L2 Unified		
	Assoc (Ways)	LineSz (Bytes)	Size (Kb)	Assoc (Ways)	LineSz (Bytes)	Size (Kb)	Assoc (Ways)	LineSz (Bytes)	Size (MB)
Opteron	2	64	64	2	64	64	8	64	1
Pentium M	8	64	32	8	64	32	8	64	1
Pentium 4	8	32	16	8	64	16	8	64	2

counter results. On the Athlon64, the LPM results are in good agreement upto 256Kw, after which it varies marginally from hardware counter results. There is good agreement for 'Sim. + LPM' results for the EM64T and Pentium 4 systems. Excellent agreement is found for cycle counts for the Pentium M. The LPM using instructions and L2 misses from simulation is able to reproduce cycle times rather well, even though the L2 misses produced by simulation had large deviations from measured values. One possible cause of this discrepancy could be due to prefetch instructions not being modelled by Callgrind.

In summary, functional cache simulation using Callgrind is able to obtain accurate instruction counts and can produce reasonable to good reproductions of the L2 misses for most systems. L1 misses are not accurately reproduced. These validation results show that Callgrind results for instruction count and L2 misses can be used for the LPM, as simulation is able to reproduce overall trends for both. Caution is to be taken as to which systems these can be applied to, as prior validation is required. For the six processor types used the Opteron, Pentium 4 and Pentium M cache configurations give reasonable results.

4.6.2 A Parametric Cache Variation Study of PRISM's performance

In the previous section, it was shown that functional cache simulation in conjunction with the LPM could reproduce cycle counts with reasonable accuracy. It was also seen that the overall trends for L2 misses were reproduced well for the Opteron, Pentium 4 and Pentium M like cache configurations. Use of fitted PPCoeffs with counts from validated functional cache simulation enables performance predictions to be made.

The aim of this section is to perform a series of parametric cache variation studies using an AMD Opteron like cache configuration, in order to identify cache parameters of interest. Specifically, the effects of varying cache associativity, cache linesize and total cache size for the k300a-04 system using a 6-31G* basis set are considered.

In depth results for the Opteron are presented in the following order (a) variation of cache associativity, (b) variation of cache linesize, (c) variation of total cache size and (d) breakdown of misses for the variation of linesize and total size.

Table 4.18: Effect of variation in cache associativity of the L1ICache, L1DCache and L2Unified caches, on the LPM predicted cycle counts for an AMD Opteron like cache hierarchy and using k300a-04 with a 6-31G* basis set. All other cache parameters are held constant as given in Table 4.17. Cycle counts are x10^{10}.

L1ICache						**L2DCache**						**L2Unified**					
	Blocking Factor (Kw)						Blocking Factor (Kw)						Blocking Factor (Kw)				
Way	2	32	64	256	512	Way	2	32	64	256	512	Way	2	32	64	256	512
1	4.1	2.6	2.5	4.4	7.3	1	4.1	2.6	2.5	4.4	7.3	1	4.5	3.4	3.3	5.2	7.7
2	4.1	2.6	2.5	4.4	7.3	2	4.1	2.6	2.5	4.4	7.3	2	4.3	2.8	2.7	4.4	7.3
8	4.1	2.6	2.5	4.4	7.3	8	4.1	2.6	2.5	4.4	7.3	8	4.1	2.6	2.5	4.4	7.3
16	4.1	2.6	2.5	4.4	7.3	16	4.1	2.6	2.5	4.4	7.3	16	4.1	2.6	2.4	4.4	7.4
32	4.1	2.6	2.5	4.4	7.3	32	4.1	2.6	2.5	4.4	7.3	32	4.1	2.6	2.4	4.5	7.4
128	4.1	2.6	2.5	4.4	7.3	128	4.1	2.6	2.5	4.4	7.3	128	4.1	2.6	2.4	4.5	7.4
256	4.1	2.6	2.5	4.4	7.3	256	4.1	2.6	2.5	4.4	7.3	256	4.1	2.6	2.4	4.5	7.4

4.6.2.1 Effect of Variation of Opteron Cache Associativity

Cycle counts obtained from the LPM for the variation of the Opteron's cache associativity, as a function of the cache blocking factor are presented in Table 4.18. There are three sub-tables which correspond to varying the associativity of the L1 instruction cache (L1ICache), the L1 data cache (L1DCache) and the L2 unified data and instruction cache (L2Unified).

Associativity is varied from a 1-way set-associate cache to 256-way. Though the LPM does not model the L1 cache, by virtue of ignoring β, small variations in instruction count were observed when both the L1 instruction and data caches were changed.

For the L1ICache and L2DCache, variation of cache associativity across all blocking factors does not give any improvements.

For the L2 cache if we consider the 64Kw result, there is a 4% improvement over the base 8-way set-associative configuration, when the associativity is increased to 16 ways. This indicates there are conflict misses being eliminated in the L2 cache, as a result of increased set-associativity.

4.6.2.2 Effect of Variation of Opteron Cache Linesize

Cycle counts obtained from the LPM for the variation of cache linesize, as a function of cache blocking, are presented in Table 4.19.

There are again three sub-tables corresponding to varying the L1ICache, L1DCache and L2Unified caches. Linesize is varied from 32 to 1024 bytes.

There is no improvement when linesize for the L1 instruction cache is increased. The L1 data cache obtains a 8% increase over the base configuration of 64 bytes if the linesize is increased to 256 bytes for the 64Kw blocking factor. For the L2 cache, this is a 4% improvement if linesize is increased to 128 bytes for the 64Kw blocking factor. The lowest cycle count is obtained for a 1024-byte L2 cacheline using a 256Kw blocking factor.

The lowest cycle counts obtained for increasing linesize, on comparing results for the L1

Table 4.19: Effect of variation in cache linesize for the L1 ICache, DCache and L2 unified caches, on the LPM predicted cycle counts for an AMD Opteron like cache hierarchy and using k300a-04 with a 6-31G* basis set. All other cache parameters are held constant as given in Table 4.17. Cycle counts are x10^{10}.

L1ICache						**L2DCache**						**L2Unified**					
	Blocking Factor (Kw)						Blocking Factor (Kw)						Blocking Factor (Kw)				
Bytes	2	32	64	256	512	Way	2	32	64	256	512	Way	2	32	64	256	512
32	4.1	2.6	2.5	4.4	7.3	32	4.1	2.6	2.5	4.4	7.3	32	4.1	2.6	2.4	4.1	7.1
64	4.1	2.6	2.5	4.4	7.3	64	4.1	2.6	2.5	4.4	7.3	64	4.1	2.6	2.5	4.4	7.3
128	4.1	2.6	2.5	4.4	7.3	128	4.1	2.6	2.4	3.3	4.7	128	4.1	2.6	2.4	3.4	4.8
256	4.1	2.6	2.5	4.4	7.3	256	4.1	2.5	2.3	2.7	3.4	256	4.1	2.6	2.4	2.8	3.5
512	4.1	2.6	2.5	4.4	7.3	512	4.1	2.5	2.3	2.4	2.8	512	4.1	2.5	2.4	2.5	2.8
1024	4.1	2.6	2.5	4.4	7.3	1024	4.1	2.5	2.3	2.3	2.5	1024	4.1	2.5	2.4	2.3	2.4

Table 4.20: Variation of L1, L2 cache size for the Opteron using k300a-04/6-31G*

	L1DCache (Cycle Count [x 10^{10}])					
	Blocking Factor (Kw)					
KiloBytes	2	32	64	256	512	1024
16	4.1	2.6	2.5	4.4	7.3	8.0
32	4.1	2.6	2.5	4.4	7.3	8.0
64	4.1	2.6	2.5	4.4	7.3	8.0
128	4.1	2.6	2.5	4.4	7.3	8.0
256	4.1	2.6	2.5	4.4	7.3	8.0
512	4.1	2.6	2.5	4.4	7.2	8.0

	L2Unified (Cycle Count [x 10^{10}])					
	Blocking Factor (Kw)					
MB	2	32	64	256	512	1024
0.5	4.3	3.0	4.0	7.9	8.1	8.2
1	4.1	2.6	2.5	4.4	7.3	8.0
2	4.0	2.5	2.3	2.7	3.9	6.7
4	4.0	2.5	2.2	2.1	2.4	3.5
8	4.0	2.5	2.2	2.1	2.0	2.2
16	4.0	2.5	2.2	2.1	2.0	2.0

and L2 caches, is 2.3 x 10^{10} cycles. This is achieved either by increasing the L1DCache linesize from 64 to 128 bytes or increasing the L2Unified linesize from 64 to 1024 bytes.

4.6.2.3 Effect of Variation of the Total Data Cache Size for the Opteron

Cycle counts obtained from the LPM for the variation of total data cache size, as a function of cache blocking, are presented in Table 4.20, for both the L1 and L2 data caches. Varying the size of the L1 instruction cache did not lead to any significant variation in total cycle time and hence this data is not shown.

Table 4.20 is divided into two sections. The first corresponds to cycle counts obtained for the variation of the L1DCache size. The second section corresponds to cycle counts obtained for the variation of the L2Unifed cache size. Blocking factors are given as columns.

The total size of the L1 data cache was varied from 16 Kb to 512 Kb and the L2 unified cache size was varied from 0.5 MB to 16 MB. The default configuration for the Opteron's L1 data cache is 64 Kb. Increasing the size of the L1 data cache does not affect execution time.

The default configuration for the L2 data cache is 1MB. For a 64Kw blocking factor, there is a 12% improvement, if the total cache size is increased to 4MB. The lowest cycle count, however, achieved from increasing the L2 size is 2.0 x 10^{10} cycles for an 8MB cache using a 512Kw blocking factor. This is a 20% improvement over the base case.

4.6.2.4 Breakdown of Cache Misses for the Variation of Linesize and Total Size

In previous sub-sections gross trends arising from individual variations of cache associativity, linesize and total size were presented. It was identified that linesize and total L2 size variations affect PRISM's performance. In this sub-section we study the combined effect of changes in L1 linesize and L2 total size on execution time.

Various event counts obtained from Callgrind for an AMD Opteron processor are given in Table 4.21. This table is divided into two sections. The first section presents data for the variation of L1 and L2 linesize. The second section presents data for the variation of the L2 cache. On the left hand side of each section, break-downs are given that correspond to instruction count (ICount), instruction cache misses (I$Misses), L1 data cache read and write misses (D$ReadMiss, D$WriteMiss), instructions which missed in L1 that were forwarded to the L2 cache (L2InstrMiss), L2 data cache read and write misses (L2DataReadMiss, L2DataWriteMiss). Towards the end of each section, the aggregate L1 and L2 misses are given as the 'Total L1 Misses' and 'Total L2 Misses' rows. Using the ICount, 'Total L2 Misses' and PPCoeffs for the Opteron (from Table 4.13) the values for 'Cycle Count (LPM)' are computed using the LPM. Columns in each sub-table correspond to the cache parameter (Linesize or total size) which were varied.

The first sub-table in Table 4.21 presents cycle counts as a function of varying the linesize for the L1 and L2 caches. The 'Original' column gives detailed breakdowns obtained for a base configuration of the Opteron like cache for a 64Kw blocking factor. From the sub-table, it can be seen that the instruction counts (ICount) and instruction cache misses (I$Misses) are constant, indicating variation of linesize does not affect the total instruction count. Variation in the L1 linesize (1024 bytes) causes D$ReadMiss, D$WriteMiss-es to be reduced. This is to be expected, as cachelines now store more data and this increases the likelihood of references being satisfied from the L1 cache. Use of a 1024 byte cacheline in the L1 cache leads to a

Table 4.21: Variation of L1 and L2 Linesizes and L2 Total size for the AMD Opteron, using k300a-04/6-31G*

	Original	L1 Linesize	L2 Linesize
Line size	64Bytes	1024Bytes	1024Bytes
Kw	64	64	64
ICount	3.3E+10	3.3E+10	3.3E+10
I$Misses	8.9E+06	8.9E+06	8.9E+06
D$ReadMiss	3.7E+08	2.0E+08	3.7E+08
D$WriteMiss	1.8E+08	3.2E+07	1.8E+08
L2InstrMiss	7.8E+05	1.1E+05	1.3E+06
L2DataReadMiss	7.4E+06	1.4E+06	5.7E+06
L2DataWriteMiss	7.0E+06	5.5E+05	3.2E+06
Total L1 Misses	5.5E+08	2.3E+08	5.5E+08
Total L2 Misses	1.4E+07	2.0E+06	8.9E+06
Cycle Count (LPM)	2.5E+10	2.3E+10	2.4E+10

		L2 Size	
Cache Size	1MB	4MB	16MB
Kw	64	256	1024
ICount	3.3E+10	3.0E+10	2.9E+10
I$Misses	8.9E+06	2.8E+06	1.4E+06
D$ReadMiss	3.7E+08	5.8E+08	7.3E+08
D$WriteMiss	1.8E+08	1.8E+08	1.9E+08
L2InstrMiss	7.8E+05	3.2E+04	1.2E+04
L2DataReadMiss	7.4E+06	1.9E+05	4.1E+04
L2DataWriteMiss	7.0E+06	5.2E+06	5.2E+06
Total L1 Misses	5.5E+08	7.6E+08	9.2E+08
Total L2 Misses	1.4E+07	5.3E+06	5.2E+06
Cycle Count (LPM)	2.5E+10	2.3E+10	2.2E+10

I$ – Instruction Cache D$ – Data Cache
ICount – Instruction Count ReadMiss – Read Misses
WriteMiss – Write Misses

reduction in the L2 instruction cache misses (L2InstrMiss). Increasing the L2 linesize results in increased L2 instruction misses. L2 data read and write misses are reduced, compared to the 'Original' configuration.

The LPM uses L2 miss information for its cycle count estimation. On comparing the respective L2 data cache read miss and write miss counts for the first sub-table, it is found that an L1 linesize of 1024 bytes results in the lowest L2 data read miss count (amongst the three columns), which in turn corresponds to a lower L1 data read miss count. The use of a 1024 byte L2 cacheline results in an overall lower L2 data write miss rate. Increasing the L1 linesize leads to a reduction in the L1 data cache write misses and also results in lower L2 data read misses. An increase of the L2 linesize results in a reduction of L2 data write misses.

The second sub-table in Table 4.21 presents data for a 1MB, 4MB and 16MB L2 cache using a 64Kw, 256Kw and 1024Kw blocking factors respectively.

Results for a 1MB L2 cache with a 64Kw blocking factor are the baseline used to compare to the 4MB and 16MB caches. Instruction counts are relatively stable across the three columns. The I$Misses reduce with an increase in L2 size. The L1 data cache read misses increase when cache size is increased. L1 data cache write misses are constant. The L2 instruction misses reduce by an order of magnitude. L2 data read misses reduce by two orders of magnitude as the cache size is increased, L2 data write misses are marginally reduced. The rows for total L1 and L2 misses show that aggregate L1 misses have increased and aggregate L2 misses decreased as the L2 size is increased. This behaviour arises from having a larger working-set being present in the L2 cache and as was pointed out in section 4.4.2, PRISM's lack of L1 cache blocking results in much larger L1 misses. Overall, the results predict that use of a 16MB cache will lead to a 12% reduction in total execution time.

4.6.2.5 Break-down Summary

Individual break downs for L1, L2 linesize variation and L2 total size variation show that (a) increasing the L1 linesize does improve performance and that (b) increasing the total cache size for the L2 also improves performance. For PRISM, increasing the L1 data cache linesize is beneficial, as it leads to a reduction in L1 data write misses, as well as L2 data read and write misses. An increase in the L2 size results in increased L2 data read misses, and in a greater reduction of the L2 data read misses compared to write misses.

4.6.3 Variation of L1, L2 Linesize and Total Size for Three Hardware Architectures

In previous sections the use of the LPM with instruction counts and cache misses obtained from functional cache simulation was validated. A cache parameter variation study of an Opteron like cache indicated the relative importance of L1, L2 read and write misses. The results suggest that the L1 and L2 linesize as well as total L2 size most strongly influenced total observed cycle count on the Opteron. In this section, the variation of L1, L2 linesize and total L2 size for the Opteron, Pentium M and Pentium 4 processors as a function of cache blocking is carried out for the k300a-04 system using a 6-31G* basis set. The aim is to assess the impact of these changes across three different hardware architectures types.

The effect of varying the L1 linesize as a function of cache blocking on execution time using the k300a-04 system and a 6-31G* basis set is shown graphically in Figure 4.8. The figure has three plots in a vertical column, which correspond to the Opteron, Pentium M and Pentium 4 systems. Data for the Opteron, previously presented as tables, are now plotted. The plots represent cycle count as a function of the blocking factor and linesize. Each point in the plot is color coded to indicate its relative weight with respect to other points. A cool color (hues of dark blue) represent regions of low cycle count, whereas a warm color (hues of deep orange) represents regions of high cycle counts. Distinct regions are then segregated by a series of contour lines. A color strip on top of the graph gives the color mapping between the lowest and highest cycle count values. In the plots, the x-axis denotes increasing PRISM blocking factors and the y-axis denotes increasing linesize (from 32 bytes to 1024 bytes). Each color coded pixel for cycle count was $\log_{10}$ scale encoded prior to plotting.

Distinct contour lines are formed for cycle count values (x10^{10}) ranging from 10.37 to 10.86 for the Opteron, 10.50 to 10.74 for the Pentium M and 10.65 to 11.04 for the Pentium 4. For the Opteron, its L1 data cache linesize is 64 bytes. The location of this value for the blocking factor that gave the lowest cycle count (i.e 32Kw), is colored in a blue hue and is surrounded by the 10.44 contour line. Increasing the linesize implies moving upwards along the y-axis. As we move towards a linesize of 1024 bytes, each location of y-axis location for 32Kw is bounded by the 10.44 contour line indicating it is still a blue hue i.e. still a low cycle count region. The lowest cycle counts however occur for the 10.32 contour line. This is visible at the (64Kw, 512Kw) and (64Kw, 1024Kw).

For the Pentium M system, a 10.53 contour line demarcates (64Kw, 64 bytes) the default cache blocking factor. Increasing the linesize for this blocking factor results in low cycles counts. The lowest cycle count are obtained in the (256Kw, 512 bytes) and (1024Kw, 1024 bytes) region.

The Pentium 4 shows (64Kw, 64 bytes) as a low cycle count region and increasing the linesize reduces cycle counts. The lowest cycle counts are obtained at (64Kw, 512 bytes) and (64Kw, 1024 bytes). These are surrounded by a 10.65 contour line.

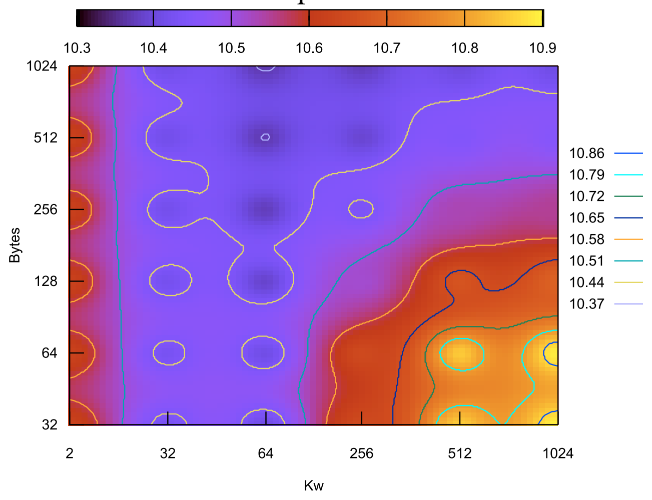
From the above, increasing L1 linesize causes cycle counts to vary very differently for all three systems.

Figure 4.9, presents similar plots for L2 linesize variation. The plot for the Opteron shows

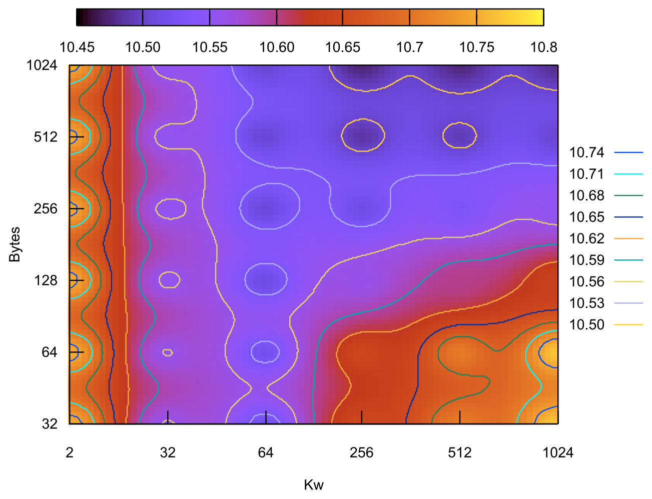
Figure 4.8: Valgrind/Callgrind cycle count results for varying L1 linesize (Bytes) for k300a-04/6-31G*.

L1 Linesize Variation

Opteron



Pentium M



Pentium 4

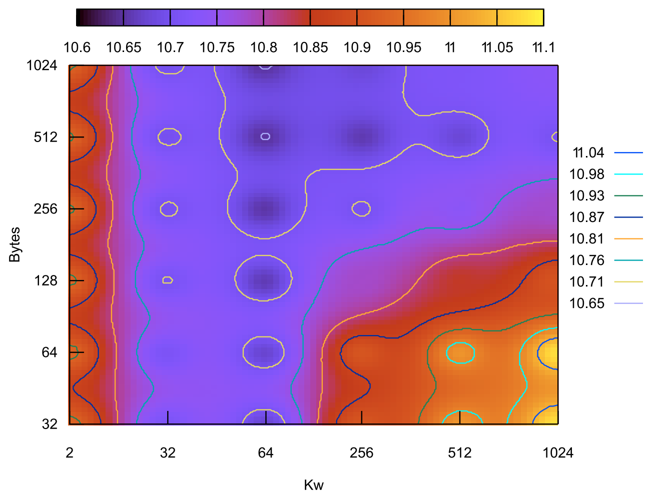


Figure 4.9: Valgrind/Callgrind cycle count results for varying L2 linesize (Bytes) for k300a-04/6-31G*. The magnitude of contour line values is x10^{10}

L2 Linesize Variation

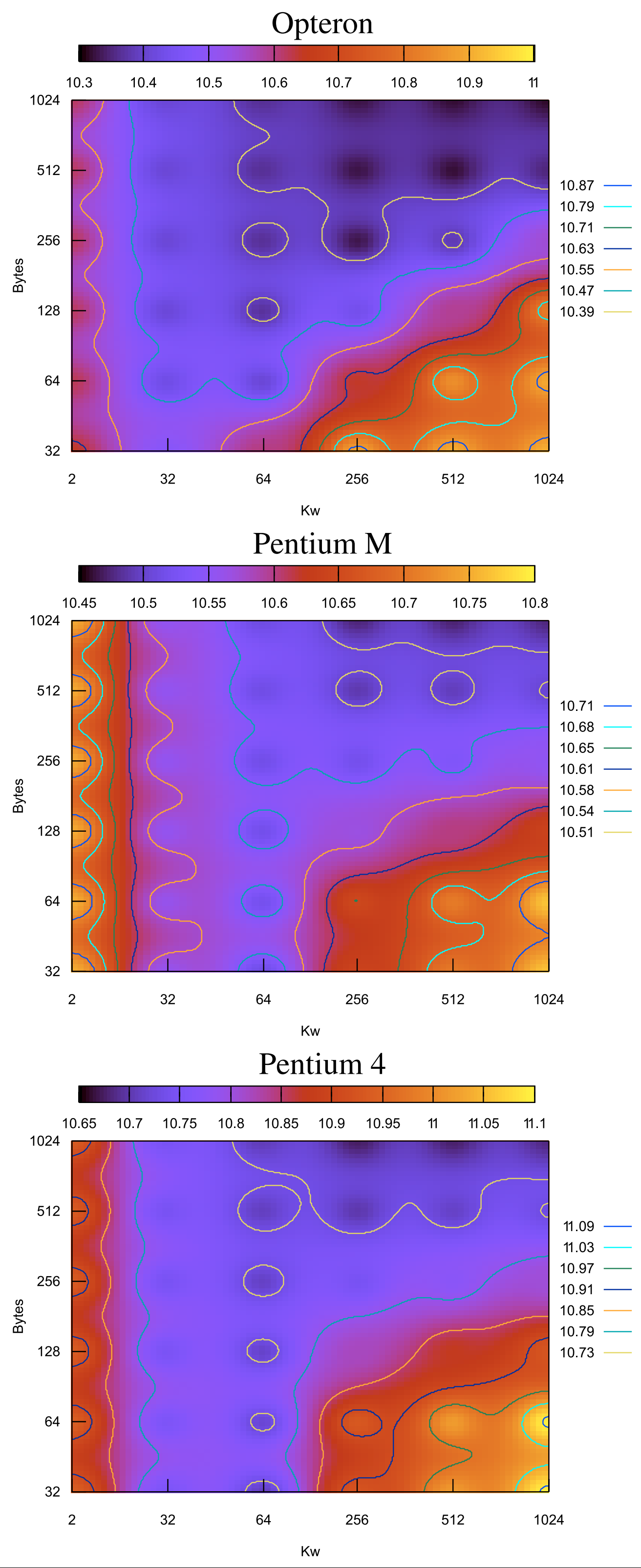
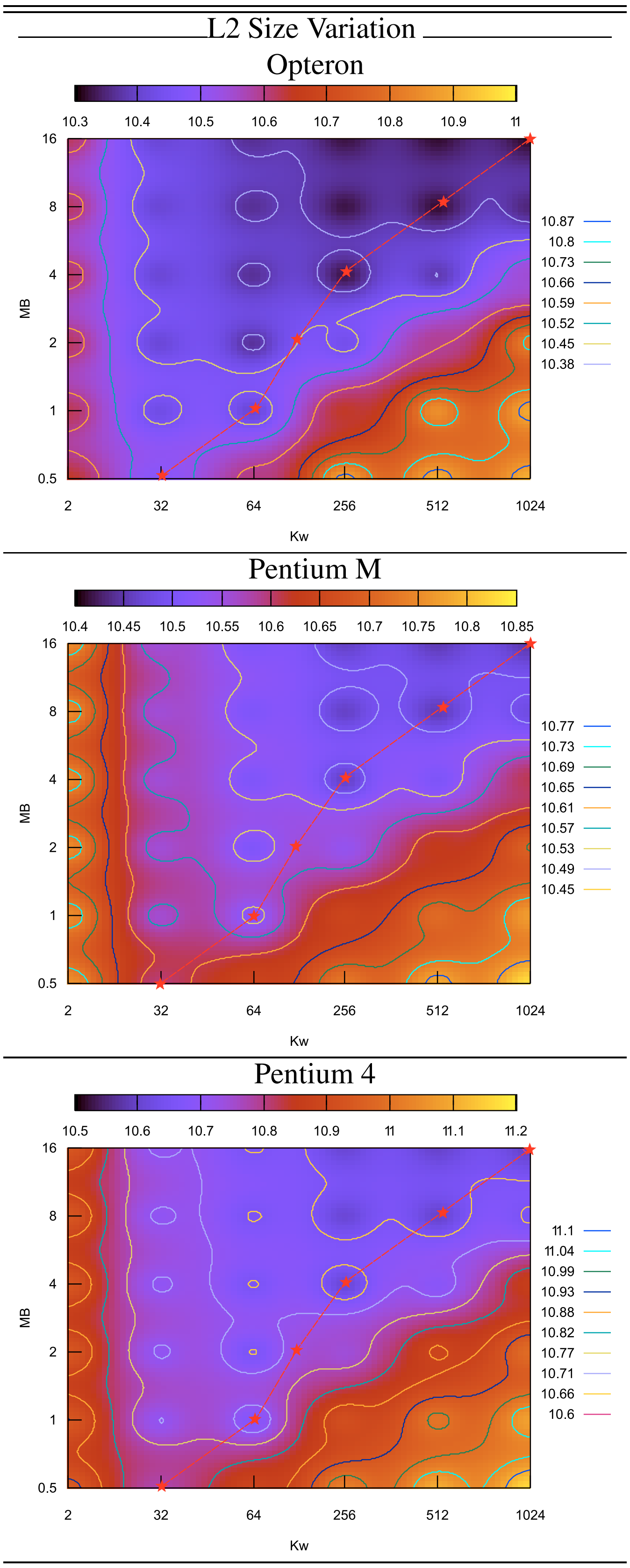


Figure 4.10: Valgrind/Callgrind cycle count results for varying L2 size (MB) for k300a-04/6-31G*.



that the lowest cycle counts are delineated by the 10.39 contour line. The lowest cycle counts occur around (256Kw, 256 bytes) to (1024Kw, 1024 bytes).

For the Pentium M system, the default cache blocking factor if (64Kw, 64 bytes). Cycle count here is a shallow minima and increasing the linesize results in minimums upto 1024 bytes. The lowest cycle counts, however, are located in regions marked by the 10.51 contour line. This occurs in the region of (256Kw, 512 bytes).

The Pentium 4 system is very different from the previous two. The default is at (64Kw, 64 bytes). Increasing the linesize does ensure this is a minimal cycle count. The 10.73 contour line bounds this upto 1024 bytes. The deepest hues of blue are between (256Kw, 512 bytes) and (512Kw, 1024 bytes).

Variation of total L2 size as a function of cache blocking are shown in Figure 4.10. Superimposed on each of the plot is a red line denoting Gaussians' default choice for cache blocking – i.e for a 0.5MB L2 cache, it chooses a blocking factor of 32Kw; for a 1MB cache, it chooses 64Kw and so on.

For the Opteron, corresponding to every value in the y-axis, i.e. the total cache size, islands of cycle count minima are to be found (10.38 and 10.45 contour lines). For the 1MB cache, this occurs at 64Kw; for 2MB at 64Kw; for 4MB at 256Kw; for 8MB at 256Kw and 16MB for 1024Kw.

In two cases the 2MB and the 8MB case, Gaussian's default blocking choice does result in the lowest possible cycle count for a given cache size, for the k300a-04/6-31G* system. For a 2MB cache, the LPM predicts a blocking factor of 64 Kw would perform better than the default choice of 128 Kw. The LPM predicts a 256Kw blocking factor would perform better for an 8MB cache than 512Kw. Similar trends are observed for the Pentium M and Pentium 4.

4.6.4 Summary: Functional Cache Simulation and the LPM

In this section the LPM has been used in conjunction with functional cache simulation for six microprocessor platforms (cf. Chapter 2, Table 2.1). The k300a-04/6-31G* system was used along with a variety of cache blocking factors for PRISM. A comparison between simulation and hardware performance counter results were made and a parametric cache variation study was performed.

The comparison of results obtained from simulation and hardware performance counters found that Instruction counts were in good agreement for all platforms. L1 misses for the Opteron, Athlon64 and Pentium M showed deviations for PRISM blocking factors less than 32Kw but were in agreement there after. In the case of the Pentium 4 and EM64T configurations the general trend was followed albeit there was a constant offset between simulation and hardware performance counters. L1 misses for the PowerPC were not in agreement. Simulation results for L2 misses from the Opteron, Pentium 4 and Pentium M configurations followed the same trends as hardware performance counters. Although the EM64T simulation did follow the general trend of values from hardware performance counters, there was a large offset between the two curves. L2 misses for the PowerPC were not in agreement. On comparing cycle counts, it was found that the LPM results were in good agreement with hardware performance counter results.

Results for a parametric cache variation study using the Opteron, Pentium 4 and Pentium M cache configurations were presented. In-depth results for the Opteron were first discussed before presenting contour plots of the cache variation study for the other two configurations. From the in-depth results it was seen that increasing the L1 linesize and total L2 size reduced the total cycle count, this was attributed to reductions in conflict misses in the L1 data cache and a greater reduction of read misses than write misses for the L2 cache. Contour plots for the three processors showed that Gaussian's default blocking factor works for k300a-04/6-31G* for all but two cases, the 4MB and 8MB L2 cache respectively.

4.7 Related Work

Ramdas et. al [220, 221] perform qualitative analysis and assess the prospects of mapping an implementation of the Rys ERI method [73, 161] onto FPGAs. They present a quantitative analysis of the 'bootstrap' phase of Rys ERI evaluation, which corresponds to 'Generate Significant Shell-Pair List' for the PRISM algorithm (cf. Algorithm 2, page 63). A discrete event simulation is used to determine the impact of arithmetic units (adders and multipliers) in the FPGA. In comparison this chapter has considered the PRISM ERI algorithm, its ERI batching behaviour, cache blocking effects on observed performance.

The LPM is lightweight in obtaining application specific performance characteristics. PP-Coeffs are obtained using hardware counter data which can then be used by either trace based or execution based simulators. The following is a review of related work covering the use of analytic modelling, synthetic benchmarks and simulation to aid in modelling application performance.

Very recently, Björn [84] developed a cycle-approximate instruction set simulation methodology which uses prior training and regression based performance prediction for a series of embedded application benchmarks. The model requires instruction and memory access counter information, which are fitted to observed cycle counts obtained from an ARM v5 cycle accurate simulator. The prediction phase uses functional simulation to obtain instruction and memory access counters, which are then used to obtain fits regression coefficients obtained from prior training runs. Cycle counts are found to be in error by 5%. Björn's general approach is very similar to that taken here in obtaining least-squares fits for the LPM. Unlike Björn, we have focused on ERI evaluation and obtained fits across a range of hardware platforms and benchmark molecular systems. The LPM's results vary from 3.3% to 7.9%.

Other related approaches to application modelling are covered below. Most of these approaches are heavy-weight in comparison to the LPM, with respect to the time taken to obtain results. The following review is biased towards related work which considered performance modelling of scientific application codes.

Using a sparse set of trace based cache simulations, Gluhovsky and O'Krafka [122] build a multivariate model of multiple cache miss rate components. This can then be used to extrapolate for other hypothetical system configurations. This is used in work carried out by Sharapov et al. [242]. They provide a methodology for characterising performance on very large parallel systems. They combine queuing theory models and cycle accurate simulation for estimating parallel performance using trace driven simulation. Traces are collected from a full machine simulator and bus traces are obtained from real hardware. These are then used to drive a trace-driven simulation from which parameters for an analytic model are created to project performance estimates.

Cheveresan et al. [53] perform detailed characterisation of scientific and commercial applications. For their study, traces are generated using an ISA simulator which allows for the capture of architectural traps, direct memory accesses and MMU activity. In their analysis they show that complete scientific codes (rather than kernels) show similar characteristics to commercial applications.

Song et al. [252] create an analytic model to quantitatively predict L2 cache misses on a multi-core chip. They use stack processing and circular sequence profiles to analyze a trace the L2 cache accesses. The model can predict L2 misses for various multi-core architectures using previously obtained traces.

Marathe et al. [171] create a framework for extracting partial access memory traces using dynamic binary re-writing. These traces are compressed using various algorithms tailored for lossless capturing instruction stream traces. These are then used for offline memory hierarchy simulations, which allow them to correlate reference statistics for cache eviction information and streaming behaviour to locations in code that cause this.

Strohmaier and Shan [260] create a synthetic performance probe called APEX-Map, which measures the performance of global data movement. This is characterised as three parameters – the global datasize 'M', temporal locality 'α' and spatial locality 'L'. APEX-Map generates a generic address stream based on non-uniform, random access to global data. From the results obtained, it is possible to generate a multi-dimensional performance surface allowing for the study of spatial and temporal effects.

Marin and Mellor-Crummey [172] create and evaluate a toolkit for semi-automatic modelling of static and dynamic components of an application's characteristics by capturing memory access traces. This stream is analyzed to generate platform independent characteristics of the scientific code being studied. This can then be used to extrapolate performance and other platforms of interest.

Grabelny et al. [104] create a framework for performance prediction based on discrete event simulation. They model systems of interest into individual components i.e. O/S, network device, processor etc. In a two-phased approach, they characterise applications of interest by capturing memory access streams and then replay these using the discrete event simulation framework to derive performance estimates.

Snavely et al. [1] use profile convolving a trace based method which involves the creation of a machine profile and an application profile. Machine profiles describe the behavior of loads and stores for the given processor, while the application profile is a runtime utility which captures and statistically records all memory references. Convolving involves creating a mapping of the machine signature and application profile; this is then fed to an interconnect simulator to create traces that aids in predicting performance.

Ahn and Vetter [7] describe multivariate statistical techniques to analyse hardware performance data from scientific codes using clustering, factor analysis and principal components analysis.

Epshteyn et al. [78] use active learning models along with empirical models to guide generation of efficient BLAS libraries.

Vera et al. [290] use cache miss equations to obtain an analytical description of cache memory behavior of loop based codes. These are used at compile time to determine near optimal cache layouts for data and code.

Andrade et al. [11] extend probabilistic miss equations using analytical modelling to model cache behavior of indirections in memory access streams. This approach works if accesses are uniformly distributed in the array of interest. The goal of this work was to create better analytical models to aid compiler optimizations.

Mathis and Keryson [175] perform analytical modelling of an unstructured mesh application. They parametrise system performance and application specific input parameters in terms of latency, bandwidth and processing rate. Using a detailed analytic model they then make performance and scalability estimates of the unstructured mesh code.

4.8 Summary, Conclusions and Future Work

A linear performance model was proposed to model the measured execution time for ERI evaluation. PPCoeffs (α, γ) were obtained for a set of benchmark systems. The α PPCoeff refers to how well the code uses the superscalar resources of the processor, γ corresponds to the average cost in cycles of an L2 miss.

In this chapter, the LPM was shown to produce good fits for measured cycle counts obtained using hardware performance counters. Using a set of test molecular systems, it was found that the optimal blocking factor is both platform and computation specific.

Evaluation of the LPM in conjunction with functional cache simulation shows it is able to reproduce the trends and cycle counts as a function of varying the cache blocking factor. A parametric cache variation study of the PRISM algorithm was performed and this showed that L1 linesize and total L2 size impact on the algorithm's performance. Detailed breakdowns of read and write misses shows that PRISM's cache performance is limited by read misses generated during ERI computation.

4.8.1 Future Work

For future work, it would be useful to test the use of the LPM at runtime to aid in searching for optimal blocking factors. Data could be gathered for the first couple of SCF cycles, to obtain PPCoeffs. Then the blocking factor could be varied by starting from the default and taking measurements for an increased and decreased blocking factor. Once the blocking factor with the lowest cycle count is measured, it can be used for the remaining SCF cycles.

Using the LPM, a training run could be performed to determine the optimal cache blocking factor for any given hardware architecture, using a range of molecular systems. The results could be weighted to determine if one cache blocking factor would be universally suitable on this hardware architecture. This training would be carried out prior to deploying Gaussian for production use.

Wallin et. al [294], point out in their paper that increased cacheline size aids scientific applications. The design of modern microprocessors is largely driven by commercial workloads, most of which have poor data-locality and suffer from false-sharing, and thus the cacheline sizes of microprocessors are usually 64 bytes. Wallin et. al advocate the judicious use of prefetching to mimic the effect of larger cachelines.

Our cache variation experiments indicate that there is scope to reduce L1 and L2 read and write misses. This can be achieved by incorporating a series of prefetch techniques [43, 278]. One possible way of achieving this is through the use of a prefetch queue [52, 88] in sections of code which exhibit large misses. A prefetch queue is a FIFO queue which holds n address that need to be prefetched. The queue is effective in handling memory references which are interspersed throughout memory. The depth of the queue, which is determined experimentally, is deep enough to ensure that once values are popped off the stack, the cacheline of interest is cache resident.

Routines and specific lines in the source code which cause read and write misses have been identified using the KCachegrind tool [140]. KCachegrind uses output from a Callgrind run to attribute cycle count costs to specific point in source. For future work, prefetch queues could be inserted into specific routines, located using the KCachegrind GUI.

Chapter 5

A Study of Thread and Memory Placement Effects using the Gaussian Electronic Structure Code

5.1 Introduction

In this chapter, we study the effects of thread and memory placement and extend the LPM to account for NUMA effects, using selected application kernels from the Gaussian computational chemistry code [86].

Shared memory, parallel platforms have increased in complexity evolving from UMA (Uniform Memory Access) to NUMA (Non-Uniform Memory Access) machines. As seen in Chapter 3, on NUMA architectures it is faster for a processor to access memory which is local to it, than remote memory. Moreover individual processor chips have now evolved towards multi-core designs [17] accelerating the shift towards asymmetry in memory latency and bandwidth. Multicore processors from various vendors also have subtle differences in their cache hierarchy, e.g. where some have shared L2 on-chip, others have dedicated L2 and shared L3 on-chip [51, 287].

Widely used shared-memory programming models like Pthreads [64] and OpenMP [228], as mentioned in Chapter 3, do not explicitly expose or handle underlying hardware asymme-

Work reported in this Chapter has been carried out in collaboration with Dr. Rui Yang (School of Computer Science, ANU). Material from this chapter was published in: (a) Proceedings of ISPAN 2008, MEMORY AND THREAD PLACEMENT EFFECTS AS A FUNCTION OF CACHE USAGE: A STUDY OF THE GAUSSIAN CHEMISTRY CODE ON THE SUNFIRE X4600 M2, `http://doi.ieeecomputersociety.org/10.1109/I-SPAN.2008.13`; (b) (Accepted) Proceedings of HPCC 2009, A SIMPLE PERFORMANCE MODEL FOR MULTI-THREADED APPLICATIONS EXECUTING ON NON-UNIFORM MEMORY ACCESS COMPUTERS tries[1]. This in effect poses challenges in obtaining good performance for scientific codes on NUMA platforms. The goal of this chapter is to study the effects of thread and memory placement on the observed performance of the Gaussian code on a contemporary multi-core NUMA platform, the SunFire X4600 M2 [265]. To facilitate this, a series of questions are addressed.

(a) What are the performance characteristics (in terms of latency and bandwidth) of the SunFire X4600 M2?

(b) Cache blocking dramatically affects the performance of the Gaussian code. What are the combined effects of cache blocking and placement on the overall runtime of the Gaussian code? How does cache blocking affect scaling as the number of processors is increased?

(c) Can the Linear Performance Model (LPM) be extended to handle NUMA systems? If so, how accurate is it?

(d) Can page migration improve the performance of Gaussian calculations run on NUMA systems?

This chapter is organized as follows: Section 5.2 discusses the architecture and performance characteristics of the X4600 M2, in terms of latency and memory bandwidth for specific thread and memory placements. Section 5.3 reviews the software environment, test molecular systems and modifications made to Gaussian. Section 5.4 demonstrates how the Placement Distribution Model (PDM) from Chapter 3 can be used to study the effects of thread and memory placement in Gaussian; Section 5.5 extends and evaluates the LPM to account for NUMA effects; Section 5.6 uses page migration to affect data locality in Gaussian. In Section 5.7 we review previous work in the area of performance modelling of NUMA systems. Section 5.8 concludes the chapter and discusses future work.

5.2 Performance Characteristics of the X4600 M2

5.2.1 Hardware Platform

The SunFire X4600 M2 platform [265] is a glueless cc-NUMA platform. There are eight sockets populated with 2.6Ghz AMD Opteron 8218 dual-core processors each of which has a 64Kb L1 data cache and 64Kb L1 instruction cache and 1MB of dedicated L2 cache per core. Each processor, shown in Figure 5.1, has all its Northbridge functionality implemented in on-chip silicon [56] i.e. a System Request Interface (SRI), memory controller (MCT), DRAM

[1]Some parallel runtime environments perform rudimentary thread and memory placement either using a first-touch approach for data or mapping logical threads to physical cores e.g the PGI runtime for compiled binaries

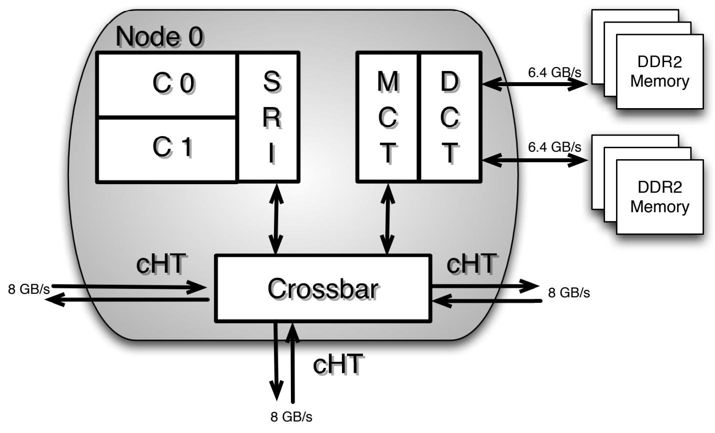


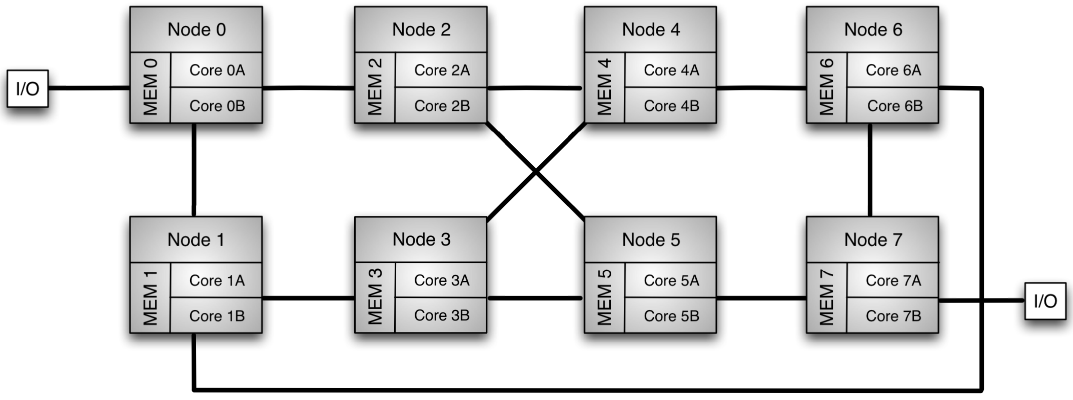
Figure 5.1: Architecture of the Opteron 800 series processor [56]

controller (DCT). The cores (C0, C1) have access to local DDR2 memory via two 6.4GB/s channels. C0 and C1's cache line requests and coherency probes for cache lines (i.e. request for the state of a cache line that is not 'Owned' by that core) are handled by the SRI. There are separate command and data paths implemented as virtual channels [62] (i.e. requests and probes are handled by the command pathways, whereas cache line transit via the data pathway). In this chapter, the grouping of the processor and memory is referred to as a Node.

5.2.1.1 Coherent HyperTransport

The HyperTransport (HT) protocol [282] defines how I/O devices can be linked together in a system, as well as linking processors. The Opteron implements a MOESI cache coherency protocol called Coherent HyperTransport protocol (cHT) on top of HT. Each processor has three duplex links, which operate at 8 Gb/s, and handle coherency (cHT) and HyperTransport (HT) related traffic. These interprocessor and I/O links are managed by a crossbar, which implements separate data and command paths. Cut-through routing of data-packets is used [62]. Cache misses that occur at a given core, are forwarded to the SRI which performs a table driven lookup to determine if the cache line is local to that node or is owned by another processor. In case it is local a request is sent to the local MCT, which is responsible for managing the DCT, in order to access pages from local DRAM. If the cache line is not local to the processor, the SRI uses the crossbar's routing table to determine which of the three cHT links ought to be used to forward the request on. From this point, the cHT implementation of the MOESI coherency protocol, overlayed on top of a snoopy bus based scheme [58], is used to obtain cache lines. Cacheable requests by individual cores in cHT are unordered with respect to each other in so it is the responsibility of each processor to maintain the program order of

Figure 5.2: Architecture of the SunFire X4600 M2



its requests (in order to ensure memory consistency).

5.2.1.2 Topology

Figure 5.2 is a schematic showing the connectivity between the eight nodes. It is referred to as a twisted ladder topology. All nodes have 3 cHT links, in the case of Node 0 and Node 7, two cHT links are used for coherence traffic, while the third link is dedicated to I/O. Using the schematic, four classifications can be made to account for the NUMA nature of the platform,

Level 0 Access to local memory e.g. Core 0A accessing MEM0

Level 1 Access to non-local memory which is one hop away e.g. Core 7B accessing MEM5

Level 2 Access to non-local memory which is two hops away e.g. Core 2A accessing MEM7

Level 3 Access to non-local memory which is three hops away e.g. Core 7A accessing MEM0

5.2.2 Latency and Bandwidth Characteristics of the X4600 M2

Latency and bandwidth measurements for the X4600 M2 are presented in Table 5.1. These were obtained using modified versions of the `lmbench` [184] and `Stream` [177, 178] triad benchmarks.

Latency is measured in Cycles and was obtained using the random pointer chasing benchmark in `lmbench`. Bandwidth is measured in GB/s and was obtained using the `Stream` triad benchmark run using an array size of 8 x 10^6 elements.

Columns in the table correspond to thread locations (i.e. the first core of any given node e.g. Core 0A in Node 0) and the rows correspond to Memory locations (i.e. MEM0 to MEM7) (see Figure 5.2). Each cell in the table is shaded, where the shading indicates thread and

Table 5.1: Latency and Bandwidths for specific memory and thread placement on the SunFire X4600 M2, using `lmbench` and the Stream Triad benchmark. Progressively darker shades of grey are used to indicate accesses that are in the same NUMA level.

		Latency (Cycles)								Bandwidth (GB/sec)							
		Thread Location								Thread Location							
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Memory Location	0	539	534	545	712	602	602	608	734	2.07	1.37	1.34	1.35	1.33	1.33	1.35	1.26
	1	529	519	719	524	590	584	534	602	1.86	2.09	1.35	1.88	1.67	1.65	1.85	1.67
	2	545	596	408	590	474	484	578	596	1.62	1.35	2.50	1.64	1.85	1.87	1.53	1.45
	3	602	534	590	404	474	470	585	602	1.36	1.60	1.65	2.51	1.90	1.84	1.53	1.48
	4	589	596	464	474	404	596	529	596	1.50	1.50	1.83	1.90	2.51	1.65	1.63	1.35
	5	596	584	479	469	584	408	584	539	1.49	1.50	1.86	1.84	1.64	2.51	1.37	1.58
	6	602	529	578	590	524	712	513	534	1.65	1.84	1.65	1.68	1.90	1.28	2.12	1.85
	7	724	602	608	596	712	534	524	529	1.22	1.39	1.26	1.26	1.39	1.26	1.43	2.05

Table 5.2: Summary of Table 5.1 categorized by NUMA level. Units for 'Average Latency' is in Cycles; 'Average Bandwidth' is GB/sec. Standard deviations (σ) for both are also given.

NUMA Level	Average Latency	σ	Average Bandwidth	σ
0	466	64.06	2.30	0.23
1	513	28.64	1.72	0.21
2	605	47.64	1.49	0.15
3	729	7.07	1.24	0.03

memory placements that have the same NUMA level. Progressively darker shades of grey are used to indicate NUMA levels 0 to 3.

Latencies measured for node local access (the diagonal of the Table 5.1) show some variation in access time. This variation arises from the cHT protocol that requires all caches to respond to other processor's coherency messages [56]. As the number of hops increases, so does the latency which peaks at 734 cycles for 3 hops. The ratio between the time to access remote memory and the time to access local memory is termed the NUMA ratio. The maximum NUMA ratio for the X4600 M2 is $\frac{734}{404} = 1.8$.

Similar trends for the bandwidth results are observed, with roughly constant values within a NUMA level and bandwidth decreasing with increasing hop count. The highest measured bandwidth results are obtained for node local access (Nodes 2 – 5: 2.5 Gb/sec), and the lowest for Node 0 accessing MEM7.

Using data given in Table 5.1 the average latency and bandwidth results for each NUMA level are presented in Table 5.2. As the NUMA level increases, average latency increases from 466 Cycle to 729 Cycles (36% variation), with a standard deviation between 1 and 10%.

Table 5.3: Highest and lowest `Stream` bandwidths (GB/s) for a single thread on the SunFire X4600 M2

	Highest Bandwidth	Lowest Bandwidth	Ratio
Copy	2.41	1.10	2.19
Scale	2.63	1.13	2.33
Add	2.75	1.21	2.27
Triad	2.51	1.22	2.06

Average bandwidth also gets worse with NUMA level, decreasing from 2.3 Gb/sec to 1.24 Gb/sec (46% variation), with a standard deviation of between 2 and 10%. These results show that the observed latency and bandwidth measurements can be roughly categorized according to NUMA levels, so the performance of a code executing on the X4600 M2 is expected to be sensitive to thread and memory placement.

For the different components of the `Stream` benchmark, aggregated highest and lowest bandwidths, are presented in Table 5.3. In all cases the highest bandwidths are obtained for NUMA level 0 and the lowest for NUMA level 3. The Add benchmark obtains the best peak of 2.75 Gb/s, while the Copy benchmark gives the lowest peak of 2.41 Gb/s. In terms of the lowest bandwidth results Copy obtained just 1.10 Gb/sec. The ratio between best and worst bandwidth ranges from 2.06 to 2.33. The operation which is most affected by thread and memory placement is Scale.

5.3 Software Environment, Test molecular systems and Modifications to Gaussian

In this section we review the software environment, page placement and migration APIs within Solaris, the test molecular systems that were used, and the code modifications made to Gaussian to enable thread and memory placement. As part of the discussion surrounding code modifications, the memory allocation model used by Gaussian is also discussed.

5.3.1 Software Environment

All experiments performed in this Chapter used the SunFire X4600 M2 system, running the Solaris 10 operating system. The Gaussian code was compiled using Sun Studio 11 compilers, using the same compiler flags as the released code. Cycle counts were obtained using hardware performance counters accessed via the Solaris `libcpc` interface [261].

5.3.2 Page Placement and Migration

The Solaris 10 operating system provides a user-space applications API to affect page placement and a means to affect page migration. The requisite page placement APIs were covered in Chapter 3. Page migration is achieved by using the `madvise` function call and specifying a policy to be used by the kernel for a given range of memory addresses. The function declaration for madvise is `int madvise(caddr_t addr, size_t len, int advice);` where `addr` is the base address of the page in which the data quantity of interest resides, `len` defines the size of the data quantity and `advice` is a set of nine values denoting how the kernel should handle either page references or thread level access to these pages.

There are two flags of interest that can be specified in an madvise call which affect the manner in which page allocations are handled by the kernel i.e. `MADV_ACCESS_LWP` and `MADV_ACCESS_MANY`. The former indicates that the region of memory will be heavily accessed by the next thread that touches the associated range of pages. This leads to pages being migrated and made local to the accessing thread. This process will be referred to later as a 'dynamic page migration strategy'. The latter, `MADV_ACCESS_MANY`, implies many threads will access this region of memory and results in the kernel migrating pages to the various `lgroups` in the system using a pseudo-random mapping [179]. This reduces the likelihood that any given set of pages, which are required by many threads of execution, are all located in one `lgroup`. This process is referred to as 'node interleaving of memory'.

5.3.3 Test Molecular Systems

The four test molecular systems used in this chapter are: 18-Crown-6 Ether [2] , Valinomycin[3] and C60[4]. Two of these (18-Crown-6 Ether and Valinomycin) are used in section 5.4, to assess serial and parallel performance of Gaussian, as a function of thread and memory placement. We note that the geometry for Valinomycin is the same as that for the 'test397' system used in Chapter 4. In section 5.5, four sets of experiments are defined using two molecules (Valinomycin and C60) with the HF and BLYP methods, and the 6-311G* [70, 111] and cc-PVTZ [277] basis sets. These are,

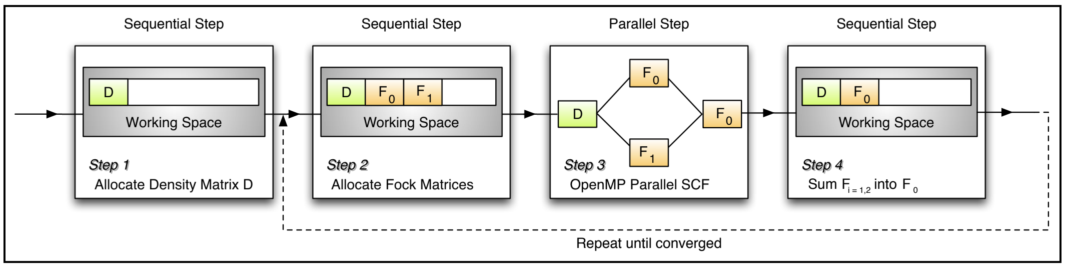
Exp_1 HF/6-311G*, Valinomycin

[2] `http://en.wikipedia.org/wiki/18-Crown-6`. ($C_{12}H_{24}O_6$). This organic compound is a widely used as a phase-transfer catalyst [162].

[3] `http://en.wikipedia.org/wiki/Valinomycin`. Valinomycin is an antibiotic which has selectivity for potassium ions and increases its transport into cell membranes, leading to cell damage in bacteria [142].

[4] `http://en.wikipedia.org/wiki/Fullerene`. C60 or Buckminsterfullerene is a novel form of Carbon which has wide industrial and biochemical applications [71].

Figure 5.3: Memory allocation in Gaussian and subsequent use of the Density and Fock matrices



Exp_2 BLYP/6-311G*, Valinomycin

Exp_3 HF/cc-PVTZ, C60

Exp_4 BLYP/cc-PVTZ, C60

5.3.4 Memory Allocation in Gaussian

Prior to considering modifications made to Gaussian to enable thread and memory placement we discuss the allocation and use of two major data quantities involved in ERI evaluation, the Density and Fock matrices. Figure 5.3 presents a schematic representation of the data quantities and their use at various stages in Gaussian.

The figure has four major steps. In step 1, a single thread of execution prepares data structures required for Fock matrix formation i.e. shell-quartet information, the Density matrix and so on. The Density matrix (D) is allocated, at the start of available memory, within a pre-defined workspace. The size of the Density matrix is dependent on problem size, the basis sets used and calculation type.

In step 2, as many copies of the Fock matrix are created as there are threads assigned to the SCF step i.e. for two threads these are F_0 and F_1. These data items are allocated by a single thread, contiguously after the Density matrix within the workspace. Thus each thread has its own copy of the Fock matrix and associated quantities.

In step 3, all threads are initialized. For HF, each thread reads the shared Density matrix and computes ERIs. In DFT, each thread performs numerical quadrature using batches of grid points. Contributions are made to each thread's private Fock matrix. On completing the evaluation of the integrals or grid points, each thread terminates, leaving its private Fock matrix in memory.

In step 4, thread zero reads in other Fock matrices and sums them into Fock matrix F_0. Once this is done, the data structures from the other threads are freed. The SCF process then repeats until convergence.

5.3.5 Modifications to Gaussian to enable thread and memory placement

The Gaussian code was modified to allow specific thread and memory placement to take place. The following changes were made to enable this,

- Prior to the SCF, unique workspaces are allocated per thread. These workspaces are node local to the thread. This allows private copies of the Fock matrix and other intermediate quantities, to be stored local to each thread.

- The PRISM subroutine computes ERIs and incorporates these into the Fock matrix, for the HF method. For the DFT method, the subroutine PRISMC evaluates the Coulomb ERI contribution to the Kohn-Sham Fock matrix. It has a structure similar to PRISM. The exchange-correlation contribution is computed separately by the CALDFT subroutine using numerical quadrature.

- Three higher-level subroutines PRISMSU, COULSU and CALDSU are used to parallelise PRISM, PRISMC and CALDFT respectively. Within these higher-level subroutines, there is a loop over the total number of threads ($N_{Threads}$) that are to be run in parallel. In this loop, the master thread determines the unique workspace of the other threads, allowing them to read the shared density matrix and accumulate ERIs or numerical quadrature contributions into their own local storage.

- On completion of the parallel region, the master thread sums the Fock matrices computed by other threads into its Fock matrix.

To summarize, each thread requires read access to a shared Density matrix and read/write access to a private Fock matrix. Other data quantities that each thread access are allocated to ensure they are physically local to the executing thread.

5.4 Effects of Thread and Memory Placement in Gaussian

In this section we study the effects of cache blocking and thread, memory placement on the serial and parallel performance of the Gaussian code using the HF method and two molecular systems – 18-Crown-6 Ether with a 6-31G* basis set and Valinomycin with a 3-21G basis set.

Table 5.4: Elapsed time (sec.) for a sequential Gaussian SCF process on a 18-Crown-6 Ether molecule using HF/6-31G*. Timings for the 64Kw and 128Kw (in brackets) cache blocking factors are presented as a function of memory and thread placement on the SunFire X4600 M2. Progressively darker shades of grey are used to indicate similar NUMA levels.

Memory	Thread Location							
Location	0	1	2	3	4	5	6	7
0	177 (224)	174 (251)	175 (254)	177 (262)	177 (263)	178 (264)	177 (262)	182 (280)
1	179 (234)	172 (220)	177 (261)	173 (229)	176 (243)	176 (244)	173 (231)	176 (244)
2	178 (239)	181 (268)	168 (205)	176 (244)	172 (226)	172 (227)	176 (247)	178 (254)
3	187 (270)	174 (236)	176 (244)	168 (205)	172 (226)	172 (227)	176 (249)	177 (251)
4	182 (251)	177 (250)	172 (227)	172 (226)	168 (205)	176 (243)	173 (235)	181 (269)
5	182 (254)	176 (249)	172 (228)	172 (227)	177 (245)	168 (204)	181 (267)	174 (239)
6	181 (246)	173 (231)	176 (243)	176 (242)	173 (228)	177 (266)	172 (219)	174 (231)
7	189 (286)	178 (259)	178 (269)	178 (268)	177 (259)	175 (261)	174 (249)	173 (222)

We use the thread and memory placement APIs to make specific placement of the Density and Fock matrices. We also use the Placement Distribution Model (PDM) to characterize the distribution of contention classes for the X4600 M2.

Timing results for serial Gaussian with the 18-Crown-6 Ether molecule, are varied w.r.t. memory and thread placement in sub-section 5.4.1. This is done in order to assess the effects of cache blocking on the results. In sub-section 5.4.2 we use the PDM to define contention classes for the X4600 M2. Following this, we present timing results for parallel Gaussian using Valinomycin in Section 5.4.3. We then present analysis and parallel speedups for it, following which we conclude this section.

5.4.1 Thread, Memory and Cache Effects for Serial Gaussian

Serial runtimes for a HF calculation on 18-Crown-6 Ether using a 6-31G* basis set are presented as a function of thread and memory location in Table 5.4. Results for two different cache blocking factors (64Kw and 1024Kw), are given.

There are two motivations in the choice of the 64Kw blocking factor used here. First, it corresponds to half of the total cache available to a core on the dual-core 8218 Opteron. Second, in the parametric variation of the blocking factor from 16Kw to 256Kw described in

Chapter 4 it was shown that 64Kw was the optimal blocking factor for the 8218 Opteron. In contrast the 128Kw blocking factor uses all of the cache, giving rise to increased cache misses. For example, when the executing thread was bound to Node 2 and the memory was also located on MEM 2, the total L2 cache misses (and accesses) where 7.88x10^6 (5.14x10^9) and 3.51x10^7 (6.21x10^9) for the 64Kw and 128Kw blocking factors respectively. In other words, the 64Kw blocking factor has a 98.5% L2 cache hit ratio compared to 94.3% L2 cache hit rate for the 128Kw blocking factor. (If alternative thread and memory placement options were used this led to less than a 10% variation in cache miss and access counts, which is much less than the effect of switching from a 64Kw to a 128Kw blocking factor).

We now consider Table 5.4, which gives elapsed times for both cache blocking factors (128Kw is in brackets) as a function of thread and memory placement. Its layout is similar to Table 5.1, with cells again shaded by NUMA level.

The elapsed times in Table 5.4 for the 64Kw blocking factor indicate that most timings within a NUMA group are similar, though some overlaps between NUMA levels does exist. Overall results vary gradually from 168 sec. to 189 sec.

For the 128Kw blocking factor, timings within a given NUMA level vary more, as do timings between NUMA groups which can now range from 204 sec. to 286 sec.

If the physical location of the actual node is taken into account, results from Node 2 – 5 have the lowest times for both blocking factors i.e. this is a direct result of the topology of the X4600M2, where Nodes 2 – 5 have three cHT links per node. In general the 64Kw blocking factor gives results that are 20% faster than for the 128Kw blocking factor.

A summary of the data from Table 5.4 is given in Table 5.5 for both blocking factors. For the 64Kw blocking factor as the NUMA level increases there is a gradual increase in the average time from 171 seconds to 186 seconds, which is an increase of 8%. For the 128Kw blocking factor, the average time increases from 213 seconds to 283 seconds, which is an increase of 33%. The observed timing variations for the 128Kw case correlates with the variation in latency measurements, presented in section 5.2. Not surprisingly the average standard deviations for the 128Kw blocking factor are significantly larger for the 64Kw cases.

Results for both blocking factors indicate that an appropriately cache blocked HF calculation (64Kw), experiences at worst a 8% performance hit, as a function of thread and memory placement. This variation increases markedly if cache blocking is poor.

5.4.2 Contention Classes for the X4600

Using the PDM developed in Chapter 3, the thread and memory placement distributions for the X4600 M2 were obtained.

The X4600 M2 uses dual core Opteron processors. To simplify its topology graph, rather

Table 5.5: Summary of Table 5.4 categorized by NUMA level. Units for 'Average Latency' is in Seconds. Standard deviations (σ) is also given.

NUMA Level	Avg. Time 64Kw	σ	Avg. Time 128Kw	σ
0	171	3.3	213	8.9
1	174	1.9	235	10.3
2	178	2.7	254	10.8
3	186	5.0	283	4.2

Table 5.6: Analysis of thread and memory placement distributions by contention class, using the PDM for the SunFire X4600 M2

Contention Class	Distribution Number	Frequency (%)
1 to <2	2156	0.01
2 to <3	5723830	34.10
3 to <4	9109808	54.30
4 to <5	1762356	10.50
5 to <6	167628	1.00
6 to <7	10976	0.07
7 to <8	448	0.00
≥8	8	0.00

than accounting for each core in a node, a dual core node is represented as one logical node in the PDM.

Thus, assuming there is one thread per node, there are a total of 8*8 possible thread and memory placement options, for eight threads there are a total of 8^8*8! placement options. This assumes memory can be allocated anywhere in the system including all memory allocated on one node.

The PDM results for the X4600 M2 are presented in Table 5.6. Eight contention classes are defined, with each class representing all those thread and memory configurations with contention values between the upper and lower bounds. Given a random thread and memory placement, there is a 54.3% chance of it being allocated in the '3 to <4' contention class, a 34.1% chance of being allocated the '2 to <3' class and so on. There are 2156 distinct placements in the minimum contention class, or a 0.01% chance of selecting this randomly. The chances of being in the maximum contention class is negligible, if placement is random.

5.4.3 Parallel HF Performance in Gaussian

The previous section discussed the distribution of contention classes for the X4600 M2. To obtain an understanding of the range of possible performance outcomes, in this section we perform Gaussian performance experiments using an instance from the minimum and maximum contention classes.

Table 5.7 presents the elapsed times obtained for the first three SCF iterations using the larger Valinomycin system with the HF method and a 3-21G basis set. The results are divided in two categories – 'Co-located Threads and Memory' and 'All Memory Located at Node 0'. The former corresponds to an allocation which occurs in the minimum contention class, the latter corresponds to the maximum contention class. Within each category, sets of experiments are defined. These sets are determined by how the threads are allocated to each node on the X4600 M2. The two options chosen are to use either one core per node (the S* or single option) or both cores per node (the D* or dual-core option), when adding threads. In this respect Set 1 and Set 3 are similar (S*), and Set 2 and Set 4 are similar (D*). Threads are allocated based on the ordering given in the 'Node' column. Thus for a two threaded calculation a 'Node' value of (0,1) applies to the S* case, whereas a value of 0 applies to the D* case. The times taken for both the 64Kw and 128Kw blocking factors are given in seconds in the 'Time' column.

Considering first the results given in Table 5.7 for 'Set 1'. As 'Set 1' corresponds to single node allocation the experiments were run using Node 0 and core 0A. This results in times of 218 sec. and 267 sec. for the 64Kw and 128Kw blocking factors respectively. Increasing the thread count to 2, has threads allocated on Node 0, core 0A and Node 1, core 1A. This gives a reduction in runtime for both blocking factors. For 4 threads Nodes 0, 1, 6 and 7 were used[5]. Timings obtained using the two blocking factors are 58 sec. and 72 sec. For eight threads all 8 nodes were used with times of 31 sec. and 42 sec. for the two cache blocking factors respectively.

For 'Set 3' using a 64Kw blocking factor execution time reduces with increased thread count. For the 128Kw blocking factor there is also a decrease with increased thread count, but scalability is worse than that observed for the 64Kw blocking factor. This is because there are more cache misses, and these are costly with non-local memory placement.

Comparing the 64Kw and 128Kw results for both 'Set 1' and 'Set 3'. For the 64Kw blocking factor, times obtained upto 8 threads, are almost identical between 'Set 1' and 'Set 3'. This indicates that cache blocking is mitigating the effects of non-local memory placement. For the 128Kw blocking factor, as thread count increases, 'Set 3' results become progressively slower than those for 'Set 1'. By 8 threads, the runtime for the 128Kw blocking factor is 50%

[5]For 4 threads these nodes were selected as firstly, Nodes 0 and 7 have 2 cHT links, whereas Nodes 1 and 6 have two cHT links and share a third cHT link. Second, the expectation is that greater variation in runtimes would be obtained for these nodes which have fewer cHT links.

Table 5.7: Elapsed time for the first three SCF iterations, as a function of memory and thread placement for a parallel Gaussian calculation on the Valinomycin molecule using HF/3-21G.

$N_{Threads}$	Co-located Threads and Memory					
	Set 1			Set 2		
	Node	Time (sec.)		Node	Time (sec.)	
	(S*)	64Kw	128Kw	(D*)	64Kw	128Kw
1	0	218	267	–	–	–
2	0,1	111	137	0	109	145
4	0,1,6,7	58	72	0,1	57	76
8	0 – 7	31	42	0,1,6,7	31	38
16	–	–	–	0 – 7	19	27
$N_{Threads}$	All Memory Located at Node 0					
	Set 3			Set 4		
	Node	Time (sec.)		Node	Time (sec.)	
	(S*)	64Kw	128Kw	(D*)	64Kw	128Kw
1	0	218	267	–	–	–
2	0,1	110	147	0	111	146
4	0,1,6,7	59	97	0,1	59	94
8	0 – 7	33	83	0,1,6,7	34	85
16	–	–	–	0 – 7	26	87

S* – Threads are allocated on a single core per node

D* – Threads are allocated per core on a node, prior to using another node

greater than for the equivalent ‘Set 1’ results, with a cache coherency traffic overheads now significantly reducing performance. This shows that when using single cores per node, it is important both to minimise cache misses and use node local thread and memory placement in order to obtain good performance on the SunFire X4600 M2.

For ‘Set 2’, which corresponds to D*, both cores are used and results are reported for 2 – 16 threads. We consider timings for the 64Kw blocking factor. For 2 threads, both cores on node 0 are used. A time of 109 sec. is measured, this is slightly less than the corresponding time in ‘Set 1’. When the thread count is increased to 4 threads and both cores on node 0 and 1 are used, the measured time, is again, slightly less than the corresponding ‘Set 1’ time. On increasing to 8 threads and using nodes 0, 1, 6 and 7 the same difference is seen. For the 128Kw results, when using 2–4 threads there is an increase in times compared to ‘Set 1’ times. For 8 threads, the ‘Set 3’ results is slightly faster than the corresponding ‘Set 1’ time. This suggests that use of both cores in each node reduces the intra-node coherency traffic overheads associated with cHT.

For ‘Set 4’, results for 2 – 8 threads using a 64Kw blocking factor follow the same trends as 64Kw for ‘Set 2’. The times get progressively longer as thread count increases and is 27%

slower than the corresponding 'Set 2' result for 16 threads. Use of a 128Kw blocking factor in 'Set 4' shows a dramatic increase in execution time compared to 64Kw results. This result indicates that with 16 threads use of the 64Kw blocking factor, even with all memory being locate at Node 0, is able to perform better than the 128Kw blocking factor.

For all timing results from 'Set 1' to 'Set 4', it is seen that a well cache blocked algorithm can significantly reduce the effects of poor thread and memory placement.

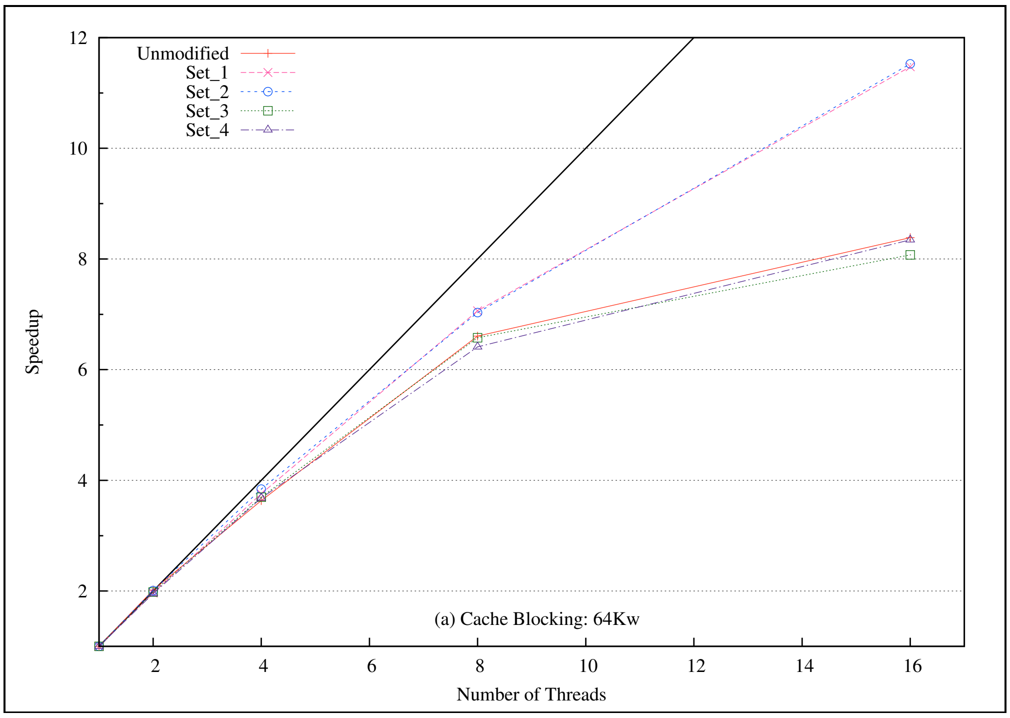
Using data from Table 5.4 five speedup curves are presented in Figure 5.4. The Table 5.4 results are augmented with those obtained from an unmodified version of the Gaussian code (i.e. one that does not perform thread or memory placement). The speedup curves are labelled Unmodified, Set_1, Set_2, Set_3 and Set_4 accordingly. The solid black line in the two plots is a reference line for perfect speedup.

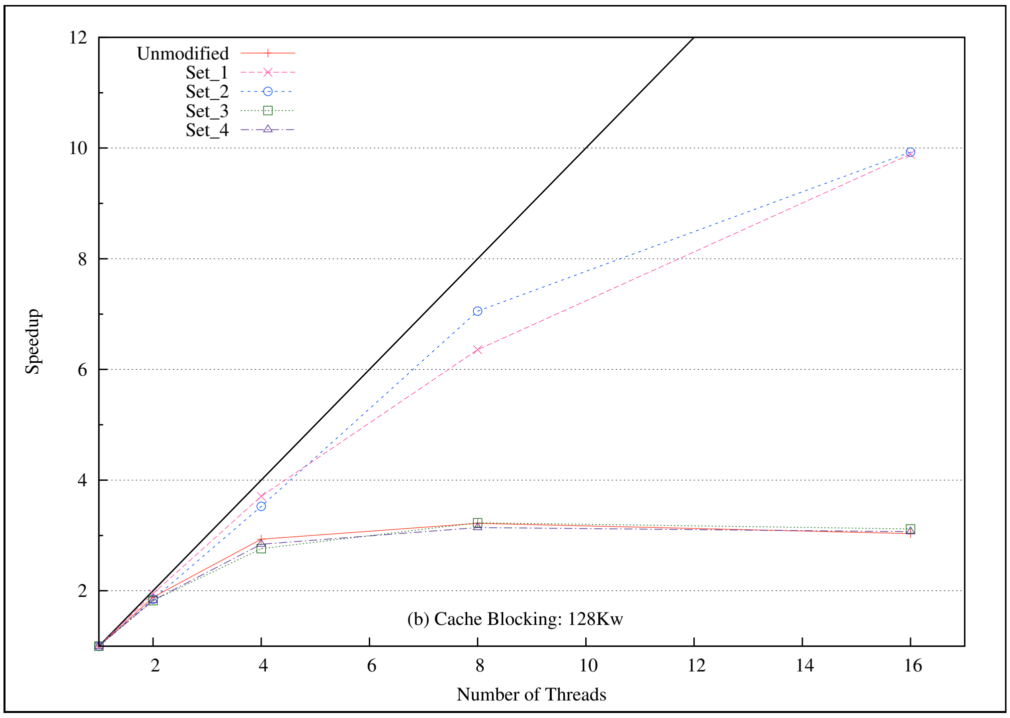
We first consider the 64Kw plot. Upto four threads, there is no significant deviation between the five curves. At eight threads a segregation occurs between Set_1 and Set_2 which are faster than Set_3, Set_4 and unmodified Gaussian. The difference between the two groups is about 10% and is similar to the difference seen between the lowest and highest times recorded for serial 18-Crown-6 Ether in Table 5.4. For sixteen threads, the difference between the two groups increases to around 30%. The best achieved speedup is 11.52 for Set_2. Results obtained for unmodified Gaussian are roughly similar to those obtained using thread and memory placements corresponding to the maximum contention class. This arises because a large block of memory (the workspace in Figure 5.3) is allocated prior to the start of the SCF iterations. Sequential code then touches this memory to create intermediate values and a section of this shared memory is then handed to each worker thread for use in its parallel section. Owing to the first-touch memory placement policy, any memory accessed prior to the parallel section will result in allocation on or near to the master thread.

If we now consider the 128Kw blocking factor. Speedups for two threads are similar. For four threads, Set_1 and Set_2 diverge from Set_3, Set_4 and the unmodified Gaussian by 30%. At eight threads, Set_1 and Set_2 differ by 9%, indicating that the policy of allocating one core per node is better, while sets 1, 2 vary from sets 3, 4 and unmodified Gaussian by 57%. For 16 threads there is a 70% difference between set 1 and 2 and sets 3, 4 and unmodified Gaussian.

These results show that thread and memory placement can produce a speedup of 10 for the co-located case, but a speedup of just 3 for the maximum contention case which is similar to unmodified Gaussian. It is to be noted that even though the maximum speedup is similar for both blocking factors, the base times used are very different; the shortest elapsed time is 19s for sixteen threads with the 64Kw blocking factor versus 29s for the 128Kw blocking factor.

Figure 5.4: Speedup results for unmodified Gaussian 03 code compared to sets 1 – 4 for two cache blocking factors (64Kw, 128Kw). Times obtained are for the first three SCF cycles for using Valinomycin, the HF method and a 3-21G basis set. Note speedups for both cases are relative to different timings, with 64Kw being faster than 128Kw.





5.4.4 Summary: Effects of Thread and Memory Placement on Gaussian Performance

In this section the effects of thread and memory placement were investigated for a HF SCF calculation using G03. The two molecular systems used were 18-Crown-16 Ether with 6-31G* basis set and Valinomycin with a 3-21G basis set.

The combination of an appropriate cache blocking factor and node local thread and memory placement gave the lowest measured elapsed times. This result is similar to that observed in Chapter 3, where a well cache blocked level 3 BLAS routine did not give as large a variation in performance as a level 1 and 2 BLAS routine. Timing and speedup results using Gaussian, showed that an appropriate blocking factor can mitigate improper thread and memory placement to some degree.

For the two instances of thread and memory placement parallel experiments were run for two memory and thread placements which correspond to the minimum and maximum contention classes. Speedup results indicate that appropriate cache blocking and thread, memory placement affects a 10% improvement when using 8 threads, which increases to 30% for 16 threads.

5.5 Extending the LPM to Account for NUMA Effects

Having established that thread and memory placement can have a significant effect on the performance of Gaussian, it is of interest to consider how the LPM, developed in Chapter 4, may be extended for multiple threads and NUMA systems.

This section is structured as follows: section 5.5.1 presents our extension to the LPM. This is evaluated for a single thread in section 5.5.2 using a modified version of Gaussian, which can perform thread and memory placement. Sections 5.5.3 – 5.5.5 then consider the NUMA extended LPM using multiple threads under various scenarios. The section then concludes with a summary of results.

5.5.1 Extending the LPM to incorporate NUMA effects

The LPM from Chapter 4 is defined as,

$$Cycles = \alpha * I_{Count} + \gamma * L_{Misses} \tag{5.1}$$

where α and γ are empirical cost factors determined by fitting measured cycles to the instruction count (I_{Count}) and total cache misses for the highest level of cache (L_{Misses}).

We now define a NUMA domain to be a set of processors and memory, where each processor can access memory within some fixed latency bound. Accessing other NUMA domains will incur a greater latency penalty. A straightforward way of extending the LPM would be to factor in the various NUMA domains that exist in a given system,

$$Cycles = \alpha * I_{Count} \ + \sum_{i=1}^{\substack{NUMA\\Domain}} \gamma_i * L^i_{Misses} \tag{5.2}$$

where L_{Misses} are total cache misses to NUMA domain i.

This leads to two questions (a) how can cache misses be obtained per NUMA level, and (b) does the approximation in Equation 5.2 give reasonable results? These issues are addressed in the following sub-sections.

5.5.1.1 Obtaining Cache misses by NUMA Domain

The original LPM used cache miss information obtained from hardware performance counters. These are event counts that are incremented irrespective of where the data resides in physical memory (i.e. there is no distinction between a cache miss being serviced by local or remote memory). Hardware performance counters on the 8218 Opteron can also measure remote coherency events related to aggregate coherency traffic on the cHT links between the eight nodes of the X4600 M2 [4], but unfortunately there is no means to distinguish between coherency traffic.

There are three possible approaches to address this problem – cache simulation, statistical sampling techniques, and (with some caveats) through the use of specific thread and memory placement experiments.

The first approach uses a simulator to obtain per NUMA domain information as outlined by Tikir and Hollingsworth [280][6]. This is however very expensive. A cheaper approach, which is being pursuing in other work, is to extend the Valgrind binary translation framework used in Chapter 4 to include a NUMA memory model [227].

The second approach is the use of statistical sampling techniques and the hardware performance counter overflow facilities available on most hardware platforms [23, 38, 79]. This involves setting a hardware counter overflow event that once triggered is used to obtain the current program counter, and then the address of the most recent load or store operation. In this respect there are two choices: Buck and Hollingsworth [39] used Data Event Address Registers (EARs) on the Itanium microprocessor with their CacheScope tool to pin-point data structures and code which lead to long latency events (i.e. cache misses and pipeline stalls).

[6]Tikir and Hollingsworth state that simulation fidelity for NUMA systems, is heavily dependent on sufficiently accurate modelling of memory contention between NUMA nodes

The EAR was sampled in order to obtain load instructions which missed the first level of cache or the TLB. (EAR registers are able to store address, instruction and latency of a cache miss). An alternative approach was suggested by Eranian [23], who claims that the AMD Barcelona and newer processors allow the use of precise "Instruction Based Sampling" (IBS) that can determine address location and eviction information for cache misses.

The third approach is to obtain timing information using experiments which consider specific thread and memory placement. This does not allow precise breakdowns of L2 misses but does permit the testing of the basic concepts behind the NUMA extended LPM. This approach is pursued in this work.

5.5.1.2 The NUMA Extended LPM

As discussed in sub-section 5.5.1.1, it is currently difficult to obtain an accurate breakdown of cache misses by NUMA domain. Inspite of this the usefulness of Equation 5.2 can still be assessed by performing specific thread and memory placement experiments.

Consider the time taken by one thread when accessing a non-local data structure a which is located one hop (h_1) away from the thread. This can be written as,

$$T^{h^a} = X_1 + \gamma_{h^a}\, L^a_{Misses} \tag{5.3}$$

where X_1 is the cost of all other terms in the LPM except those associated with cache misses to data quantity a, $\gamma_{h_1^a}$ is the cost penalty associated with accessing a at a distance of one hop, and L^a_{Misses} is the number of cache misses that are associated with accessing a. The time taken for the same computation, when a is local to the thread of execution, i.e. a is 0 hops away (h_0^a), is

$$T^{h_0^a} = X_0 + \gamma_{h_0^a}\, L^a_{Misses} \tag{5.4}$$

If to a first approximation we assume that X_1 and X_0 are equal, and that the number of cache misses associated with accessing a is constant, then the difference between the two timing results is

$$\begin{aligned} \Delta T^{h_1^a} &= T^{h_1^a} - T^{h_0^a} \\ &= \gamma_{h^a}\, L^a_{Misses} - \gamma_{h_0^a} L^a_{Misses} \\ &= (\gamma_{h_1^a} - \gamma_{h_0^a}) L^a_{Misses} \\ &= \Delta\gamma_{h^a} L^a_{Misses} \end{aligned} \tag{5.5}$$

Here $\Delta T^{h_1^a}$ is an estimate of the cost associated with moving a from its local NUMA domain to NUMA domain that is 1 hop away.

As mentioned, this result depends on two assumptions, namely $X_1 \approx X_0$ and that the cache miss count does not change with different placements of a. Both are approximations, since movement of a to a different cache gives a larger effective cache size and will inevitably reduce conflict misses.

Extending this further to multiple data items (a, b, c, ...) which are moved (i, j, k, ...) hops away gives a cost penalty of

$$\begin{aligned} \Delta T^{(h_i^a,\, h_j^b,\, h_k^c,\, \dots)} \quad = & \quad \Delta\gamma_{h_i^a} L^a_{Misses} \\ & + \Delta\gamma_{h_j^b} L^b_{Misses} \\ & + \Delta\gamma_{h_k^c} L^c_{Misses} \\ & + \dots \end{aligned} \tag{5.6}$$

The implication of Equation 5.6 is that each value on the R.H.S can be determined individually and used to predict the overall effect of moving multiple data items. Later, we assess this assumption by using explicit data placement of the Density and Fock matrices in the Gaussian code.

At this point the NUMA extended LPM does not account for multi-threaded execution. One possible way of extending Equation 5.1 is to divide the instruction count (I_{Count}) and total cache misses (L_{Misses}) by the total number of threads. This would assumes perfect parallelization and that there are no changes in the total number of instructions executed and cache misses in the parallel code. This is not true. In the first instance the total instruction count can increase if there is duplicate instructions. Second the cache misses may change; the number of cold misses would increase due to their being multiple caches, although the combined cache size from using multiple cores should reduce the number of capacity misses. These effects can be partially accounted for by only considering parallel portions of application code and instead of using the number of threads to divide the instruction and L2 miss counts, through the use of an empirically determined parameter, f_n, defined as

$$f_n = \frac{T_1^{(h_0^a,\, h_0^b,\, h_0^c,\, \dots)}}{T_n^{(h_0^a,\, h_0^b,\, h_0^c,\, \dots)}} \tag{5.7}$$

where the numerator and the denominator are the times taken for 1 and n threads respectively where all data quantities are allocated local to each thread.

Values of f_n less than n, indicate replicated computation in the parallel region, communication overheads, or a load imbalance between the threads. A value of f_n greater than n indicates some superlinear effect. Using f_n the NUMA extended LPM for multi-threaded cases should conform with,

$$T_n^{(h_i^a,\, h_j^b,\, h_k^c,\, \dots)} = \frac{T^{(h_0^a,\, h_0^b,\, h_0^c,\, \dots)} + [\Delta\gamma_{h_i^a} L^a_{Misses} + \Delta\gamma_{h_j^b} L^b_{Misses} + \Delta\gamma_{h_k^c} L^c_{Misses} + \dots]}{f_n} \tag{5.8}$$

where, $T^{(h_0^a,\, h_0^b,\, h_0^c,\, \dots)}$ is the time taken using a single thread and with all data being local to the thread; and $\Delta\gamma_h L_{Misses}$ are timings measured in a single threaded calculation where individual data items are migrated separately to remote nodes.

In the sections that follow, we test the validity of Equation 5.8 under a number of different scenarios.

5.5.2 Single Threaded Placement Experiments

As discussed above, to a first approximation the penalty costs in the extended LPM that are associated with moving individual data items to different NUMA domains can be treated separately according to Equation 5.6 In this section we explore this issue by carrying out single thread placement experiments with varying number of hops for two data items, the Density and Fock matrices.

Table 5.8 present measured and modelled execution times for a variety of Gaussian single threaded calculations with specific thread and memory placement. The table presents results for the PRISM, PRISMC, CALDFT routines obtained from experiments Exp_1 (Valinomycin using the HF method and a 6-311G* basis set), Exp_2 (Valinomycin using the BLYP method and a 6-311G* basis set), Exp_3 (C60 using the HF method and a cc-PVTZ basis set), and Exp_4 (C60 using the BLYP method and a cc-PVTZ basis set). For each Exp there are two groups of results: 'Basic Experiments' and 'Modelling Predictions'. The basic experiments give the times taken in seconds by the PRISM, PRISMC and CALDFT routines when the location of either the Fock or Density matrix, but not both, is varied by a certain number of hops from the home NUMA domain. The number of hops is given in the columns 'h_D' and 'h_F' for the Density and Fock matrix respectively. The data given under 'Modelling Prediction' are for cases where the number of hops for both the Density and Fock matrices are varied. Times for these cases are both measured and predicted. We now consider the 'Basic Experiments' section of Table 5.8, for Exp_1 and Exp_2. Measured times for each placement are given in the $T_{Measured}$ column. The difference in time between the all local case (h_D = 0, h_F = 0) and other basic experiments is given in column ΔT_1. For example, the measured time for the single threaded, node-local placement for 'PRISM from Exp_1' is $(h_D{=}0, h_F{=}0)$ = 1884 sec. The difference between this value and a placement where the Fock matrix is one hop away is 39 s.

Table 5.8: Measured and modelled results in seconds for single-thread calculations for Exp_1 – 4 obtained for first SCF cycle, on the SunFire X4600 M2 system.

Exp_1 (Valinomycin – HF/6-311G*) and Exp_2 (Valinomycin – BLYP/6-311G*)										
		PRISM from Exp_1			PRISMC from Exp_2			CALDFT from Exp_2		
h_D	h_F	$T_{Measured}$	$T_{Modelling}$	ΔT_1	$T_{Measured}$	$T_{Modelling}$	ΔT_1	$T_{Measured}$	$T_{Modelling}$	ΔT_1
Basic Experiments										
0	0	1884	–	0	584	–	0	969	–	0
0	1	1923	–	39	593	–	9	1040	–	71
0	2	1961	–	77	602	–	17	1082	–	113
1	0	1932	–	48	588	–	4	973	–	4
2	0	2001	–	117	593	–	8	979	–	10
Modelling Predictions										
h_D	h_F	$T_{Measured}$	$T_{Modelling}$	Err%	$T_{Measured}$	$T_{Modelling}$	Err%	$T_{Measured}$	$T_{Modelling}$	Err%
1	1	1970	1971	0.06	598	597	-0.07	1044	1044	-0.07
1	2	2008	2009	0.04	606	606	-0.02	1087	1086	-0.08
2	1	2033	2040	0.33	601	602	0.03	1050	1049	-0.02
2	2	2072	2078	0.28	611	610	-0.23	1095	1092	-0.27

Exp_3 (C60 – HF/cc-PVTZ) and Exp_4 (C60 – BLYP/cc-PVTZ)										
		PRISM from Exp_3			PRISMC from Exp_4			CALDFT from Exp_4		
h_D	h_F	$T_{Measured}$	$T_{Modelling}$	ΔT_1	$T_{Measured}$	$T_{Modelling}$	ΔT_1	$T_{Measured}$	$T_{Modelling}$	ΔT_1
Basic Experiments										
0	0	1073	–	0	488	–	0	471	–	0
0	1	1091	–	18	493	–	6	506	–	35
0	2	1114	–	41	498	–	11	526	–	55
1	0	1095	–	22	489	–	1	472	–	2
2	0	1128	–	55	491	–	3	475	–	4
Modelling Predictions										
h_D	h_F	$T_{Measured}$	$T_{Modelling}$	Err%	$T_{Measured}$	$T_{Modelling}$	Err%	$T_{Measured}$	$T_{Modelling}$	Err%
1	1	1118	1113	-0.42	495	495	-0.04	506	507	0.20
1	2	1133	1136	0.29	500	498	-0.28	528	528	-0.06
2	1	1146	1146	-0.02	496	496	-0.04	510	509	-0.04
2	2	1167	1169	0.14	501	501	0.00	528	530	0.36

The values under ΔT_1 for Exp_1 are therefore those given by Equation 5.5 and as follows:

$$\begin{aligned}
\Delta\gamma_{h_1^F} L^F_{Misses} &= \Delta T^{h^F} = T_1^{h_1^F} - T^{h_0^F} = 39 \text{ sec.} \\
\Delta\gamma_{h_2^F} L^F_{Misses} &= \Delta T^{h^F} = T^{h_1^F} - T^{h_0^F} = 77 \text{ sec.} \\
\Delta\gamma_{h_1^D} L^D_{Misses} &= \Delta T^{h^D} = T^{h_1^D} - T^{h_0^D} = 48 \text{ sec.} \\
\Delta\gamma_{h_2^D} L^D_{Misses} &= \Delta T^{h^D} = T^{h_1^D} - T^{h_0^D} = 117 \text{ sec.}
\end{aligned} \tag{5.9}$$

These basic placement experiments show that for PRISM, the placement of the Density matrix has greater influence than the Fock matrix, but for PRISM and CALDFT the opposite is true.

Using the results obtained from the 'Basic Experiments', it is possible to create modelling predictions for varying the number of hops for the Fock and Density matrices. As an example, the predicted timing for positioning the Fock and Density matrices one hop away i.e. $(h_D{=}1, h_F{=}1)$ for 'PRISM from Exp_1' is,

$$\begin{aligned}
T^{(h_1^D, h_1^F)} &= T^{(h_0^D, h_0^F)} + \Delta T^{h_1^D} + \Delta T^{h_1^F} \\
&= 1884 + 39 + 48 \\
&= 1971
\end{aligned} \tag{5.10}$$

While the measured value for $T^{(h_1^D, h_1^F)}$ corresponding to 'Prism from Exp_1' is 1970 sec.

If we consider modelling prediction errors, we find that all results are in error by 1% at most. This indicates the simple NUMA extension to the LPM can effectively model single threaded HF and DFT calculations.

In the second part of Table 5.8, corresponding to Exp_3 and Exp_4, similar trends are observed as for Exp_1 and Exp_2.

Thus a simple extension to the LPM gives reasonable results for single threaded Gaussian calculations using the HF and DFT methods.

5.5.3 Multi-Threaded, Single-Core Placement Experiments

The multi-threaded placement penalties defined by Equation 5.8, requires a scaling factor f_n to be determined. These values are obtained using specific thread and memory placement timing experiments, where the requisite data quantities are local to all the threads for both serial and parallel computation. This required a modified version of Gaussian, where each thread uses a node-local copy of both the Density and Fock matrix. (The Density matrix is normally shared by all threads).

Results for these timing experiments are given in Table 5.9. This table gives the average speedup 'S_n' and its standard deviation (σ) for calculations with 2, 4 and 8 threads. Results

Table 5.9: The average Speedup S$_n$ and its standard deviation (σ) for n-thread calculations with ideal local access to both Density and Fock matrices/blocks.

n-thread		PRISM from Exp_1	PRISMC from Exp_2	CALDFT from Exp_2	PRISM from Exp_3	PRISMC of Exp_4	CALDFT of Exp_4
n = 2	S_2	1.941	1.848	2.004	1.979	1.906	2.022
	σ	0.001	0.001	0.001	0.001	0.006	0.220
n = 4	S_4	3.863	3.367	4.008	3.929	3.700	3.979
	σ	0.022	0.010	0.009	0.005	0.027	0.018
n = 8	S_8	7.548	6.088	7.477	7.728	7.176	7.799
	σ	0.002	0.073	0.030	0.009	0.079	0.018

for PRISM, PRISMC and CALDFT for Exp_1 – 4 are given. The results were obtained for computations run with Nodes 2, 3, 4 and 5 (cf. Figure 5.2) of the SunFire X4600 M2 and with a single core per node. For the 8 thread results, all 8 nodes were used, but jobs were executed twice. The first time this was done using threads 1 – 4 running on Nodes 2 – 5. The second time, threads 5 – 8 were run on Nodes 2 – 5. Speedups were obtained by measuring the time taken for each thread. By default, dynamic load balancing of work is used in PRISM, PRISMC and CALDFT. For the purpose of this work a static load balancing scheme was used in order to avoid skewing f_n due to work shifting between threads.

For the PRISM and PRISMC subroutines which compute ERIs, the speedups given in Table 5.9 are less than n. This is a result of replicated work being done in these routines, e.g. all threads need to calculate ERI quantities relating to pairs of basis functions. The standard deviation (σ) is small indicating that static load balancing is good. The CALDFT routine in some cases gives a slight superlinear speedup, and for the 2 thread case and Exp_2 there is a large standard deviation. The latter indicates that static load balancing is poor and this was confirmed to be the case. (Note load balancing had been deliberately disabled as noted previously).

5.5.3.1 Two Threads, Single Core Thread Assignment

Using the values of 'f_n' presented in Table 5.9 we can now derive execution times for multi-threaded Gaussian calculations, with specific thread and memory placement by using Equation 5.8. In this section we consider the case where only a single core is used at each node.

We consider first the case of using two threads executing on Nodes 2 and 5 with 1 thread per node. Timing predictions for various threads are given in Table 5.10.

The table is divided into two sections corresponding to thread 1 and thread 2. Variations in hops are in columns h_D and h_F. The measured execution time for each thread is given for the 4 experiments and different routines. In addition, the percentage error that the extended LPM gave is reported. In each case if there were multiple ways of performing a placement

Table 5.10: Modelling error in percent for 2-thread calculations performed at each NUMA level using f_n, for single core thread assignment

			Exp_1		Exp_2				Exp_3		Exp_4			
			PRISM		PRISMC		CALDFT		PRISM		PRISMC		CALDFT	
Thread	h_D	h_F	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$
1	0	0	973	-0.2	316	-0.1	485	-0.2	543	-0.2	257	-0.2	255	-8.7
	0	1	993	-0.3	324	-1.0	520	-0.1	552	-0.2	261	-0.9	273	-8.6
	0	2	1016	-0.6	332	-1.9	544	-0.8	563	0.2	265	-1.5	286	-9.0
2	1	0	1001	-0.5	320	-0.6	482	0.7	557	-0.6	257	-0.2	220	6.2
	2	0	1042	-1.0	324	-1.0	484	1.9	573	-0.5	258	-0.4	221	6.3
	1	1	1020	-0.5	328	-1.6	516	0.9	565	-0.4	261	-0.7	234	7.1
	2	1	1058	-0.7	331	-1.8	517	1.4	582	-0.5	262	-0.8	234	7.6
	1	2	1045	-0.9	336	-2.3	538	0.8	575	-0.2	265	-1.4	244	7.0
	2	2	1087	-1.5	342	-3.4	546	-0.2	595	-0.7	268	-1.7	247	6.1

experiment, then each of these was timed and the lowest value reported.

In all experiments thread 1 was bound to Node 2 and the shared Density matrix was placed in MEM2. Thus the values for h_D are always zero for thread 1. The Fock matrices for the two threads were allocated on any of the following Nodes: 2, 3, 4, 5. For thread 1 the Fock matrix is varied to be 1 or 2 hops away by allocating it on Nodes 5 or (3 or 4) respectively. This corresponds to entries (0,1) and (0,2) for thread 1. For thread 2, the Density matrix can only be either 1 or 2 hops away depending if thread 2 is running on nodes 3, 4 or 5. This gives rise to six possible entries in Table 5.10.

Examining the errors across all the experiments we find that apart from CALDFT in Exp_4, which is known to have load balancing issues, the maximum errors obtained by use of the extended LPM and f_n is less than 2%.

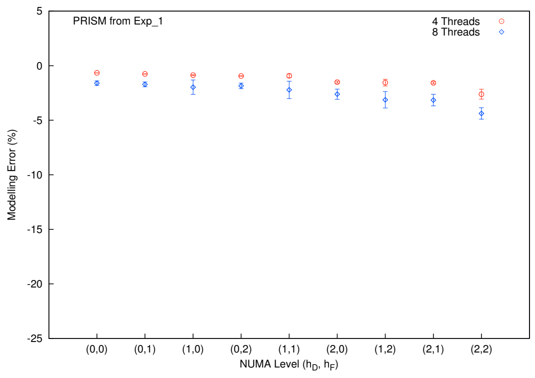
5.5.3.2 Four and Eight Threads, Single Core Thread Assignment

In this sub-section we extend the 2 thread experiments detailed above to 4 and 8 threads but still use single core thread assignment.

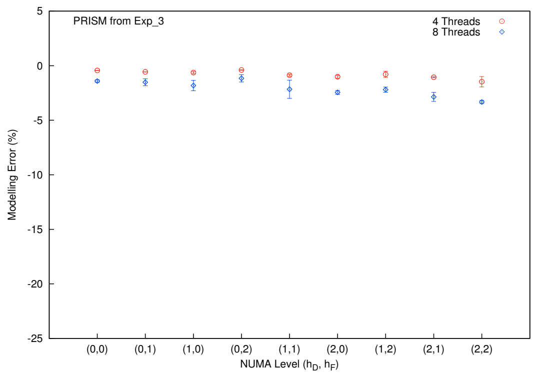
This time the modelling errors are presented graphically in Figure 5.5. This figure is composed of 6 plots, with results for PRISM, PRISMC and CALDFT ordered by row. The figures are labelled (a) to (f), where each corresponds to the following: (a) PRISM from Exp_1; (b) PRISMC from Exp_2; (c) CALDFT from Exp_2; (d) PRISM from Exp_3; (e) PRISMC from Exp_4 and (f) CALDFT from Exp_4.

Each sub-plot in Figure 5.5 has error bars to indicate the difference in thread timings as a result of load imbalance between threads. The x-axis is the number of hops for the Density and Fock matrices (h_D, h_F) and the y-axis denotes modelling error (expressed as a percentage). The values of (h_D, h_F) along the x-axis are arranged according to increasing execution time.

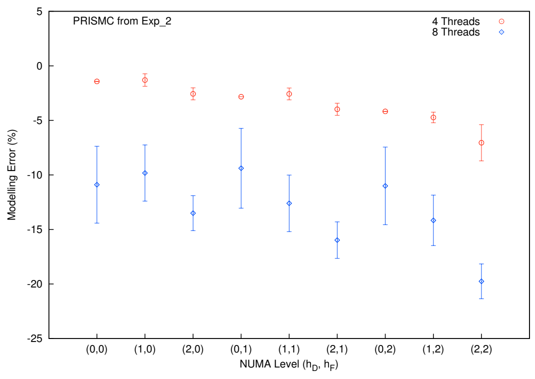
Figure 5.5: Modelling error for 4, 8 thread calculations at each NUMA level corresponding to: (a) PRISM from Exp_1; (b) PRISMC from Exp_2; (c) CALDFT from Exp_2; (d) PRISM from Exp_3; (e) PRISMC from Exp_4 and (f) CALDFT from Exp_4



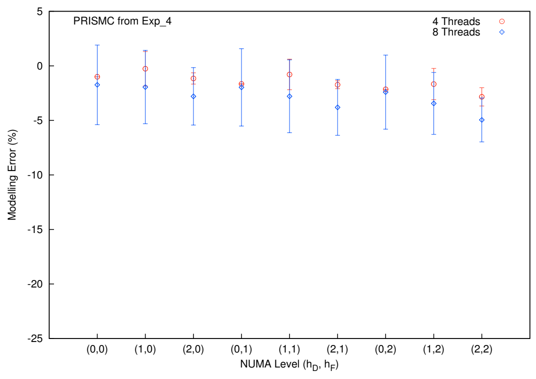
(a)



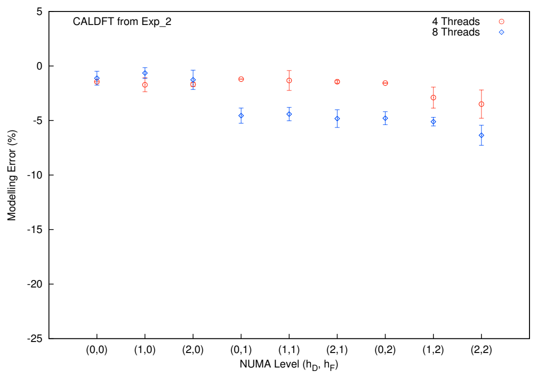
(d)



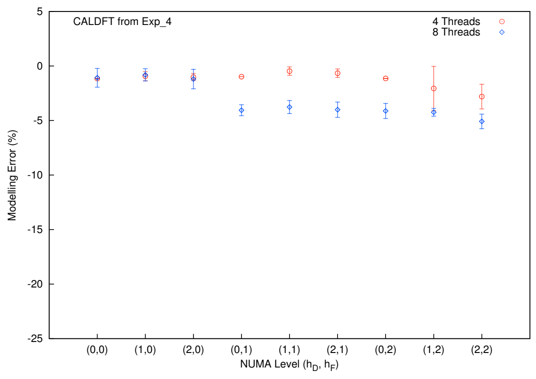
(b)



(e)



(c)



(f)

Table 5.11: Modelling error in percent for 2 thread calculations using f_n and dual-core thread assignment

			Exp_1		Exp_2				Exp_3		Exp_4			
			PRISM		PRISMC		CALDFT		PRISM		PRISMC		CALDFT	
Thread	h_D	h_F	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$	$T_{Measured}$	$\frac{Err}{\%}$
1	0	0	981	-1.0	320	-1.3	320	-3.1	545	-0.5	258	-0.7	261	-8.9
	0	1	1004	-1.4	329	-2.3	329	-2.9	555	-0.7	263	-1.4	280	-8.8
	0	2	1030	-1.9	340	-4.3	340	-5.7	568	-0.9	268	-2.4	299	-11.1
2	0	0	982	-1.5	321	-1.6	321	-2.1	546	-0.7	257	-0.5	226	5.2
	0	1	1002	-1.2	327	-1.9	327	-0.5	554	-0.5	261	-0.9	238	7.3
	0	2	1024	-1.4	336	-3.0	336	-1.9	564	-0.3	266	-1.5	250	6.2

Three common characteristics are evident in the plots: first, the majority of predicted differences are negative. This is due to interconnect contention not being explicitly included in Equation 5.8. Second, the modelling error is much less for 4-threads than for 8-threads. This reflects the fact that interconnect contention increases with increased threads. Third, the modelling errors increase slightly with larger NUMA levels. This is to be expected as the model attempts to predict performance at greater hop counts using execution times obtained from those of lower hop counts.

Overall the majority of predictions are able to reproduce elapsed times to within 5%. The largest differences are seen for the PRISMC 8 thread results from Exp_2 indicating that interconnect contention is limiting parallel performance.

5.5.4 Multi-Threaded, Dual-Core Placement Experiments

In the previous sub-section new threads were allocated to different nodes. In this section we consider the effects of dual-core thread assignment on measured performance. The use of both cores on the 8218 Opteron will invariably introduce contention between both threads for available bandwidth on the three cHT links.

Table 5.11 presents the results for the four test systems, using two threads. The layout of the table is similar to Table 5.10. As both threads are resident on the same node, both threads have node local access to the Density matrix which is 0 hops away. This means the Fock matrix can only be 0, 1 or 2 hops away from the two threads. This results in 3 entries per thread in the Table 5.10. The threads were bound to core 2A and core 2B on node 2.

As expected the measured times for each thread are similar in all cases except for CALDFT of Exp_4 where load imbalance is large. Execution time increases as the number of hops for the Fock matrix is increased. Modelling errors, apart from CALDFT in Exp_4, are less than 5%.

In comparison with the single core per node results, given in Table 5.10, using two cores per

Table 5.12: Average modelling error in percent for dual-core and single core thread assignment. S_n is used for single core cases and f_n^{DC} is used in dual-core cases.

	Exp_1		Exp_2				Exp_3		Exp_4			
	PRISM		PRISMC		CALDFT		PRISM		PRISMC		CALDFT	
	S_n	f_n^{DC}	S_n	f_n^{DC}	S_n	f_n^{DC}	S_n	f_n^{DC}	S_n	f_n^{DC}	S_n	f_n^{DC}
Single-Core												
2-thread	-0.7	–	-1.5	–	0.4	–	-0.4	–	-0.9	–	1.6	–
4-thread	-1.3	–	-3.4	–	-1.9	–	-0.9	–	-1.4	–	-1.2	–
8-thread	-2.5	–	-12.8	–	-3.7	–	-2.1	–	-2.7	–	-3.1	–
Dual-Core												
2-thread	-1.3	-0.3	-2.4	-0.4	-2.7	0.3	-0.6	0.4	-1.2	0.8	-3.7	-0.7
4-thread	-1.9	-0.9	-5.2	-3.3	-4.4	-1.5	-1.4	-0.4	-1.6	0.4	-3.3	-0.3
8-thread	-3.6	-2.6	-17.0	-15.3	-8.0	-5.2	-3.1	-2.1	-3.7	-1.7	-7.4	-4.6

node gives rise to slight increases in execution times and slightly larger modelling errors. This is a consequence of intra-core contention which is not accounted for in the NUMA extended LPM.

5.5.4.1 Four and Eight Threads, Dual Core Assignment

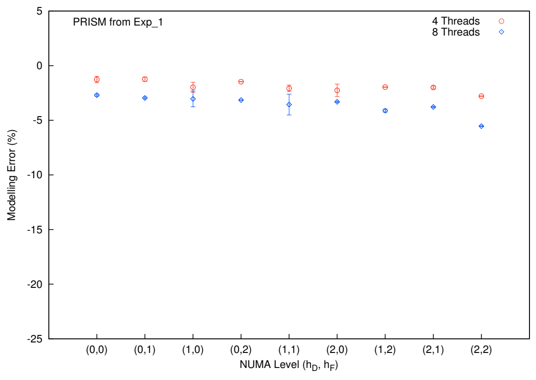
Figure 5.6 presents modelling error plots, obtained for four and eight threads with dual core assignment. The results show similar trends to those seen in Figure 5.5 i.e. the majority of prediction errors are negative and the absolute error increases from four threads to eight threads, and from lower to higher NUMA levels (i.e. across the x-axis).

The modelling errors for both PRISM cases are less than 5%. For the PRISMC and CALDFT subroutines the modelling errors are less than 15% except for PRISMC from Exp_2 were the errors are greater than 15%. As the extended LPM does not explicitly model interconnect contention, the errors obtained for PRISMC and CALDFT would suggest that intra-core contention and the cHT protocol overheads are limiting the observed performance.

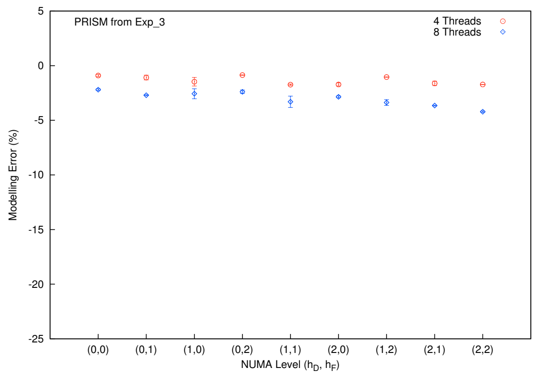
5.5.5 Accounting for Dual-Core contention

Results presented in the previous section show that modelling errors systematically increase for cases involving dual-core thread assignment (cf. Figure 5.5 with 5.6). One possible way to account for dual-core contention is to introduce a dual-core scaling factor S_{DC}, which is a ratio between dual-core and single-core thread placement results obtained for PRISM, PRISMC and CALDFT i.e we use (0,0) values from Table 5.8. Values for S_{DC} are 0.99, 0.98, 0.97 for PRISM, PRISMC and CALDFT respectively. These results indicate that CALDFT is most affected by intra-core contention.

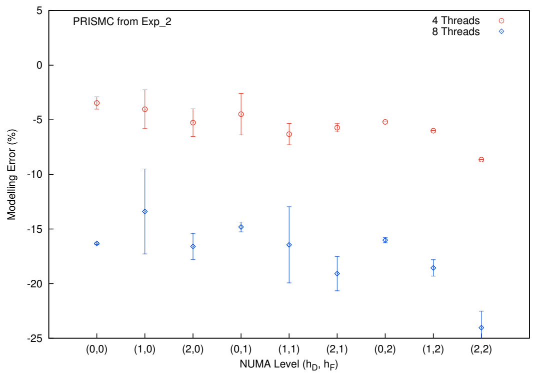
Figure 5.6: Modelling Error for 4, 8 thread calculations using f_n at each NUMA level: (a) PRISM from Exp_1; (b) PRISMC from Exp_2; (c) CALDFT from Exp_2; (d) PRISM from Exp_3; (e) PRISMC from Exp_4 and (f) CALDFT from Exp_4



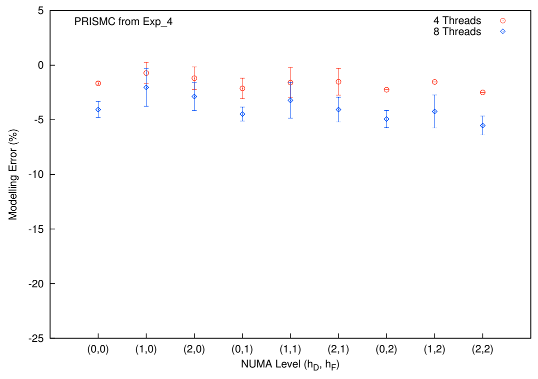
(a)



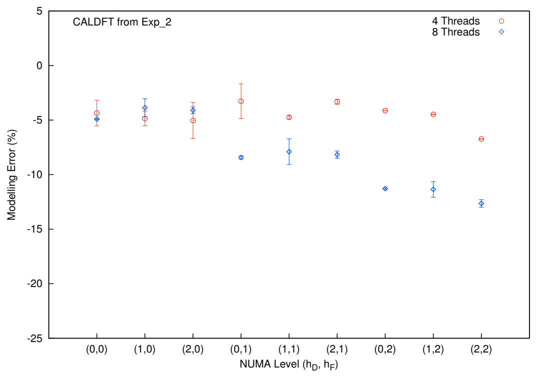
(d)



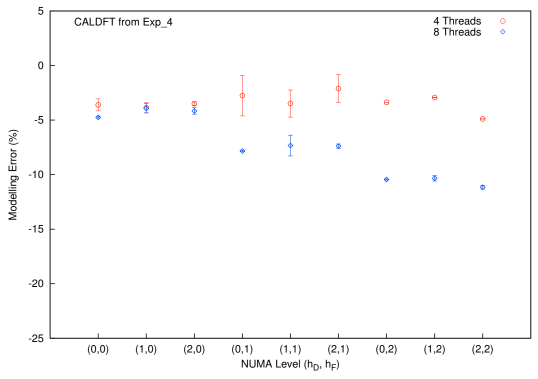
(b)



(e)



(c)



(f)

If we assume that S_{DC} is specific to the system environment (i.e the number of cores and arrangement of on-chip caches, the specific routing and number of threads) and not on the size of the problem, a dual-core scaling factor f_n^{DC} can be defined as,

$$f_n^{DC} = S_n * S_{DC} \tag{5.11}$$

where, S_n is the same scaling factor used previously in Table 5.9.

Using the dual-core scaling factor, the average modelling prediction error for dual core and single core thread assignments are given in Table 5.12 for two, four and eight threads. From this table it can be seen that the use of f_n^{DC}, gives a systematic error reduction for all of the dual-core cases.

5.5.6 Summary: NUMA and Multi-threaded Extended LPM

In this section, the LPM was extended to account for NUMA locality by use of an additive cost model for each data item which was not local to a given thread of execution.

A series of base timings were determined by varying the number of hops for both the Density and Fock matrices for two molecular systems (Valinomycin using a 6-311G* basis set and C60 using a cc-PVTZ basis set) and the HF and BLYP methods. The extended LPM was able to reproduce single threaded runtimes to within 1%.

Modelling errors not explicitly accounted for by the extended LPM includes interconnect contention which increases with thread count. A possible modification to account for contention would be to incorporate, a variable scale factor 'f_n' into the model as the function of the number of threads.

For dual-core the use of a variable scaling factor f_n^{DC} is able to reduce modelling prediction errors.

5.6 Use of Page Migration to Affect Data Locality

In previous sections, thread and memory placement and cache blocking were shown to influence strongly the performance of the Gaussian code on the SunFire X4600 M2. The aim of this section is to evaluate the use of application initiated page migration, as a means of improving data locality.

To achieve this aim, three types of data placement schemes are identified. These are then used to affect the placement of the Density and Fock matrix. These placements are then evaluated using the Valinomycin and C60 benchmarks for the HF, BLYP and B3LYP methods.

Table 5.13: Three data placement schemes for the Fock and Density Matrices

Placement	Density Matrix (D)	Fock Matrix (F)
$\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{1-Node}}$	Placed on one node	Placed on one node
$\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{Dynamic}}$	Placed on one node	Node local
$\mathbf{D}_{\text{All-Nodes}} - \mathbf{F}_{\text{Dynamic}}$	Interleaved over all nodes	Node local

5.6.1 Data Placement Strategies

Prior to considering data placement schemes, we briefly recap the two page management techniques used to improve data locality in Solaris. The first is the ability of a user-space application to affect page mappings within its address space by using the madvice call for the range of pages requiring migration. This is a ‘dynamic page migration strategy’. The second strategy involves reducing contention for any given set of pages by re-mapping these pages using a pseudo-random mapping of node locations in order to ensure that there are no access hot spots. This is referred to as a ‘node interleaving of memory’.

Using these two strategies, three placement schemes are outlined in Table 5.13 and illustrated in Figure 5.7, for the Density (D) and Fock (F) matrix placements. These are elaborated below

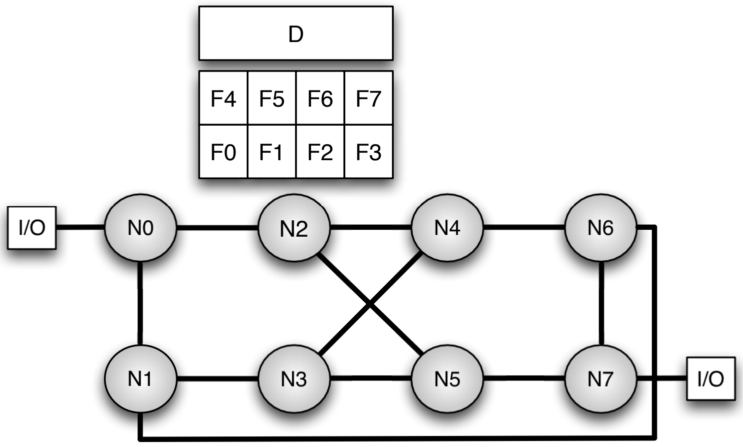
- The $\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{1-Node}}$ scheme places the Fock matrix and density matrix on one node. It resembles the placement behaviour of an unmodified instance of Gaussian, which arises from the first-touch policy used by the operating system. This placement corresponds to Figure 5.7, (a).

- The $\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{Dynamic}}$ scheme places the Density matrix on one node, but through user initiated page migration, makes the Fock matrix node local to each thread in the parallel region. This placement corresponds to Figure 5.7, (b).

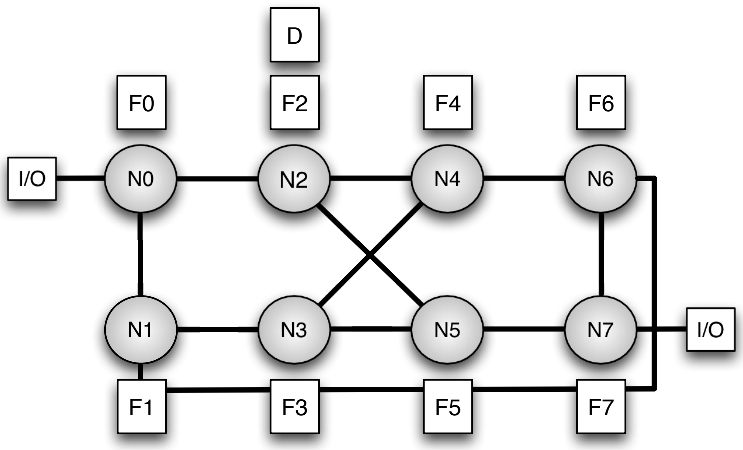
- The $\mathbf{D}_{\text{All-Nodes}} - \mathbf{F}_{\text{Dynamic}}$ scheme uses node interleaving of memory so that pages of the Density matrix are spread throughout all the nodes in the system. Again the Fock matrix is migrated to be local to each parallel thread. This placement corresponds to Figure 5.7, (c).

The $\mathbf{D}_{\text{All-Nodes}} - \mathbf{F}_{\text{Dynamic}}$ scheme is used to asses the effectiveness of interleaving the Density matrix, whereas the $\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{Dynamic}}$ scheme is used to assess the influence of dynamic Fock matrix migration on parallel performance.

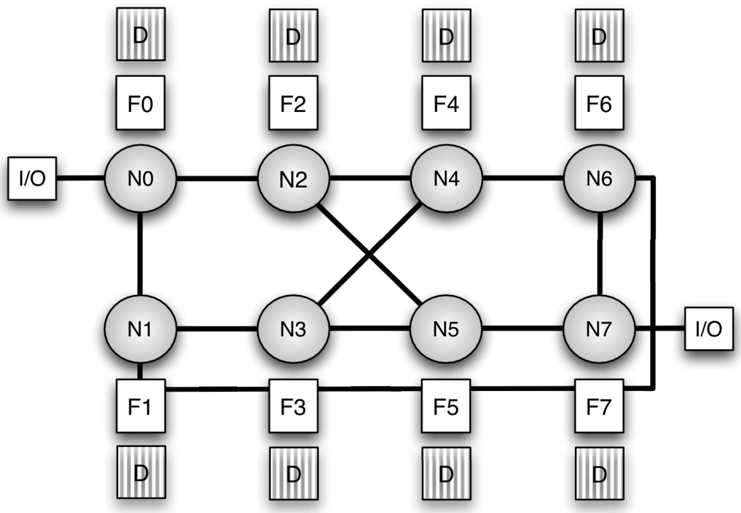
Figure 5.7: Node Mappings used for the Density and Fock matrices



(a) Shared Density matrix (D) and
Fock matrices (F{0-8}) on one node
i.e. $\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{1-Node}}$



(b) Shared Density matrix (D) and
Fock matrices (F{0-8}) local to each node
i.e. $\mathbf{D}_{\text{1-Node}} - \mathbf{F}_{\text{Dynamic}}$



(c) Interleaved Density matrix (D) and
Fock matrices (F{0-8}) local to each node
i.e. $\mathbf{D}_{\text{All-Nodes}} - \mathbf{F}_{\text{Dynamic}}$

5.6.2 Speedup Plots for Valinomycin and C60

By using the placement schemes previously mentioned, we perform a series of timing experiments for the Valinomycin and C60 systems. The objective of this is to assess HF and DFT performance as a function of placement of the Density and Fock matrices, for increasing thread count.

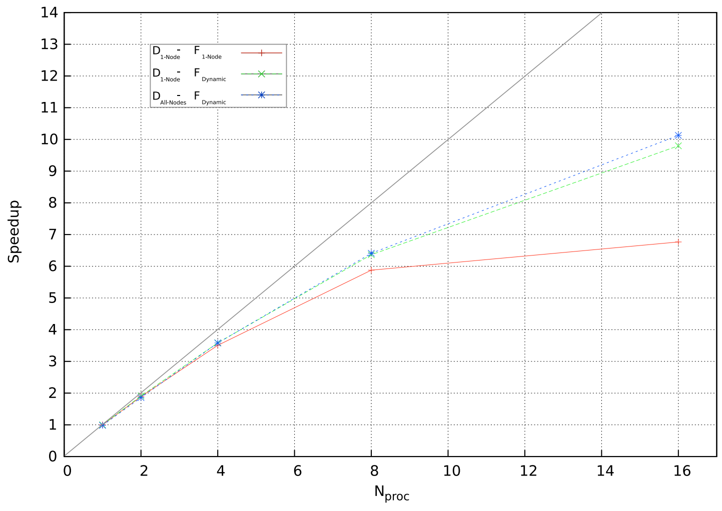
Figures 5.8 and 5.9 refer to Valinomycin and C60 respectively. The figures have three sub-plots, corresponding to the HF, BLYP and B3LYP methods. Timings are obtained for one SCF iteration in both cases. This time comprises the time taken for the parallel formation of the Fock matrix and its serial diagonalization. The base time which is use to compute speed up corresponds to a single threaded calculation.

There are three curves for each sub-plot, which we refer to as curves 1, 2 and 3. Curve 1, the red curve, corresponds to the $\mathbf{D}_{\text{1-Node}}$ – $\mathbf{F}_{\text{1-Node}}$ placement. This curve indicates the behaviour of unmodified Gaussian. Curve 2, the green curve, corresponds to the $\mathbf{D}_{\text{1-Node}}$ – $\mathbf{F}_{\text{Dynamic}}$ placement. This curve shows the influence of local Fock matrix placement on performance. Curve 3, the blue curve, corresponds to the $\mathbf{D}_{\text{All-Nodes}}$ – $\mathbf{F}_{\text{Dynamic}}$ placement. This curve shows the effectiveness of interleaving the Density matrix. On each sub-plot the grey line corresponds to ideal speedup. The x-axis of each plot has the thread count (N_{proc}) which is varied from 1 to 16. The y-axis is the speedup. For each of the three methods the base time, which was used to compute the speedup, is also given. We have opted not to present the base time for all three placement curves as these vary from each other by less than 1%. We note that the 4 thread experiments used Nodes 2 – 5, the 8 thread experiments used one core from each node, and the 16 thread experiments used all cores on all the nodes of the SunFire X4600 M2.

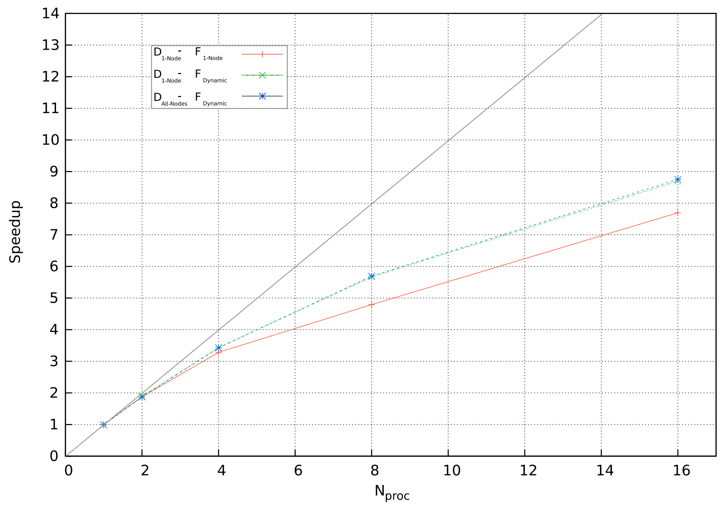
For the Valinomycin molecule and the HF method, all three curves in Figure 5.8 are nearly identical from 1 – 4 threads, having a speedup very close to 3.5 with 4 threads. From this point, curve 1 starts to diverge from 2 and 3. For 8 threads, this separation is clear, with curves 2 and 3 being quite distinct from curve 1. This indicates that the overhead if threads having to access remote Fock matrices (for the red curve) starts to be a limiting factor. On increasing to 16 threads, we see that curve 1 has not been able to make use of the extra thread count owing to the poor memory placement of the both the Density and Fock matrices. It attains a speedup of just 6.7. In comparison, curves 2 and 3 attain speedups of 9.8 and 10.1 respectively. The separation between curves 2 and 3, indicates that interleaving the Density matrix improves speedup when using 16 threads.

For the BLYP method the speedup curves are identical for 2 threads. Starting from four threads, there is a separation between curve 1 and the other curves. For 8 threads, the speedup for curve 1 is less than curves 2 and 3, which are near identical. Increasing to 16 threads,

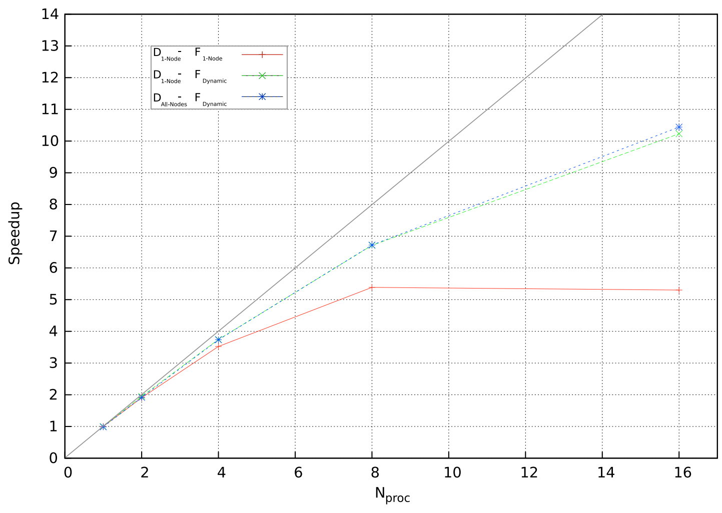
Figure 5.8: Speedup plot for Valinomycin using a 6-311G* basis set and HF, BLYP, B3LYP methods.



(a) HF (Base time: 1996 seconds)



(b) BLYP (Base time: 1649 seconds)



(c) B3LYP (Base time: 2942 seconds)

curve 1 attains at best a speedup of 7.7 whereas curves 2 and 3 have speedups of 8.7 and 8.8 respectively. The implementation of the BLYP method does scale with increasing processor count but, it is much less than what was observed for the HF method.

For the B3LYP method speedups follow similar trends to those seen for the HF method, as PRISM dominates the runtime rather than CALDFT.

For C60 speedups are shown in Figure 5.9. The trends are similar to those seen for Valinomycin for all three methods.

To summarize this section, three placement schemes were used to evaluate the speedups for specific memory placements of the Density and Fock matrices. Two of these placements varied the location of the Fock matrix and interleaving of the Density matrix.

It was observed that all calculations benefit from thread local access to Fock matrices. For the HF and B3LYP methods interleaving of the Density matrix helps improve speedups obtained. The BLYP method was not sensitive to Density matrix interleaving, but was sensitive to the use of node local Fock matrices.

For 16 thread calculations the dynamic migration of the Fock matrix and interleaving of the Density matrix resulted in significantly better HF performance. For BLYP calculations, the use of dynamic migration of the Fock matrix improved performance. Speedups obtained for B3LYP were similar to those for HF.

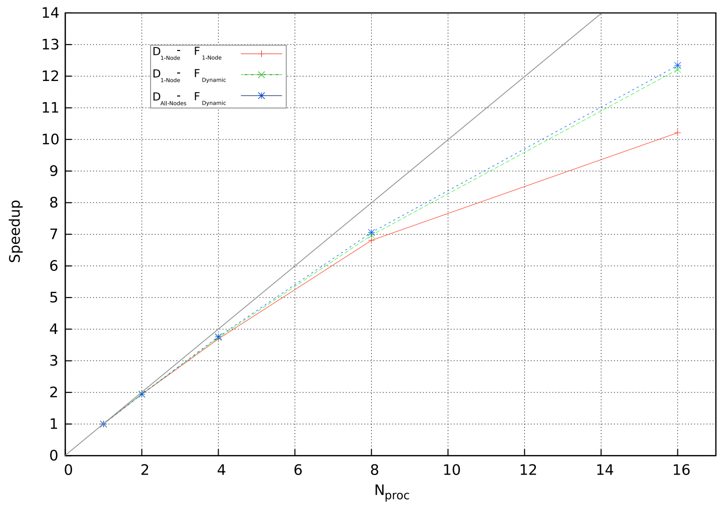
Most importantly the cost of dynamic migration and interleaving were found to be small in comparison to the gains to be had from accessing node local data and reducing interconnect contention.

5.7 Related and Previous Work

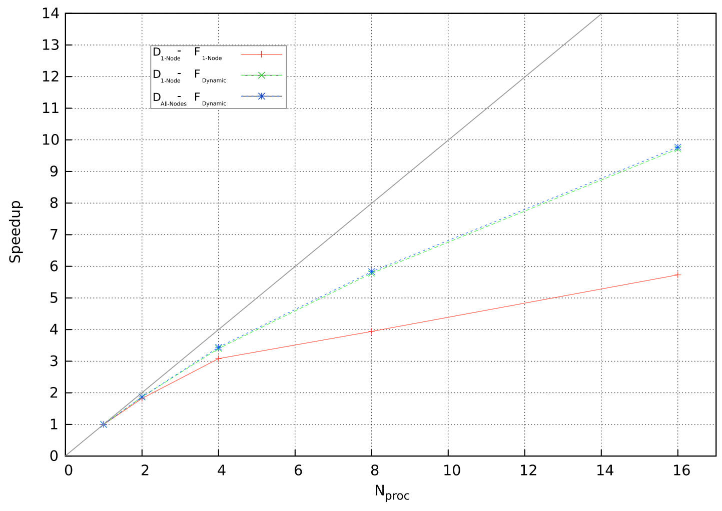
Very recently Gomperts et al. [98] reported the scalability of the Gaussian code for frequency calculations on the SGI Altix 450 cc-NUMA machine [240], for a coupled perturbed Hartree-Fock (CPHF) [74, 203, 215] calculation to obtain the infra-red and vibrational circular dichroism (VCD) spectra [68] for the α-Pinene[7] molecule. They observed a performance anomaly where the first four threads' execution time was far less than those of the other twenty-eight threads, within link l1002. The cause was shown to be the near simulataneous data-loads of the Density matrix by all the worker threads. By using code modifications, which involve creating thread-local datastructures at runtime, they were able to improve performance by a factor of 2. Performance analysis was done using an SGI developed profiling tool called 'Histx', which works by sampling the hardware performance counters for the Itanium2 processor during application execution. Profiling revealed the default first-touch policy led to all memory being

[7] α-Pinene is commonly found in 'Oil of Turpentine' extracted from Pine trees [301].

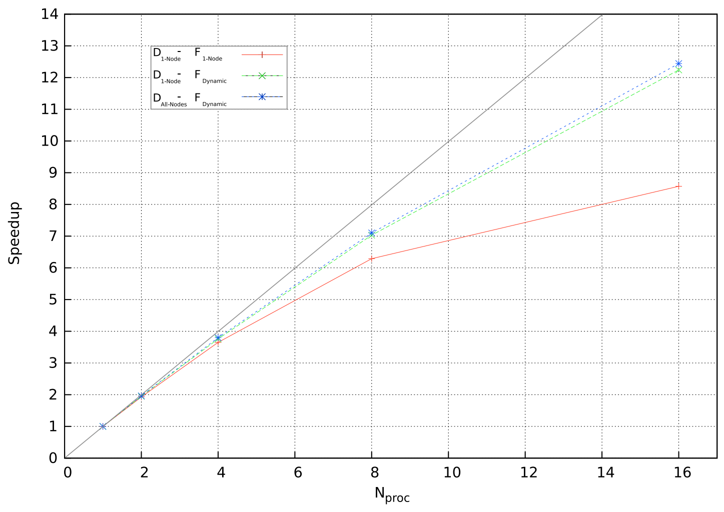
Figure 5.9: Speedup plot for C60 using a cc-PVTZ basis set for the HF, BLYP, B3LYP methods.



(a) HF (Base time: 1275 seconds)



(b) BLYP (Base time: 500 seconds)



(c) B3LYP (Base time: 1475 seconds)

allocated on one NUMA node. They re-coded Gaussian to create node local copies through the creation of temporary arrays in parallel regions and relying on the first-touch policy to ensure pages were local to the thread of execution. (An approach similar to this was used in 5.4 of this chapter). They go on to propose modifications to the `FirstPrivate` OpenMP clause to affect node local placement. There is general agreement with the conclusions reached in [98] and work performed in this chapter, wherein for different code paths within Gaussian (the CPHF code) and different test molecule, the need for locality aware access to data impacts performance. In our work we were able to affect application driven thread and memory placement to ensure node local placement. We found the use of dynamic migration of the Fock matrix and node interleaving of the Density matrix beneficial in improving scalability for the HF, BLYP and B3LYP methods. The overheads for migration were outweighed by the benefits of node local data placement.

The NUMA and multi-thread extended LPM is a simple model which uses an additive cost model of data items and scaling factors to predict performance. In the following section, we review previous work done in the area of performance modelling of NUMA platforms.

Yang [309] develops a queuing model for a cache-based multiprocessor system that uses hierarchical buses, and which resembles modern multi-core CPUs. The model captures system bus contention as well as cache and memory interference for a general memory reference pattern. The modelling was validated against simulation and it was found that bus traffic for enforcing coherency was significant and hence Yang proposes an adaptive cache coherence protocol which allows for multiple copies of shared data within a set of processors that share the same interconnect link to main memory.

Torrellas et. al. [281] study the performance of a hierarchical shared-memory multiprocessor. They develop an analytic model of traffic in a machine which mimics the Stanford DASH [158] shared-memory multiprocessor system. By using traces collected for a 16 processor configuration they predict performance for a 256 processor system. Three major factors are identified in their study as influencing performance (a) locality of data access, (b) the amount of data sharing between threads, and (c) the available bandwidth in a cluster. A key recommendation for reducing bus contention is to facilitate direct access to memory without involving the bus used by the attached processors. The SGI Altix system implements this recommendation by virtue of its hardware SHUB [166] that responds to remote memory requests for a group of Itanium2 processors that share the same bus.

Zhang and Qin [316] present several analytic models to predict the overhead of operations (scheduling, synchronization, data layout and access patterns) which impact the performance of a NUMA multiprocessor system. Their models incorporate memory and network contention. It assumes that memory requests are uniformly spread across all the memory modules, this is done to facilitate quick numerical solutions for their models. A key finding of their modelling is that the rate of remote-memory requests could be used to predict the remote-memory access delay.

LaRowe et. al. [143] implement parametrized dynamic page migration strategies and follow on to measure the performance of parallel programs using these strategies as well as developing an analytical model of the memory system performance for a NUMA system using mean-value analysis. Their measurements and modelling shows that replication of commonly used pages is beneficial and avoids pages bouncing between NUMA memory nodes.

Ring based interconnects are used in multi-core microprocessors notably the Cell BE [8], the Core i7 [152] and Larabee [238]. Holliday and Stumm [117] investigate the performance of hierarchical, ring-based shared memory multiprocessors using simulation. The simulator was validated against the Hector shared-memory system [293]. Instead of using trace or execution driven simulation, a synthetic workload model was used to enable fast turn-around for a 1024 processor system. Their key findings were that maximizing locality in applications reduces memory contention; multiple memory banks are required to facilitate multiple outstanding requests to memory; an adaptive maximum number of outstanding memory transactions is needed to adjust for and aid computation or communication efficiency subject to changes in communication locality; and processor and memory subsystem design needs to be balanced to avoid creating system hot-spots.

Bhuyan et. al. [25] use simulation to quantify the effects of memory management policies for scientific applications on NUMA platforms which use a multistage switching network. A key finding was that the degree of performance improvement of an application is both dependent on the memory management technique and the switch architecture.

Kaeli et. al. [144] present performance analysis of a CC-NUMA prototype machine developed at the IBM T. J. Watson research center. By using hardware instrumentation, traces were obtained for transaction processing benchmarks to identify which elements in the OS or user code were responsible for inter-node references. An analytic model of the prototype system was created to evaluate the effect of architectural changes. Some key results are (a) replication of read-only pages is critical in reducing the number of non-local references; (b) process pages should be placed local to the owing process during initial page allocation; (c) there needs to be an API to provide the OS with semantic hints about page contents and its placement.

Schmollinger and Kaufmann [236] present an extension to the BSP model [288] of parallel computation to aid in mapping algorithms onto clusters of SMP machines (NUMA machines with a hierarchical interconnect).

Nordén [196] presents an analytic model describing OpenMP PDE solvers which take into account the NUMA ratio; a locality and optimal locality factor. Using this model Nordén shows that ordered local PDE methods are insensitive to high NUMA ratios and allows these algorithms to scale well on any NUMA system. The modelling also indicates that there are great gains to be had from using an optimal data distribution.

5.8 Conclusions and Future Work

In this chapter a study of thread and memory placement effects on the Gaussian code was undertaken. It comprises three parts:

First, a 18-Crown-6 Ether molecule using a 6-31G* basis set and Valinomycin using a 3-21G basis was used to obtain serial and parallel timings using a modified version of Gaussian, which could perform thread and memory placement. Timing results were obtained for two cache blocking factors and speedup plots were presented. Serial results, using 18-Crown-6 Ether, showed that good cache blocking can potentially mitigate the effect of poor thread and memory placement on a NUMA machine like the SunFire X4600 M2. Parallel results, using Valinomycin, indicate that good speedups can be obtained by the use of cache blocking and co-locating threads and memory. Speedup results for an unmodified instance of the Gaussian code indicated that it scales similarly to an instance which has all memory being allocated on one node.

Second, a straightforward extension of the LPM to NUMA systems was proposed and evaluated. The extension uses a simple additive model to account for the cost penalty associated with fetching data items from non-local NUMA domains. The NUMA and multi-threaded LPM requires cache misses for each NUMA domain. By using targeted thread and memory placements a set of base timings were obtained with combinations of these used to calculate times for new thread and memory placements. The extended model was then evaluated for single and multi-threaded experiments. Results upto 4 threads had a 5% error in prediction, whereas for 8 threads the error increased to 15% for the dual-core case. The extended model does not account for interconnect contention explicitly. It was proposed that two additional scaling factors could be used to reduce the modelling errors on multi-core platforms.

Third, the use of page migration to affect data locality was assessed using the memory placement APIs in the Solaris operating system. These were used to place Fock matrices locally and to node interleave pages for the Density matrix to reduce contention. A series of placement experiments were performed for the HF, BLYP and B3LYP methods. All calculation types benefited from the use of page migration for the Fock matrix and node interleaving of the Density matrix.

For future work, it would be useful to incorporate the extended LPM as a feedback loop into PRISM, PRISMC and CALDFT to adaptively alter thread and memory placement decisions based on the topology of the underlying hardware.

Chapter 6

A Comparative Study of Charges Obtained for a Set of Water Cluster Complexes

6.1 Introduction

The primary focus of this thesis is towards the creation and use of performance models for the Gaussian quantum chemistry code on NUMA platforms. The electronic structure calculations performed in previous chapters generated a wavefunction for a fixed molecular geometry. As the wavefunction is a mathematical construct, computational chemists have devised procedures to extract meaning from it in line with their 'chemical intuition'. One such model relates to the assignment of charge onto constituent atoms in a molecular system. Charges are experimentally observable quantities [213], and relate to chemical processes such as to bond formation, electronegativity, polarization and sites for electrophilic or nucleophilic attack [188, 315].

In this chapter the Gaussian code is used to perform charge analysis of the electronic wavefunction for a set of molecular systems. This analysis is an example of how Gaussian is used by computational chemists. In chapter 4 two test molecular systems (k300a-04 and k300a-08) were used in assessing the Linear Performance Model (LPM). These two test systems are part of a larger ensemble of molecular systems used in this chapter. The larger ensemble of systems lie at the forefront of system sizes that are amenable to calculation at present.

In 2004, Bliznyuk and Rendell [28] studied the electronic charge on a potassium ion (K+) located at two specific positions in a large molecular structure known as a Potassium ion channel [167]. (The potassium ion channel is a pore-like protein, present in all biological cells which is responsible for regulating the flow of ions into and out of the cell. A schematic is shown in Figure 6.1). Charge results have been reproduced in Table 6.1. The table gives the nett charge on the K+ ion obtained using HF and B3LYP methods with a 6-31G* basis set. Charges were obtained using the Mulliken Population Analysis (MPA) method [189].

Table 6.1: Mulliken Charges (au) obtained using HF and the B3LYP DFT functional on a K+ ion positioned at two locations (Point B, Point C). The 6-31G* basis set was used for both methods. Reproduced from Table 5 in [28].

Shell	Structure 2 (Point B)		Structure 3 (Point C)	
Cutoff	HF	DFT	HF	DFT
3	0.745	0.580	0.572	0.277
5	0.617	0.369	0.565	0.261
6	0.490	0.169	0.540	0.221
8	0.433	0.079	0.532	0.195
10	0.426	0.068	0.530	0.193

In Table 6.1, there are references to two structures (‘Structure 2’ and ‘Structure 3’), in which the potassium ion is located at ‘Point B’ or ‘Point C’. These two locations were chosen as they represent different electronic environments for the passage of the K+ ion through the channel. The two structures were taken from a molecular dynamics (MD) study performed in earlier work by Bliznyuk et. al. [29] . The ‘Shell Cutoff’ column, in the table, refers to the distance criteria used to determine if a given molecule was to be included when computing the charge on the K+ ion. A key point is the variation of charge on the K+ ion; specifically the DFT results shows that the K+ ion loses almost all of its charge as the system size is increased, and this is true for both structures. The dramatic reduction in charge on the K+ ion using DFT is clearly an unphysical result.

Bliznyuk and Rendell point out that this result reinforces “the view that electron density results obtained from DFT calculations should be viewed with extreme caution, especially for large molecules”. They note also that the dramatic reduction in charge on K+ is due to DFT methods overemphasising the importance of polarization in large molecules as has been observed elsewhere [103], but they do not prescribe alternative strategies to perform charge analysis using DFT methods[1].

The aim of this study is to further investigate the issue of how to obtain charges for K+ ions using DFT wavefunctions, including the use of two different charge analysis methods – the Mulliken Population Analysis and Natural Population Analysis (NPA).

The chapter layout is as follows: Section 6.2 presents background material. Section 6.3 briefly discusses the test molecular systems, software and methodologies used in this chapter. Section 6.4 presents results for charges on a K+ obtained from HF, BLYP and B3LYP methods using both the MPA and NPA methods. The distribution of charge within a water cluster system is analyzed in Section 6.5. Following this Section 6.6 presents an analysis of charge distribution in a water cluster as a function of the radial distribution function. In Section 6.7

[1]DFT methods are widely used due to their $O(N^2)$ scaling for large systems and its ability to obtain chemically accurate results [137]

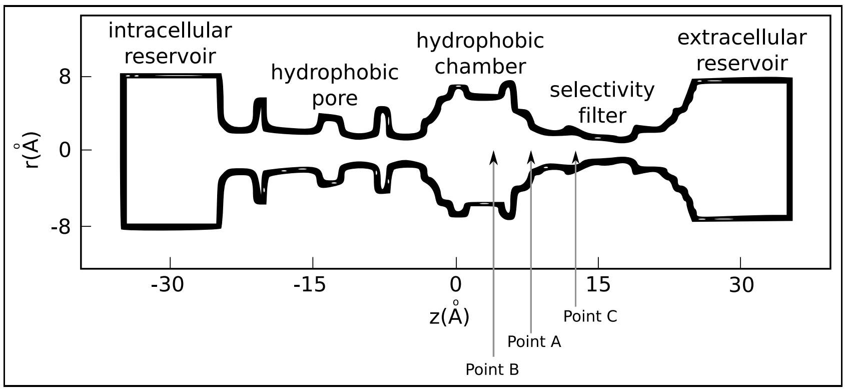


Figure 6.1: Schematic illustration of the KcsA potassium ion channel showing the location of Point B and C. Taken from [28].

we compare spherically integrated electron density, obtained from the HF, BLYP and B3LYP methods. Section 6.8 considers the use alternative basis sets. Previous work is discussed in Section in 6.9 and the Chapter concludes in Section 6.10.

6.2 Background

In this section the Mulliken and Natural charge analysis methods, used in this chapter, are discussed. Charge analysis techniques are a means of interpreting Schrödinger's wave equation; the MPA and NPA methods achieve this by partitioning the wavefunction with reference to the basis functions used to compute it. This then allows qualitative analysis of the chemical processes underpinning the system e.g. determining sites for nucleophilic or electrophilic attacks [188].

The electron density function $\rho(\mathbf{r})$ is defined as the probability of finding an electron in a volume d$\mathbf{r}$,

$$\int \rho(\mathbf{r})\, d\mathbf{r} = N \tag{6.1}$$

where integration is carried out over all space and this results in N electrons. For HF theory,

$$\rho(\mathbf{r}) = \sum_{\mu}^{N}\sum_{\nu}^{N} P_{\mu\nu}\ \phi_{\mu}(\mathbf{r})\ \phi_{\nu}(\mathbf{r}) \tag{6.2}$$

where $P_{\mu\nu}$ is an element of the density matrix and $\phi(\mathbf{r})$ are atom centered basis functions. On integrating $\rho(\mathbf{r})$ over all space,

$$\begin{aligned}
\int \rho(\mathbf{r})\, d\mathbf{r} &= \sum_{\mu}^{N}\sum_{\nu}^{N} P_{\mu\nu} \int \phi_{\mu}(\mathbf{r})\, \phi_{\nu}(\mathbf{r})\, d\mathbf{r} && (6.3)\\
&= \sum_{\mu}^{N}\sum_{\nu}^{N} P_{\mu\nu}\, S_{\mu\nu} && (6.4)\\
&= N && (6.5)\\
& && (6.6)
\end{aligned}$$

where $S_{\mu\nu}$ is the overlap matrix.

6.2.1 Mulliken Population Analysis

Mulliken proposed the first population analysis technique [189]. The implicit assumption is that the electron density can be partitioned according to the basis functions used to describe it. If density is attributed to a product of two functions on different centers, then the density is split evenly.

The MPA scheme uses the $D_{\alpha\beta} S_{\alpha\beta}$ matrix to apportion charge i.e. the matrix arising from the product of the Density and overlap matrices. The number of electrons associated with AO α is the diagonal element $D_{\alpha\alpha} S_{\alpha\alpha}$, whereas half the value of an off-diagonal element $D_{\alpha\beta} S_{\alpha\beta}$ is the number of electrons shared between AOs α and β. By summing all the contributions from AOs of a given atomic center, the number of electrons attributed to that center can be obtained. Thus the electron population on a given atomic center A is defined as,

$$\rho_A = \sum_{\alpha \in A}^{AO} \sum_{\beta}^{AO} D_{\alpha\beta} S_{\alpha\beta} \tag{6.7}$$

which is then used to define the gross charge Q_A on the atomic center according to

$$Q_A = Z_A - \rho_A \tag{6.8}$$

where Z_A is the atomic number of atom A (i.e the number of positively charged protons in the nucleus of Atom A).

6.2.2 Natural Population Analysis

Wienhold et al. [224,298] describes the use of Natural population analysis to obtain nett charge on atoms. The NPA techniques uses natural orbitals to apportion electron density onto atomic and molecular orbitals.

Natural orbitals are the eigenfunctions of a quantity called the 'first-order reduced Density matrix'. We now discuss the first-order density matrix, the definition of a natural orbital and its subsequent use in the NPA procedure to obtain nett charge.

6.2.2.1 First-order reduced density matrix

The motion of electrons in a system are described by the wavefunction ψ. The electron density function $\rho(\mathbf{r})$ is obtained from the wavefunction,

$$\rho(\mathbf{r}) = |\psi(\mathbf{r})|^2 \tag{6.9}$$

Given 'N' electrons in a system, the probability P, of finding electron 1 located at $\mathbf{x}_1$, simultaneously when electron 2 is located at $\mathbf{x}_2$ and so on for 'N' electrons is given by,

$$P = \psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)\ \ \psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) \tag{6.10}$$

Thus the probability of finding electron 1, regardless of the locations of the other electrons is defined as,

$$P(\mathbf{x}_1) = N \int d\mathbf{x}_1\, d\mathbf{x}_2 \ldots d\mathbf{x}_N\ \psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)\ \ \psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) \tag{6.11}$$

where, N is a normalization factor defined such

$$\int d\mathbf{x}_1 \rho(\mathbf{x}_1) = N \tag{6.12}$$

The probability function $P(\mathbf{x}_1)$ can be generalized to a density matrix $\gamma(\mathbf{x}_1, \mathbf{x}_1^{'})$, which is termed the first-order reduced density matrix,

$$\gamma(\mathbf{x}_1, \mathbf{x}_1^{'}) = N \int d\mathbf{x}_1\, d\mathbf{x}_2 \ldots d\mathbf{x}_N\ \psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N)\ \ \psi(\mathbf{x}_1^{'}, \mathbf{x}_2, \ldots, \mathbf{x}_N) \tag{6.13}$$

For HF theory, this can be obtained by expanding $\gamma(\mathbf{x}_1, \mathbf{x}_1^{'})$ using functions χ_i,

$$\gamma^{HF}(\mathbf{x}_1, \mathbf{x}_1^{'}) = \sum_a \chi_a(\mathbf{x}_1)\chi_a^*(\mathbf{x}_1) \tag{6.14}$$

6.2.2.2 Natural Atomic Orbitals and NPA

The first-order reduced density matrix (γ) is Hermitian. On diagonalizing γ the eigenfunctions are called natural atomic orbitals (NAO) and the eigenvalues are occupation numbers. NAOs can be used to determine nett atomic charge [83]. We now briefly cover the process of obtaining charges using NPA. Assume the basis functions for the system of interest have been arranged such that all orbitals for center A are before those on center B and so on i.e.,

$$\chi_1^A,\, \chi_2^A,\, \chi_3^A,\, \cdots\, ,\, \chi_{k+1}^B,\, \chi_{k+2}^B,\, \chi_{k+3}^B,\, \ldots,\, \chi_{l+1}^C,\, \chi_{l+2}^C,\, \chi_{l+3}^C, \tag{6.15}$$

A Density matrix **D** can be written in terms of blocks of basis functions at a given center,

$$\mathbf{D} \;=\; \begin{pmatrix} \mathbf{D}^{\mathrm{AA}} & \mathbf{D}^{\mathrm{AB}} & \mathbf{D}^{\mathrm{AC}} & \ldots \\ \mathbf{D}^{\mathrm{AB}} & \mathbf{D}^{\mathrm{BB}} & \mathbf{D}^{\mathrm{BC}} & \ldots \\ \mathbf{D}^{\mathrm{AC}} & \mathbf{D}^{\mathrm{BC}} & \mathbf{D}^{\mathrm{CC}} & \ldots \\ \multicolumn{4}{c}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \end{pmatrix} \tag{6.16}$$

The NAOs for atom A are defined as those which diagonalize the $\mathbf{D}^{\mathrm{AA}}$ block and similarly for atom B and so on. The definition of the sub-blocks are constrained to ensure eigenfunctions are orthonormal within the sub-block and to all other eigenfunctions.

The process of obtaining NPA charges begins by partitioning both the density and overlap matrices into sub-blocks ordered by the Atom (A) that the sub-block represents, its angular momentum (l) and the symmetry element (m) for the given l.

All $2l+1$ symmetry elements in the sub-block are averaged and the sub-blocks are independently diagonalized to give eigenfunctions which form what are called pre-NAOs.

These pre-NAOs are classified by their occupancy into two groups, the ones with the highest occupancy are called the Natural Minimum Basis (NMBs) and all the rest are called the Natural Rydberg Basis (NRBs).

As pre-NAOs from one center overlap with other pre-NAOs on another center, two sets of diagonalizations are done to obtain the intermediate NAOs – the NRBs are Schmidt orthogonalized w.r.t the NMBs and the NRBs are separately diagonalized using a weighted occupancy symmetric orthogonalization scheme [224].

This process removes overlaps and localized density onto specific atomic centers. The intermediate NAOs from the two sets of orthogonalizations are re-blocked and diagonalized to give the final NAOs. The diagonal elements of the density matrix formed using NAOs give the atomic population of each NAO. If the atomic populations for all the NAOs centered on a given atom are summed, the resulting charge population is called the natural atomic population.

6.2.3 The Radial Distribution Function

The RDF is a means of describing the structure of systems like gases and liquids [9, 156]. It measures the correlation between particles in a system and denotes the average probability of finding some particle at a distance **r** from a reference particle i.e. it is a spatial measure for the packing of particles around some central point.

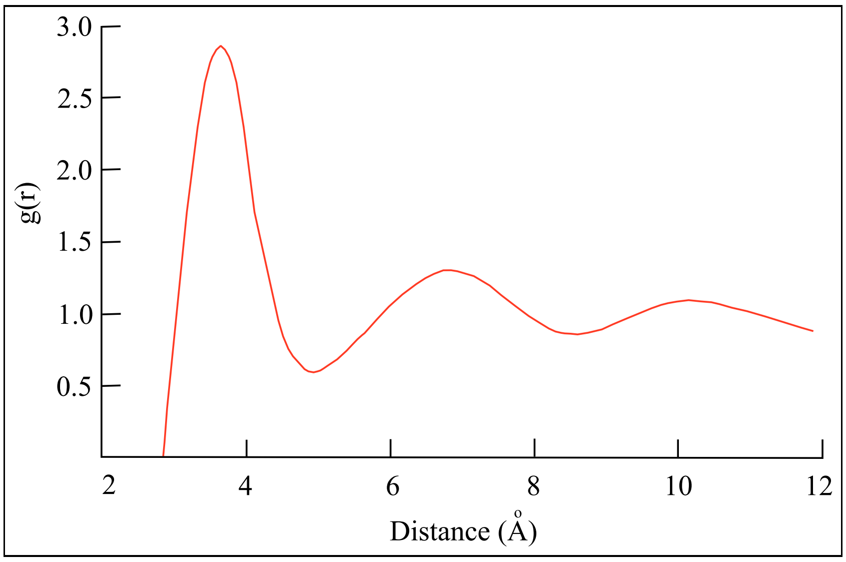


Figure 6.2: Radial distribution function determined from a 100 ps molecular dynamics simulation of liquid argon at a temperature of 100 Kelvin and a density of 1.396 g/cm^3. Taken from [156].

The RDF is computed relative to an ideal gas[2] and is a dimensionless quantity [115, 223]. In this section the pair-wise RDF $g(r)$ is utilized.

Figure 6.2 is a typical RDF obtained from an MD simulation. The graph, taken from [156], is for a 100 ps MD simulation of liquid argon. From the graph, at short distances $g(r)$ is zero owing to the large short-range repulsive force between atoms. At 3.7 Å $g(r)$ peaks at 3 and this indicates that it is three times more likely that a pair of Argon atoms are separated at this distance. The RDF reduces at 5Å approaching a minimum, indicating that at this distance the chances of finding a pair of particles is diminished compared to the 3.7Å . At distances greater than 7Å the value of $g(r)$ approaches one indicating a distinct lack of long range structure.

The pair-wise RDF is generated by computing the distance between a particle of interest and other particles in the system. These distances are sorted by distance and binned to create a histogram. Each population is then normalized by the number of particles that would be present in an ideal gas.

A Python code for computing the RDF was written to process the test systems used in this Chapter, this has been reproduced in the Appendix, Section A.4.

[2]An ideal gas is a theoretical system which has a uniform distribution of its particles over all of space

6.3 Experimental Details

This section discusses the molecular systems and the methodology used in this study.

6.3.1 Test Molecular Systems

A set of test molecular systems which consist of a solvated K+ ion, surrounded by progressively larger numbers of water molecules were used.

Geometries for these water clusters are obtained from MD simulations run at 300 and 345 Kelvin respectively using the Amber program [46]. These snapshots have been labeled as k300a, k300b, k300c and k345a i.e. there are three snapshots at 300 Kelvin and one snapshot at 345 Kelvin (cf. Appendix A.3). Four snapshots for the two temperatures are taken for a water cluster which has a radius of 12Å. Then for each of these snapshots a set of six sub-systems are derived by using different cutoff radii. These are labelled as (k300a-{03 – 10}), where each suffix i.e. 03, 04, 06, 08 and 10 denotes the cutoff distance from the center of the K+ ion.

Using Appendix Table A.9 as an example, k300a-03 is a subset of the k300a-12 snapshot. It has five water molecules which are at a distance of 3Å from the central K+ ion.

Thus the k300a-12 is a superset of previous five subsets (k300a-{03 – 10}). In the same manner subsets for the k300b (Table A.10), k300c (Table A.11) and k345a (Table A.12) geometries are defined.

6.3.2 Software and Methodology

The Gaussian code implements various charge analysis methods. Of these we opted to consider those which either used input basis functions or the electron density. The MPA and NPA methods use input basis sets. Two alternative methods, which use electron density for charge analysis, called Atoms in Molecules (AIM) [20] and Hirshfeld charges [105] exist within Gaussian. We explored the use of the two electron density methods and found that the AIM implementation in Gaussian was deprecated, and could not handle the large water cluster systems. It also became apparent that the Hirshfeld charge implementation was not fully functional and to our knowledge there are no peer reviewed publications based on using it. Thus, only MPA and NPA charges are considered here.

MPA and NPA charges were calculated for all the snapshots using Gaussian G03D02 [86], with a 6-31G* basis set using the HF, BLYP and B3LYP methods.

Custom Python scripts were used to extract K+ charges, plots of relative charges were obtained using a Gnuplot [216] interface from Python. Similarly radial distribution functions

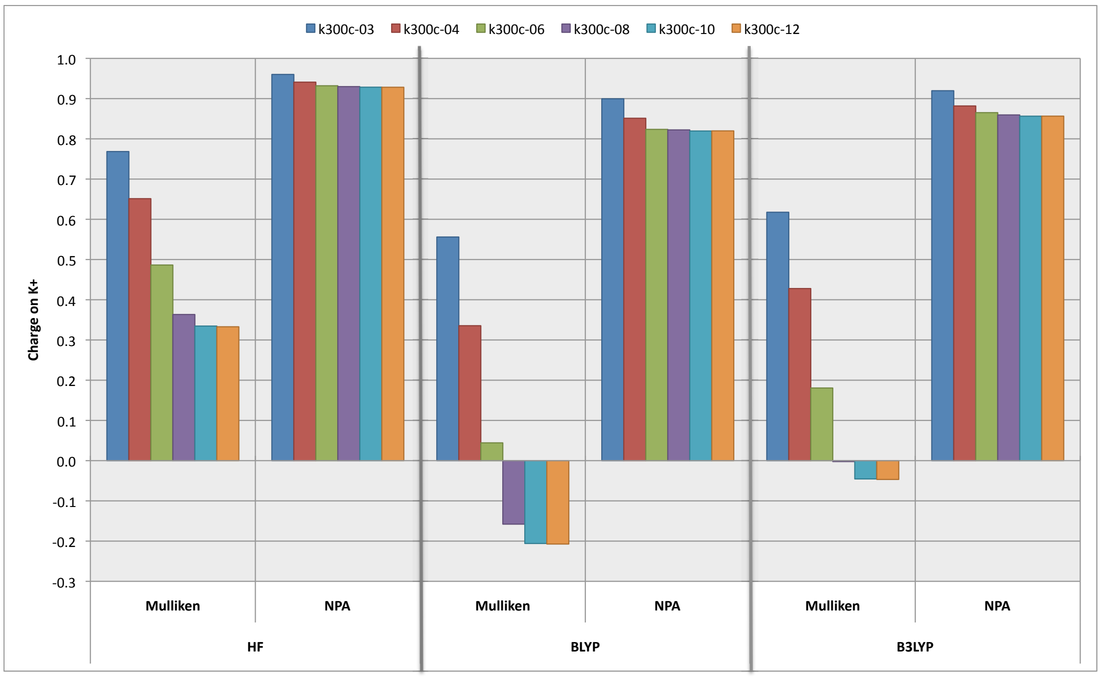


Figure 6.3: Charge on K+ for the k300c system

for each snapshot were computed using Numerical Python [63] and plotted using the same interface.

6.4 MPA and NPA Charges on the Potassium Ion Obtained Across All Water Complexes

Figure 6.3 presents results for the MPA and NPA charges on the Potassium ion, for the k300c set of snapshots.

The figure is divided into 3 columns. Charges for k300c-{03, 04, 06, 08, 10, 12} are grouped by the computational method (HF, BLYP, B3LYP) and sub-grouped by the charge analysis method used (MPA, NPA). In each column the colored vertical bars are ordered by distance i.e. 3, 4, 6, 8, 10 and 12 Å. (Results for other snapshots are not shown as these display similar trends to the k300c snapshot).

For HF, all charges are positive for both MPA and NPA. As the water cluster size increases from 3Å to 12Å, there is a reduction in charge ($\approx 57\%$) in the MPA results. For the HF NPA results there is only a 3% reduction.

For BLYP results the reduction in charge is dramatic for MPA, the results indicate that from the 8 Å onwards, the charge on the K+ is negative! This implies all charge from the K+ ion has been lost to atoms surrounding it, clearly an unphysical result. For NPA k300c charges are positive decreasing by just 9% as the system size is increased.

The B3LYP results follow the same trends as the BLYP results. The MPA charges are negative starting from the k300c-08 system, but are not of the same magnitude as those calculated for BLYP. NPA results are all positive and show a 7% decrease as system size is increased. In short the B3LYP results are mid-way between HF and BLYP, which is to be expected given that the functional is a hybrid of HF and BLYP.

As noted earlier the MPA procedure is heavily basis set dependent, apportioning charge equally to atomic centers which share in an overlap density. These deficiencies are reduced when using NPA. Yet, this does not satisfactorily explain the dramatic reduction in HF charge as the system size increases and why the Kohn-Sham DFT procedure leads to a dramatic reduction in charges using MPA, yet the NPA charges obtained from DFT are slightly less than those obtained for HF.

6.5 Relative Charge on Potassium as a Function of Distance

In the previous section, an increase in the size of the water cluster system resulted in a decrease in the positive charge on the K+ ion. In this section we seek to identify which water molecules attract the most amount of positive charge from the K+ ion. The expectation is that water molecules nearest to the K+ ion will have the greatest influence, and this influence will reduce with distance.

To address the above we present plots for MPA and NPA charges obtained from the HF, BLYP and B3LYP methods for the relative charge lost from the K+ ion. Figure 6.4 presents these plots for the k300c-12 system, as a function of distance.

The two plots have "Distance from K+" expressed in Ångstroms, on the x-axis. The y-axis is the 'Percentage Relative Charge' using a log scale. Both plots were generated by performing two sets of calculations: first, using MD snapshots for each system (k300a,b,c and k345a) the MPA and NPA charges were obtained; second, the K atom was removed from the input geometry and the charges were computed. The relative charge was then calculated by subtracting charges obtained for each atom in the snapshot without the K+ ion from the snapshot with the K+ ion. The 'Percentage Relative Charge' is this number expressed as a percentage. Results are plotted for each of the HF, BLYP and B3LYP methods.

For the MPA plot, in Figure 6.4, charges obtained from the HF, BLYP and B3LYP methods are near identical upto a distance of 3Å . The magnitude of these six data points indicates that the majority of the K+ charge has been attracted to the electronegative water molecules closest to the K+ ion, as expected.

For distances greater than 3Å , variations in the relative charges obtained from HF, BLYP reduce to around 1%. There is an increase around 3.6Å that then reduces until 6Å . Beyond 6Å the relative charge is much less than 1% indicating that water molecules beyond here play little part in reducing the charge on the K+ ion.

NPA results vary much more than those for MPA. Upto 3Å all HF, BLYP and B3LYP methods follow roughly the same trends, as seen for MPA charges, although the BLYP charges indicate that K+ loses its charge much more quickly than for the other two methods. There is also a pronounced variation in charge on progressing from 2Å towards 6Å , by comparison the MPA results are smoother and better correlated between HF, BLYP and B3LYP upto 6Å.

Overall the MPA results looks entirely reasonable, while the NPA results are some what spurious. Trends observed here, for the k300c plot, were also observed for the k300a, b and k345a snapshots.

6.6 Charges Binned with Respect to the RDF

The previous section presented results for the variation of relative charge with respect to distance from the K+ ion. From this it was determined that water molecules which were at a distance of 6Å or less from K+ played an important role in attracting charge from the K+ ion.

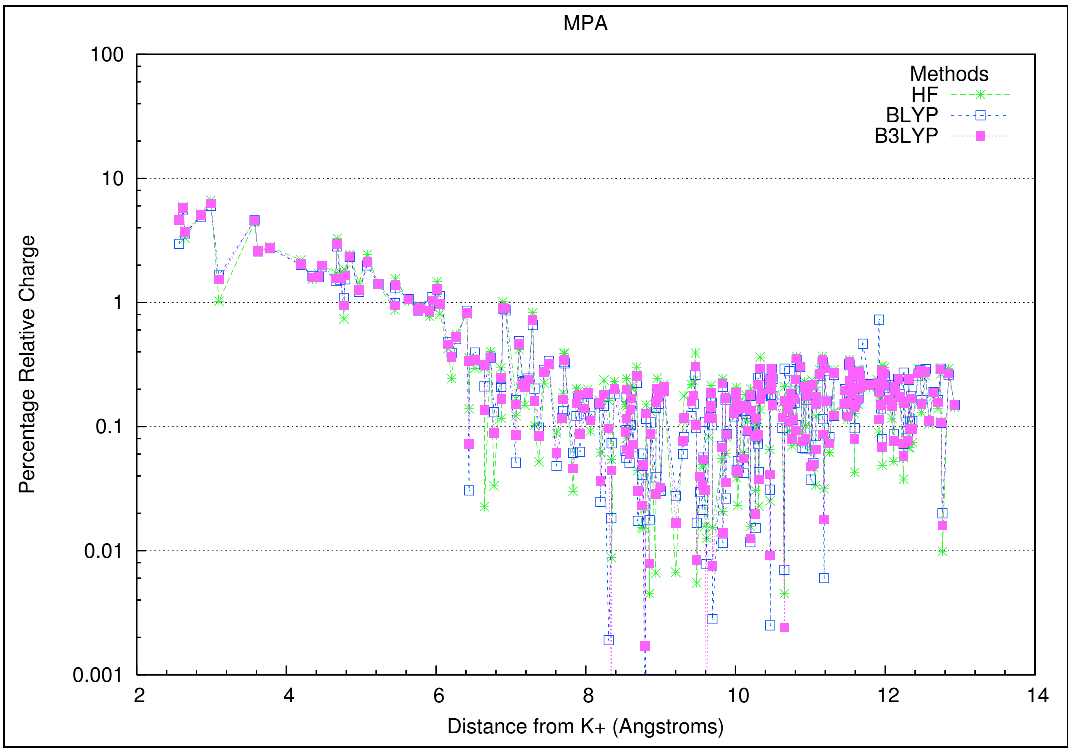
In this section, the Radial Distribution Function (RDF) for the k300c-12 snapshot is presented to (a) characterize the distribution of water molecules surrounding the K+ ion and (b) to bin charges as a function of the RDF. Following this discussion, in sub-section 6.6.1, we present results for a break-down of charges obtained as a function of the RDF.

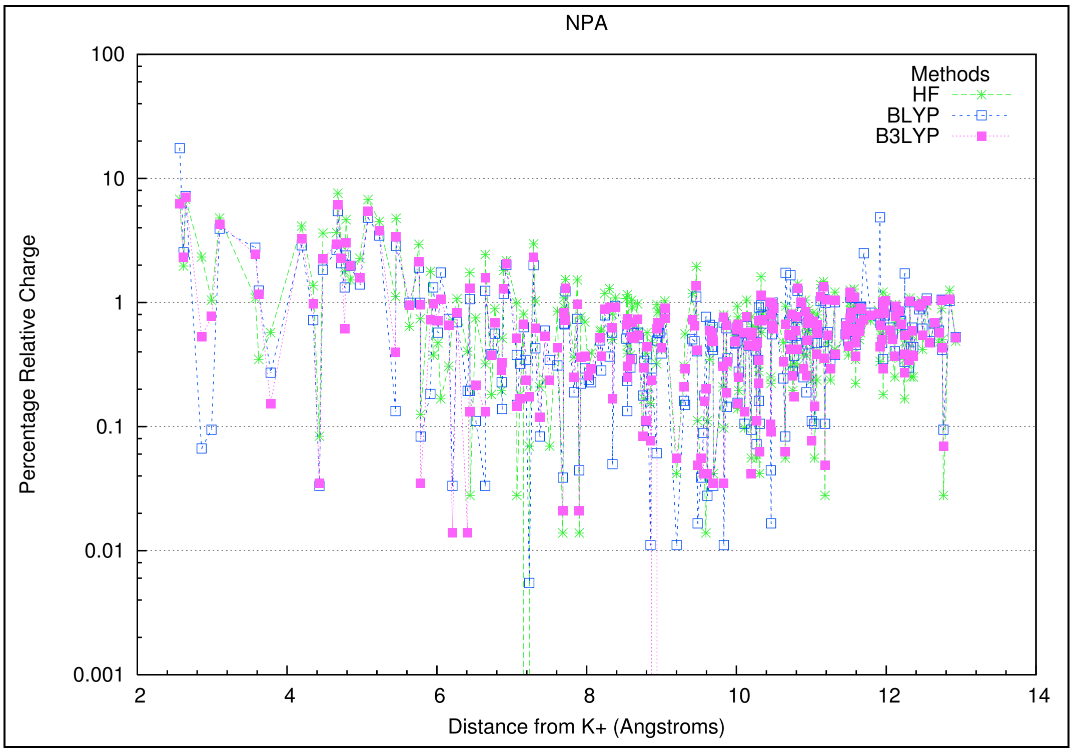
In typical MD simulations the RDF is usually computed by averaging over multiple MD snapshots, whereas we computed RDFs using each of the individual k300a, b, c and k345a snapshots.

The RDF for all snapshots were computed first. This was then followed by the binning of charges based on computed RDF peaks. Results for k300c-12 MPA and NPA charges, binned by distance, are presented in Figure 6.5. Charge plots were created by obtaining NPA and MPA relative charges and binning charges on the Oxygen atom using the same binning width as the RDF. In both plots, the x-axis denotes distance 'r' in Ångstroms from the K+ ion. The left hand side of the graphs is used for the y-axis, it denotes the RDF '$g(r)$'. The right hand side is used to denote charges obtained from the HF, BLYP and B3LYP methods using MPA and NPA.

The RDF (the red curve) for both plots in Figure 6.5 were created using distance histogram bins separated by 0.5 Å . A peak for $g(r) = 3$ at 2.6 Å and a second peak for $g(r) = 1.45$ at 4.16 Å are observed. There are two other peaks for $g(r) = 1.19$ at 5.75 Å and $g(r) = 1.12$ at 9.5 Å . Water molecules occurring at each of these values of $g(r)$, create what are known as solvation

Figure 6.4: Variation of relative charge as a function of distance from K+ for the k300c-12 system.





shells. In the case of the water cluster systems, the water acts as a solvent around the K+ ion. The electronegative oxygen atoms, in all the water molecules nearest to the K+ ion, attracts the positive charge of the K+ ion. This results in a shell of water molecules surrounding the ion.

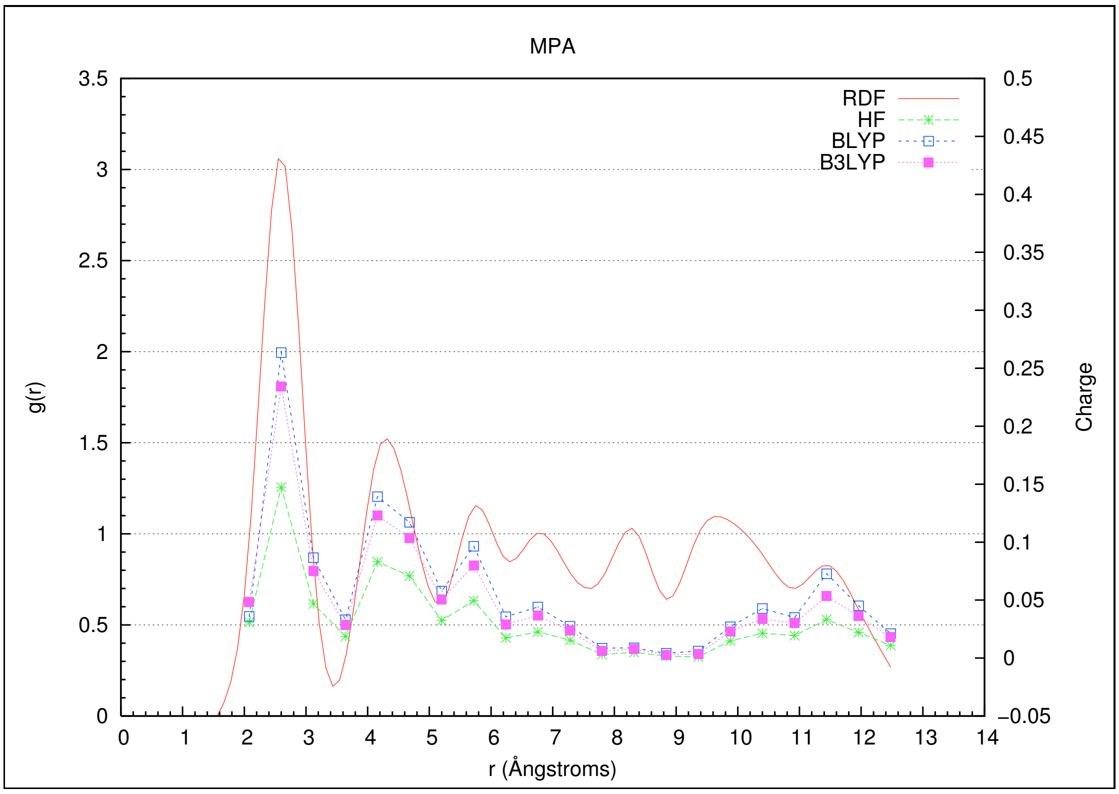
The $g(r)$ values indicate there are two major solvation shells [30] around 2.6 Å and 4.16 Å, followed by a minor shell at 5.75 Å. Thus, it is three times more likely to find water molecules at a distance of 2.25 Å from the K+ ion and around 1.5 times more likely to find molecules at a distance of 4.25 Å from the K+ ion. From this observation the expectation is that most of charge on K+ is drawn away from it by the first solvation shell followed by the second solvation shell.

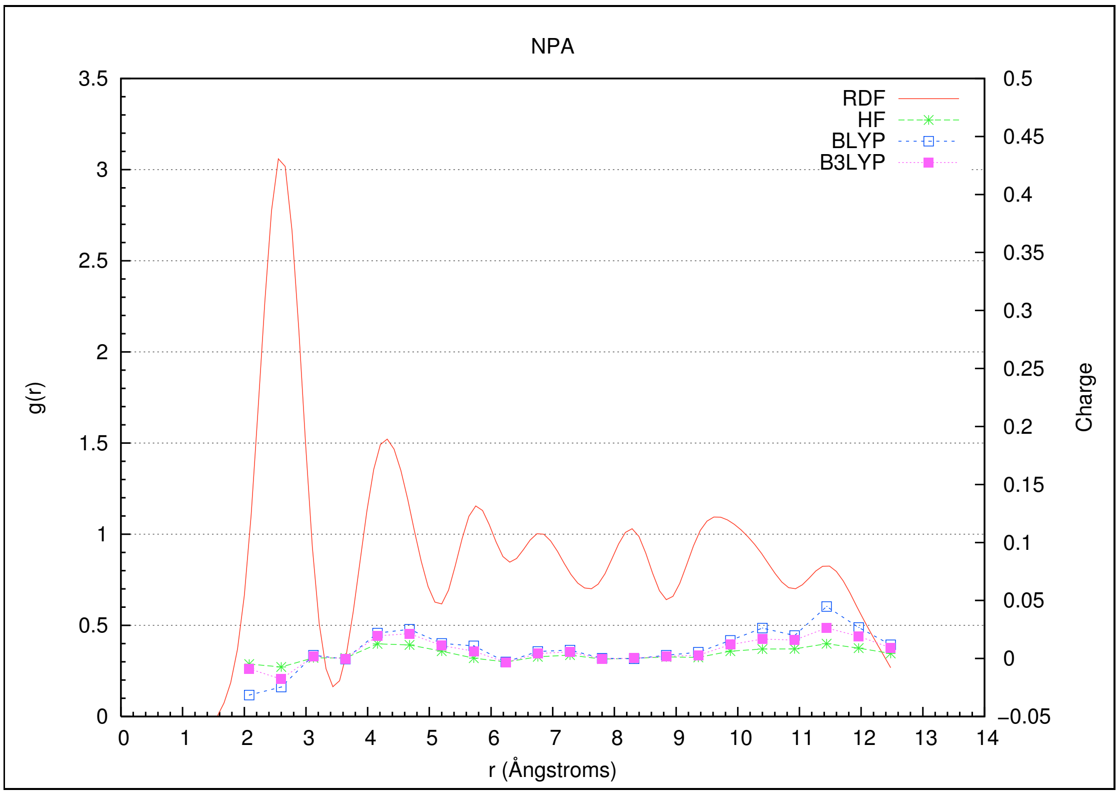
We now consider results for charges that are binned with respect to the radial distribution function. For the MPA plot, there is a strong correlation between RDF peaks and aggregate charge obtained by binning. The highest charge concentration corresponds to the first solvation shell, as expected. MPA charges obtained from HF are the lowest followed by B3LYP and BLYP. The second largest charge concentration occurs at the second solvation shell. There are two other minor charge peaks which occur at 5.75 Å and 11.44 Å. Charges obtained from MPA peak at points corresponding to the location of the first, second and third solvation shells, clearly indicate that the charge distribution within the water cluster is directly influenced by the aggregate distribution of water molecules i.e. where there is a greater likelihood of finding water molecules, there is a clustering of charge. This observation explains the trends observed in Figure 6.4 for the variation of the ‘Percentage Relative Charge’ as function of distance.

Charges obtained using NPA for the HF, BLYP and B3LYP methods follow very different trends from those observed for MPA. Charges upto 3.7 Å are negative and furthermore there is no direct correlation between binned charges and RDF peaks. From the NPA plot, the highest charge peak occurs at 11.44 Å, followed by a peak at 4.68 Å. These results do not reflect expectations of how the charge ought to vary with distance from the K+ ion.

In summary the overall trends for NPA charges do not correlate with the location and number of water molecules for a given solvation shell. We believe this result arises as a consequence of the NPA procedure, and the way it localizes density onto specific atomic centers. The process of generating NPA charges, while accounting for the electronegativity of each atom [105], seems to modify charge densities in a manner such that, for very large water cluster calculations it causes the charge distribution not follow the expected physical trends i.e the expectation that charge peaks occur for corresponding $g(r)$ peaks is not evident.

Figure 6.5: RDF and MPA, NPA Charges for k300c-12 obtained for the HF, BLYP and B3LYP methods. Binning width is 0.5 Å .





6.6.1 Breakdown of RDF Charges By Solvation Shell

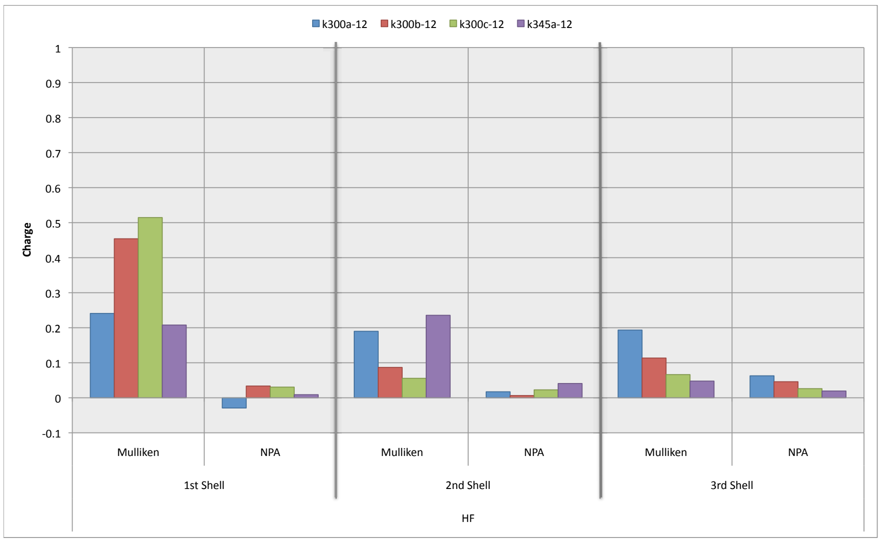
In this sub-section, a detailed breakdown of charges by solvation shell is presented. The aim is to characterize how the total charge attracted away from the K+ ion varies as a function of the solvation shell. This is done by aggregating the results in Figure 6.5 according to the solvation shell. The results are given in Figure 6.6. Figures (a), (b) and (c) correspond to HF, BLYP and B3LYP results respectively.

If we consider Mulliken charges for the first solvation shell obtained using HF, BLYP and B3LYP, we see that the relative ordering between the k300a-12, k300b-12, k300c-12 and k345a-12 is the same. The same trend is seen for Mulliken charges for the 2nd and 3rd solvation shells.

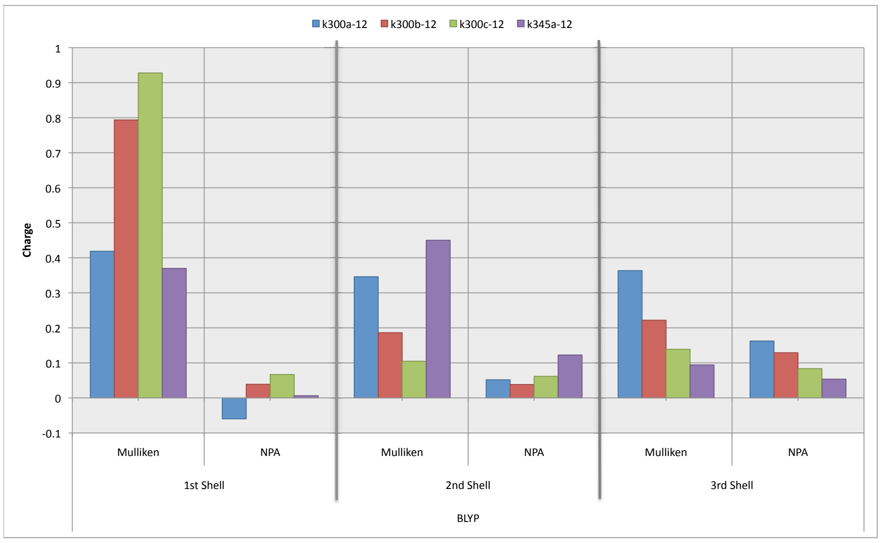
The NPA charges in the first shell, across HF, BLYP and B3LYP are all negative. The second solvation shell charges roughly follow the corresponding MPA trends. For the third shell results show a similar trend to the 3rd shell results for MPA populations but for NPA the third shell results are larger than what they are for the first shell. The latter is unphysical as it suggests the 3rd shell (at a distance of ≈ 10Å) is attracting more charge from the K+ ion than the first shell.

From break-down data presented here, we conclude that trends for MPA charges are in line with the expectation that a solvation shell near the K+ ion attracts more charge than one further away.

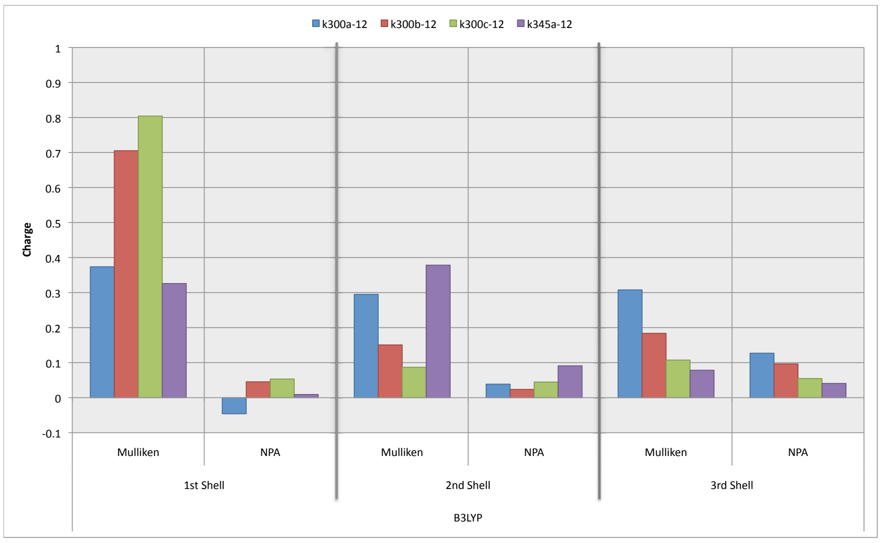
Figure 6.6: MPA and NPA charges per RDF shell for all 12 Å systems, using a 6-31G* basis set.



(a) Results for HF



(b) Results for BLYP



(c) Results for B3LYP

6.7 Density Plots for Charges

In previous sections, the MPA and NPA charge on the K+ ion obtained for the BLYP and B3LYP methods indicated a dramatic loss of charge from the K+ as the water cluster size increased. We then considered how the charge distribution varies with respect to relative distance and as a function of the RDF. None of these investigations were able to pin-point a physical reason (i.e. geometry of the snapshots or structure of solvation shells), that would lead to this.

In this section, we propose and test the hypothesis that the dramatic loss of charge that arises when using DFT methods arises due to a fundamental difference in the densities produced by the BLYP and B3LYP methods, in comparison to the HF method. In the MPA and NPA methods electron density is apportioned to various atomic centers, thus fundamental variations in density are likely to cause differences in the nett charge on atomic centers.

To test this hypothesis, we obtained the discrete representation of the density, using a numerical grid, and carried out spherical integration as a function of distance from the center of the K+ ion. The results are given in Figure 6.7. This was performed using the Gaussian cube option, which generates a discrete representation of electron density, outputs the results as a 'cube' file. Cube files were generated for the k300{a,b,c} and k345a 12 Å systems using the 'Fine' grid option. (This corresponds to 12 points per Bohr for the entire water cluster system). The x-axis in Figure 6.8 is the distance from the K+ ion. The number of electrons was determined by locating the K+ ion within the cube file, and then defining cubic volumes of dimension 1Å upto 24Å with increments of 0.5Å . The values contained within each cube were integrated in order to obtain the electronic charge.

In Figure 6.8 the electronic charge obtained from HF, BLYP and B3LYP densities for the k300c-12 system, as a function of distance are given. These show that for the 248 water molecules and the K+ there are a total of 2501 electrons (as expected). Results obtained from integrating the density for HF, BLYP and B3LYP were essentially identical with no significant variations in the electron density between the three methods. Thus variation in electron density does not appear to be responsible for the negative charges obtained using MPA and DFT methods for the large water cluster systems. The only other factor that determines the charges for MPA and NPA is the input basis sets, hence in the following section we consider the use of various basis sets.

We note that the result in Figure 6.8 are similar to those obtained for all other snapshots.

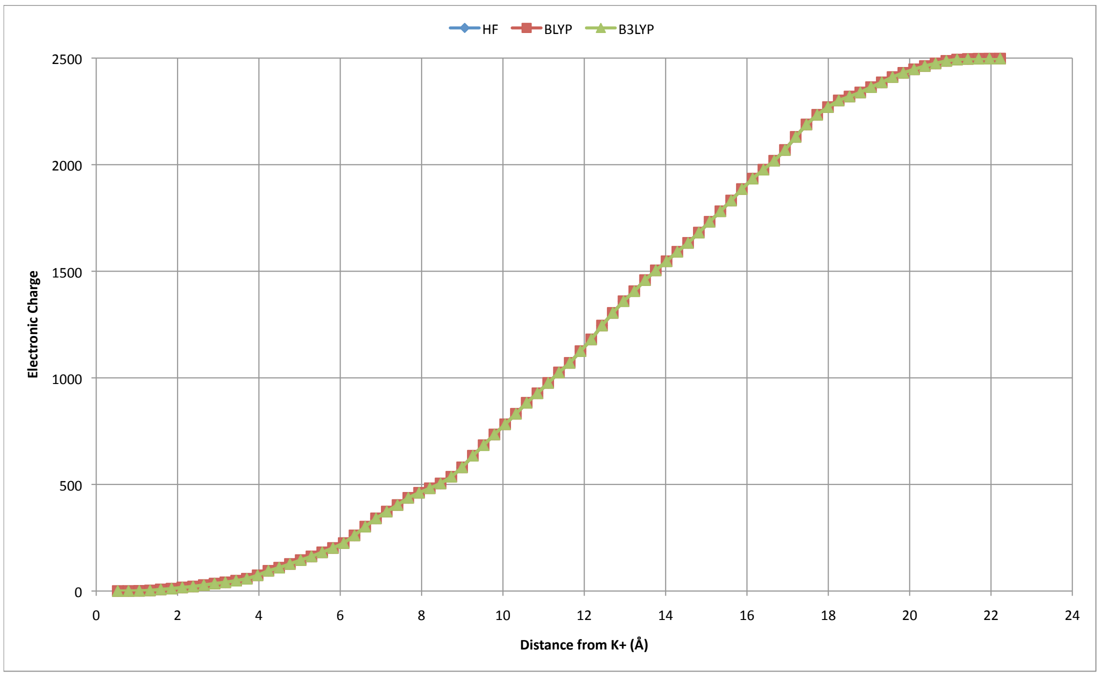


Figure 6.7: Spherical integration of k300c-12 electron density obtained using a 6-31G* basis set for the HF, BLYP and B3LYP methods.

6.8 Charge on K+ as a Function of Different Basis Sets

In the previous sections we explored possible physical causes for charge variation and analyzed the density between those obtained from the HF, BLYP and B3LYP methods for the water cluster complexes. These showed that physical effects such as system geometry did not play a role in the dramatic reduction of charge on the K+ ion. Also, there were no meaningful variations in electron density obtained from the HF, BLYP and B3LYP methods.

In this section, we assess the effect of using different basis sets for the charge obtained on the K+ ion. The k300c system is used as an exemplar since, as in previous sections, results for the various k300{a,b,c} snapshots showed similar trends. Two other reasons for choosing k300c-08 are (a) it is the first snapshot in the k300c series which exhibits negative charges when using DFT and (b) it is small enough to perform a variational study w.r.t. change of basis set.

The charges on the K+ ion in the k300c system computed using MPA and various basis sets are given in Table 6.8. The first set of results are for k300c-08 and the second set are for k300c-12 using just two basis sets. The first column in the table, lists the basis sets used. There are three subsequent columns which give the electronic energy and charge observed on the K+ ion using the HF, BLYP and B3LYP methods.

Three types of 6-31G* style basis sets are used – 6-31G*, 6-31G*(5D) and 6-311++G(3df,3pd). The (5D) in 6-31G*(5D) denotes 5 pure d functions rather than the 6 Cartesian d functions used by the 6-31G* basis set . The rationale for doing this was to test whether the additional "s" contaminant function that is present in the in the 6-d case is responsible for the spurious results. The 6-311++G(3df,3pd) basis set has pure spherical diffuse and polarization functions [82, 111]. The cc-pVDZ and cc-pVTZ are Dunning style basis sets which are correlation consistent, polarized valence double zeta (cc-pVDZ) and triple zeta (cc-pVTZ) sets [277]. The TZVP basis set [235] have a single contraction to define inner shells, and three basis functions describing the valence shells. The def2-TZVP basis set was specifically designed for "quantitatively accurate DFT treatments" [81].

It is to be noted that NPA charges for the k300c-08/TZVP and k300c-12/def2-TZVP could not be obtained as NPA analysis, it was later found, expects 'd' functions to be present in all third-row elements. The two basis sets used for the potassium ion did not incorporate this and this resulted in Gaussian giving an error message – "`'Error in SR CORTBL'`", in a subroutine which determines the number of sub-shells for core orbitals.

The results show that the charge on K+ using MPA is positive for the three 6-31G* style basis sets using the HF method, although it reduces as the number of basis functions increases. For BLYP the MPA method yields negative charges for these basis sets. The B3LYP MPA charges are very small and become negative for the 6-311++G(3df,3pd) case. NPA charges for HF, BLYP and B3LYP using the three 6-31G* style basis sets are positive. The NPA charges are observed to reduce as the number of basis functions increase.

Using the cc-pVDZ and cc-pVTZ basis sets, the MPA charges are now positive for BLYP and B3LYP. NPA charges for BLYP are less than those obtained for B3LYP; HF NPA charges are the largest amongst the three methods.

Use of the TZVP basis set results in positive MPA charges for all methods. The use of def2-TZVP gives positive charges for the MPA and NPA charges across all methods for the k300c-08 snapshot. The absolute values obtained by the def2-TZVP basis set also seems reasonable.

Given that def2-TZVP gives reasonable results for the k300c-08 snapshot, a set of calculations were performed using the k300c-12 snapshot. The def2-TZVP results for k300c-12 obtained from HF, BLYP and B3LYP methods are all positive which is not the case when using the 6-31G* basis.

In summary the results show that when using the def2-TZVP basis set with the systems studied here, meaningful MPA charges can be obtained for DFT methods.

Table 6.2: Charge on K+ for the k300c-08 system using eight basis sets for the HF, BLYP and B3LYP methods. Charge on K+ for the k300c-12 system using 6-31G* and def2-TZVP are also given.

		HF			BLYP			B3LYP		
k300c-08	# Basis Fn.	Energy	MPA	NPA	Energy	MPA	NPA	Energy	MPA	NPA
6-31G*	1543	-6680.63	0.4071	0.9551	-6711.89	-0.0748	0.8239	-6713.60	0.0723	0.8612
6-31G*(5D)	1462	-6680.54	0.3907	0.8943	-6711.76	-0.0811	0.7410	-6713.48	0.0642	0.7968
6-311++G(3df,3pd)	6060	-6684.17	0.3091	0.7409	-6716.04	-0.1966	0.6668	-6717.54	-0.0419	0.6907
cc-pVDZ†	1951	-6682.03	0.5467	0.8265	-6712.97	0.1984	0.6843	-6714.74	0.2931	0.7278
cc-pVTZ‡	4703	-6684.17	0.6168	0.7543	-6715.92	0.3146	0.6209	-6717.47	0.3914	0.6612
TZVP	2498	-6681.56	0.9748	–	-6715.89	0.8335	–	-6715.24	0.9441	–
def2-TZVP⋆	3473	-6684.24	0.9246	0.7425	-6716.02	0.7383	0.6344	-6717.55	0.7921	0.6677
k300a-12										
6-31G*	5039	-20668.67	0.3757	0.9496	-20768.99	-0.1317	0.8450	-20775.86	0.0234	0.88042
def2-TZVP	11385	-20680.38	0.9245	–	-20788.67	0.7900	–	-20783.63	0.7353	–

† The cc-pVDZ basis set available in Gaussian does not define K. In this calculation, the basis set used for K was Feller Misc. CVDZ from [207]

‡ The cc-pVDZ basis set available in Gaussian does not define K. In this calculation, the basis set used for K was Feller Misc. CVTZ from [207]

⋆ def2-TZVP basis set obtained from [207]

6.9 Previous Work

The major objective of this chapter was to explore quantitatively the issue of negative charges on the K+ ion, that were obtained using DFT methods and the 6-31G* basis sets for large water cluster systems. While we were unable to find explicit mention of this problem in literature, the following two items of previous work are relevant.

Martin and Zipse [174] present charge analysis for a single water molecule using a variety of basis sets, four methods (HF, B3LYP, MP2[3], and QCISD[4]), and six population analysis methods (Mulliken, NPA, AIM, CHELPG, Merz-Kollman, and Resp). They find that charges for the water molecule are best reproduced by use of electrostatic potential (ESP) methods[5]. To achieve good results a Dunning style correlation consistent basis set and a correlated method (B3LYP, MP2 and QCISD) are also required. (Correlation here refers to the explicit treatment of electron-electron motion as a function of its spin [55, 83]). As we consider very large water cluster systems in this chapter, the use of B3LYP is attractive, as it is not as computationally intensive as MP2 or QCSID. Consistent results were obtained using a def2-TVZP basis set for the B3LYP method.

Tanaka and Aida [273] present an analysis of the orbital interaction between a K+ ion and the surrounding solvent water molecules using a quantum mechanics/molecular mechanics (QM/MM) method. A total of 171 water molecules were used. Water molecules were clustered in shells around a central K+ ion. The K+ ion was treated computationally using the HF method and a 6-31G* basis set. The MM part, essentially all the solvent molecules, was handled using a TIP3P MM force-field [139]. Charge distributions were computed using the Mulliken, NPA and CHELPG [36] charge analysis methods. In their analysis of charges, they conclude that the NPA method does not give reasonable charge distributions in molecular clusters. It was found that the influence of the K+ orbitals reached as far as 6 Å to 8 Å from the K atom (cf. Section 6.5). Unlike Tanaka and Aida, in this chapter, we have used QM methods applied to a maximum of 264 water molecules (cf. k300a-12, Appendix A.9). Results presented here for variation of relative charge and for variation of charge as a function of the RDF indicates the influence of MPA charges is upto a maximum distance of 6 Å (cf. Figure 6.4, 6.5) . We also observed that NPA charges did not give a realistic distribution for the water cluster systems considered.

[3]MP2 is a post-HF method, which builds on HF theory by incorporating electron correlation effects by using Rayleigh-Schrödinger perturbation theory [55, 83, 112].

[4]Quadratic Configuration Interaction Singles and Doubles (QCISD) is a post-HF method which is a type of configuration interaction [112].

[5]In ESP methods, atomic charges are fitted to reproduce the electrostatic potential for a number of points around a molecule. This technique does not work for very large systems.

6.10 Conclusions

In this chapter quantum chemical charge analysis was performed on a series of MD snapshots. These snapshots comprised a potassium ion which is surrounded, concentrically by water molecules at distances ranging from 3Å to 12Å . The systems were labeled k300-{a,b,c} and k345a, and snapshots were generated at two temperatures (300 and 345 Kelvin). Using these snapshots, trends observed for the charge on the K+ ion using the HF, BLYP and B3LYP methods and the 6-31G* basis set were presented. These trends follow those observed by Bliznyuk and Rendell [28] using MPA, where the charge the K+ ion was found to be dramatically reduced.

In addition to using MPA, the NPA technique was used to obtain charges. When using DFT methods, for increasing water cluster sizes, a negative charge on the K+ ion was obtained using MPA, but NPA did not produce such negative charges.

Following this the distribution of charge around the K+ was examined using a 'Percentage Relative Charge' measure. Results for the HF, BLYP and B3LYP methods using MPA showed that the water molecules within 5Å of the K+ ion were able to affect the movement of charge away from the K+ ion due to the electronegative Oxygen atoms in the water molecules. NPA results for this displayed similar trends (upto 3 Å), but had large variations between the results from the HF, BLYP and B3LYP methods.

The radial distribution functions for all the snapshots were computed and charges on the water molecules were then binned as a function of distance from the K+ ion. The RDF results for the MPA charges obtained from the HF, BLYP and B3LYP methods showed good correlation with the RDF peaks. This conforms with the expectation that the charge distribution within the water cluster will be a function of the RDF. This correlation was not observed using NPA charges and it was posited that this is due to the process used to determine the NPA charges i.e. the electron density is localized onto the individual atoms. Thus even though NPA assigns a positive charge to the K+ ion, which is in line with expectation, it does not produce physically consistent results for the charge distribution in large water cluster systems.

We investigated electron densities obtained from the HF, BLYP and B3LYP methods. This was examined with the expectation that variations in density, produced by different methods, led to negative charges being computed for DFT methods when using MPA. The electronic charge was determined as a function of distance from the K+ ion, using spherical integration. This in turn showed that the electron densities produced by the HF, BLYP and B3LYP methods were near identical. This observation led to the final set of experiments which involved the use of a moderately large system (k300c-08) and a parametric variation of the basis set.

Results using six different basis sets were presented, these showed that the use of double and triple zeta with valence polarization led to positive charges on K+ when using DFT methods and MPA. Particularly, the use of the def2-TZVP basis set, which was specifically designed for DFT, gave the best results.

A large water cluster system was then considered (k300c-12). Using the def2-TZVP basis set this also gave reasonable results for MPA populations with DFT methods.

In conclusion, it appears that chemically meaningful charge populations can be obtained for large water cluster systems using DFT methods, if an appropriate basis set like def2-TZVP is employed. The MPA method, while splitting equally density which is shared by pairs of basis functions, does produce charge distributions that correlate well with the location of solvation shells in large water cluster systems.

Chapter 7

Conclusions and Future Work

The primary objective of this thesis has been the creation, validation and use of performance models for electronic structure methods on modern computer architectures. Performance analysis of complex, parallel, scientific codes like electronic structure methods are complicated by the fact that these application codes often execute on complex, multi-core, NUMA systems.

This chapter briefly summarizes the contributions made in this thesis and provides a few directions for future work.

7.1 Summary of Contributions

This thesis has made the following contributions:

(a) Creation of software tools to characterize NUMA architectures i.e. a cross-platform thread and memory placement infrastructure and a placement distribution model to characterize experiments

(a.1) A defined set of basic performance metrics for any NUMA platform in terms of latency and bandwidth benchmarks. Experimental protocols that aid in thread and memory placement experiments using thread and memory placement APIs

(a.2) Creation and validation of the Placement Distribution Model (PDM) to characterise latency and bandwidth experiments without resorting to an exhaustive set of thread and memory placement experiments

(b) Performance model for ERI evaluation

(b.1) In-depth characterization of ERI evaluation using PRISM was presented in terms of platform dependent and independent components.

(b.2) A methodology for assessing platform specific cache blocking factors for PRISM.

(b.2) Validation of a Linear Performance Model (LPM) using seven microprocessor platforms and four test molecular systems.

(b.3) Use of the LPM in combination with functional cache simulation

(b.3.1) Validation of the functional cache simulation counts with those obtained from hardware performance counters

(b.3.2) Use of the LPM to characterise cache miss behaviour which affects PRISM's scalability: L1 read misses

(b.3.3) Use of the LPM to identify key cache parameters which influence performance: L1 linesize and L2 size

(c) Studying the effect of thread and memory placement within Gaussian

(c.1) Performance characterisation of Gaussian in terms of thread, memory placement and its subsequent influence on cache behavior

(c.2) Extending and validating the LPM to account for NUMA, multi-threaded effects

(c.3) Use of dynamic page migration and node interleaving improves Gaussian's performance

(d) A comparative study of charges for a set of water cluster complexes

(d.1) Characterization and analysis of atomic charges obtained from density functional methods

(d.2) Demonstration of basis set sensitivity for atomic charges computed using density functional wavefunctions

7.2 Discussion

In this thesis, a body of work was presented which aimed at creating tools (in terms of APIs) and models which aid in performance modelling of electronic structure codes.

An observation to be made, from work done here, is that at this time sophisticated scientific codes need to account for the underlying platform characteristics on which they execute, rather than relying on compiler technology or operating systems to affect thread and memory placement decisions. A key reason, to be aware of the underlying hardware environment, is so that the application developer is cognizant of the hardware environment the code executes on i.e. the developer does not code against an abstract programming model which makes implicit simplifications of what the target hardware would be. This in turn reflects the need for programming models like OpenMP to provide information about hardware elements that affect performance e.g. cache sizes, cache hierarchies and linesizes, physical layout of processor and memory locations as well as the locality information that affects memory placement decisions from within the application code. In effect this would allow a developer to reason about the target platform onto which the code eventually runs. One could argue that the use of domain specific languages would make the job of executing complex scientific applications an exercise in runtime environment construction i.e. an end-user interacts a Mathematica or Matlab like tool to define and execute their problem. The techniques and performance modelling discussed in this thesis could potentially aid implementors of such runtime systems but importantly it would aid developers who are either developing new code using contemporary languages or maintaining large, evolving codes like Gaussian.

The work done here also shows that judicious use of thread, memory placement and appropriate cache blocking of key application data structures plays a crucial part in obtaining good performance from modern computer architectures.

The applications area work performed on large water cluster systems identified that for large systems, for which DFT methods are competitive, it is essential that a basis set like def2-TZVP is used in order to obtain meaningful charge results.

7.3 Future Work

Contributions arising from this thesis lead to further questions that remain unanswered. The following avenues for future work are highlighted.

The implementation of the PDM qualitatively accounts for interconnect contention. It would be beneficial to incorporate a detailed contention model as it would aid it reducing errors in the PDM.

The LPM uses instruction counts and Level 2 misses to derive the PPCoeffs and ultimately the Cycle count. As microprocessors have numerous performance counter events, it might be beneficial to design a fitting scheme which can evaluate various counter events and synthesize a set of PPCoeffs which better reproduces Cycle counts.

An in-depth study into the interaction between PRISM's floating-point workload and the L1 data cache needs to be carried out, as results obtained in the thesis show that while PRISM is cache blocked for the L2 cache, as it is not being cache blocked for L1 the utilization of on-chip floating point units is being hampered.

The LPM does not incorporate the L1 data cache PPCoeff (β) nor does it take into account the effects of hardware prefetch. As part of future work, avenues for incorporating β and hardware prefetch into the LPM should be pursued.

In order to affect runtime thread and memory placement decisions, it would be useful to incorporate the LPM within a feedback loop in the Gaussian code i.e. a larger set of PPCoeffs could be obtained at runtime and this could be used by the code in guiding thread, memory placement and the appropriate cache blocking factor for a given input molecule, computational method and basis set.

The range of hardware platforms used to evaluate the LPM should be expanded to cover multi-core, heterogenous and novel architectures (GPGPUs, the Cell processor and embedded systems (Intel's Atom, IBM PowerPC 440 and the nVidia Tegra)).

The Valgrind/Callgrind tool set could be extended to include a memory model which can account for NUMA in its functional cache simulation.

It would be useful to consider the role of High Productivity Computer Systems (HPCS) languages like X10 [26] to implement electronic structure methods. Using such languages it may be possible to give high-level hints that a runtime system for the HPCS language could use.

Appendix

A.1 PAPI native hardware performance counter events

Table A.1: PAPI native hardware performance counter events

Processor	Instr Count	L1 Misses	L2 Misses	FLOPS	Cycles
Opteron Athlon	FR_X86_INS	1) IC_L2_REFILL 2) DC_L2_REFILL_MOES 3) DC_SYS_REFILL_MOES 4) IC_SYS_REFILL	1) DC_SYS_REFILL_MOES 2) IC_SYS_REFILL	FP_MULT_AND_ADD_PIPE	CPU_CLK_UNHALTED
EM64T Pentium 4	inst_retired- _NBOGUSNTAG- _NBOGUSTAG	1) BPU_fetch_request_TCMISS 2) replay_event- _NBOGUS_PEBS with load, store and L1 bits set	BSQ_cache_reference_RD_- _MISS_WR_- 2ndL_MISS	1) execution_event_NBOGUS1 2) scalar_DP_uop_TAG1_ALL 3) x87_FP_uop_TAG1_ALL	global_power- _events_RUNNING
Pentium M	INST_RETIRED	L2_RQSTS_MESI	L2_TOT_LINES_IN	FLOPS	CPU_CLK_UNHALTED
G5 G5-XServe	PM_INST_CMPL	PM_LD_MISS_L1 PM_ST_MISS_L1	PM_DATA_FROM_MEM	PM_FPU_ALL PM_FPU_FMA	PM_CYC

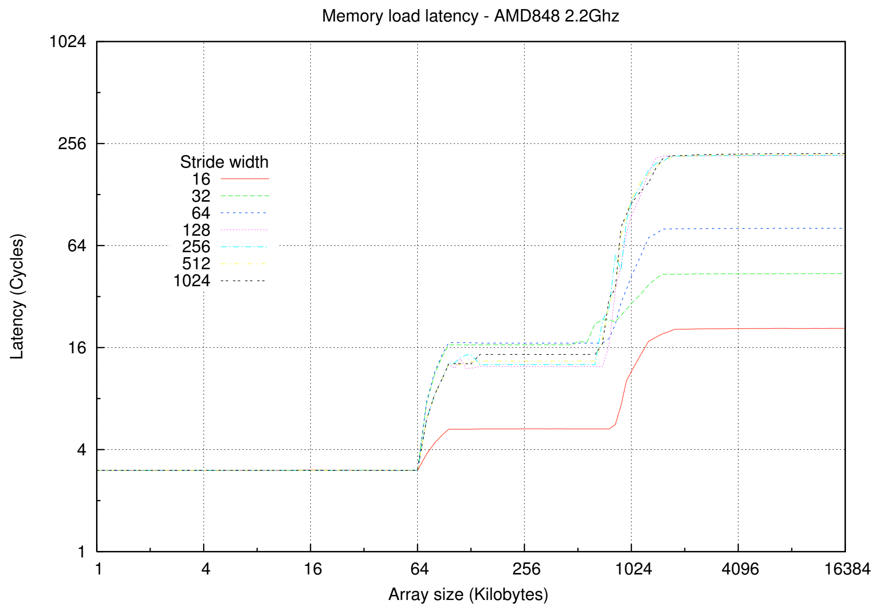
A.2 `lmbench` Plots

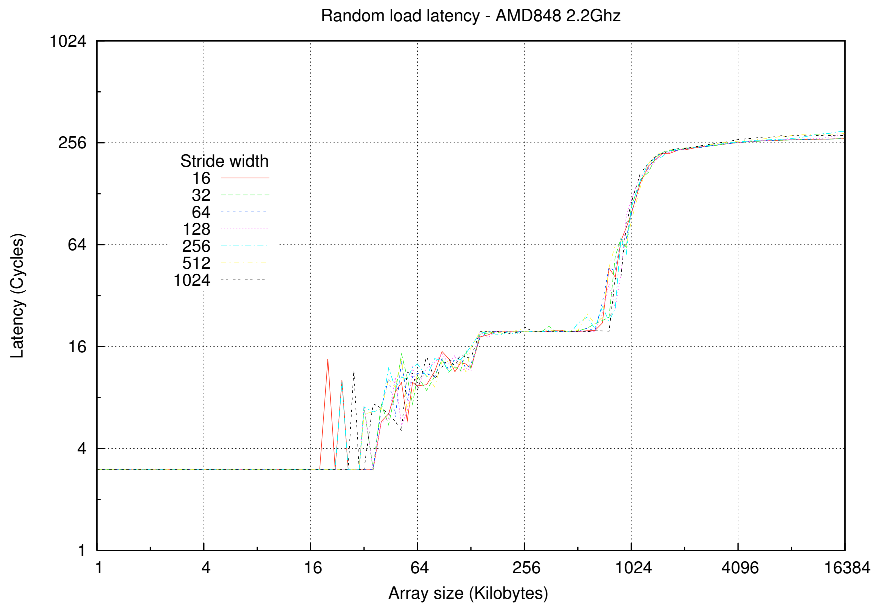
A.2.1 AMD Opteron 848

AMD Opteron/HyperTransport

The Opteron [146] system used contains four 2.2Ghz AMD848 processors each with a 64 Kb L1 data and instruction cache and a 1024 Kb L2 cache. The Celestica A8440 motherboard [47] is configured with 2GB of memory per processor giving a total of 8GB for the entire system. The AMD848 Opterons have an on-chip memory controller and uses coherent HyperTransport to link processor coherency traffic. The Opteron has two coherent HyperTransport links, each operating at 6.4 GB/s bi-directionally. The processors are arranged in a ring topology resulting in processors having at most two hops to reach the most distant processor.

Table A.2: AMD Opteron 848

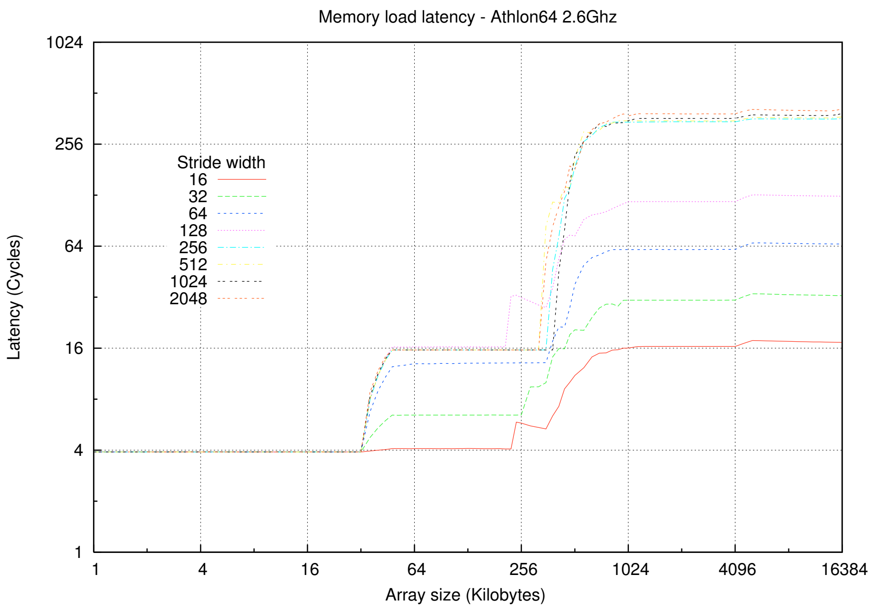


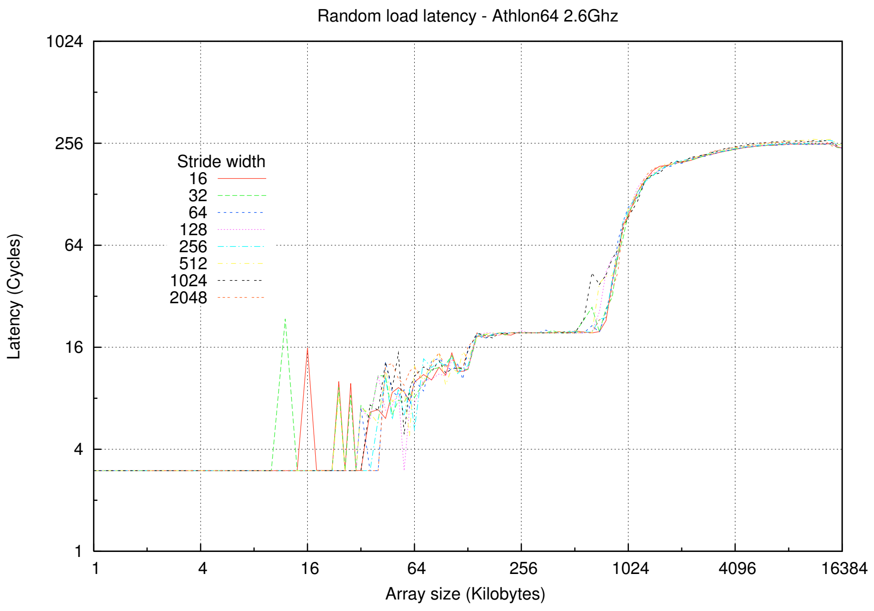


A.2.2 AMD Athlon64

The Athlon64 system was based on a commodity nVidia nForce motherboard.

Table A.3: Athlon64

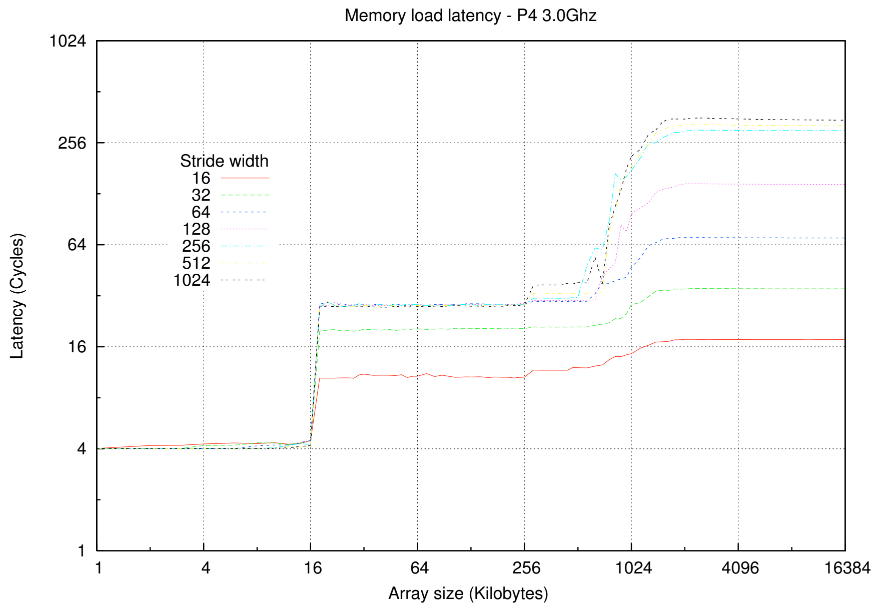


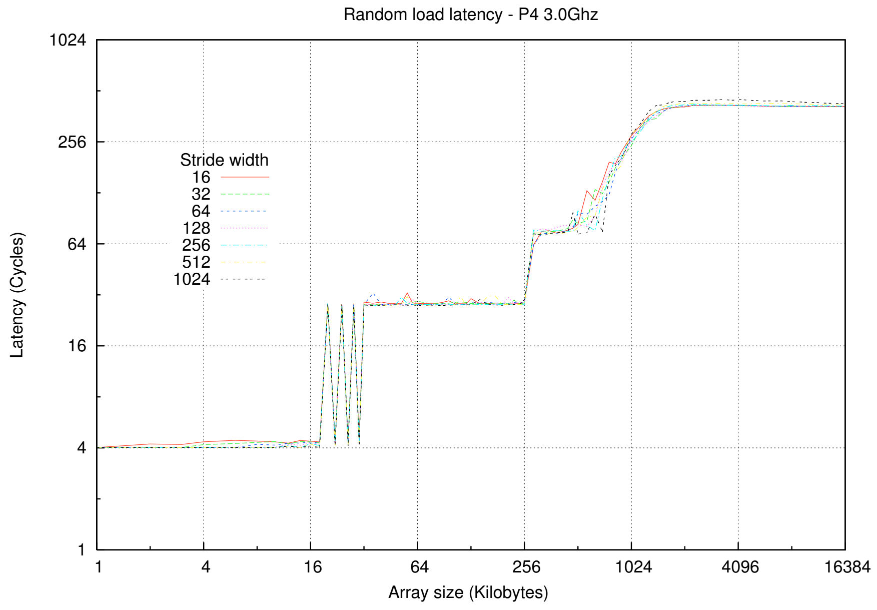


A.2.3 Intel NetBurst P4

The Intel Pentium M [245] and P4 [87] microprocessors used in this study were on commodity hardware (Pentium M: Dell Latitude D800; P4: Dell PowerEdge 2850). The systems use a Northbridge to link the microprocessors to memory.

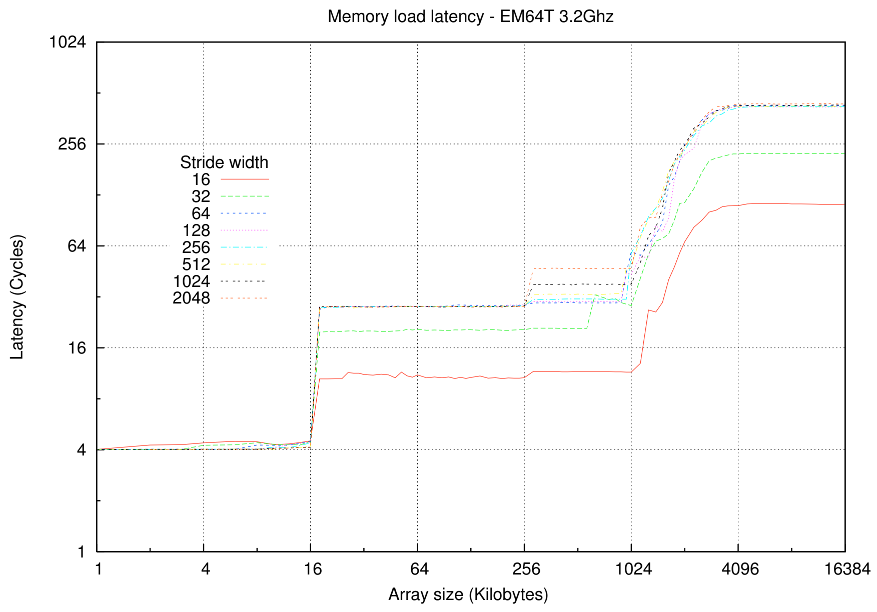
Table A.4: NetBurst - P4

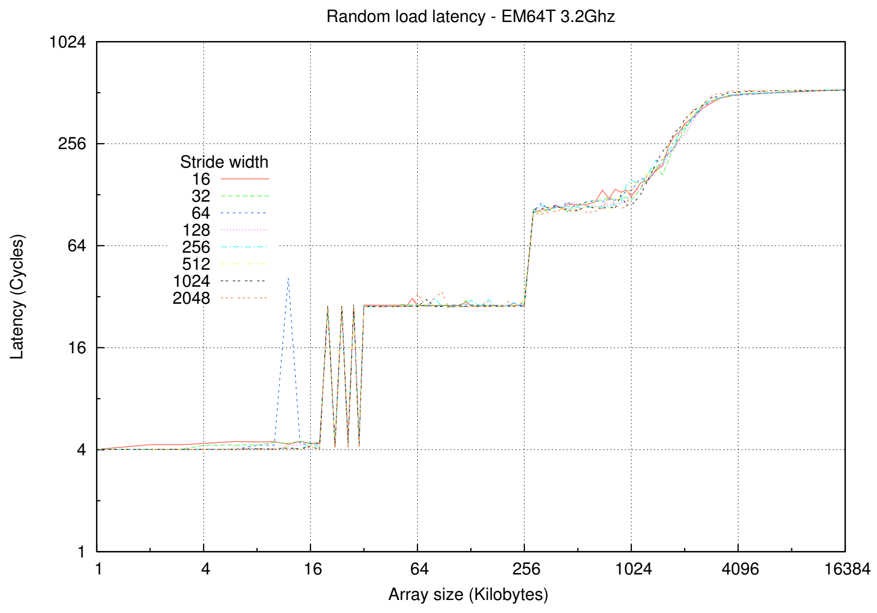




A.2.4 Intel NetBurst EM64T

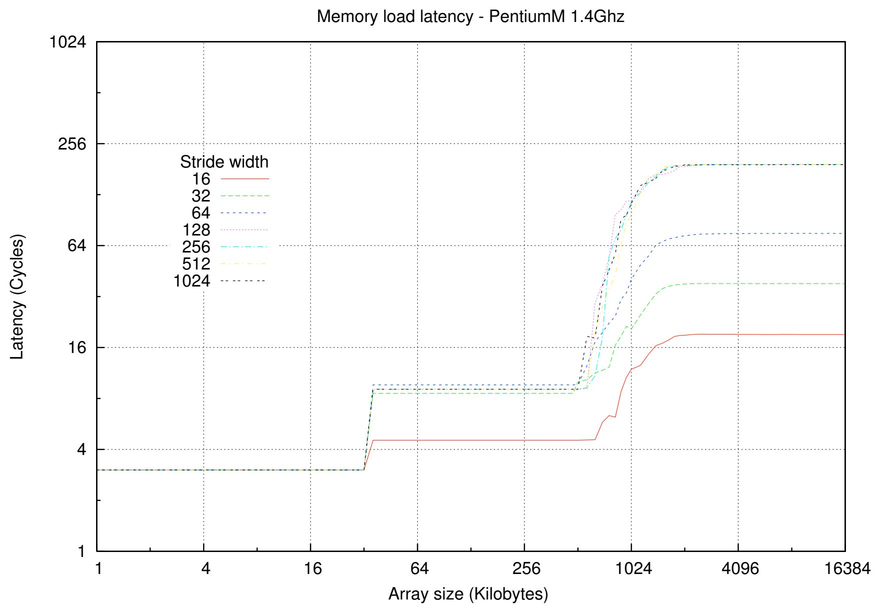
Table A.5: NetBurst - EM64T

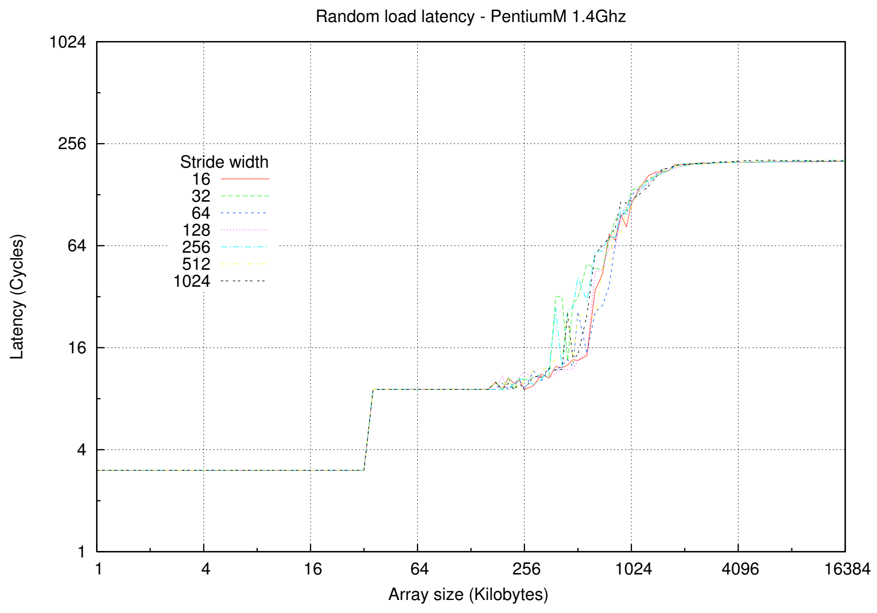




A.2.5 Intel Pentium M

Table A.6: Pentium M

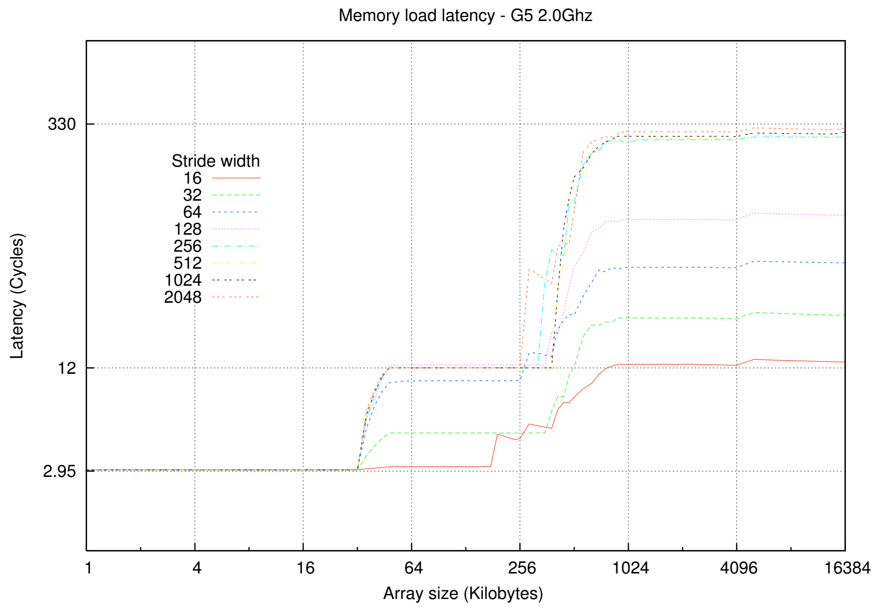


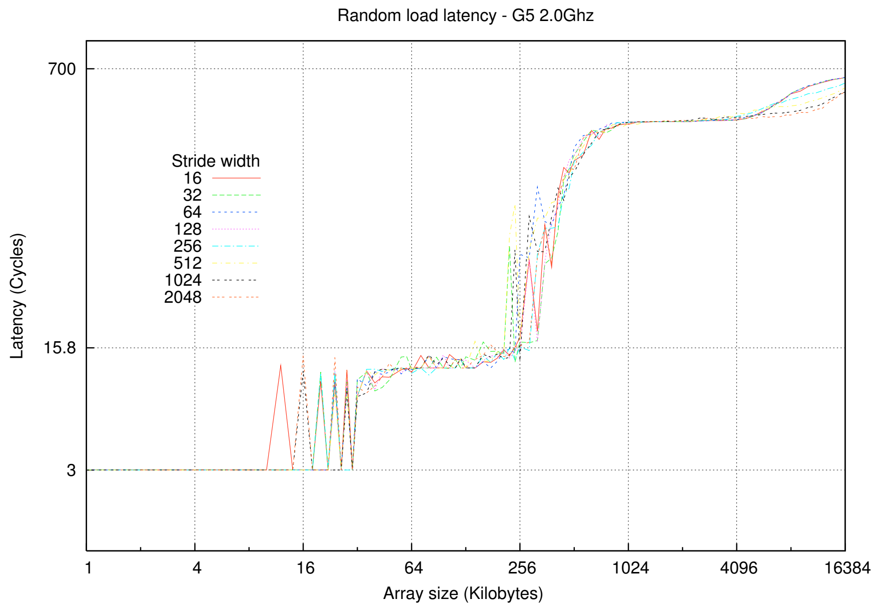


A.2.6 IBM G5/PPC970Fx

An Apple XServe G5 [16, 247] was used as a platform for the G5/PPC970Fx microprocessor. The XServe is a two socket system which uses Apples U3H Northbridge bus [247] to link the two G5s via HyperTransport to memory and I/O devices.

Table A.7: G5



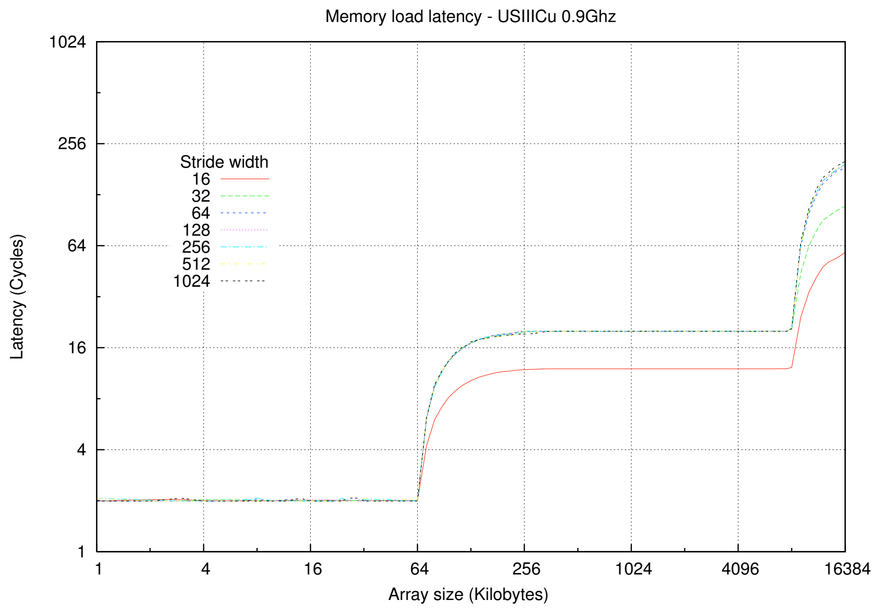


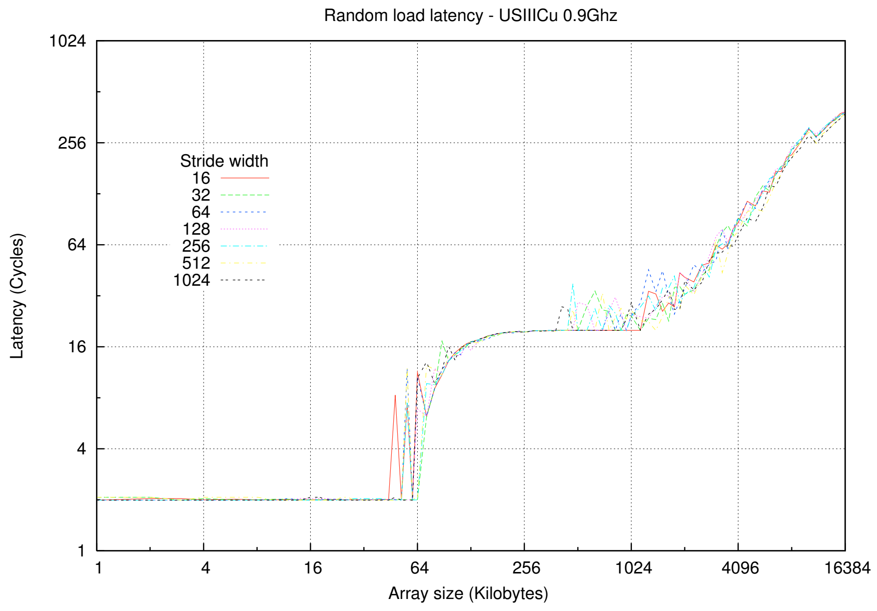
A.2.7 Sun UltraSPARC IIICu

A SunFire V1280 system [266] was used. It has twelve 900MHz UltraSPARC III Cu processors [119, 267] each with a 32Kb L1 instruction cache, 64 Kb L1 data cache and 8192 Kb L2 cache which is off-chip. The system contains three boards that contain four processors each and are joined by the FirePlane [50] interconnect. The system contains 8GB of memory per board giving a total of 24GB for the entire system. The three boards form a combined snooping based coherency domain. For larger systems, i.e. >24 processors, a directory based protocol is used at the point-to-point level.

A pair of processors and their associated memories are all linked using a Dual CPU Data Switch (DCDS), i.e. there are four separate data paths each running at 2.4GB/s from processors or memories to the DCDS. A processor can directly access its own local memory for addresses, but needs to use the DCDS to access data from its local memory or from the memory associated with the second processor. There are two DCDS per board, that are linked to each other using a board data switch. The DCDSs can sustain 4.8 GB/s to the board data switch. Since memory on the boards is 16-way interleaved across a board, a peak of 6.4 GB/s per board is achieved. The point-to-point links among boards have a bi-directional bandwidth of 4.8 GB/s per board, approaching a peak of 9.6 GB/s for the whole system. Since the four processors on a board have the same memory access latencies, it is referred to as one node.

Table A.8: UltraSPARC IIICu





A.3 Water Clusters Used in this Thesis

Table A.9: k300a Water Clusters; 6-31G* basis set

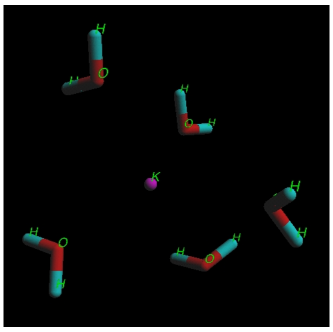
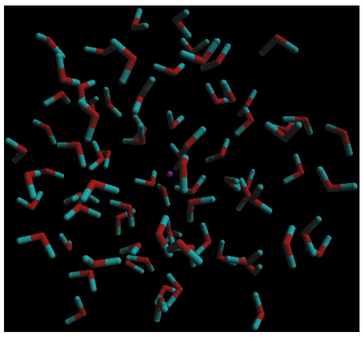
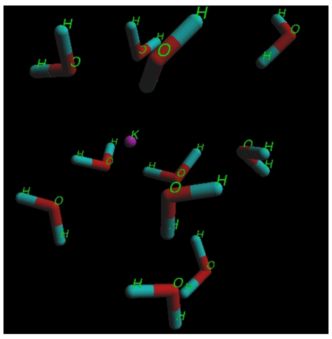
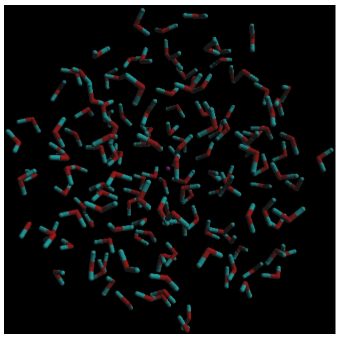
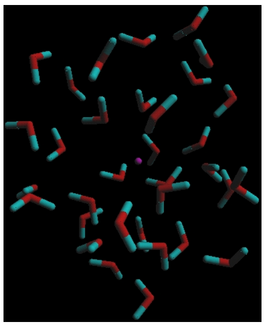
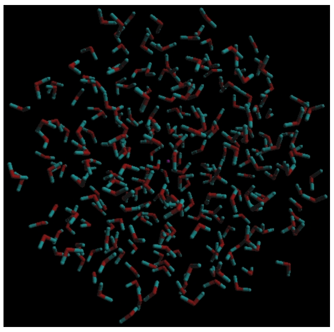
	k300a-03 NAtoms 17 NBasis 118 NWaters 5		k300a-08 NAtoms 454 NBasis 1543 NWaters 80
	k300a-04 NAtoms 35 NBasis 232 NWaters 11		k300a-10 NAtoms 455 NBasis 2892 NWaters 151
	k300a-06 NAtoms 101 NBasis 650 NWaters 33		k300a-12 NAtoms 794 NBasis 5039 NWaters 264

Table A.10: k300b Water Clusters; 6-31G* basis set

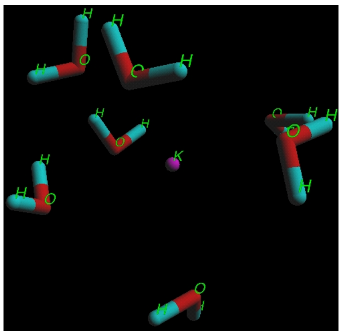
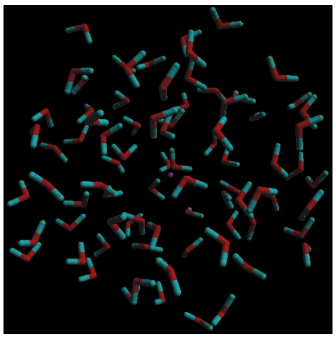
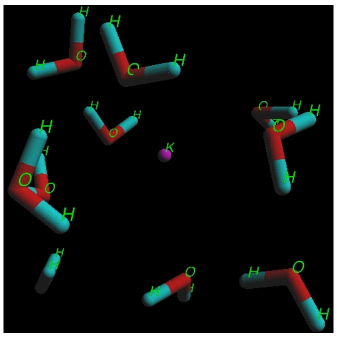
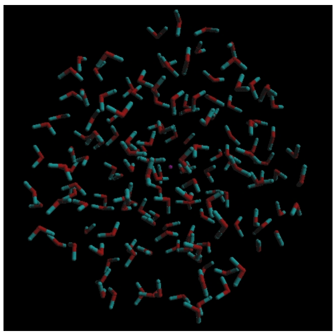
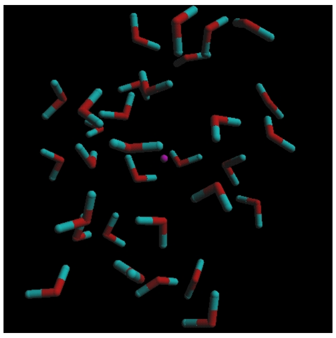
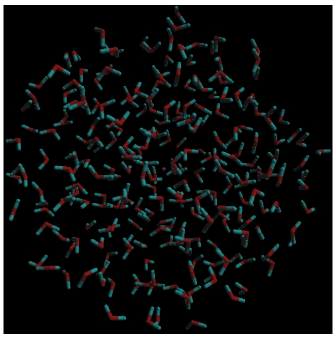
	k300b-03 NAtoms 23 NBasis 156 NWaters 7		k300b-08 NAtoms 245 NBasis 1562 NWaters 81
	k300b-04 NAtoms 32 NBasis 213 NWaters 10		k300b-10 NAtoms 458 NBasis 2911 NWaters 152
	k300b-06 NAtoms 95 NBasis 612 NWaters 31		k300b-12 NAtoms 740 NBasis 4697 NWaters 246

Table A.11: k300c Water Clusters; 6-31G* basis set

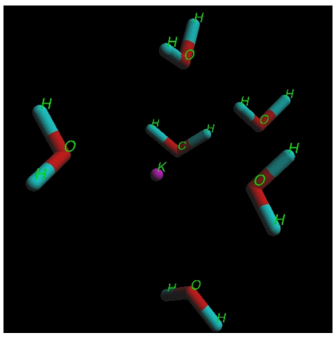
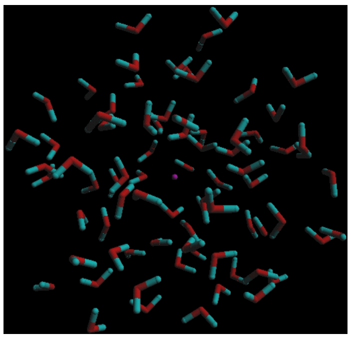
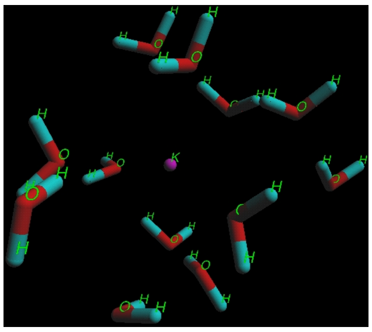
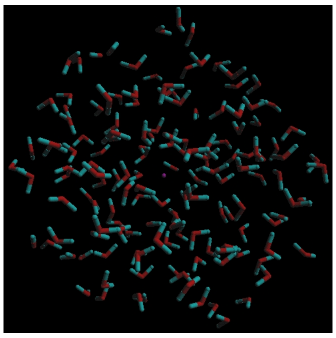
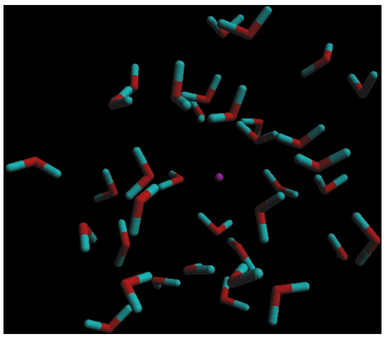
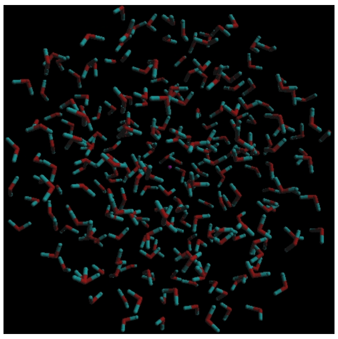
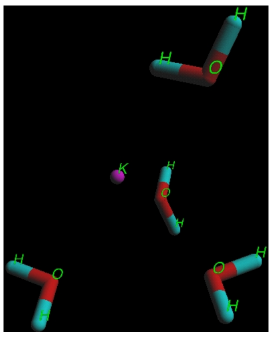
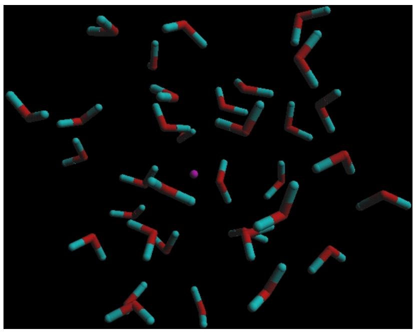
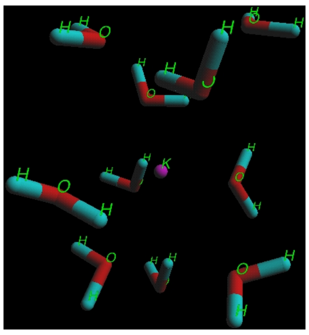
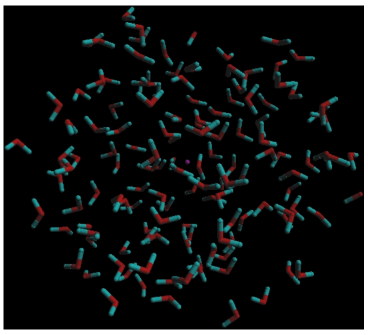
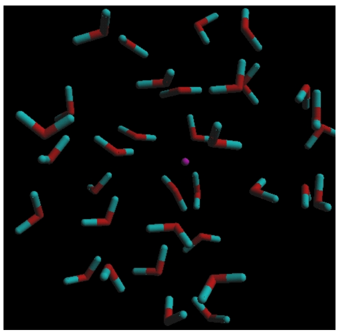
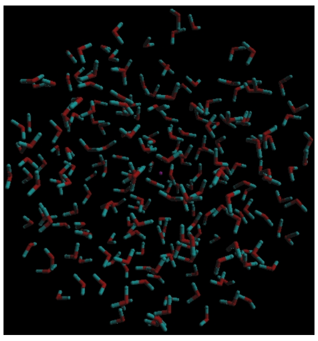
	k300c-03 NAtoms 20 NBasis 137 NWaters 6		k300c-08 NAtoms 233 NBasis 1486 NWaters 77
	k300c-04 NAtoms 38 NBasis 251 NWaters 12		k300c-10 NAtoms 443 NBasis 2816 NWaters 147
	k300c-06 NAtoms 107 NBasis 688 NWaters 35		k300c-12 NAtoms 746 NBasis 4735 NWaters 248

Table A.12: k345a Water Clusters; 6-31G* basis set

	k345a-03			k345a-08	
	NAtoms	14		NAtoms	221
	NBasis	99		NBasis	1410
	NWaters	4		NWaters	73
	k345a-04			k345a-10	
	NAtoms	32		NAtoms	398
	NBasis	213		NBasis	2531
	NWaters	10		NWaters	132
	k345a-06			k345a-12	
	NAtoms	104		NAtoms	665
	NBasis	669		NBasis	4222
	NWaters	34		NWaters	221

A.4 Python code used for generating the pair-wise RDF

Code Listing A.1: Generate a pair-wise radial distribution function

```python
#============================================
# generate the distance histogram and RDF
#============================================
def getDistanceHistogram(store, atomID, binWidth):
    # input args:
    # -store- is a dictionary of datastructures
    # -atomID- corresponds to the atom of interest e.g. Oxygen
    # -binWidth- is the spacing between bins

    # get the keys from the dictionary
    tmpKeys = store.keys()

    # initialize tuples for distance and charge
    distances = []
    charges = []

    # what is the largest distance?
    for i in tmpKeys:

        # store[i]['E'] holds the elementID i.e.
        # atomic number of the atom
        if (store[i]['E'][0] == atomID):

            # store[i]['Distance'] holds the distance
            # between this atom and the K+ ion
            tmpVal = float(store[i]['Distance'])

            # add this atom's 'Distance' to
            # the distance tuple
            distances.append(tmpVal)
        else:
            # this is an atom we are not interested in
            del(store[i])

    # use Numeric Python
    import Numeric

    # what is the maximum and minimum distances?
    distMax = Numeric.maximum.reduce(distances)
    distMin = Numeric.minimum.reduce(distances)
```

```
# number of bins
numBins = int(distMax/binWidth)

# create histograms for the pair-wise RDF
# define bins and initialize to zero

# bins to accumulate distances into
distBins = Numeric.zeros(numBins)

# initialize another bin to Floating point zeros
# this it to hold the actual distance from K+
# for a given bin
tmpDistBins = Numeric.zeros(numBins, Numeric.Float)

# how many elements in the distance tuple?
numPoints = len(distances)

# index into bins then accumulate distances
# first, accumulate distances between atomID atoms and
# K+ into distBins
for i in distances:
    indx = int((i/binWidth)) - 1
    distBins[indx] = distBins[indx] + 1

# second, initialize tmpDistBins to hold the actual
# distances i.e. binWidth, 2*binWidth, 3*binWidth ...
for i in xrange(0, numBins):
    if (i == 0):
        tmpDistBins[i] =  binWidth
    else:
        tmpDistBins[i] =  tmpDistBins[i-1] + binWidth

# third, create a 2D matrix having distances and
# populations
hist = Numeric.transpose((tmpDistBins, distBins))

# fourth, compute the pair-wise
# Radial Distribution Function
#
# Computing the RDF involves finding particles which are
# confined in within a spherical shell of
# thickness r + dr
#
# Volume of a Sphere of radius distMax
volume = (4/3)*Numeric.pi*(distMax)**3
```

```
    # number of atoms
    # i.e number of entries in the distances tuple
    numAtoms = len(distances)

    # Calculate the RDF
    # hist[:,1] refers to elements belonging to distBins
    # hist[:,2] refers to elements belonging to tmpDistBins
    #
    # 4 * Pi * hist[:,0]**2 i.e 4 * Pi * r^2
    #  is the volume of the spherical shell

    hist[:,1] = hist[:,1] * volume      \
                /                       \
                (4 * Numeric.pi         \
                   * hist[:,0]**2       \
                   * binWidth           \
                   * numAtoms)


    # hist now has distances for each bin in
    # column 1 and the RDF g(r) in column 2
    return hist
```

Bibliography

[1] A. Snavely and N. Wolter and L. Carrington, *Modelling Application Performance by Convolving Machine Signatures with Application Profiles*, IEEE Workshop on Workload Characterization, (2001).

[2] R. D. Adamson, *Novel Methods for Large Molecules in Quantum Chemistry*, PhD in Chemistry, Trinity College, Cambridge University, 1998.

[3] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, Santa Clara, California, USA, April 2003. Publication Number: 24594.

[4] ——, *Software Optimization Guide for AMD64 Processors*, Advanced Micro Devices, Santa Clara, California, USA, September 2005. Version 3.06.

[5] ——, *AMD64 Architecture Programmer's Manual*, vol. 2, Advanced Micro Devices, Santa Clara, California, USA, September 2007. Version 3.14.

[6] S. V. Adve, V. S. Pai, and P. Ranganathan, *Recent advances in memory consistency models for hardware shared memory systems*, Proceedings of the IEEE, 87 (1999), pp. 445–455.

[7] D. H. Ahn and J. S. Vetter, *Scalable analysis techniques for microprocessor performance counter metrics*, in Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, 2002, IEEE Computer Society Press, pp. 1–16.

[8] T. W. Ainsworth and T. M. Pinkston, *Characterizing the Cell EIB On-Chip Network*, IEEE Micro, 27 (2007), pp. 6–14.

[9] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, Clarendon Press, New York, NY, USA, 1989.

[10] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' guide (third ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[11] D. Andrade, B. B. Fraguela, and R. Doallo, *Precise automatable analytical modeling of the cache behavior of codes with indirections*, ACM Trans. Archit. Code Optim., 4 (2007), p. 16.

[12] J. Antony, P. P. Janes, and A. P. Rendell, *Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport*, in HiPC, Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, eds., vol. 4297 of Lecture Notes in Computer Science, Springer, 2006, pp. 338–352.

[13] F. Aono and M. Kimura, *The AzusA 16-way Itanium server*, Micro, IEEE, 20 (2000), pp. 54–60.

[14] A. W. Appel, *Modern Compiler Implementation in C: Basic Techniques*, Cambridge University Press, 1997.

[15] Apple Inc., *Power Mac G5 Users Guide*. `http://manuals.info.apple.com/en/Power_Mac_G5_Late_2005.pdf`.

[16] ——, *Xserve G5 Developer Note*. `http://developer.apple.com/documentation/Hardware/Developer_Notes/Servers/XServeG5/XserveG5.pdf`.

[17] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[18] T. Austin, E. Larson, and D. Ernst, *Simplescalar: An infrastructure for computer system modeling*, Computer, 35 (2002), pp. 59–67.

[19] R. Azimi, M. Stumm, and R. W. Wisniewski, *Online performance analysis by statistical sampling of microprocessor performance counters*, in ICS '05: Proceedings of the 19th annual international conference on Supercomputing, New York, NY, USA, 2005, ACM, pp. 101–110.

[20] R. F. W. Bader, *Atoms in Molecules: A Quantum Theory (The International Series of Monographs on Chemistry, No 22)*, Oxford University Press, June 1994.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, *Xen and the art of virtualization*, in SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, 2003, ACM, pp. 164–177.

[22] A. D. BECKE, *A Multicenter Numerical Integration Scheme for Polyatomic molecules*, The Journal of Chemical Physics, 88 (1988), pp. 2547–2553.

[23] E. BERG AND E. HAGERSTEN, *Statcache: a probabilistic approach to efficient and accurate data locality analysis*, in ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, Washington, DC, USA, 2004, IEEE Computer Society, pp. 20–27.

[24] E. BERG, H. ZEFFER, AND E. HAGERSTEN, *A Statistical Multiprocessor Cache Model*, Performance Analysis of Systems and Software, 2006 IEEE International Symposium on, (19-21 March 2006), pp. 89–99.

[25] L. N. BHUYAN, R. R. IYER, H.-J. WANG, AND A. KUMAR, *Impact of CC-NUMA Memory Management Policies on the Application Performance of Multistage Switching Networks*, IEEE Trans. Parallel Distrib. Syst., 11 (2000), pp. 230–246.

[26] G. BIKSHANDI, J. G. CASTANOS, S. B. KODALI, V. K. NANDIVADA, I. PESHANSKY, V. A. SARASWAT, S. SUR, P. VARMA, AND T. WEN, *Efficient, portable implementation of asynchronous multi-place programs*, in PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, 2009, ACM, pp. 271–282.

[27] N. L. BINKERT, R. G. DRESLINSKI, L. R. HSU, K. T. LIM, A. G. SAIDI, AND S. K. REINHARDT, *The M5 Simulator: Modeling Networked Systems*, IEEE Micro, 26 (2006), pp. 52–60.

[28] A. A. BLIZNYUK AND A. P. RENDELL, *Electronic effects in biomolecular simulations: Investigation of the kcsa potassium ion channel*, The Journal of Physical Chemistry B, 108 (2004), pp. 13866–13873.

[29] A. A. BLIZNYUK, A. P. RENDELL, T. W. ALLEN, AND S.-H. CHUNG, *The potassium ion channel: Comparison of linear scaling semiempirical and molecular mechanics representations of the electrostatic potential*, The Journal of Physical Chemistry B, 105 (2001), pp. 12674–12679.

[30] C. W. BOCK, G. D. MARKHAM, A. K. KATZ, AND J. P. GLUSKER, *The Arrangement of First- and Second-Shell Water Molecules in Trivalent Aluminum Complexes: Results from Density Functional Theory and Structural Crystallography*, Inorganic Chemistry, 42 (2003), pp. 1538–1548.

[31] H. G. BOCK, E. KOSTINA, X. P. HOANG, AND R. RANNACHER, *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the Third International Conference on High Performance Scientific Computing, March 6-10, 2006, Hanoi, Vietnam*, Springer Publishing Company, Incorporated, 2008.

[32] P. A. Boncz, M. L. Kersten, and S. Manegold, *Breaking the memory wall in monetdb*, Commun. ACM, 51 (2008), pp. 77–85.

[33] S. F. Boys, *Electronic Wave Functions. I. A General Method of Calculation for the Stationary States of Any Molecular System*, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 200 (1950), pp. 542–554.

[34] ——, *Quantum Theory of Atoms, Molecules and the Solid State*, Academic, New York, 1966.

[35] T. Brecht, *On the Importance of Parallel Application Placement in NUMA Multiprocessors*, in Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), 1993, pp. 1–18.

[36] C. M. Breneman and K. B. Wiberg, *Determining atom-centered monopoles from molecular electrostatic potentials. the need for high sampling density in formamide conformational analysis*, J. Comput. Chem., 11 (1990), pp. 361–373.

[37] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, *PAPI*, Intl. Journal of HPC Applications, 14 (2000), pp. 189–204.

[38] B. R. Buck and J. K. Hollingsworth, *Data Centric Cache Measurement on the Intel Itanium 2 Processor*, in In: Proceedings of SuperComputing. (2004, 2004.

[39] B. R. Buck and J. K. Hollingsworth, *A New Hardware Monitor Design to Measure Data Structure-Specific Cache Eviction Information*, International Journal of High Performance Computing Applications, 20 (2006), pp. 353–363.

[40] P. P. Bungale and C.-K. Luk, *PinOS: a programmable framework for whole-system dynamic instrumentation*, in VEE '07: Proceedings of the 3rd international conference on Virtual execution environments, New York, NY, USA, 2007, ACM, pp. 137–147.

[41] D. Burger and T. Austin, *The SimpleScalar Tool Set, Version 2.0*, Tech. Rep. TR-1342, University of Wisconsin-Madison Computer Sciences Department, 1997.

[42] A. W. Burks, H. H. Goldstine, and J. von Neumann, *Preliminary discussion of the logical design of an electronic computing instrument*. Report to the U.S. Army Ordnance Department, 1946.

[43] S. Byna, Y. Chen, and X.-H. Sun, *A Taxonomy of Data Prefetching Mechanisms*, Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on, (2008), pp. 19–24.

[44] H. W. Cain, K. M. Lepak, and M. H. Lipasti, *A Dynamic Binary Translation Approach to Architectural Simulation*, Computer Architecture News, 29 (2001).

[45] B. M. CANTRILL, M. W. SHAPIRO, AND A. H. LEVENTHAL, *Dynamic instrumentation of production systems*, in ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2004, USENIX Association, pp. 2–2.

[46] D. A. CASE, T. E. C. III, T. DARDEN, H. GOHLKE, R. LUO, K. M. M. JR., A. ONUFRIEV, C. SIMMERLING, B. WANG, AND R. J. WOODS, *The Amber biomolecular simulation programs*, Journal of Computational Chemistry, 26 (2005), pp. 1668–1688.

[47] CELESTICA INC., *AMD A8440 4U 4 Processor SCSI System*. `http://www.celestica.com/products/A8440.asp`.

[48] R. CHANDRA, D.-K. CHEN, R. COX, D. E. MAYDAN, N. NEDELJKOVIC, AND J.-A. M. ANDERSON, *Data Distribution Support on Distributed Shared Memory Multiprocessors.*, in PLDI, 1997, pp. 334–345.

[49] B. CHAPMAN, G. JOST, AND R. VAN DER PAS, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, October 2007.

[50] A. CHARLESWORTH, *The Sun Fireplane System Interconnect*, in Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), ACM Press, New York, New York, USA, November 2001.

[51] J. CHEN AND W. W. III, *Software Barrier Performance on Dual Quad-Core Opterons*, in NAS '08: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage, Washington, DC, USA, 2008, IEEE Computer Society, pp. 303–309.

[52] C.-Y. CHER, A. L. HOSKING, AND T. N. VIJAYKUMAR, *Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign*, SIGARCH Comput. Archit. News, 32 (2004), pp. 199–210.

[53] R. CHEVERESAN, M. RAMSAY, C. FEUCHT, AND I. SHARAPOV, *Characteristics of workloads used in high performance and technical computing*, in ICS, B. J. Smith, ed., ACM, 2007, pp. 73–82.

[54] CHRISTOPH LAMETER, *Local and Remote Memory: Memory in a NUMA System*. `http://ftp.kernel.org/pub/linux/kernel/people/christoph/ols2006`.

[55] CHRISTOPHER J. CRAMER, *Essentials of Computational Chemistry*, John Wiley & Sons, 2004.

[56] B. CONWAY, P.; HUGHES, *The AMD Opteron Northbridge Architecture*, IEEE Micro, 27 (March-April 2007), pp. 10–21.

[57] J. CORBALÁN, X. MARTORELL, AND J. LABARTA, *Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000.*, in ICS, U. Banerjee, K. Gallivan, and A. González, eds., ACM, 2003, pp. 121–129.

[58] D. E. CULLER, A. GUPTA, AND J. P. SINGH, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1999.

[59] V. CUPPU, B. JACOB, B. DAVIS, AND T. MUDGE, *High-Performance DRAMs in Workstation Environments*, IEEE Transactions on Computers, 50 (2001), pp. 1133–1153.

[60] B. W. CURRAN, E. FLUHR, J. PAREDES, L. J. SIGAL, J. FRIEDRICH, Y.-H. CHAN, AND C. HWANG, *Power-constrained high-frequency circuits for the ibm power6 microprocessor*, IBM Journal of Research and Development, 51 (2007), pp. 715–732.

[61] F. DAHLGREN AND P. STENSTROM, *Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors*, Parallel and Distributed Systems, IEEE Transactions on, 7 (1996), pp. 385–398.

[62] W. DALLY AND B. TOWLES, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[63] DAVID ASCHER AND PAUL F. DUBOIS AND KONRAD HINSEN AND JIM HUGUNIN AND TRAVIS OLIPHANT, *Numerical Python*. `http://numpy.scipy.org/`.

[64] DAVID R. BUTENHOF, *Programming with POSIX Threads*, Addison-Wesley Professional, 1997.

[65] E. R. DAVIDSON AND D. FELLER, *Basis set selection for molecular calculations*, Chemical Reviews, 86 (1986), pp. 681–696.

[66] P. J. DENNING, *The Locality Principle*, Commun. ACM, 48 (2005), pp. 19–24.

[67] P. J. DENNING AND J. P. BUZEN, *The operational analysis of queueing network models*, ACM Comput. Surv., 10 (1978), pp. 225–261.

[68] F. J. DEVLIN, P. J. STEPHENS, J. R. CHEESEMAN, AND M. J. FRISCH, *Ab Initio Prediction of Vibrational Absorption and Circular Dichroism Spectra of Chiral Natural Products Using Density Functional Theory: Camphor and Fenchone*, The Journal of Physical Chemistry A, 101 (1997), pp. 6322–6333.

[69] J. DONGARRA, K. LONDON, S. MOORE, P. MUCCI, D. TERPSTRA, H. YOU, AND M. ZHOU, *Experiences and lessons learned with a portable interface to hardware performance counters*, 2003.

[70] H. DORSETT, A. WHITE, D. SCIENCE, T. O. (AUSTRALIA), AERONAUTICAL, AND M. R. L. (AUSTRALIA)., *Overview of molecular modelling and ab initio molecular orbital methods suitable for use with energetic materials / H. Dorsett and A. White*, DSTO Aeronautical and Maritime Research Laboratory, Salisbury, S. Aust. :, 2000.

[71] M. S. DRESSELHAUS, G. DRESSELHAUS, AND P. C. EKLUND, *Science of Fullerenes and Carbon Nanotubes: Their Properties and Applications*, Academic Press, 1996.

[72] M. DUBOIS, J. SKEPPSTEDT, L. RICCIULLI, K. RAMAMURTHY, AND P. STENSTRÖM, *The Detection and Elimination of Useless misses in Multiprocessors*, in ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture, New York, NY, USA, 1993, ACM, pp. 88–97.

[73] M. DUPUIS, J. RYS, AND H. F. KING, *Evaluation of molecular integrals over gaussian basis functions*, The Journal of Chemical Physics, 65 (1976), pp. 111–116.

[74] C. E. DYKSTRA AND P. G. JASIEN, *Derivative hartree–fock theory to all orders*, Chemical Physics Letters, 109 (1984), pp. 388 – 393.

[75] L. EECKHOUT, D. STROOBANDT, AND K. D. BOSSCHERE, *Efficient microprocessor design space exploration through statistical simulation*, in ANSS '03: Proceedings of the 36th annual symposium on Simulation, Washington, DC, USA, 2003, IEEE Computer Society, p. 233.

[76] P. G. EMMA, A. HARTSTEIN, T. R. PUZAK, AND V. SRINIVASAN, *Exploring the limits of prefetching*, IBM J. Res. Dev., 49 (2005), pp. 127–144.

[77] ENRICO CLEMENTI, *Ab Initio Computations in Atoms and Molecules*, in IBM Journal of Research and Development, vol. 1, 1965, pp. 2 – 19.

[78] A. EPSHTEYN, M. J. GARZARÁN, G. DEJONG, D. A. PADUA, G. REN, X. LI, K. YOTOV, AND K. PINGALI, *Analytic models and empirical search: A hybrid approach to code optimization*, in LCPC, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, eds., vol. 4339 of Lecture Notes in Computer Science, Springer, 2005, pp. 259–273.

[79] S. ERANIAN, *What can performance counters do for memory subsystem analysis?*, in MSPC, E. D. Berger and B. Chen, eds., ACM, 2008, pp. 26–30.

[80] D. FELLER, *The Role of Databases in Support of Computational Chemistry Calculations*, The Journal of Computational Chemistry, 17 (1996), pp. 1571–1586.

[81] FLORIAN WEIGEND AND REINHART AHLRICHS, *Balanced basis sets of split valence, triple zeta valence and quadruple zeta valence quality for H to Rn: Design and assessment of accuracy*, Physical Chemistry Chemical Physics, 7 (2005), pp. 3297–3305.

[82] J. B. Foresman and A. Frisch, *Exploring Chemistry with Electronic Structure Methods*, Gaussian, Inc., New Haven, CT, 1996.

[83] Frank Jensen, *Introduction to Computational Chemistry*, John Wiley & Sons, 1999.

[84] B. Franke, *Fast cycle-approximate instruction set simulation*, in SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems, 2008, pp. 69–78.

[85] M. J. Frisch, B. G. Johnson, P. M. W. Gill, D. J. Fox, and R. H. Nobes, *An improved criterion for evaluating the efficiency of two-electron integral algorithms*, Chemical Physics Letters, 206 (1993), pp. 225 – 228.

[86] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, J. A. Montgomery, Jr., T. Vreven, K. N. Kudin, J. C. Burant, J. M. Millam, S. S. Iyengar, J. Tomasi, V. Barone, B. Mennucci, M. Cossi, G. Scalmani, N. Rega, G. A. Petersson, H. Nakatsuji, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, M. Klene, X. Li, J. E. Knox, H. P. Hratchian, J. B. Cross, V. Bakken, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, P. Y. Ayala, K. Morokuma, G. A. Voth, P. Salvador, J. J. Dannenberg, V. G. Zakrzewski, S. Dapprich, A. D. Daniels, M. C. Strain, O. Farkas, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. V. Ortiz, Q. Cui, A. G. Baboul, S. Clifford, J. Cioslowski, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, C. Gonzalez, and J. A. Pople, *Gaussian 03, Revision D.02*. Gaussian, Inc., Wallingford, CT, 2004.

[87] G. Hinton and D. Sager and M. Upton and D. Boggs and D. Carmean and A. Kyker and P. Roussel, *The Microarchitecture of the Pentium 4 processor*, Intel Technical Journal, (2001).

[88] R. Garner, S. M. Blackburn, and D. Frampton, *Effective prefetch for mark-sweep garbage collection*, in The 2007 International Symposium on Memory Management, Oct. 2007.

[89] Gaussian Inc., *Gaussian 03: Expanding the limits of Computational Chemistry*. `http://www.gaussian.com/g_brochures/g03_intro.htm`.

[90] R. Ghiya, D. Lavery, and D. Sehr, *On the importance of points-to analysis and other memory disambiguation methods for c programs*, SIGPLAN Not., 36 (2001), pp. 47–58.

[91] S. Ghosh, A. Kanhere, R. Krishnaiyer, D. Kulkarni, W. Li, C.-C. Lim, and J. Ng, *Integrating high-level optimizations in a production compiler: Design and implementation experience*, in CC, G. Hedin, ed., vol. 2622 of Lecture Notes in Computer Science, Springer, 2003, pp. 303–319.

[92] P. M. W. Gill, *Molecular Integrals over Gaussian Basis Functions*, Advances in Quantum Chemistry, 25 (1994), pp. 141–205.

[93] P. M. W. Gill, *Density Functional Theory (DFT), Hartree-Fock (HF), and the Self-Consistent Field*, in Encyclopaedia of Computational Chemistry, P. v. R. Schleyer et. al, ed., vol. 2, Wiley, 1998, pp. 678 – 688.

[94] P. M. W. Gill and S.-H. Chien, *Radial quadrature for multiexponential integrands*, Journal of Computational Chemistry, 24 (2003), pp. 732–740.

[95] P. M. W. Gill, M. Head-Gordon, and J. A. Pople, *Efficient computation of two-electron - repulsion integrals and their nth-order derivatives using contracted gaussian basis sets*, The Journal of Physical Chemistry, 94 (1990), pp. 5564–5572.

[96] P. M. W. Gill and J. A. Pople, *The PRISM algorithm for two-electron integrals*, International Journal of Quantum Chemistry, 40 (1991), pp. 753–772.

[97] S. R. Goldschmidt and J. L. Hennessy, *The Accuracy of Trace-Driven Simulations of Multiprocessors*, in Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, ACM Press, 1993, pp. 146–157.

[98] R. Gomperts, M. Frisch, and J.-P. Panziera, *Scalability of Gaussian 03 on SGI Altix: The Importance of Data Locality on CC-NUMA Architecture*, in Evolving OpenMP in an Age of Extreme Parallelism, vol. 5568 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 93–103.

[99] J. R. Goodman, *Using cache memory to reduce processor-memory traffic*, in ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture, Los Alamitos, CA, USA, 1983, IEEE Computer Society Press, pp. 124–131.

[100] K. Goto and R. A. van de Geijn, *Anatomy of high-performance matrix multiplication*, ACM Trans. Math. Softw., 34 (2008), pp. 1–25.

[101] S. L. Graham, P. B. Kessler, and M. K. McKusick, *gprof: a call graph execution profiler*, SIGPLAN Notices, 39 (2004), pp. 49–57.

[102] A. Grama, G. Karypis, A. Gupta, and V. Kumar, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Addison-Wesley, 2003.

[103] S. P. Greatbanks, J. E. Gready, A. C. Limaye, and A. P. Rendell, *Enzyme polarization of substrates of dihydrofolate reductase by different theoretical methods*, Proteins: Structure, Function, and Genetics, 37 (1999), pp. 157–165.

[104] E. Grobelny, D. Bueno, I. Troxel, A. D. George, and J. S. Vetter, *FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications*, SIMULATION, 83 (2007), pp. 721–745.

[105] C. F. Guerra, J.-W. Handgraaf, E. J. Baerends, and F. M. Bickelhaupt, *Voronoi deformation density (vdd) charges: Assessment of the mulliken, bader, hirshfeld, weinhold, and vdd methods for charge analysis*, Journal of Computational Chemistry, 25 (2004), pp. 189–210.

[106] T. P. Hamilton and H. F. Schaefer, *New variations in two-electron integral evaluation in the context of direct scf procedures*, Chemical Physics, 150 (1991), pp. 163 – 171.

[107] J. Handy, *The cache memory book (2nd ed.): the authoritative reference on cache design*, Academic Press, Inc., Orlando, FL, USA, 1998.

[108] M. Hässer and R. Ahlrichs, *Improvements on the direct SCF method*, Journal of Computational Chemistry, 10 (1989), pp. 104–111.

[109] M. Head-Gordon and J. A. Pople, *A method for two-electron gaussian integral and integral derivative evaluation using recurrence relations*, J. Chem. Phys., 89 (1988), pp. 5777–5786.

[110] W. J. Hehre, R. Ditchfield, and J. A. Pople, *Self—consistent molecular orbital methods. xii. further extensions of gaussian—type basis sets for use in molecular orbital studies of organic molecules*, The Journal of Chemical Physics, 56 (1972), pp. 2257–2261.

[111] W. J. Hehre, L. Radom, P. V. Schleyer, and J. Pople, *Ab-Initio Molecular Orbital Theory*, Wiley-Interscience, 1986.

[112] T. Helgaker, P. Jorgensen, and J. Olsen, *Molecular Electronic-Structure Theory*, John Wiley & Sons, 2001.

[113] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[114] W. D. Hillis and G. L. Steele, Jr., *Data parallel algorithms*, Commun. ACM, 29 (1986), pp. 1170–1183.

[115] R. W. Hockney and Eastwood, *Computer simulation using particles*, MacGraw-Hill, New York, 1981.

[116] P. Hohenberg and W. Kohn, *Inhomogeneous electron gas*, Phys. Rev., 136 (1964), pp. B864–B871.

[117] M. A. Holliday and M. Stumm, *Performance evaluation of hierarchical ring-based shared memory multiprocessors*, IEEE Trans. Computers, 43 (1994), pp. 52–67.

[118] J. K. Hollingsworth, B. P. Miller, and J. Cargille, *Dynamic program instrumentation for scalable performance tools*, in Proceedings of the Scalable High Performance Computing Conference (SHPCC), 1994, pp. 841–850.

[119] T. Horel and G. Lauterbach, *UltraSPARC-III: Designing Third-Generation 64-Bit Performance*, IEEE Micro, 19 (1999), pp. 73–85.

[120] A. Horváth and M. Telek, eds., *Formal Methods and Stochastic Models for Performance Evaluation, Third European Performance Engineering Workshop, EPEW 2006, Budapest, Hungary, June 21-22, 2006, Proceedings*, vol. 4054 of Lecture Notes in Computer Science, Springer, 2006.

[121] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, *RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors*, Computer, 35 (2002), pp. 40–49.

[122] Iilya Gluhovsky and Brian O'Krafka, *Comprehensive Multiprocessor Cache Miss Rate Generation Using Multivariate Models*, ACM Transactions on Computer Systems, (2005).

[123] Intel Corporation Inc., *Intel Core Microarchitecture and Smart Memory Access*. `http://download.intel.com/technology/architecture/sma.pdf`.

[124] Intel Corporation Inc., *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, Santa Clara, California, USA, 2002. Document Number: 245470-012.

[125] ——, *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, Santa Clara, California, USA, 2002. Document Number: 245472-012.

[126] ——, *Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, Santa Clara, California, USA, October 2002. Document Number 245319-004.

[127] Intel Corporation Inc., *Intel VTune Performance Analyzer*. `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm`, 2008.

[128] International Business Machines, *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-bit Microprocessors*, New York, USA, June 2003. Available as pem_64bit.d20030611.pdf.

[129] E. ÏPEK, S. A. MCKEE, R. CARUANA, B. R. DE SUPINSKI, AND M. SCHULZ, *Efficiently exploring architectural design spaces via predictive modeling*, SIGOPS Oper. Syst. Rev., 40 (2006), pp. 195–206.

[130] A. F. IZMAYLOV, G. E. SCUSERIA, AND M. J. FRISCH, *Efficient evaluation of short-range hartree-fock exchange in large molecules and periodic systems*, The Journal of Chemical Physics, 125 (2006), p. 104103.

[131] B. JACOB, S. NG, AND D. WANG, *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann, September 2007.

[132] JAN ALMLÖF, *Direct Methods in Electronic Structure Theory*, in Modern Electronic Structure Theory, D. R. Yarkony, ed., World Scientific, 1995.

[133] T. E. JEREMIASSEN AND S. J. EGGERS, *Reducing false sharing on shared memory multiprocessors through compile time data transformations*, SIGPLAN Not., 30 (1995), pp. 179–188.

[134] JIE TAO AND WOLFGANG KARL AND MARTIN SCHULZ, *Memory access behavior analysis of NUMA-based shared memory programs*, Scientific Programming, 10 (2002), pp. 45–53.

[135] B. G. JOHNSON, P. M. W. GILL, AND J. A. POPLE, *Exact and approximate solutions to the one-center mcmurchie-davidson tree-search problem*, International Journal of Quantum Chemistry, 40 (1991), pp. 809–827.

[136] ——, *Exact and approximate solutions to the one-center mcmurchie-davidson tree-search problem*, International Journal of Quantum Chemistry, 40 (1991), pp. 809–827.

[137] ——, *Preliminary results on the performance of a family of density functional methods*, The Journal of Chemical Physics, 97 (1992), pp. 7846–7848.

[138] JONATHAN CHEW, *Memory Placement Optimisation*. `http://www.opensolaris.org/os/community/performance/`.

[139] W. L. JORGENSEN, J. CHANDRASEKHAR, J. D. MADURA, R. W. IMPEY, AND M. L. KLEIN, *Comparison of simple potential functions for simulating liquid water*, The Journal of Chemical Physics, 79 (1983), pp. 926–935.

[140] JOSEF WEIDENDORF, *KCachegrind – A Profile Visualisation Tool*. `http://kcachegrind.sourceforge.net/`.

[141] P. J. JOSEPH, K. VASWANI, AND M. J. THAZHUTHAVEETIL, *A Predictive Performance Model for Superscalar Processors*, in MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, 2006, IEEE Computer Society, pp. 161–170.

[142] JOYCE J. DIWAN, *Biochemistry of Metabolism: Membrane Transport*. `http://www.rpi.edu/dept/bcbp/molbiochem/MBWeb/mb1/part2/4-transport.ppt`.

[143] R. P. L. JR., C. S. ELLIS, AND M. A. HOLLIDAY, *Evaluation of numa memory management through modeling and measurements*, IEEE Trans. Parallel Distrib. Syst., 3 (1992), pp. 686–701.

[144] D. R. KAELI, L. L. FONG, R. C. BOOTH, K. C. IMMING, AND J. P. WEIGEL, *Performance analysis on a cc-numa prototype*, IBM Journal of Research and Development, 41 (1997), pp. 205–214.

[145] T. S. KARKHANIS AND J. E. SMITH, *A First-Order Superscalar Processor Model*, in ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture, Washington, DC, USA, 2004, IEEE Computer Society, p. 338.

[146] C. N. KELTCHER, K. J. MCGRATH, A. AHMED, AND P. CONWAY, *The AMD Opteron Processor for Multiprocessor Servers.*, IEEE Micro, 23 (2003), pp. 66–76.

[147] W. KOCH AND M. C. HOLTHAUSEN, *A Chemist's Guide to Density Functional Theory*, Wiley-VCH, 2001.

[148] J. KOHANOFF, *Electronic structure calculations for solids and molecules: theory and computational methods*, Cambridge Univ. Press, Cambridge, 2006.

[149] P. KONGETIRA, K. AINGARAN, AND K. OLUKOTUN, *Niagara: A 32-way multithreaded sparc processor*, IEEE Micro, 25 (2005), pp. 21–29.

[150] D. KOUFATY AND D. T. MARR, *Hyperthreading technology in the NetBurst Microarchitecture*, Micro, IEEE, 23 (2003), pp. 56–65.

[151] V. KRISHNAN AND J. TORRELLAS, *Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-multiprocessor*, in International Conference on Supercomputing, 1998, pp. 85–92.

[152] N. KURD, J. DOUGLAS, P. MOSALIKANTI, AND R. KUMAR, *Next generation Intel micro-architecture (Nehalem) clocking architecture*, June 2008, pp. 62–63.

[153] M. D. LAM, E. E. ROTHBERG, AND M. E. WOLF, *The cache performance and optimizations of blocked algorithms*, SIGOPS Oper. Syst. Rev., 25 (1991), pp. 63–74.

[154] J. LAUDON AND D. LENOSKI, *The SGI Origin: A ccNUMA highly scalable server*, in Proceedings of the 24th Annual IEEE International Symposium on Computer Architecture, IEEE, 1997, pp. 241–251.

[155] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. Sauer, E. M. Schwarz, and M. T. Vaden, *Ibm power6 microarchitecture*, IBM Journal of Research and Development, 51 (2007), pp. 639–662.

[156] A. Leach, *Molecular Modelling: Principles and Applications (2nd Edition)*, Prentice Hall, March 2001.

[157] C. Lee, W. Yang, and R. G. Parr, *Development of the colle-salvetti correlation-energy formula into a functional of the electron density*, Phys. Rev. B, 37 (1988), pp. 785–789.

[158] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, *The Stanford DASH Multiprocessor*, Computer, 25 (1992), pp. 63–79.

[159] D. E. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1995.

[160] J. Levon, *OProfile Manual*, Victoria University of Manchester, 2004.

[161] R. Lindh, U. Ryu, and B. Liu, *The reduced multiplication scheme of the rys quadrature and new recurrence relations for auxiliary function based two-electron integral evaluation*, The Journal of Chemical Physics, 95 (1991), pp. 5889–5897.

[162] Liotta,, C. L. and Berknerin,, J., Encyclopedia of Reagents for Organic Synthesis (Ed: L. Paquette), (2004).

[163] Lizy Kurian John and Lieven Eeckhout, in Performance Evaluation and Benchmarking, CRC Press Inc., 2006.

[164] H. Löf and S. Holmgren, *affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System*, in ICS '05: Proceedings of the 19th annual international conference on Supercomputing, New York, NY, USA, 2005, ACM Press, pp. 387–392.

[165] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, in PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, 2005, ACM, pp. 190–200.

[166] M. Woodacre and D. Robb and D. Roe and K. Feind, *The SGI Altix 3000 Global Shared-Memory Architecture*, 2003.

[167] R. MacKinnon, S. L. Cohen, A. Kuo, A. Lee, and B. T. Chait, *Structural Conservation in Prokaryotic and Eukaryotic Potassium Channels*, Science, 280 (1998), pp. 106–109.

[168] P. S. MAGNUSSON, M. . CHRISTENSSON, J. ESKILSON, D. FORSGREN, AND G. HALLBERG, *SimICS: A Full System Simulation Platform*, IEEE Computer, 35 (2002), pp. 50–58.

[169] G. S. MANKU, *Theoretical Principles of Inorganic Chemistry*, Tata McGraw-Hill Publishing Company, 1980.

[170] J. MARATHE, F. MUELLER, AND B. R. DE SUPINSKI, *Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques*, ACM Trans. Archit. Code Optim., 3 (2006), pp. 390–423.

[171] J. MARATHE, F. MUELLER, T. MOHAN, S. A. MCKEE, B. R. D. SUPINSKI, AND A. YOO, *Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies*, ACM Trans. Program. Lang. Syst., 29 (2007), p. 12.

[172] G. MARIN AND J. MELLOR-CRUMMEY, *Cross-architecture performance predictions for scientific applications using parameterized models*, in SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems, New York, NY, USA, 2004, ACM Press, pp. 2–13.

[173] D. T. MARR, D. P. GROUP, AND I. CORP, *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal, 6 (2002), p. 2002.

[174] F. MARTIN AND H. ZIPSE, *Charge distribution in the water molecule - a comparison of methods*, Journal of Computational Chemistry, 26 (2005), pp. 97–105.

[175] M. M. MATHIS AND D. J. KERBYSON, *A general performance model of structured and unstructured mesh particle transport computations*, J. Supercomput., 34 (2005), pp. 181–199.

[176] W. MATHUR AND J. COOK, *Improved estimation for software multiplexing of performance counters*, in MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Washington, DC, USA, 2005, IEEE Computer Society, pp. 23–34.

[177] J. D. MCCALPIN, *Stream: Sustainable memory bandwidth in high performance computers*, tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[178] ——, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (1995), pp. 19–25.

[179] R. MCDOUGAL AND J. MAURO, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, Prentice Hall PTR, 2nd ed., 2006.

[180] S. McFarling, *Combining branch predictors*, Digital WRL Technical Note TN-36, (1993).

[181] S. A. McKee, *Reflections on the memory wall*, in CF '04: Proceedings of the 1st conference on Computing frontiers, New York, NY, USA, 2004, ACM, p. 162.

[182] L. E. McMurchie and E. R. Davidson, *One- and two-electron integrals over cartesian gaussian functions*, Journal of Computational Physics, 26 (1978), pp. 218 – 231.

[183] C. McNairy and D. Soltis, *Itanium 2 processor microarchitecture*, IEEE Micro, 23 (2003), pp. 44–55.

[184] L. W. McVoy and C. Staelin, *lmbench: Portable tools for performance analysis.*, in USENIX Annual Technical Conference, 1996, pp. 279–294.

[185] Mick Point, *High-Performance Math Libraries*, Dr. Dobb's Journal, 3 (2005), pp. 39–41.

[186] Mikael Petterson, *Linux kernel support for hardware performance counters – perfctrs*. `http://user.it.uu.se/~mikpe/linux/perfctr`.

[187] MIPS Technologies, Inc., *MIPS64 Architecture for Programmers Volume I: Introduction to the MIPS64 Architecture*, Mountain View, California, USA, June 2003. Document Number: MD00083.

[188] R. T. Morrison and R. N. Boyd, *Organic Chemistry*, Allyn and Bacon Boston,, 3d ed. ed., 1973.

[189] R. S. Mulliken, *Electronic population analysis on lcao[single bond]mo molecular wave functions. i*, The Journal of Chemical Physics, 23 (1955), pp. 1833–1840.

[190] N. Nethercote, *Dynamic Binary Analysis and Instrumentation*, PhD thesis, University of Cambridge, November 2004.

[191] N. Nethercote and A. Mycroft, *The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware*, in MSP '02: Proceedings of the 2002 workshop on Memory system performance, New York, NY, USA, 2002, ACM Press, pp. 44–55.

[192] N. Nethercote and J. Seward, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, in PLDI, J. Ferrante and K. S. McKinley, eds., ACM, 2007, pp. 89–100.

[193] H. Nguyen and L. K. John, *Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology*, in ICS '99: Proceedings of the 13th international conference on Supercomputing, New York, NY, USA, 1999, ACM, pp. 11–20.

[194] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, *Leveraging Transparent Data Distribution in*

OpenMP via User-Level Dynamic Page Migration., in ISHPC, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, eds., vol. 1940 of Lecture Notes in Computer Science, Springer, 2000, pp. 415–427.

[195] D. B. NOONBURG AND J. P. SHEN, *A framework for statistical modeling of superscalar processor performance*, in HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, Washington, DC, USA, 1997, IEEE Computer Society, p. 298.

[196] M. NORDÉN, *Performance Modelling for Parallel PDE Solvers on cc-NUMA Systems*, Technical reports from the Department of Information Technology, 2006-041 (2006). Part of urn:nbn:se:uu:diva-7149.

[197] S. OBARA AND A. SAIKA, *Efficient recursive computation of molecular integrals over cartesian gaussian functions*, The Journal of Chemical Physics, 84 (1986), pp. 3963–3974.

[198] ——, *General recurrence formulas for molecular integrals over cartesian gaussian functions*, The Journal of Chemical Physics, 89 (1988), pp. 1540–1559.

[199] V. G. OKLOBDZIJA, *Performance Evaluation: Techniques, Tools and Benchmarks*, in The Computer Engineering Handbook: Electrical Engineering Handbook, Boca Raton, FL, USA, 2001, CRC Press, Inc.

[200] L. OLIKER, A. CANNING, J. CARTER, C. LANCU, M. LIJEWSKI, S. KAMIL, J. SHALF, H. SHAN, E. STROHMAIER, S. ETHIER, AND T. GOODALE, *Scientific application performance on candidate petascale platforms*, March 2007, pp. 1–12.

[201] M. OLSZEWSKI, K. MIERLE, A. CZAJKOWSKI, AND A. D. BROWN, *Jit instrumentation: a novel approach to dynamically instrument operating systems*, SIGOPS Oper. Syst. Rev., 41 (2007), pp. 3–16.

[202] K. OLUKOTUN, B. A. NAYFEH, L. HAMMOND, K. WILSON, AND K. CHANG, *The case for a single-chip multiprocessor*, in ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 1996, ACM, pp. 2–11.

[203] Y. OSAMURA, Y. YAMAGUCHI, AND H. F. S. III, *Generalization of analytic energy derivatives for configuration interaction wave functions*, Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta), 72 (1987), pp. 71–91.

[204] A. J. OVER, *Multiprocessor Memory-System Simulation*, PhD in Computer Science, ANU Department of Computer Science, 2009.

[205] P. C. HARIHARAN AND J. A. POPLE, *The influence of polarization functions on molecular orbital hydrogenation energies*, Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta), 28 (1973), pp. 213– 222.

[206] P. M. W. GILL, BENNY G. JOHNSON, JOHN A. POPLE, *A simple yet powerful upper bound for Coulomb integrals*, Chemical Physics Letters, 217 (1994), pp. 65–68.

[207] PACIFIC NORTHWEST LABS (PNL), *EMSL Basis Set Exchange ver 1.2.2*, 2009.

[208] V. PAI, P. RANGANATHAN, AND S. ADVE, *RSIM Reference Manual Version 1.0*, Tech. Rep. Tech. Rep. 9705, Department of Electrical and Computer Engineering, Rice University, 1997.

[209] D. A. PATTERSON AND J. L. HENNESSY, *Computer organization and design (2nd ed.): the hardware/software interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[210] PAUL S. BAGUS AND ARTHUR R. WILLIAMS, *Electronic Structure Theory*, in IBM Journal of Research and Development, vol. 25, 1981, pp. 793 – 810.

[211] PER EKMAN, *Linux kernel memory-to-node mappings*. `http://www.pdc.kth.se/~pek/linux/NUMA/`.

[212] J. P. PERDEW, K. BURKE, AND M. ERNZERHOF, *Generalized Gradient Approximation Made Simple*, Phys. Rev. Lett., 77 (1996), pp. 3865–3868.

[213] F. PLUCINSKI AND A. MAZUREK, *An experimental method of determination of the point charges located on the carbon atoms in benzene ring*, Journal of Molecular Structure, 597 (2001), pp. 269 – 272.

[214] V. PRASAD, F. EIGLER, AND J. KENISTON, *Abstract locating system problems using dynamic instrumentation*, in Proceedings of the Linux Symposium, 2005.

[215] P. PULAY, *Second and third derivatives of variational energy expressions: Application to multiconfigurational self-consistent field wave functions*, The Journal of Chemical Physics, 78 (1983), pp. 5043–5051.

[216] J. RACINE, *gnuplot 4.0: a portable interactive plotting utility*, Journal of Applied Econometrics, 21 (2006), pp. 133–141.

[217] S. RADHAKRISHNAN, S. CHINTHAMANI, AND K. CHENG, *The blackford northbridge chipset for the intel 5000*, IEEE Micro, 27 (2007), pp. 22–33.

[218] R. RAJESH KOTA; OEHLER, *Horus: large-scale symmetric multiprocessing for opteron systems*, IEEE Micro, 25 (March-April 2005), pp. 30–40.

[219] A. Ralston and J. Edwin D. Reilly, *Encyclopedia of Computer Science and Engineering*, Thomson Learning, 1982.

[220] T. Ramdas, G. K. Egan, D. Abramson, and K. Baldridge, *Towards a special-purpose computer for Hartree-Fock computations*.

[221] ——, *Converting massive tlp to dlp: a special-purpose processor for molecular orbital computations*, in CF '07: Proceedings of the 4th international conference on Computing frontiers, New York, NY, USA, 2007, ACM, pp. 267–276.

[222] R. R. Ramseyer and A. van Dam, *A multi-microprocessor implementation of a general purpose pipelined cpu*, in ISCA '77: Proceedings of the 4th annual symposium on Computer architecture, New York, NY, USA, 1977, ACM, pp. 29–34.

[223] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, Cambridge University Press, New York, NY, USA, 1996.

[224] A. E. Reed, R. B. Weinstock, and F. Weinhold, *Natural population analysis*, J. Chem. Phys., (1985), pp. 735–746.

[225] A. P. Rendell, J. Antony, W. Armstrong, P. Janes, and R. Yang, *Building fast, reliable, and adaptive software for computational science*, Journal of Physics: Conference Series, 125 (2008), p. 012015 (10pp).

[226] N. Robertson and A. P. Rendell, *OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000.*, in International Conference on Computational Science, P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, eds., vol. 2660 of Lecture Notes in Computer Science, Springer, 2003, pp. 648–656.

[227] D. Robson and P. E. Strazdins, *Parallelisation of the valgrind dynamic binary instrumentation framework*, in ISPA, IEEE, 2008, pp. 113–121.

[228] Rohit Chandra and Ramesh Menon et. al., *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000.

[229] Roland Lindh, *Integrals of Electron Repulsion*, in Encyclopaedia of Computational Chemistry, P. v. R. Schleyer et. al, ed., vol. 2, Wiley, 1998, p. 1337.

[230] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, *Using the SimOS machine simulator to study complex computer systems*, ACM Transactions on Modelling and Computer Simulation, 7 (1997), pp. 78–103.

[231] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, *Complete computer system simulation: the SimOS approach*, IEEE Parallel and Distributed Technology: Systems and Applications, 3 (1995), pp. 34–43.

[232] Rui Yang and A. P. Rendell, *First principles study of gallium atom adsorption on the α-al$_2$o$_3$(0001) surface*, Journal of Physical Chemistry B, 110 (2006), pp. 9608–9618.

[233] S. Rusu, H. Muljono, and B. Cherkauer, *Itanium 2 processor 6m: higher frequency and larger l3 cache*, Micro, IEEE, 24 (2004), pp. 10–18.

[234] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, and R. Biswas, *Scientific application-based performance comparison of sgi altix 4700, ibm power5+, and sgi ice 8200 supercomputers*, in SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, 2008, IEEE Press, pp. 1–12.

[235] A. Schäfer, C. Huber, and R. Ahlrichs, *Fully optimized contracted gaussian basis sets of triple zeta valence quality for atoms li to kr*, The Journal of Chemical Physics, 100 (1994), pp. 5829–5835.

[236] M. Schmollinger and M. Kaufmann, *kappa numa: A model for clusters of smp-machines*, in PPAM, R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, eds., vol. 2328 of Lecture Notes in Computer Science, Springer, 2001, pp. 42–50.

[237] K. L. Schuchardt, B. T. Didier, T. Elsethagen, L. Sun, V. Gurumoorthi, J. Chase, J. Li, and T. L. Windus, *Basis set exchange: A community database for computational sciences*, J. Chem. Inf. Model., 47 (2007), pp. 1045–1052.

[238] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, *Larrabee: A many-core x86 architecture for visual computing*, Micro, IEEE, 29 (2009), pp. 10–21.

[239] SGI, *SGI NUMA Tools*. `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/linux/bks/SGI_Admin/books/LX_Resource_AG/sgi_html/ch07.html`.

[240] SGI Corporation, *The SGI Altix 450 System User's Guide*.

[241] I. Sharapov, R. Kroeger, G. Delamarter, R. Cheveresan, and M. Ramsay, *A case study in top-down performance estimation for a large-scale parallel application*, in PPOPP, J. Torrellas and S. Chatterjee, eds., ACM, 2006, pp. 81–89.

[242] I. Sharapov, R. Kroeger, G. Delamarter, R. Cheveresan, and M. Ramsay, *A case study in top-down performance estimation for a large-scale parallel application*, in PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, 2006, ACM, pp. 81–89.

[243] I. Shavitt, *The Gaussian Function in Calculations of Statistical Mechanics and Quantum Mechanics*, vol. 2, Academic Press, 1963.

[244] S. S. Shende and A. D. Malony, *The tau parallel performance system*, The International Journal of High Performance Computing Applications, 20 (2006), pp. 287–331.

[245] Simcha Gochman and Ronny Ronen and Ittai Anati and Ariel Berkovits and Tsvika Kurts and Alon Naveh and Ali Saeed and Zeev Sperber and Robert C. Valentine , *The Intel Pentium M Processor: Microarchitecture and Performance*, Intel Technical Journal, 7 (2003).

[246] J. Simons, *An experimental chemist's guide to ab initio quantum chemistry*, The Journal of Physical Chemistry, 95 (1991), pp. 1017–1029.

[247] A. Singh, *Mac OS X Internals*, Addison-Wesley Professional, 2006.

[248] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, *Challenges in computer architecture evaluation*, Computer, 36 (2003), pp. 30–36.

[249] A. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[250] A. J. Smith, *Cache memories*, ACM Comput. Surv., 14 (1982), pp. 473–530.

[251] J. E. Smith and A. R. Pleszkun, *Implementation of precise interrupts in pipelined processors*, SIGARCH Comput. Archit. News, 13 (1985), pp. 36–44.

[252] F. Song, S. Moore, and J. Dongarra, *L2 cache modeling for scientific applications on chip multi-processors*, in ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007), Washington, DC, USA, 2007, IEEE Computer Society, p. 51.

[253] D. J. Sorin, J. L. Lemon, D. L. Eager, and M. K. Vernon, *Analytic evaluation of shared-memory architectures*, IEEE Transactions on Parallel and Distributed Systems, 14 (2003), pp. 166–180.

[254] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, *Analytic evaluation of shared-memory systems with ilp processors*, SIGARCH Comput. Archit. News, 26 (1998), pp. 380–391.

[255] B. Sprunt, *Managing the complexity of performance monitoring hardware: The brink and abyss approach*, Int. J. High Perform. Comput. Appl., 20 (2006), pp. 533–540.

[256] R. Srinivasan and O. Lubeck, *Montesim: a monte carlo performance model for in-order microachitectures*, SIGARCH Comput. Archit. News, 33 (2005), pp. 75–80.

[257] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and

S. TROYER, *A 65 nm 2-billion transistor quad-core itanium processor*, Solid-State Circuits, IEEE Journal of, 44 (2009), pp. 18–31.

[258] M. C. STRAIN, G. E. SCUSERIA, AND M. J. FRISCH, *Achieving Linear Scaling for the Electronic Quantum Coulomb Problem*, Science, 271 (1996), pp. 51–53.

[259] A. STREY AND M. BANGE, *Performance Analysis of Intel's MMX and SSE: A Case Study*, in Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, London, UK, 2001, Springer-Verlag, pp. 142–147.

[260] E. STROHMAIER AND H. SHAN, *Apex-map: a parameterized scalable memory access probe for high-performance computing systems*, Concurrency and Computation: Practice and Experience, 19 (2007), pp. 2185–2205.

[261] SUN MICROSYSTEMS INC., *Solaris 10 : Extended Library Functions*. `http://docs.sun.com/app/docs/doc/817-0679`.

[262] ——, *Solaris 10 : Programming Interfaces Guide*. `http://docs.sun.com/app/docs/doc/817-4415`.

[263] ——, *Sun Studio 11 Collection*. `http://docs.sun.com/app/docs/coll/771.7`.

[264] ——, *Sun Studio 11: Sun Performance Library User's Guide*. `http://docs.sun.com/app/docs/doc/819-3692`.

[265] ——, *The SunFire X4600 M2 Architecture*.

[266] ——, *The Sun Fire V1280 Server Architecture*, November 2002.

[267] ——, *UltraSPARC III Cu User's Manual*, Sun Microsystems Inc., Santa Clara, California, USA, January 2004. Version 2.2.1.

[268] ——, *Sun studio performance analyzer*. `http://developers.sun.com/sunstudio/overview/topics/analyzing.jsp`, 2010.

[269] S. SWANSON, L. K. MCDOWELL, M. M. SWIFT, S. J. EGGERS, AND H. M. LEVY, *An evaluation of speculative instruction execution on simultaneous multithreaded processors*, ACM Trans. Comput. Syst., 21 (2003), pp. 314–340.

[270] P. SWEAZEY AND A. J. SMITH, *A class of compatible cache consistency protocols and their support by the ieee futurebus*, SIGARCH Comput. Archit. News, 14 (1986), pp. 414–423.

[271] A. SZABO AND N. S. OSTLUND, *Modern Quantum Chemistry*, Dover Publications, 1996.

[272] T. HELGAKER AND P. R. TAYLOR, *Gaussian Basis Sets and Molecular Integrals*, in Modern Electronic Structure Theory, D. R. Yarkony, ed., World Scientific, 1995.

[273] M. Tanaka and M. Aida, *An Ab-Initio mo study on orbital interaction and charge distribution in alkali metal aqueous solution: Li+, na+, and k+*, Journal of Solution Chemistry, 33 (2004), pp. 887–901.

[274] J. Tao and J. Weidendorfer, *Cache Simulation Based on Runtime Instrumentation for OpenMP Applications*, in ANSS '04: Proceedings of the 37th annual symposium on Simulation, IEEE Computer Society, 2004, p. 97.

[275] Terrance Tao, *The Euler-Maclaurin formula, Bernoulli numbers, the zeta function, and real-variable analytic continuation.* `http://terrytao.wordpress.com/2010/04/10/the-euler-maclaurin-formula-bernoulli-numbers-the-zeta-function-and-` 2010.

[276] The Portland Group Inc., *Fortran, C, C++ Compilers*. `http://www.pgroup.com/`.

[277] Thom H. Dunning, Kirk A. Peterson and David E. Woon, *Basis sets: Correlation Consistent Sets*, in Encyclopaedia of Computational Chemistry, P. v. R. Schleyer et. al, ed., vol. 1, Wiley, 1998, p. 88.

[278] X. Tian, R. Krishnaiyer, H. Saito, M. Girkar, and W. Li, *Impact of compiler-based data-prefetching techniques on spec omp application performance*, Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, (2005), pp. 53a–53a.

[279] M. M. Tikir and J. K. Hollingsworth, *Using Hardware Counters to Automatically Improve Memory Performance.*, in SC, IEEE Computer Society, 2004, p. 46.

[280] M. M. Tikir and J. K. Hollingsworth, *Hardware monitors for dynamic page migration*, J. Parallel Distrib. Comput., 68 (2008), pp. 1186–1200.

[281] J. Torrellas, J. L. Hennessy, and T. Weil, *Analysis of critical architectural and program parameters in a hierarchical shared memory multiprocessor*, in SIGMETRICS, 1990, pp. 163–172.

[282] J. Trodden and D. Anderson, *HyperTransport System Architecture*, Addison-Wesley Professional, 2003.

[283] A. M. Turing, *On Computable Numbers, with an application to the Entscheidungsproblem*, Proc. London Math. Soc., 2 (1936), pp. 230–265.

[284] U. Prestor and A. Davis, *An Application-Centric ccNUMA Memory Profiler*, in IEEE International Workshop on Workload Characterization, 2001, pp. 101–110.

[285] R. A. Uhlig and T. N. Mudge, *Trace-Driven Memory Simulation: A Survey*, ACM Computing Surveys (CSUR), 29 (1997), pp. 128–170.

[286] Ulrich Drepper, *What Every Programmer Should Know About Memory*. `http://people.redhat.com/drepper/cpumemory.pdf`.

[287] S. Vadlamani and S. Jenks, *Architectural considerations for efficient software execution on parallel microprocessors*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, (2007), pp. 1–10.

[288] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), pp. 103–111.

[289] S. P. Vanderwiel and D. J. Lilja, *Data prefetch mechanisms*, ACM Computing Surveys, 32 (2000), pp. 174–199.

[290] X. Vera, N. Bermudo, J. Llosa, and A. González, *A fast and accurate framework to analyze and optimize cache memory behavior*, ACM Trans. Program. Lang. Syst., 26 (2004), pp. 263–300.

[291] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, *Operating System Support for Improving Data Locality on CC-NUMA Compute Servers.*, in ASPLOS, 1996, pp. 279–289.

[292] S. H. Vosko, L. Wilk, and M. Nusair, *Accurate spin-dependent electron liquid correlation energies for local spin density calculations: a critical analysis*, Canadian Journal of Physics, 58 (1980), pp. 1200–1211.

[293] Z. Vranesic, M. Stumm, D. Lewis, and R. White, *Hector-a hierarchically structured shared memory multiprocessor*, vol. i, Jan 1991, pp. 444–453 vol.1.

[294] D. Wallin, H. Johansson, and S. Holmgren, *Cache Memory Behavior of Advanced PDE Solvers*, in Processing of Parallel Computing 2003 (ParCo2003), Dresden, Germany, Sept. 2003.

[295] W.-H. Wang and J.-L. Baer, *Efficient Trace-Driven Simulation Methods for Cache Performance Analysis*, ACM Transactions on Computer Systems (TOCS), 9 (1991), pp. 222–241.

[296] J. Weidendorfer, M. Kowarschik, and C. Trinitis, *A tool suite for simulation based analysis of memory access behavior*, in International Conference on Computational Science, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds., vol. 3038 of Lecture Notes in Computer Science, Springer, 2004, pp. 440–447.

[297] J. Weidendorfer and C. Trinitis, *Collecting and exploiting cache-reuse metrics*, in International Conference on Computational Science (2), V. S. Sunderam, G. D. van

Albada, P. M. A. Sloot, and J. Dongarra, eds., vol. 3515 of Lecture Notes in Computer Science, Springer, 2005, pp. 191–198.

[298] F. WEINHOLD AND C. R. LANDIS, *Valency and Bonding: A Natural Bond Orbital Donor-Acceptor Perspective*, Cambridge University Press, 2005.

[299] R. C. WHALEY, A. PETITET, AND J. DONGARRA, *Automated empirical optimizations of software and the ATLAS project.*, Parallel Computing, 27 (2001), pp. 3–35.

[300] C. A. WHITE, B. G. JOHNSON, P. M. W. GILL, AND M. HEAD-GORDON, *The continuous fast multipole mmthod*, Chemical Physics Letters, 230 (1994), pp. 8 – 16.

[301] WIKIPEDIA, *Alpha-Pinene*. `http://en.wikipedia.org/wiki/Alpha-Pinene`.

[302] ——, *Basic Block*. `http://en.wikipedia.org/wiki/Basic_block`.

[303] ——, *Diffuse Functions*. `http://en.wikipedia.org/wiki/Basis_set_(chemistry)`.

[304] ——, *Euler-Maclaurin Formula*. `http://en.wikipedia.org/wiki/Euler-Maclaurin_formula`.

[305] S. WILLIAMS, J. CARTER, L. OLIKER, J. SHALF, AND K. YELICK, *Lattice boltzmann simulation optimization on leading multicore platforms*, April 2008, pp. 1–14.

[306] S. WINKEL, R. KRISHNAIYER, AND R. SAMPSON, *Latency-tolerant software pipelining in a production compiler*, in CGO, M. L. Soffa and E. Duesterwald, eds., ACM, 2008, pp. 104–113.

[307] M. E. WOLF AND M. S. LAM, *A data locality optimizing algorithm*, SIGPLAN Not., 26 (1991), pp. 30–44.

[308] WOLFRAM MATHWORLD, *Hermite Polynomial*. `http://mathworld.wolfram.com/HermitePolynomial.html`.

[309] Q. YANG, *Performance Analysis of a Cache-Coherent Multiprocessor Based on Hierarchical Multiple Buses*, in PARBASE / Architectures, 1990, pp. 260–275.

[310] R. YANG, J. ANTONY, P. P. JANES, AND A. P. RENDELL, *Memory and thread placement effects as a function of cache usage: A study of the gaussian chemistry code on the sunfire x4600 m2*, in ISPAN, IEEE Computer Society, 2008, pp. 31–36.

[311] R. YANG, J. ANTONY, AND A. RENDELL, *Effective use of dynamic page migration on numa platforms: The gaussian chemistry code on the sunfire x4600m2 system*, Parallel Architectures, Algorithms, and Networks, International Symposium on, 0 (2009), pp. 63–68.

[312] R. YANG, J. ANTONY, AND A. P. RENDELL, *A simple performance model for multi-threaded applications executing on non-uniform memory access computers*, in HPCC '09: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications, Washington, DC, USA, 2009, IEEE Computer Society, pp. 79–86.

[313] K. C. YEAGER, *The MIPS R10000 Superscalar Microprocessor*, IEEE Micro, 16 (1996), pp. 28–40.

[314] T.-Y. YEH AND Y. N. PATT, *A comparison of dynamic branch predictors that use two levels of branch history*, SIGARCH Comput. Archit. News, 21 (1993), pp. 257–266.

[315] D. C. YOUNG, *Computational Chemistry: A Practical Guide for Applying Techniques to Real-World Problems*, Wiley, 2001.

[316] X. ZHANG AND X. QIN, *Performance prediction and evaluation of parallel processing on a numa multiprocessor*, IEEE Trans. Software Eng., 17 (1991), pp. 1059–1068.

Index