Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 281–291

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Modelling the Runtime of the Gaussian Computational Chemistry Application and Assessing the Impacts of Microarchitectural Variations

Joseph Antony[a,1], Alistair P. Rendell[b], Rui Yang[c], Gary Trucks[d], Michael J. Frisch[d]

[a] *ANU Supercomputer Facility, Australian National University*
[b] *School of Computer Science, Australian National University*
[c] *Information Technology Services, University of Wollongong*
[d] *Gaussian Inc., Wallingford, C.T.*

Abstract

This paper explores the use of a simple linear performance model, that determines execution time based instruction and cache miss counts, for describing the behaviour of two-electron integral evaluation algorithm in the Gaussian computational chemistry package. Four different microarchitecture platforms are considered with a total of seven individual microprocessors. Both Hartree-Fock and hybrid Hartree-Fock/Density Functional Theory electronic structure methods are assessed.

In most cases the model is found to be accurate to within 3%. Least agreement is for an Athlon64 system (ranging from 1.8% to 6.5%) and a periodic boundary computation on an Opteron where errors of up to 6.8% are observed. These errors arise as the model does not account for the intricacies of out-of-order execution, on-chip write-back buffers and prefetch techniques that modern microprocessors implement.

The parameters from the linear performance model are combined with instruction and cache miss counts obtained from functional cache simulation to predict the effect of cache modification on total execution time. Variations in level 1 and 2 linesize and level 2 total size are considered, we find there is some benefit if linesizes are increased (L1: 8%, L2: 4%). Increasing the level 2 cache size is also predicted to be beneficial, although the cache blocking approach already implemented in the Gaussian integral evaluation code was found to be working well.

Keywords: Performance model, Gaussian, LPM, hardware performance counters, cache simulation, PRISM, ERI, HF, B3LYP

1. Introduction

Effective performance models of fundamental algorithms, which constitute computational science applications, allow a computational scientist and/or developer to reason about the intrinsic costs associated with the data structures and algorithms being used [1, 2]. Models that can be used to understand memory performance are particularly

Email address: joseph.antony@anu.edu.au (Joseph Antony)
[1] Corresponding author

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.030

important. This is because all modern microprocessors utilize complex cache memory hierarchies, and the efficient use of this hierarchy can have a huge effect on overall performance. Consider for example the effort that has been devoted towards the design of algorithms that carefully orchestrate computation in synchrony with data movement [3, 4, 5].

In this paper we demonstrate the use of a simple linear performance model (LPM) to describe the runtime performance of an algorithm that is fundamental to electronic structure theory, namely of the PRISM algorithm for computing two-electron repulsion integrals (ERI) [6, 7], as implemented in the widely used Gaussian G03 computational chemistry package [8]. We build on previous work [9, 10], where preliminary results using the LPM to model the performance of PRISM for Hartree-Fock (HF) calculations on x86 microprocessors were presented, to consider both HF and hybrid HF/density functional theory (DFT) methods and a total of seven different processors. We also show how the LPM can be combined with functional cache simulation to predict the effect of system hardware changes on algorithmic runtime performance.

The paper is organized into eight sections. Section 2 discusses background material pertaining to the LPM, the HF and DFT methods, and the PRISM ERI algorithm. Section 3 details the experimental methodology used, including the microprocessors, the hardware performance counters, the functional cache simulator, the data analysis performed and molecular the systems used. In Section 4 we demonstrate the effect of cache blocking on the performance of the PRISM ERI algorithm on one microprocessor and how this data can be used to parametrize the LPM. Section 5 considers the accuracy of the LPM for the seven microprocessors and a range of computation types. The use of functional cache simulation and how it may be combined with the LPM to predict performance as a function of architectural changes is presented in Section 6. Finally Sections 7 and 8 present related work and conclusions respectively.

2. Background

The Linear Performance Model (LPM)

The LPM [11] defines application performance as a simple linear combination of instructions issued and cache misses,

$$Cycles = \alpha * (I_{Count}) + \beta * (L1_{Misses}) + \gamma * (L2_{Misses}) \quad (1)$$

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 α, β and γ are penalty factors. Loosely speaking 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 term α, β and γ 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 can be obtained by running the code on similar processor family revisions which have different cache sizes or by varying a 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 to the observed cycle time according to Equation 1.

The Hartree Fock (HF) and Density Functional Theory (DFT) Methods

Over the past five decades computational chemists have devised a hierarchy of methods based around evaluating a self-consistent field (SCF) for the quantum chemical system being studied [12, 13]. This hierarchy affords increasing chemical accuracy at added computational cost. The HF method, the starting point for many of the more sophisticated methods, uses molecular orbitals (MO) [2] to compute the total electronic energy of the system. An alternative method known as DFT uses the electronic density to derive a total system energy. Between these two alternative approaches there lies a variety of hybrid HF/DFT schemes that compute some interaction terms based on the molecular orbitals and some based on the electron density. B3LYP [12, 13] is one such hybrid method.

[2] A MO (ϕ_i)is a function which describes the motion of each electron in a system. MOs are often expanded using a finite set of basis functions i.e. $\phi_i = \sum_{\alpha}^{n} c_{\alpha i}\chi_{\alpha}$, where $c_{\alpha i}$ are the MO coefficients. Both HF and DFT methods optimize coefficients MO coefficients to minimize total electronic energy.

In all cases performing a SCF computation involves solving the following matrix equation,

$$FC = SC\varepsilon \tag{2}$$

where F is the so called Fock matrix, C the matrix of MO coefficients we seek to evaluate, S a matrix of integrals representing the overlap between pairs of basis functions and ε is a vector the elements of which correspond to the energies of the MOs. The F matrix is defined as,

$$F(i) = h(i) + \sum_{j=1}^{N} 2J_j(i) - K_j(i) \tag{3}$$

where $h(i)$ is the one-electron Hamiltonian operator, J_j the Coulomb operator and K_j the exchange-correlation (XC) operator. The latter two operators capture the interactions between and among the N electrons in the system. For HF and hybrid DFT computations evaluation of J_j and K_j requires the evaluation of two-electron repulsion integrals (ERIs). In the Gaussian application, these ERIs for HF are evaluated using the PRISM ERI evaluation algorithm[3].

The PRISM ERI Algorithm and DFT XC

Gaussian uses an efficient implementation of the PRISM algorithm [6, 7] to compute ERIs. Each ERI involves four basis functions, where each basis function is a linear combination of one or more Cartesian Gaussian functions located at the same centre. The latter is referred to as a contracted Gaussian function. For each contracted Gaussian function all the Cartesian components are treated together, with all ERIs arising from all components of all the four contracted Gaussian functions being collectively referred to as a shell-quartet. The PRISM algorithm tailors its operation according to the shell-quartet type, that is according to the angular momentum of the constituent contracted Gaussian functions and how each contracted function is assembled from individual Gaussian functions. Shell quartets of the same type, but with functions that have different exponents and/or are located at different centres, are computed in a similar manner and are referred to as an ERI batch.

Treating all ERIs of a given type in one batch is advantageous in that it minimizes the need to recompute common intermediates. The algorithm can also be vectorized over the batch size, and batch sizes can be large leading to good vectorisation or good instruction pipelining. On cache based machines large batch sizes can, however, also lead to data structures that no longer fit in cache. This causes expensive cache misses that degrade performance. As an example of batch size consider a computation performed on a large water cluster system [14]. Usually all oxygen and hydrogen atoms in each water molecule will use the same basis set so there will be many ERIs with the same quartet type but with functions located on different oxygen or hydrogen centers.

Evaluation of the $J_j(i)$ and $K_j(i)$ terms vary between HF and DFT methods. In HF methods both terms are evaluated with four-center ERIs using PRISM, with the exception that HF methods have exchange integrals but do not include terms for electron-electron correlated motion. DFT methods, in contrast, evaluate the Coulomb, exchange and correlation terms separately. In hybrid DFT methods like B3LYP, PRISM is used for the Coulomb terms and numerical quadrature on a 3D grid is used for the XC terms [12, 13, 15].

When a batch size becomes too large it is better to split the batch into multiple smaller batches and process each sub-batch separately. Such cache blocking of ERI batches and batching of grid points in DFT numerical quadrature has been implemented in Gaussian for many years now, with a blocking factor that was derived based largely on empirical observation. In this work we use modification of the cache blocking factor as a means for creating different instruction and cache miss counts and execution times. This data is then used to fit the LPM.

3. Experimental Methodology

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

[3] For B3LYP, the K_j operator also requires some terms to be evaluated using numerical quadrature.

Microprocessors and Compilers

Four distinct microprocessor families are used: the AMD64, Intel's NetBurst, P6 architectures and the IBM PowerPC. Within these families data for a total of seven distinct microprocessor platforms were obtained. We note that the microarchitecture for the AMD64, P6 and PowerPC systems are very similar to those found in contemporary microprocessors. From AMD64 family, we used (a) Opteron AMD848, clocked at 2.2Ghz with 64Kb L1 data-cache and 1 MB L2 unified-cache and (b) Athlon64 X2 4200+, 2.2Ghz, 64Kb L1 data-cache, 512Kb unified-L2 cache. From the Intel NetBurst family, we used (a) Pentium 4, 3.0Ghz, 64Kb L1 data-cache and 1 MB unified-L2 cache and (b) Pentium 4 EM64T, 3.0Ghz, 16Kb L1 data-cache and 2 MB L2 unified-cache. From the Intel P6 family we used a Pentium M, 1.4Ghz, 32Kb L1 data-cache and 1 MB L2 unified-cache. Lastly from the IBM PowerPC family we used (a) G5 PPC970Fx, 1.8Ghz, 32Kb L1 data-cache and 512Kb L2 unified-cache and (b) G5-XServe PPC970Fx, 2.0Ghz, 32Kb L1 data-cache and 512Kb L2 unified-cache. 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.

Hardware Performance Counters

Hardware performance counter values were read by inserting instrumentation code into G03 source code (version G03D01 [8]) and recompiling . For portability the PAPI cross-platform hardware performance counter infrastructure (version 3.2.1) was used [16]. All platforms ran a Linux 2.6.11.4 kernel with the SuSE 9.2 distribution and Mikael Peterson's perfctrs [17] Linux kernel patch (version 2.6.19).

Functional Cache Simulation

Functional cache simulation was performed using the Callgrind simulator [18] along with the Valgrind [19] 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. 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.

Molecular Systems and Benchmarks

Four molecular systems were chosen for performance experiments, for which both HF and B3LYP methods were used. Benchmark systems 1 and 2 are respectively $K^+(H_2O)_{11}$ and $K^+(H_2O)_{80}$ water cluster complexes. The geometries for these systems were obtained from molecular dynamics (MD) simulations of a potassium ion (K^+) surrounded by a cluster of water molecules at 300 Kelvin [14]. $K^+(H_2O)_{11}$ contains all water molecules within a radius of 4 Å from the central potassium ion, while $K^+(H_2O)_{80}$ has a radius of 8 Å. 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 are in principle more accurate but lead to longer execution times and greater memory use.

Benchmark system 3 is based on the Valinomycin molecule. It is derived from the Gaussian G03 quality assurance (QA) suite and, reflecting its number in the QA suite, is referred to as *test397*. It is a widely used benchmark.

Benchmark system 4, referred to as α - Al_2O_3, is based on a Periodic Boundary Condition (PBC) [15] calculation and consists of three Aluminum trioxide molecules which form an infinite sheet in the (0001) orientation [20]. The system was chosen as it exercises a slightly different code path through the Gaussian code.

In summary, a total of four distinct molecular systems were chosen. For the $K^+(H_2O)_{11}$ and $K^+(H_2O)_{80}$ 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 a total of five benchmark systems.

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. Blocking factors ranging from 2 Kw (KiloWords) up to 1024 Kw (16Kb to 8 MB) were used with the PRISM algorithm; where a KiloWord is equal to 8 Kilobytes.

Table 1: Cycle count per LTot for $K^+(H_2O)_{11}$ using HF/6-31G* on a 2.2Ghz AMD848 Opteron.

	Total	Asy.	Avg.	Cycle count ($\times 10^9$)					
LTot	NoQrt	Lim	QPB.	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 ($\times 10^{10}$)			3.85	3.00	**2.69**	2.75	3.58	4.64
	% Increase from 32 Kw			+30.17	+10.56	0.00	+2.32	+24.94	+42.10

Asy. Lim – Asymptotic Limit
Avg. QPB – Average number of Quartets per batch, without the use of cache blocking

4. The Effect of Cache Blocking on the PRISM ERI Algorithm

The objective of this section is to investigate the effects of cache blocking on the batch sizes and execution time for different quartet types within the PRISM algorithm. One HF SCF cycle on the AMD848 using a single CPU for $K^+(H_2O)_{11}$/6-31G* takes 13 sec., $K^+(H_2O)_{11}$/6-31++G(3df,3pd) takes 21.4 min., $K^+(H_2O)_{80}$/6-31G* takes 20 min. and $K^+(H_2O)_{80}$/6-31++G(3df,3pd) takes 3.4 days. Detailed data for one cycle of a HF SCF using the $K^+(H_2O)_{11}$ system with the 6-31G* basis set on the 2.2Ghz AMD848 Opteron system is presented in Table 1. The data and trends presented here are representative of the what is observed for larger systems. There are five major columns in the table which are divided into two sections. The first major section has four columns for the total angular momentum of the shell quartet (LTot), the total number of quartets (Total NoQrt) of that angular momentum, the asymptotic limit (Asy. Lim) for the number of shell quartets types of this angular momentum, and the average number of quartets per batch.

Expanding on the above, the value of LTot is given by the sum of the angular momenta for the four Cartesian Gaussian functions that make up a shell quartet. Since in this system the maximum angular momentum value for any basis function is 2 (a *d* function), the maximum LTot value is 4×2 or 8. As all four functions in the quartet must have the maximum angular momentum value possible, there are relatively few quartets of this type (623). Likewise when LTot is zero every function in the quartet must have the minimum possible angular momentum value of zero. This is somewhat more common since the construction of the 6-31G* basis sets results in more functions with low angular momentum than high angular momentum. Thus we find more quartets of this type (352176). In between these two LTot values we find the number of quartets increases to some maximum. In this case the maximum is 806845 quartets that occurs for an LTot value of 2.

The LTot value does not, however, fully characterise the quartet type. In addition to the angular momentum of each function, the contraction scheme used to create each function is a distinguishing factor. The third column in Table 1 labelled 'Asy.Lim.', gives the number of unique quartet types of a given LTot value. The sum of these values is the minimal number of ERI batches that could be used to compute the ERIs if no cache blocking was implemented. If this were the case then the average number of quartets per batch is the value given in column four; and this would be the average vector length for the inner loop in the PRISM algorithm.

Evaluating shell quartets of high LTot value requires more operations than for shell quartets with low LTot values, but there are more quartets of low LTot value than of high LTot value. Thus predicting the LTot value for the shell quartet that will take the most number of cycles to compute is hard, and this is anyway likely to vary from machine to machine and be a function of cache blocking. This is evident from Table 1; the LTot value that consumes most of the cycles is found by looking down anyone of the cycle count columns and observing where the maximum count occurs. Thus, for the 4 Kw cache blocking size the maximum cycle count is 9.12×10^9 at an LTot value of 4, while for the 256 Kw cache blocking size the maximum is 8.22×10^9 that occurs at a different LTot value of 3.

The results in Table 1 also show that there is no single optimal cache blocking size. That is, looking across each row for the various LTot values we find that the smallest cycle counts occur with a blocking factor of 32 Kw for LTot values from 0-4, but for LTot values from 5-8 the smallest counts occur for the 64 Kw blocking factor. An ideal algorithm would therefore modify the cache blocking according to LTot value, and possibly at a finer level than this. This is not done in the current version of the G03 code, rather there is a single blocking factor. In this respect the final two rows of the table show that if a single cache blocking size is used, then it should be 32 Kw; that increasing this

Table 2: Self-fitting errors for the HF and B3LYP methods, between measured Cycle counts and those obtained using the LPM.

	Micro-processor	% Self-Fitting Error					% Avg. Error	σ
		$K^+(H_2O)_{11}$		$K^+(H_2O)_{80}$	test397	α-Al_2O_3		
		A*	B†	A*	3-21G	3-21G*		
HF	Opteron	3.0	1.0	2.3	1.9	1.3	1.9	0.8
	Athlon64	4.5	5.8	5.3	6.0	1.8	4.7	1.7
	EM64T	2.7	1.4	2.0	2.7	0.8	1.9	0.8
	Pentium 4	2.3	1.8	2.3	2.2	0.5	1.8	0.8
	Pentium M	2.0	0.8	1.9	2.5	1.0	1.6	0.7
	G5	1.8	1.5	2.3	2.7	2.2	2.1	0.5
	G5-XServe	2.4	1.7	2.0	2.9	2.4	2.3	0.5
	% Avg. μp error	2.7	2.0	2.6	3.0	1.4		
	σ	0.9	1.7	2.0	2.9	2.4		
B3LYP	Opteron	1.2	2.4	1.8	1.3	6.8	2.7	2.3
	Athlon64	3.3	6.5	4.8	4.5	5.4	4.9	1.2
	EM64T	1.0	1.3	5.3	0.8	4.3	2.5	2.1
	Pentium 4	0.9	1.4	1.5	1.3	3.7	1.8	1.1
	Pentium M	2.0	0.8	1.2	1.3	2.1	1.5	0.6
	G5	1.1	1.4	2.8	2.1	2.0	1.9	0.7
	G5-XServe	1.3	1.6	3.1	2.5	2.8	2.3	0.8
	% Avg. μp error	1.5	2.2	2.9	2.0	3.9		
	σ	0.9	2.0	1.6	1.3	1.8		

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

to 64 Kw causes the execution to go up by a modest 2.32%, while reducing it to 16 Kw has a much great penalty of 10.56%.

Although not presented here[4], it was found that for all of the benchmark systems, (a) there are different optimal blocking factors depending on the system and basis set used; and (b) use of a single blocking factor skews the blocking factor in favour of those values of LTot with large number of batches and shell-quartets with moderate total angular momentum.

5. A Linear Performance Model

In Section 4 it was shown that the 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. Using five of the benchmark molecular systems, hardware performance counter results were obtained for cycle count, L1 & L2 misses and instruction counts for the various hardware platforms. PPCoeffs were then obtained for each benchmark system separately. 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. On fitting some of the systems, negative values for β were obtained. This is clearly unphysical since it implies that an L1 cache miss is beneficial. To remedy this we modified the LPM for all systems to include only instruction counts and L2 cache misses. For α the values range from 0.63 to 1.1 (i.e. on average instructions were taking 0.63 to 1.1 cycles to complete), while values of γ range from 100 to 500 (i.e. the average cost of an L2 miss which required data to be fetched from DRAM).

In Table 2 we summarize the observed self-fitting errors for the HF and B3LYP cycle counts (i.e. execution times) across all hardware platforms expressed as a percentage. It 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 i.e. using PPCoeffs for the given hardware platform. In the case of HF calculations PRISM is responsible for the majority of the runtime, whereas in the DFT B3LYP calculation the XC computation dominates with PRISM making a slightly smaller contribution. If for HF we consider the error distribution for any given system across the set of processors, i.e. going down the columns, then the average error ranges from 1.4% to 3.0% (σ ranges from 0.9 to 1.7). The error distribution for all processors, i.e. going down the % Avg. Error column, apart from the Athlon64 ranges from 1.6% to 2.3% (σ ranges from 0.5 to 0.8). For the Athlon64 the average error across all systems is 4.7% (σ = 1.7). For B3LYP the fitting errors range from 1.5% to 3.9% (σ ranges from 0.9 to 2.0). The errors for any

[4] An extended version can be found in Chapter 4 of [21].

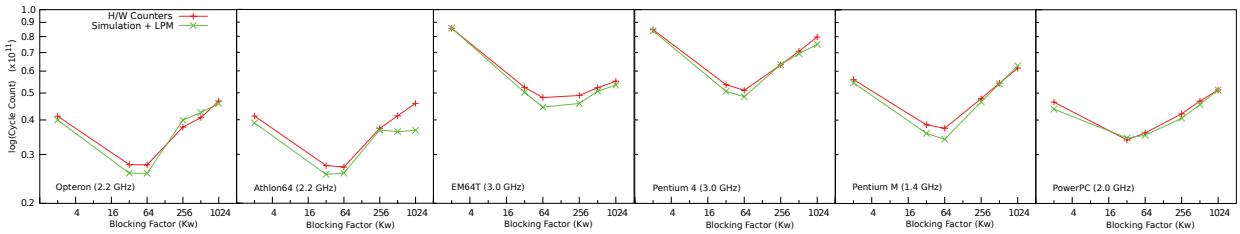

Figure 1: Plots for Cycle counts obtained from simulation and hardware performance counters for $K^+(H_2O)_{11}$/6-31G*. Simulation results were obtained using Valgrind/Callgrind.

given processor across all systems for B3LYP 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%.

To summarize, the self-fitting errors presented here 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 of processors and benchmark systems. We also find that the LPM can obtain good cycle count fits of between 4.3% and 6.0% for the HF method and between 3.1% to 6.7% for the B3LYP method, across all hardware platforms.

6. Performance Estimation using Functional Cache Simulation and the LPM

In the previous section it was shown that the LPM can be used to obtain cycle counts that are reasonably accurate, if the instruction count, total L2 misses and PPCoeffs are known. In this section we combine the PPCoeffs obtained by fitting measured results for the various hardware platforms with instruction counts and L2 misses obtained from functional cache simulation, in order to predict algorithmic performance as a function of cache design. We used the Callgrind tool, which is part of the Valgrind program supervision framework, to perform execution driven functional cache simulation. We considered all seven hardware architecture types, but present results only for the $K^+(H_2O)_{11}$ using HF/6-31G*.

Prior to exploring the effect of architectural changes on performance, it is important to validate the Callgrind functional cache simulator. Thus, we present results comparing measured cycle counts with those obtained using the LPM and functional cache simulation. Following this, we perform a detailed parametric study to identify parameters of interest. This was performed using a cache configuration similar to an AMD Opteron. In this section, we also consider results for varying L1, L2 cache linesize and total cache size for the Opteron, Pentium M and Pentium 4 architectures.

Validation of the Callgrind Functional Cache Simulator

Figure 1 presents plots comparing cycle counts obtained from simulation and hardware performance counters. The 'H/W Counters' curve corresponds to hardware performance counter results and the 'Simulation + LPM' curve corresponds to the use of PPCoeffs and the LPM (Equation 1) for each platform using I_{Count} and $L2_{Misses}$ which were obtained from Callgrind. For the Opteron, the use of the LPM and PPCoeffs with values for instruction count, total L2 misses from simulation produces a result which is in very good agreement with the measured cycles counts. On the Athlon64, the LPM results are in good agreement up to 256 Kw, after which it varies marginally from hardware counter results. There is good agreement of results for the EM64T, Pentium 4 and Pentium M systems. Excellent agreement is found for cycle counts for the PowerPC. 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. This variation is due to prefetch instructions and out-of-order execution features not being modelled by Callgrind.

Though we do not present results for total instruction counts and L1 misses here, we found that Callgrind is able to obtain accurate instruction counts and can produce reasonable to good reproductions of the L2 misses for most systems. L1 misses were not accurately reproduced. 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 for PRISM.

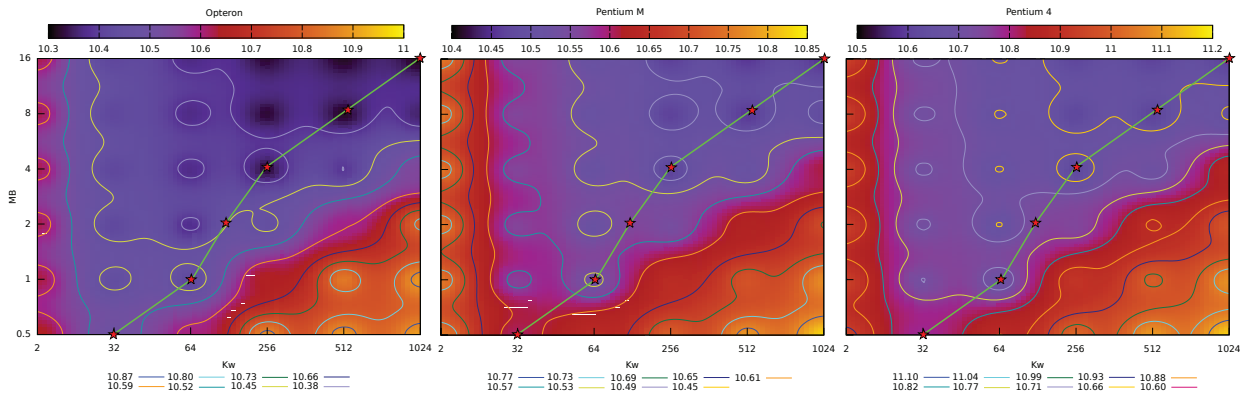

Figure 2: Valgrind/Callgrind cycle count results for varying L2 size (MB) for $K^+(H_2O)_{11}$ with HF/6-31G*.

A Parametric Cache Variation Study of PRISM's Performance

As functional cache simulation in conjunction with the LPM can reproduce cycle counts with reasonable accuracy, we now use fitted PPCoeffs with counts from functional cache simulation to make performance predictions for possible future cache configurations. Though we focus on cache based microarchitectural changes here, the LPM could be easily re-parametrized to capture other microarchitectural elements [22].

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

The use of the LPM with instruction counts and cache misses obtained from functional cache simulation was validated in the previous section. In previous work [10], we'd performed an in depth cache parameter variation study of an Opteron like cache which indicated the relative importance of L1 and L2 read, write misses and total unified L2 size. Detailed breakdowns of read and write misses showed that PRISM's cache performance is being limited by read misses generated during ERI computation [10]. We obtained an 8% improvement by increasing the L1 linesize (from 64 bytes to 256 bytes) and a 4% improvement by increasing the L2 linesize (from 64 bytes to 128 bytes) [5]. A 20% improvement was observed when the unified L2 cache was increased from 1 MB to 8 MB. In this section, we present results for varying total L2 size for the Opteron, Pentium M and Pentium 4 processors for a HF computation performed on the $K^+(H_2O)_{11}$ system using the 6-31G* basis set. A variety of cache blocking factors are used.

The aim is to assess the impact L2 cache size changes across three different hardware architectures types on the PRISM ERI algorithm.

Results for the three microprocessors types with various L2 cache sizes and different G03 cache blocking factors are shown in Figure 2. The x-axis has the blocking factor in Kw, the y-axis has the total unified L2 cache size (0.5 to 16 MB). The plot is a heat map of cycle counts obtained for the $K^+(H_2O)_{11}$ system with HF/6-31G*, where each pixel is color coded to indicate its relative weight to other pixels. We plot cycle counts using a $\log_{10}$ scale. The color strip on top of each graph denotes the range of values with cool to warm colors denoting low to high cycle counts. Contour lines for each plot are given at the bottom of each plot. Superimposed on each of the plot is a green line denoting Gaussians' default choice for cache blocking – i.e for a 0.5 MB L2 cache, it chooses a blocking factor of 32 Kw; for a 1 MB cache, it chooses 64 Kw 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 (e.g. the 10.38 and 10.45 contour lines). For the 1 MB cache, this occurs at 64 Kw; for 2 MB at 64 Kw; for 4 MB at 256 Kw; for 8 MB at 256 Kw and 16 MB for 1024 Kw. In two cases the 2 MB and the 8 MB case, Gaussian G03's default blocking choice does result in the lowest possible cycle count for a given cache size, for the $K^+(H_2O)_{11}$/6-31G* system. For a 2 MB cache, the LPM predicts a blocking factor of 64 Kw would perform better than the default choice of 128 Kw. The LPM predicts a 256 Kw blocking factor would perform better for an 8 MB cache than 512 Kw. Similar trends are observed for the Pentium M and Pentium 4.

[5] In-depth results can be found in Chapter 4 of [21].

To summarize, we find that results obtained from simulation and the LPM were able to reproduce trends observed using hardware counter measurements. Results for a parametric cache variation study using the Opteron, Pentium 4 and Pentium M cache configurations show that Gaussian's default blocking factor works well for k300a-04/6-31G* except in the case of 4 MB and 8 MB L2 cache sizes.

7. Related Work

Ramdas et. al [23] perform qualitative analysis and assess the prospects of mapping an implementation of the Rys ERI method 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. A discrete event simulation is used to determine the impact of arithmetic units (adders and multipliers) in the FPGA. In comparison we considered the PRISM ERI algorithm, its ERI batching behaviour, cache blocking effects on observed performance.

Franke et. al. [24, 25] 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%. The LPM is lightweight in obtaining application specific performance characteristics. PPCoeffs are obtained using hardware counter data which can then be used by either trace based or execution based simulators. Franke's general approach is very similar to that taken here in obtaining least-squares fits for the LPM. Unlike Franke, 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%.

8. Discussion, Conclusions and Future Work

A linear performance model was proposed to model the measured execution time for the PRISM ERI algorithm. 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, which required cache lines to be fetched from DRAM. It was shown that the LPM can 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.

In [26, 27, 28] we have extended the LPM to account for modern multi-core processors, NUMA memory placement effects and use the LPM for the parallel OpenMP version of PRISM. For future work, it would be useful to test the use of the LPM at runtime to aid in searching for optimal blocking factors for an entire SCF cycle. 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. We also intend, for future work, to study the performance impact of shared last-level of cache on multi-core systems using the LPM.

Wallin et. al [29], 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. One possible way of achieving this, in future work, is through the use of a prefetch queue [30] 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.

Acknowledgments

This work was possible due to funding from the Australian Research Council, Gaussian Inc. and Sun Oracle Inc. under ARC Linkage Grant LP0347178. JA wishes to thank Tess Collins from the Apple Developer Connection for access to G5 hardware, IBM's academic program for access to compilers, ANU's DoI for access to G5 servers and Mikael Petterson for helping port perfctrs for the G5-XServe.

References

[1] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.
[2] R. Cheveresan, M. Ramsay, C. Feucht, I. Sharapov, Characteristics of workloads used in high performance and technical computing, in: B. J. Smith (Ed.), ICS, ACM, 2007, pp. 73–82.
[3] K. Goto, R. A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 1–25.
[4] R. Nishtala, R. W. Vuduc, J. W. Demmel, K. A. Yelick, When cache blocking of sparse matrix vector multiply works and why, Appl. Algebra Eng., Commun. Comput. 18 (2007) 297–311.
[5] S. K. Sahoo, S. Krishnamoorthy, R. Panuganti, P. Sadayappan, Integrated loop optimizations for data locality enhancement of tensor contraction expressions, in: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, SC '05, 2005.
[6] P. M. W. Gill, Molecular Integrals over Gaussian Basis Functions, Advances in Quantum Chemistry 25 (1994) 141–205.
[7] Roland Lindh, Integrals of Electron Repulsion, in: P. v. R. Schleyer et. al (Ed.), Encyclopaedia of Computational Chemistry, Vol. 2, Wiley, 1998, p. 1337.
[8] M. J. Frisch, G. W. Trucks, et al., Gaussian 03, Revision D.02, Gaussian, Inc., Wallingford, CT, 2004.
[9] J. Antony, M. J. Frisch, A. P. Rendell, Modelling the Performance of the Gaussian Chemistry Code on x86 Architectures, in: Modeling, Simulation and Optimization of Complex Processes, Springer Berlin Heidelberg, 2008, pp. 49–58.
[10] A. P. Rendell, J. Antony, W. Armstrong, P. Janes, R. Yang, Building fast, reliable, and adaptive software for computational science, Journal of Physics: Conference Series 125 (2008) 012015 (10pp).
[11] N. Nethercote, 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, ACM Press, New York, NY, USA, 2002, pp. 44–55.
[12] P. M. W. Gill, Density Functional Theory (DFT), Hartree-Fock (HF), and the Self-Consistent Field, in: P. v. R. Schleyer et. al (Ed.), Encyclopaedia of Computational Chemistry, Vol. 2, Wiley, 1998, pp. 678 – 688.
[13] Frank Jensen, Introduction to Computational Chemistry, John Wiley & Sons, 1999.
[14] A. A. Bliznyuk, A. P. Rendell, Electronic Effects in Biomolecular Simulations: Investigation of the KcsA Potassium Ion Channel, The Journal of Physical Chemistry B 108 (36) (2004) 13866–13873. arXiv:http://pubs.acs.org/doi/pdf/10.1021/jp0487298, doi:10.1021/jp0487298.
[15] J. Kohanoff, Electronic structure calculations for solids and molecules: theory and computational methods, Cambridge Univ. Press, Cambridge, 2006.
[16] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, PAPI, Intl. Journal of HPC Applications 14 (3) (2000) 189–204.
[17] Mikael Petterson, Linux kernel support for hardware performance counters – perfctrs, `http://user.it.uu.se/~mikpe/linux/perfctr`.
[18] J. Weidendorfer, M. Kowarschik, C. Trinitis, A Tool Suite for Simulation Based Analysis of Memory Access Behavior, in: M. Bubak, G. D. van Albada, P. M. A. Sloot, J. Dongarra (Eds.), ICCS, Vol. 3038 of Lecture Notes in Computer Science, Springer, 2004, pp. 440–447.
[19] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: J. Ferrante, K. S. McKinley (Eds.), PLDI, ACM, 2007, pp. 89–100.
[20] Rui Yang and A. P. Rendell, First principles study of gallium atom adsorption on the α-al_2o_3(0001) surface, Journal of Physical Chemistry B 110 (19) (2006) 9608–9618.
[21] J. Antony, Performance Models for Electronic Structure Methods on Modern Computer Architectures, Ph.D. thesis, The Australian National University, `http://thesis.anu.edu.au/public/adt-ANU20101222.165637` (2010).
[22] F. Ryckbosch, S. Polfliet, L. Eeckhout, Fast, accurate, and validated full-system software simulation of x86 hardware, Micro, IEEE 30 (6) (2010) 46 –56. doi:10.1109/MM.2010.95.
[23] T. Ramdas, G. Egan, D. Abramson, K. Baldridge, Towards a special-purpose computer for Hartree-Fock computations, Vol. 120, Springer Berlin / Heidelberg, 2008, pp. 133–153, 10.1007/s00214-007-0306-6.
[24] 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.
[25] D. C. Powell, B. Franke, Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators, in: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '09, ACM, New York, NY, USA, 2009, pp. 315–324.
[26] R. Yang, J. Antony, A. P. Rendell, A Simple Performance Model for Multithreaded 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, IEEE Computer Society, Washington, DC, USA, 2009, pp. 79–86.
[27] R. Yang, J. Antony, 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) 63–68.
[28] R. Yang, J. Antony, P. P. Janes, 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.
[29] D. Wallin, H. Johansson, S. Holmgren, Cache Memory Behavior of Advanced PDE Solvers, in: Processing of Parallel Computing 2003 (ParCo2003), Dresden, Germany, 2003.

[30] R. Garner, S. M. Blackburn, D. Frampton, Effective prefetch for mark-sweep garbage collection, in: The 2007 International Symposium on Memory Management, ACM, 2007.