Modelling the Performance of the Gaussian Chemistry Code on x86 Architectures

Joseph Antony[1], Mike J. Frisch[2], and Alistair P. Rendell[1]

[1] Department of Computer Science, The Australian National University, ACT 0200, Australia. {joseph.antony, alistair.rendell}@anu.edu.au
[2] Gaussian Inc. 340 Quinnipiac St., Bldg. 40, Wallingford CT 06492, USA.

Summary. *Gaussian* is a widely used scientific code with application areas in chemistry, biochemistry and material sciences. To operate efficiently on modern architectures *Gaussian* employs cache blocking in the generation and processing of the two-electron integrals that are used by many of its electronic structure methods. This study uses hardware performance counters to characterise the cache and memory behavior of the integral generation code used by *Gaussian* in Hartree-Fock calculations. A simple performance model is proposed that aims to predict overall performance as a function of total instruction and cache miss counts. The model is parameterised for three different x86 processors – the Intel Pentium M, the P4 and the AMD Opteron. Results suggest that the model is capable of predicting execution times to an accuracy of between 5 and 15%. Use of this model in developing a dynamic cache blocking scheme is also discussed.

1 Introduction

It is well known that technological advances have driven processor speeds faster than main memory speeds, and that to address this issue complex cache based memory hierarchies have been developed. Obtaining good performance on cache based systems requires that the vast majority of the load/store instructions issued by the processor are serviced using data that resides in cache. In other words, to achieve good performance it is necessary to minimize the number of cache misses [4].

One approach to achieving this goal is to implement some form of cache blocking [10]. The objective here is to structure the computational algorithm in such a way that it spends most of its time working with blocks of data that are sufficiently small to reside in cache, and only periodically does it move data between main memory and cache.

Gaussian [5] is a widely used computational chemistry code that employs cache blocking to perform more efficient *integral computation*s [8,14]. The integrals in question lie at the heart of many of the electronic structure methods implemented within *Gaussian*, and are associated with the various interactions between and among the electrons and nuclei in the system under study.

Since many electronic structure methods are iterative, and the number of integrals involved too numerous for them to be stored in memory, the integrals are usually re-computed several times during the course of a typical calculation. For this reason algorithms that compute electronic structure integrals fast and on-demand are extremely important to the computational chemistry community.

To minimize the operation count, integrals are usually computed in batches, where all integrals in a given batch share a number of common intermediates [7]. In the PRISM algorithm used by *Gaussian*, large batch sizes give rise to large inner loop lengths. This is good for pipelining, but poor if it causes cache overflows and the need to fetch data from main memory. To address this problem *Gaussian* imposes a cache blocking by limiting the maximum size of an integral batch. This in effect says that the time required to recompute the common shared intermediates is less than the time penalty associated with having inner loops fetch data quantities from main memory.

In the current version of *Gaussian* there is a "one size fits all" approach to cache blocking, in that the same block size is used regardless of the exact characteristics of the integrals being computed. A long term motivation for our work is to move away from this model towards a dynamic model where cache blocking is tailored to each and every integral batch. As a first step towards this goal, this paper explores the ability of a simple Linear Performance Model (LPM) to predict the performance of *Gaussian*'s integral evaluation code purely as a function of instruction count and cache misses.

It is important to note that the LPM is very different to that used in typical analytic or simulation based performance studies. Analytic models attempt to weight various system parameters and present an empirical equation for performance, whereas simulation studies are either trace[3] or execution[4] driven with each instruction considered in order to derive performance metrics. Analytic models fail, however, to capture dynamic aspects of code execution that are only evident at runtime, while execution or trace driven simulations are extremely slow, often being 100-1000 times slower than execution of the actual code. The LPM on the other hand, effectively ignores all the intricate details of program execution and assumes that, over time these details can be averaged out and incorporated into penalty factors associated with the average cost of issuing an instruction and the average cost of a cache miss.

In this study, three x86 platforms – the Intel Pentium M, P4 and AMD Opteron – are considered. On-chip hardware performance counters are used to gather instruction and cache miss data from which the LPM is derived. The paper is broken into the following sections: section 2 discusses the background to this study, the tools and methodology used, and introduces the

[3] Trace driven simulation uses a pre-recorded list of instructions in a tracefile for later interpretation by a simulator.

[4] An execution driven simulator interprets instructions from a binary source to perform its simulation.

LPM; section 3 uses the LPM for a series of experiments on the three different platforms and discusses the results. Previous work, conclusions and future work are covered in sections 4 and 5.

2 Background

2.1 The Hartree-Fock Method

Electronic structure methods aim to solve Schrödinger's wave equation for atomic and molecular systems. For all but the most trivial systems it is necessary to make approximations. Computational chemists have developed a hierarchy of methods each with varying computational cost and accuracy. Within this hierarchy the Hartree-Fock (HF) method is relatively inaccurate, but it is also the bedrock on which more advanced and accurate methods are built. For these reasons the HF method was chosen as the focus of this work.

At the core of HF theory is the concept of a molecular orbital (MO), where one MO is used to describe the motion of each electron in the system. The MOs (ϕ) are expanded in terms of a number (N) of basis functions (χ) such that for MO ϕ_i:

$$\phi_i = \sum_{\alpha}^{N} c_{\alpha i}\chi_\alpha \tag{1}$$

where $c_{\alpha i}$ are the expansion or molecular orbital coefficients. In HF methods the form of these coefficients is optimized so that the total energy of the system is minimized. The basis functions used are normally located at the various atomic nuclei, and are a product of a radial function that depends on the distance from that nuclei, and an angular function such as a spherical harmonic ${Y_{lm}}^5$ [8]. Usually the radial function is a Gaussian $G_{nl}(r)$ [6], and it is for this reason that the *Gaussian* code is so named.

The matrix form of HF equations is given by:

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

where C is the matrix of molecular orbital coefficients, S a matrix of (overlap) integrals between pairs of basis functions, ϵ a vector with elements corresponding to the energy of each MO, and F is the so called Fock matrix defined by:

$$F_{\mu\nu} = H_{\mu\nu}^{core} + \sum_{\lambda\sigma}^{N}\sum_{i}^{N_e} C_{\lambda i}C_{\sigma i}^{*}[(\mu\nu \,|\, \lambda\sigma) - (\mu\lambda \,|\, \nu\sigma)] \tag{3}$$

where N_e is the number of electrons in the system. In equation 3 each element of the Fock matrix is expressed in terms of another two-index quantity ($H_{\mu\nu}^{core}$) that involves other integrals between pairs of basis functions, and

[5] $Y_{lm}(\theta,\varphi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(cos\,\theta)(e^{im\varphi})$

[6] $G_{nl}(r) = \frac{2(2\alpha)^{3/4}}{\pi^{1/4}}\sqrt{\frac{2^{2n-l-2}}{(4n-2l-3)!!}}(\sqrt{2\alpha}r)^{2n-l-2}\exp(-\alpha r^2)$

the molecular orbital coefficients (C) contracted with a four-index quantity $(\mu\nu \,|\, \lambda\sigma)$. Since F depends on C, which is the same quantity that we seek to determine,equation 2 is solved iteratively by guessing C, building a Fock matrix, solving equation 2 and then repeating this process until convergence is reached. The four-index quantities, $(\mu\nu \,|\, \lambda\sigma)$, are the electron repulsion integrals (ERIs) that are of interest to this work, and arise due to repulsive interactions between pairs of electrons. They are given by:

$$(\mu\nu \,|\, \lambda\sigma) = \iint \chi_\mu(r_1)\, \chi_\nu(r_1)\, \frac{1}{|r_1 - r_2|}\, \chi_\lambda(r_2)\, \chi_\sigma(r_2)\; dr_1\, dr_2 \tag{4}$$

where r_1 and r_2 are the coordinates of two electrons. For a given basis the number of two-electron integrals grows as $O(N^4)$, so evaluation and processing of these quantities quickly becomes a bottleneck. (We note that for large systems it is possible to reduce this asymptotic scaling through the use of pre-screening and other techniques [13], but these alternative approaches still require a substantial number of ERIs to be evaluated and processed.)

In the outline given above it has been assumed that each basis function is a single Gaussian function multiplied by a spherical harmonic (or similar). In fact it is common to combine several Gaussian functions with different exponents together in a fixed linear combination, treating the result as one *contracted* Gaussian basis function. We note also that when a basis function involves a spherical harmonic (or similar) of rank one or higher (i.e. $l \geq 1$), it is normal to include all orders of spherical harmonic functions within that rank (i.e. $\forall m : -l \leq m \leq l$). Thus if a basis function involves a spherical harmonic of rank 2, all 5 components are included as basis functions. Thus there are three parameters that characterise a basis function; i) its location, ii) its degree of contraction and the exponents of the constituent Gaussians, and iii) the rank of its angular component. In the PRISM [12] algorithm functions that have the same items ii) and iii) the same are treated together, with a batch of ERI integrals defined by doing this for all of the four functions involved. The size of these batches can quickly become very large since the same basis set is generally applied to all atoms of the same type within the system under study, e.g. all oxygen or hydrogen atoms in the system. It is for this reason that *Gaussian* imposes cache blocking to limit maximum batch sizes.

2.2 Linear Performance Model

The Linear Performance Model (LPM) gives the total number of cycles required to execute a given code segment as:

$$Cycles = \alpha * (I_{Count}) + \beta * (L1_{Misses}) + \gamma * (L2_{Misses}) \tag{5}$$

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 fitting parameters. Intuitively the value of α reflects the ability of

the code to exploit the underlying superscalar architecture, β is the average cost of an L1 cache miss, and γ is the average cost of an L2 cache miss. We will collectively refer to α, β and γ as the Processor and Platform specific coefficients (PPCoeffs). They will be derived by performing a least squares fit of the $Cycles, I_{Count}, L1_{Misses}$ and $L2_{Misses}$ counts obtained from hardware performance counters for a variety of cache blocking sizes.

2.3 PAPI

PAPI [3], a cross platform performance counter library, is used to obtain hardware counter data. It uses on-chip performance counters to measure application events of interest like instruction and cycle counts as well as other cache events. The three x86 machines used, the Intel Pentium M, P4 and the AMD Opteron, have different numbers of on-chip performance counters. Each on-chip performance counter can count one particular hardware event. The following hardware events are used in this study; `PAPI_L1_TCM` (Total level one (L1) misses (data and instruction)), `PAPI_L2_TCM` (Total level two (L2) misses), `PAPI_TOT_INS` (Total instructions) and `PAPI_TOT_CYC` (Total cycles). PAPI also supports hardware performance counter event multiplexing. 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.

It is noted that on the P4 processor PAPI does not have a `PAPI_L1_TCM` preset event, as it is not exposed by the underlying hardware counters. Instead `PAPI_L1_DCM` and `PAPI_L1_ICM` are used to count the total number of data and instruction cache misses respectively.

Table 1 lists processor characteristics and the cache and memory latencies measured using `lmbench` [11]. The P4's L1 instruction cache is a trace cache [6], unlike the Pentium M and Opteron. Hyperthreading on the P4 was turned off for this study. PAPI's event multiplexing was used on the Pentium M, as this processor has only two hardware counters, but four hardware events are required by the LPM.

2.4 Methodology

Gaussian computations are performed on a small system consisting of a solvated potassium ion surrounded by 11 water molecules with the geometry obtained from a snapshot of a molecular dynamics simulation. This work uses two basis sets denoted as 6-31G* and 6-31G++(3df,3pd) [8]. The former is a relatively modest basis set, while the latter would be considered large. Cache blocking in *Gaussian* is controlled by an input parameter `cachesize`, this was set to values of 2, 8, 32, 128, 256 and 512 kilowords. The default value of this parameter equates to the approximate size of the highest level of cache on the machine being used, and from this value the sizes of various intermediate buffers are derived. For each blocking size performance counter results were recorded for one complete iteration of the HF procedure and averaged over five runs.

		Pentium M	P4	Opteron
Clock Rate	(Ghz)	1.4	3.0	2.2
Ops. per Cycle	(Cy)	3	3	3
Memory Subsystem		NtBr	NtBr	HT
Perf. Counters	(No.)	2	18	4
L1 DCache	Size (Kb)	32	16	64
	Associativity (Ways)	8	8	2
	Line size (Bytes)	64	64	64
	Cache Policies	LRU, WB	P-LRU	LRU, WB, WA
L2 Unified	Size (MB)	1	1	1
	Associativity (Ways)	8	8	16
	Line size (Bytes)	64	64	64
	Relation to L1	Inclusive	Inclusive	Exclusive
	Cache Policies	LRU	P-LRU	P-LRU
`lmbench` Latencies for				
L1 DCache	Latency (Cy)	3	4	3
L2 Unified	Latency (Cy)	9	28	20
Main Memory	Latency (Cy, $\approx$)	201	285	405

NtBr = Northbridge, HT = HyperTransport,
LRU = Least Recently Used, P-LRU = Pseudo-LRU,
WB = Write Back, WA = Allocate on Write

Table 1. Processor characteristics of clock rate, cache sizes and measured latencies for L1 DCache, L2 cache and main memory latencies for the three x86 processors used in the study.

Block	6-31G*			6-31G++(3df,3pd)		
Size	Pentium M	P4	Opteron	Pentium M	P4	Opteron
2	42.0	28.2	20.8	4440	3169	2298
8	36.7	24.2	17.0	3849	2821	1970
32	30.0	19.8	13.6	2914	2210	1484
128	31.8	20.2	17.0	2869	2121	1701
256	37.0	22.0	20.2	3349	2259	1856
512	42.0	24.8	22.0	3900	2516	2214
$\overline{x}$	36.6	23.2	18.4	3554	2516	1921
σ	5.0	3.2	3.1	618	409	308

Table 2. Timings (seconds) for HF benchmark using the 6-31G* and 6-31G++(3df,3pd) basis sets as a function of the cache blocking parameter. Also shown are the average ($\overline{x}$) times and their standard deviations (σ).

3 Observed timings and hardware counter data

Observed timings

Table 2 shows the execution times obtained on the three different hardware platforms as a function of the different cache block sizes and when using the 6-31G* and 6-31G++(3df,3pd) basis sets. These results clearly show that cache blocking for integral evaluation has a major effect on the overall performance of the HF code in *Gaussian*. As the block size is increased from 2 to 512 kilowords the total execution time initially decreases, reaches a minimum,

and then increases again. Exactly where the minimum is located is seen to vary slightly across the different platforms, and between the two different basis sets. Also shown in Table 2 are the execution times averaged over all the different cache block sizes on a given platform, together with the corresponding standard deviation. Although, the absolute value of the standard deviations are significantly smaller for the 6-31G* basis, as a percentage of average total execution times they are roughly equal for both basis sets at around 15%.

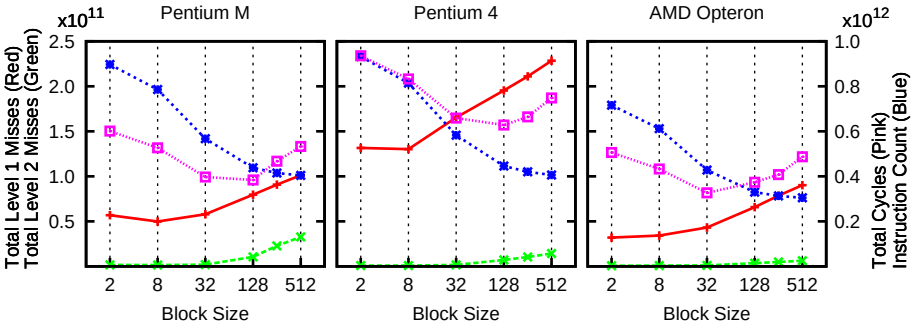

Fig. 1. Hardware counter data as a function of the cache blocking parameter for the HF method using the 6-31G++(3df,3pd) basis set on the three different hardware platforms

Hardware counter data

Hardware counter data for the 6-31G++(3df,3pd) basis set is given in Figure 1. The left hand axis of the graph (y_1) has two quantities namely Total Level 1 Misses ($L1_{Misses}$) and Total Level 2 Misses ($L2_{Misses}$), while the right hand axis (y_2) has Total Cycles ($Cycles$) and Instruction Count (I_{Count}). The x axis is plotted using a log_2 scale.

The cycle counts shown in Figure 1 are directly related to the times given in Table 2 by the clock speeds (see Table 1). Hence they show a similar behavior, decreasing initially as the block size increases, reaching a minimum and then increasing. In contrast the instruction counts show a steep initial decrease, but then appear to level off for large block sizes. This behavior reflects the fact that similar integrals, previously split into multiple batches, will be computed in fewer batches as the block size increases. Mirroring this behavior the L1 and L2 cache misses are initially low, increase when the blocking size is expanded, and ultimately will plateau when there are no more split batches to be combined (although this is not evident for the block sizes given in the figure).

Obtained PPCoeffs

Using the LPM (equation 5) and the hardware performance counter data for the HF/6-31G* calculations, a least squares fit was performed in order to obtain the PPCoeffs values given in Table 3. For the Pentium M and Opteron the values of α, β and γ appear reasonable. Specifically a value of 0.67 for

Processor	α	β	γ
Pentium M	0.67	13.39	63.60
P4	2.87	-59.35	588.46
Opteron	0.64	7.13	388.23
P4[a]	0.86	–	323.18

Table 3. PPCoeff (α, β, γ) values for the Pentium M, P4 and Opteron obtained from HF, 6-31G* results. See text for further details. [a] Results obtained when ignoring counts for L1 cache misses.

α on the Pentium M and 0.64 on the Opteron, implies that the processors are issuing 1.5 and 1.6 instructions per cycle respectively. Given that both processors (and also the P4) can issue upto three instructions per cycle these values are in the typical range of what might be expected.

The values for β and γ are average L1 and L2 cache miss penalties respectively, or alternatively β is the average cost of referencing data in L2 cache, while γ is the average cost of referencing data in main memory. The actual costs for referencing the L2 cache and main memory as measured using `lmbench` are given in Table 1. Thus for the Pentium M a value for β of 13.39 can be compared with the L2 latency of 9 cycles (Table 1), and a value for γ of 63.60 can be compared with 201. On the Opteron the equivalent comparisons are 7.13 to 20, and 388.23 to 405. These results for β, and particularly those for γ are roughly in line with what we might expect if we note that they are averages, while those measured by `lmbench` are worst case scenarios; hardware smarts such as prefetching and out-of-order execution are likely to mask some of the latencies associated with a cache miss in *Gaussian*, but not for `lmbench` (by design).

In contrast to the Pentium M and Opteron systems the results for the P4 are clearly unphysical with a negative value for β. The reason for this will be outlined in a future publication [2], but in essence it is due to the nature of the P4 micro-architecture which makes it very hard to count accurately the L1 cache misses. If, however, we ignore L1 misses and restrict the LPM to just the instruction count and L2 cache misses we obtain the second set of P4 data given in Table 3. This is far more reasonable,with a value for α that now equates to 1.2 instructions per cycle compared to an unlikely previous value of 0.37. Similarly the latency for a main memory reference is now less than that recorded by `lmbench`.

The PPCoeffs in Table 3 were derived using performance counter data obtained from running with the 6-31G* basis set. It is of interest to combine these values for α, β and γ with the instruction and cache miss counts recorded with the larger 6-31G++(3df, 3pd) basis set, and thereby obtain predicted cycle counts for this larger calculation. The difference between these predicted cycle counts and the actual cycle counts gives a measure of the ability of the LPM to make predictions outside of the domain in which it was originally parameterised. Doing this we find RMS differences between the predicted and measured execution times of 456, 268 and 95 seconds for the Pentium M, P4

(2 parameter LPM) and Opteron processors respectively. Compared to the average execution times given in Table 2, this represents an error of ∼13% on the Pentium M, ∼10% on the P4, and ∼5% on the Opteron. Since the total execution time varies by over 50% as the block size is changed, these results suggest that the LPM is accurate enough to make useful predictions concerning the performance of *Gaussian* as a function of total instruction and cache misses.

4 Previous work

Using a sparse set of trace based cache simulations, Gluhovsky and O'Krafka [9] build a multivariate model of multiple cache miss rate components. This can then be used to extrapolate for other hypothetical system configurations. Vera et al. use cache miss equations [15] 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. Snavely et. al use profile convolving [1] 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. The LPM, in comparison to these methods 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.

5 Conclusions and Future work

A linear performance model was proposed to model the cache performance of *Gaussian*. PPCoeffs (α, β, γ) obtained intuitively correspond to how well the code uses the superscalar resources of the processor, the average cost in cycles of an L1 cache miss and the average cost in cycles of an L2 miss.

Experiments show optimal batch sizes are both platform and computation specific, hinting that a dynamic means of varying batch sizes at runtime might be useful. In which case the LPM could be used to determine cache blocking sizes prior to computing a batch of integrals. On completing each batch cache metrics gather could then be used to guide a runtime search toward the most optimal blocking size. The predictive ability of the LPM can be used to aid experiments which use cache simulation tools. These tools are capable of simulating caches of current and possible future processors and yield instruction counts, number of L1 and L2 misses. In tandem with the LPM, cycle counts can be computed thus allowing determination of which microarchitectural features have the greatest impact on code performance.

For future work we propose to test the usefulness of the LPM at runtime to aid in searching for optimal blocking factors and use it to study the effect of microarchitectural changes on code performance.

Acknowledgments

This work was possible due to funding from the Australian Research Council, Gaussian Inc. and Sun Microsystems Inc. under ARC Linkage Grant LP0347178. JA and APR wish to thank Alexander Technology for access to an AMD Opteron cluster and DCS TSG.

References

1. A. Snavely and N. Wolter and L. Carrington. Modelling Application Performance by Convolving Machine Signatures with Application Profiles. *IEEE Workshop on Workload Characterization*, December 2001.
2. Joseph Antony, M. J. Frisch, and A. P. Rendell. Future Publication.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. PAPI. *Intl. Journal of HPC Applications*, 14(3):189–204, 2000.
4. David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1999.
5. M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, and J. R. Cheeseman et. al. *Gaussian 03, Revision C.01*. Gaussian Inc., Gaussian, Inc., Wallingford CT, USA, 2004.
6. 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.
7. Martin Head-Gordon and John A. Pople. A method for two-electron gaussian integral and integral derivative evaluation using recurrence relations. *J. Chem. Phys.*, 89(9):5777–5786, 1988.
8. Trygve Helgaker, Poul Jorgensen, and Jeppe Olsen. *Molecular Electronic-Structure Theory*. John Wiley & Sons, 2001.
9. Iilya Gluhovsky and Brian O'Krafka. Comprehensive Multiprocessor Cache Miss Rate Generation Using Multivariate Models. *ACM Transactions on Computer Systems*, May 2005.
10. Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGOPS Oper. Syst. Rev.*, 25:63–74, 1991.
11. Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
12. P. M. W. Gill. Molecular Integrals over Gaussian Basis Functions. *Advances in Quantum Chemistry*, 25:141–205, 1994.
13. P. M. W. Gill, Benny G. Johnson, John A. Pople. A simple yet powerful upper bound for Coulomb integrals. *Chemical Physics Letters*, 217:65–68, 1994.
14. Roland Lindh. Integrals of Electron Repulsion. In P. v. R. Schleyer et. al, editor, *Encyclopaedia of Computational Chemistry*, volume 2, page 1337. Wiley, 1998.
15. X. Vera, N. Bermudo, and A. González J. Llosa. A Fast and Accurate Framework to Analyze and Optimize Cache Memory Behavior. *ACM Transactions on Programming Languages and Systems*, March 2004.