Building fast, reliable, and adaptive software for computational science

Alistair P Rendell, Joseph Antony, Warren Armstrong, Pete Janes and Rui Yang
Department of Computer Science, Australian National University, Canberra, ACT0200 Australia
E-mail: Alistair.Rendell@anu.edu.au

Abstract. Building fast, reliable, and adaptive software is a constant challenge for computational science, especially given recent developments in computer architecture. This paper outlines some of our efforts to address these three issues in the context of computational chemistry. First, a simple linear performance that can be used to model and predict the performance of Hartree-Fock calculations is discussed. Second, the use of interval arithmetic to assess the numerical reliability of the sort of integrals used in electronic structure methods is presented. Third, use of dynamic code modification as part of a framework to support adaptive software is outlined.

1. Introduction

It's an exciting time for computational science. After years of incremental changes to computer architectures we are at the dawn of a new era. Multicore is now on the desktop with processor counts steadily increasing. Graphics cards and special-purpose processors such as the NVIDIA GeForce 8800 GTX and Cell Broadband Engine provide flexible programming environments and astonishing performance for just a few hundred dollars. And if this were not enough, the release of the Sun ROCK processor sees transactional memory move from an academic curiosity into mainstream.

How to harness these new developments and construct code that is fast, reliable, and adaptive to its environment is a real challenge for the computational scientist. In this paper we outline some of our work to address these issues. To build fast software, we are interested in designing performance models that can be used to predict performance and thereby guide program execution. Detailed here is our attempt to use a simple linear performance model to describe and predict the performance of the Gaussian quantum chemistry code [1] as a function of cache usage and memory page placement on a variety of systems. For reliable software we are seeking to include the effects of rounding and truncation errors into scientific computations. To achieve this, we are developing interval versions of standard quantum chemistry methods. To facilitate the development of adaptive software, we are using DynInst [2] to dynamically modify running executables.

2. Building fast software

Central to understanding the performance of any computational application is an appreciation of the costs associated with accessing the various data structures used. Increasingly processors have multiple levels of cache that shadow a memory system that has non-uniform memory access (NUMA) characteristics. Two key issues are therefore understanding application performance as a function of cache usage, and as a function of memory page placement. With these in mind our goal has been to develop simple...
performance models that capture these effects and can be used to predict performance as a function of algorithmic or system hardware modifications.

Below we outline briefly some of our work that has explored the use of a simple linear performance model to characterize the cache performance of the Gaussian quantum chemistry code [1] on a variety of modern hardware systems. We then show how this model can be combined with cache simulation to study the effect of cache variation on application performance, before discussing how our model can be extended to include the effects of NUMA architectures.

2.1. A simple linear performance model
Recently we proposed that under certain circumstances a simple linear performance model (LPM) may be sufficiently accurate to provide predictive information for use in calculating cache blocking factors for quantum chemistry calculations [3]. In this model the overall performance is predicted as a simple linear combination of instructions issued and cache misses (i.e. requests by the processor to load or store an item from/to memory where that item is not found to be resident in cache):

\[ Cycles = \alpha * (I_{\text{Count}}) + \beta * (L1_{\text{Misses}}) + \gamma * (L2_{\text{Misses}}), \]

where \(I_{\text{Count}}\) is the instruction count, \(L1_{\text{Misses}}\) the total number of Level 1 cache misses, \(L2_{\text{Misses}}\) the total number of Level 2 cache misses, and \(\alpha, \beta, \gamma\) are penalty factors. Loosely speaking, the value of \(\alpha\) reflects the ability of the code to exploit the underlying super-scalar architecture, \(\beta\) the average cost of an L1 cache miss, and \(\gamma\) the average cost of an L2 cache miss.

We have investigated the above in the context of Hartree-Fock (HF) calculations as implemented within the Gaussian quantum chemistry code [1]. Under most circumstances the time required to perform an HF calculation is dominated by formation of the so-called Fock matrix (\(F\)). This has the following form:

\[ F_{\mu\nu} = H_{\mu\nu} + \sum_{\lambda\sigma} \sum_{i} C_{\lambda i} C_{\sigma i}^{*} [2(\mu\nu | \lambda\sigma) - (\mu\lambda | \nu\sigma)], \]

where \(\mu\nu\cdots\) and \(i\) are indices for the basis functions and molecular orbitals respectively, \(N\) is the total number of basis functions, and \(N_{e}\) is the total number of electrons in the system. In equation 2 each \(F_{\mu\nu}\) involves another two-index quantity \(H_{\mu\nu}\), and a four-index quantity \(2(\mu\nu | \lambda\sigma)\) contracted with a two-index quantity \(C_{\lambda\sigma}\). The \((\mu\nu | \lambda\sigma)\) are the electron repulsion integrals (ERIs), and for most HF calculations evaluation of these integrals and their contraction with \(C\) is what dominates the overall computation time.

In the PRISM algorithm [4] used by Gaussian the ERIs are evaluated in batches, where ERIs in the same batch share some common intermediates and involve the same set of operations, the latter make each batch amenable to processing on single instruction multiple data (SIMD) or vector architecture. There is a balance between larger batch sizes causing improved performance because of smaller instruction counts and better exploitation of SIMD architectural features, but too large batches causing the cache to overflow leading to a loss in performance. The Gaussian code uses cache blocking to limit the maximum size of an integral batch and avoid cache overflow.

To illustrate this effect, we show performance data table 1 for an HF calculation as the cache blocking size is varied from 4–1024 KiW. (The blocking factor relates to the size of an internally allocated array within the Gaussian program and is measured in 8-byte words). The test system is taken from a molecular dynamics simulation of a solvated potassium ion equilibrated at 300K and then truncated to contain only the central potassium ion and any water molecules that have atoms within 4Å of it. (In subsequent results the same system will be used but truncated at 8Å and containing 80 water molecules.) This shows cycle times of \(5.38 \times 10^{10}\) with a block size of 4 KiW that reduce to \(2.75 \times 10^{10}\) when the block size is expanded to 64 KiW, but then increases to \(4.64 \times 10^{10}\) as the block size is further increased to 1024 KiW. Between best and worse case scenario this represents a performance difference of nearly a factor of 2. Clearly the goal is to determine the optimal blocking size, but as this may change according to
the particular calculation being performed and hardware being used, this should really be done using an
adaptive algorithm.

Table 1. Performance of the PRISM integral evaluation code as a function of cache blocking size for a
Hartree-Fock calculation on a solvated potassium ion (K\(^+\)(H\(_2\)O\(_{11}\)) using 6-31G* basis on a 2.2 GHz
AMD848 Opteron. See text for further details.

| Cache Blocking Size (KiW) | Number of Batches | Instruction Count (x10\(^{10}\)) | L1 Cache Misses (x10\(^{8}\)) | L2 Cache Misses (x10\(^{7}\)) | Floating Point Operations (x10\(^{9}\)) | CPU Cycles (x10\(^{10}\)) |
|--------------------------|-------------------|---------------------------------|---------------------------|----------------------------|--------------------------------|--------------------------|
| 4                        | 226922            | 5.38                           | 3.79                      | 1.68                       | 6.24                          | 5.38                     |
| 16                       | 75729             | 4.11                           | 3.83                      | 2.35                       | 6.15                          | 2.99                     |
| 64                       | 20049             | 3.18                           | 5.43                      | 1.39                       | 6.18                          | 2.75                     |
| 256                      | 5624              | 2.91                           | 7.50                      | 4.89                       | 6.14                          | 3.58                     |
| 1024                     | 2174              | 2.84                           | 8.82                      | 6.51                       | 6.08                          | 4.64                     |

Using the data in Table 1, we can perform a least squares fit to equation 1 and obtain values for \(\alpha\), \(\beta\) and \(\gamma\). Sample results for four different computations (molecular system, basis set) obtained on an Opteron system are given in Table 2. The values for \(\alpha\) are all positive and suggest that on average between 1.4 and 1.6 instructions are being executed each cycle. For \(\beta\) two values are, however, negative with the unphysical interpretation that it takes negative time to retrieve data from the L1 cache! For \(\gamma\) all values are positive and roughly what we might expect a level 2 cache miss penalty to cost (e.g. if measured using LMBench [5]). On other platforms we found similar problems extracting meaningful values for \(\beta\) and believe this is partly due to the effects of out-of-order execution (although there are also issues identifying exactly which hardware counter(s) to use in order to compute total L1 cache miss). For this reason we chose to remove L1 misses as a parameter in the LPM.

Table 2. LPM Parameters for 5 different HF Calculations on the Opteron.

| System Basis | K\(^+\)(H\(_2\)O\(_{11}\)) | K\(^+\)(H\(_2\)O\(_{80}\)) | K\(^+\)(H\(_2\)O\(_{11}\)) | Valinomycin | \(\alpha\)-Al\(_2\)O\(_3\) |
|--------------|------------------------|-----------------|-----------------|-------------|-----------------|
| K\(^+\)(H\(_2\)O\(_{11}\)) | 6-31G*                 | 6-31+G(3df,3pd) | 6-31G*          | 3-21G       | 3-21G*          |
| \(\alpha\)   | 0.7                    | 0.6             | 0.7             | 0.7         | 0.6             |
| \(\beta\)    | 5.1                    | 2.0             | -0.4            | -4.5        | 4.5             |
| \(\gamma\)   | 320.3                  | 403.1           | 411.0           | 367.4       | 452.6           |

To access the accuracy of the LPM for a range of different processors, we present in table 3, the
average relative errors obtained when using the LPM to fit cycle times for five different HF calculation
types on seven different compute platforms. The results show that on average the LPM is able to fit the
observed performance to an accuracy of around 5%, and in some cases much better than this.

2.2. Performance prediction using the LPM
As demonstrated above the LPM is capable of describing the observed cycle count for Gaussian HF
calculations with an accuracy of \(\sim 5\%\) across a range of different calculation types and architectures. To
obtain this did, however, require that the computation first be run on that hardware recording instructions
issued and cache misses. An alternative to using hardware performance counters to obtain this data is to
use functional cache simulation; this simulates cache behavior given the load/store memory operations
generated by the program. Callgrind [6] is a tool that permits such simulations. It performs dynamic,
execution driven cache simulation by using the Valgrind framework for dynamic binary instrumentation
Table 3. Percentage average relative error in fitted cycle time using LPM for a range of different calculations and systems.

| Molecular System and Basis | K⁺(H₂O)₁₁ | K⁺(H₂O)₈₀ | K⁺(H₂O)₁₁ | Valinomycin | α-Al₂O₃ |
|---------------------------|------------|------------|------------|-------------|--------|
|                           | 6-31G*     | 6-31++G(3df,3pd) | 6-31G* | 3-21G | 3-21G* |
| Opteron                   | 3.2        | 1.0        | 2.7        | 1.3        | 2.0    |
| EM64T                     | 2.7        | 1.4        | 2.0        | 2.7        | 0.7    |
| Pentium 4                 | 2.3        | 1.8        | 2.7        | 0.5        | 2.2    |
| Pentium M                 | 2.0        | 0.8        | 1.9        | 1.0        | 2.6    |
| G5                        | 1.8        | 1.5        | 2.7        | 2.2        | 2.7    |
| G5-XServe                 | 2.4        | 1.7        | 2.2        | 2.4        | 3.2    |

[7]. Thus, by combining the values of α and γ derived by fitting to data obtained on existing hardware, with instruction and cache miss counts given by Callgrind for a “similar” system, we can perform architectural studies. This would include addressing questions such as “What performance effect would we predict for Gaussian HF calculations if we doubled the cache size, or modifying some other characteristic?”

To illustrate this, we show in table 4 the various event counts obtained via hardware performance counters on an Opteron with those given by Callgrind for the original Opteron cache configuration, the same system but with the L1 and L2 cache line size increased and the same system with a larger L2 cache (while also increasing the cache blocking parameter used by Gaussian). (The cache line size determines the number of bytes transferred between main memory and cache when a request is made by the processor to access a data item that is currently not resident in the cache.) By combining these counts with the values for α and γ obtained when running on real hardware, we can predict the overall cycle times on the modified hardware. Not surprisingly, the results show that decreasing the L1 line size to 32B has no effect, but increasing the L2 line size causes a slight decrease from $2.50 \times 10^{10}$ to $2.36 \times 10^{10}$ cycles. Increasing the L2 cache size to 16MB while also adjusting the cache blocking factor is much more significant, resulting in nearly a 20% reduction in the predicted cycle count.

Table 4. Hardware and Callgrind counts for various L1 and L2 cache configurations on AMD Opteron based system system together with LPM predicted cycle counts.

| Hardware | Callgrind |
|----------|-----------|
|          | 64KiB     | 64KiB     | 64KiB     | 64KiB     | 64KiB     |
| L1 Size  | 64B       | 64B       | 32B       | 64B       | 64B       |
| L1 Line size | 1MB      | 1MB      | 1MB      | 1MB      | 16MB      |
| L2 Size  | 64B       | 64B       | 64B       | 64B       | 1024B     |
| L2 Line size | 64KiW    | 64KiW    | 64KiW    | 64KiW    | 1MW       |
| ICount   | 3.28E+10  | 3.27E+10  | 3.27E+10  | 3.27E+10  | 2.93E+10  |
| L2$Miss$ | 9.72E+06  | 1.44E+07  | 1.44E+07  | 8.90E+06  | 5.24E+06  |
| Est. Cycles | 2.72E+10 | 2.50E+10  | 2.50E+10  | 2.36E+10  | 2.01E+10  |

2.3. Extending the LPM for NUMA
The LPM developed above predicts performance based solely on the instruction count and total L2 cache misses. On many multicore shared memory platforms all L2 cache misses are not equal; rather the
memory has non-uniform access characteristics. A simple extension to the LPM would be to divide the L2 cache miss penalty \( \gamma \) into multiple terms that are each associated with a cache miss to a particular domain within the NUMA system. While in principle this would be easy, in practice hardware counters are currently not able to provide such a breakdown.

We are addressing the above limitation in two ways. First, we have performed a number of studies looking at the effect of memory placement on the performance of the Gaussian application. This work involves strategically placing certain data structures at specific locations within the NUMA architecture and building up a performance model based on the resulting execution times and data from various hardware counters [8, 9]. In [8] we show that even with good cache blocking it is possible to see a difference in runtime of over 40% between for a Gaussian HF calculation that uses good or poor memory placement on 16 cores of a Sunfire X4600 M2 system.

Our second approach is to extend Callgrind to include a NUMA memory model on top of the existing cache model. This work is still in progress, but will allow us to map easily cache misses to particular memory domains and determine which memory pages are best located near a particular process.

3. Building reliable software

To date, the majority of computational science is performed by using 64-bit floating-point arithmetic. This provides \( \sim 15 \) significant figures. On a petaflop machine performing in excess of \( 10^{15} \) operations per second it is not at all clear whether this is sufficient accuracy. To address this issue, we have been investigating the use of interval arithmetic to bound errors in computational science applications.

Interval arithmetic is not new; it but was proposed in the early days of computing as a means of bounding rounding errors that occur in floating-point calculations [10]. Formally, an interval \( \mathbf{X} = [X, \bar{X}] \) contains the set of all real numbers between two endpoints corresponding to the lower \( X \) and upper bound \( \bar{X} \), referred to as the infimum and supremum respectively. Thus, an interval may represent, among other things, a bound on the true result of an uncertain quantity, where the source of uncertainty may include a combination of rounding and truncation errors.

The set of basic arithmetic operations \( \bullet \in \{+, -, \times, \div\} \) is defined such that uncertainties in \( X \) and \( Y \) are propagated onto the result:

\[
X \bullet Y = \{x \bullet y | x \in X, y \in Y\}
\]  

where for example \( X + Y = [\bar{X} + \bar{Y}, \underline{X} + \underline{Y}] \), which if implemented using IEEE754 [11] floating-point arithmetic will use downward \( \underline{\ } \) and upward rounding \( \bar{\ } \) for the infimum and supremum respectively.

There exists a duality between expressions in floating-point and interval arithmetic. An interval equivalent of a floating-point expression, referred to as a natural interval extension, can be obtained by simply replacing all floating-point arithmetic operations and functions such as \( \exp, \sin, \cos \) with their interval equivalents. The interval obtained by this translation contains both the true numerical result and the result that is obtained by the equivalent ordinary floating-point calculation [12]. Thus, if a floating-point calculation is unstable due to large rounding or truncation errors, then its interval extension will return wide interval bounds.

According to the containment principle an interval implementation must guarantee that the result it returns will bound the true result of the particular calculation in question, taking into account uncertainties introduced by rounding and truncation errors [13]. Thus if, for example, an expression for \( \sin(x) \) is computed by evaluating its Taylor series expansion, then the interval returned must include the effects of both the rounding errors associated with the operations used to compute the series, and the truncation error resulting from only evaluating a finite number of terms in an infinite series. One of the most challenging problems faced when writing interval code is to find the interval with the narrowest width, as this places the most precise bounds on the true result.

We are interested in using intervals to place rigorous bounds on the sort of integrals used in quantum chemical calculations, and propagating these through an entire calculation to quantify the errors in the
The final property being computed. Specifically we have considered the same ERIs as given in Equation 1. The majority of algorithms for computing these rely on the evaluation of the incomplete gamma function [14], denoted here as $F_m(T)$ and defined as follows:

$$F_m(T) = \int_0^1 t^{2m} e^{-T t^2} \, dt,$$  

(4)

where $T$ is a non-negative real value (dependent on the distance between the basis functions) and $m$ is an integer (the value of which depends on the total angular momentum of the functions involved). Values of $T$ range from 0 to over 1,000, while $m$ typically ranges from 0 up to 16. In a typical calculation $F_m(T)$ may be evaluated many millions of times.

In modern quantum chemistry codes exactly how $F_m(T)$ is evaluated depends on both the values of $T$ and $m$. In general there are two alternative approaches. For a given $m$ if $T$ is less than some threshold ($T_F$) the value of $F_m(T)$ is based on finite evaluation of an infinite series, while above the threshold an asymptotic approximation is used. In order to speed evaluation of the finite series, interpolation tables based on either Taylor [15, 16] or Chebyshev [17] polynomial approximations are constructed. Interpolation points are generated along the center of each uniformly discretized partition $[X-h, X+h]$ of the domain $[0, T_F]$, with values at each interpolation point computed using the truncated series.

In short, when evaluating $F_m(T)$ errors can result from fundamental approximations made before any computation has begun, such as use of the asymptotic approximation or a particular form of interpolation, as well as from rounding errors associated with the various floating-point operations involved. Using interval arithmetic we have been able to place rigorous error bounds on all of these errors [18]. This is illustrated in table 5 where the average relative interval widths for $F_m(T)$ computed using five different polynomial approximation schemes and two different implementations are given. The results clearly show that the use of ChebyA, a three term interpolation in $T$ gives rise to large interval widths, particularly for large $m$ values. The alternative approaches are all roughly equal, giving about 14 significant figures of accuracy in $F_m(T)$. Alg2, which uses compensated summation to minimize the effect of rounding errors in series evaluation, is able to decrease the interval width by a factor of about 4.

| Scheme | $m$ |
|-------|-----|
|       | 0   | 4   | 8   | 12  |
| ChebyA Alg1 | 8.3e-14 | 4.3e-12 | 2.2e-11 | 6.7e-11 |
| ChebyA Alg2 | 3.2e-14 | 1.6e-12 | 7.9e-12 | 2.3e-11 |
| ChebyB Alg1 | 1.4e-14 | 1.7e-14 | 1.9e-14 | 2.1e-14 |
| ChebyB Alg2 | 5.7e-15 | 7.2e-15 | 8.1e-15 | 8.9e-15 |
| ChebyC Alg1 | 1.5e-14 | 1.8e-14 | 2.0e-14 | 2.1e-14 |
| ChebyC Alg2 | 7.1e-15 | 8.5e-15 | 9.3e-15 | 1.0e-14 |
| TaylorA Alg1 | 1.4e-14 | 1.7e-14 | 1.8e-14 | 2.0e-14 |
| TaylorA Alg2 | 4.8e-15 | 6.2e-15 | 7.1e-15 | 8.0e-15 |
| TaylorB Alg1 | 1.4e-14 | 1.7e-14 | 1.9e-14 | 2.0e-14 |
| TaylorB Alg2 | 5.6e-15 | 7.0e-15 | 7.9e-15 | 8.7e-15 |

In other work [19] we have used interval analysis to compare electrostatic energies for a 3D distribution of point charges computed using a pairwise summation with those evaluated using fast multipole methods (FMM). Using intervals, we were able to show how adding terms to the FMM treatment improves accuracy as truncation errors decrease, but then errors begin to increase as rounding errors become significant.
4. Building adaptive software

Traditional methods for program optimization have involved repeated cycles of performance measurement, analysis and tuning. Increasingly, however, tuning is runtime dependent reflecting what else is running on the system at that time. For instance, on a multicore system with a shared cache it may be useful to adjust cache usage dynamically depending on what is happening on other cores. Moreover, there can be cases where the algorithm of choice can change during the course of the computation. For example, in particle simulations it is often advantageous to exploit linear scaling algorithms, but only if the particles involved are well spaced. If they move during the course of the simulation, whether they are well spaced or not can change over time.

While there are a number of possible approaches to achieving dynamic optimization, we have been investigating the use of DynInst [2] to perform dynamic modifications to a running executable. Our basic design goal is to produce a tool that can attach to a running program, inject sensors into it to measure its performance, analyze the results using machine learning techniques, and then affect changes in the execution path of that program [20]. This high-level design is illustrated in figure 1. In principle our goals are similar to what can be achieved using a managed language run on a virtual machine (e.g., Java), but the advantage of our approach is that it is applicable to the large body of existing computational science applications written using C/C++ or Fortran.

As with any optimization that aims to improve performance, it is important to quantify the costs associated with introducing those optimizations. In this case it is important to understand the costs associated with using DynInst to modify a running executable. DynInst provides three main methods for dynamic modification. The first involves inserting fragments of code at designated points - for example, inserting code to measure how many cache misses a function is incurring. The second modification involves manipulating global variables within the address space of the target process, for example, reading them and writing to them. The third modification involves redirecting function calls; for example, if $f$ and $g$ are procedures with identical parameter counts, parameter types, and return types, then any call to $f$ may be replaced by a call to $g$, while maintaining syntactic validity. (Note that caution must be employed when doing this in the face of interprocedural optimization by the compiler and or the linker.)

The basic mechanism for modifying the behavior of a running binary is known as a trampoline. This can be thought of as a diversion of the target program’s flow of control. There are two main kinds of trampoline: base trampolines and mini trampolines. Figure 2 shows how these trampolines interact with each other, and with the target application. The base trampolines are responsible for saving and restoring the state of the CPU, while the mini trampolines contain the inserted code. Only mini trampolines must be explicitly inserted by the user of DynInst; base trampolines are placed automatically where required.

As mentioned above, the base trampoline is responsible for saving and restoring the state of the processor. At a minimum, this state consists of all general-purpose registers and CPU flags. However, if the inserted code is going to make use of floating-point computations, it is also necessary that floating-point state be saved and restored as well. In the general case it is also necessary to guard against the
possibility that the modified code is inserted into a routine that calls itself, leading to infinite recursion. To handle such situations, DynInst provides guard constructs.

Beyond the one-off cost of modifying the running executable (something that involves stopping the process, making the modifications and then restarting it, and should therefore not be taken lightly), it is of interest to consider the overhead associated with executing the trampolines since this will occur every time that portion of the original code is executed. We have designed a benchmark to measure this overhead [21]. Results are shown in table 6 for a “do-nothing” trampoline located at various points within the target routine. The results show that execution of a trampoline with no checks for recursion or saving of floating point registers adds roughly 110 cycles to the execution time every time this code is executed. Including checks for recursion is relatively cheap, but saving floating point state can be expensive. The latter is true when the code is inserted in the body of a routine, rather than at its head, reflecting the fact that if the trampoline is inserted at the head of a routine floating point state is already saved as part of the calling convention and is therefore not repeated by DynInst.

5. Conclusions and discussion
This paper has outlined some of our efforts to develop fast, reliable and adaptive software for computational science applications. The linear performance model was shown to describe the cache behavior of HF calculations to an accuracy of around 5% across a wide spectrum of hardware. This suggests that over time the intricate details of program execution average out and are dominated by instruction counts and level 2 cache misses. As presented here, the model was parameterized for specific HF calculations, it is of interest to explore its domain of applicability. For example, using values for $\alpha$ and $\gamma$ obtained by fitting to a specific HF calculation, how well can we predict execution times for a different basis set, molecular system, or method based only on the measured instruction counts and cache misses. Extending the model to include effects of NUMA is a challenge due to lack of hardware counters to breakdown cache misses by memory domain. If page placement was random over the entire NUMA system, using a single $\gamma$ that was in some way an average would probably suffice. We are investigating these issues by extending the Callgrind tool to include a NUMA memory model.

For the use of interval methods to place rigorous numerical error bounds on computational science, applications support by the Sun compiler of an interval data type for Fortran and a C++ template library has greatly eased our work. Performance of interval codes remains a major issue, not least because they require additional operations to be performed. For example an interval addition requires two floating point addition operations one for the infimum and one for the supremum, while interval multiplication
Table 6. Per-invocation overhead for executing DynInst modified code on an x86 platform as a function of mechanism and location.

| Safety Checks                  | None          | Recursion     | Floating Point | Recursion & Floating Point |
|-------------------------------|---------------|---------------|----------------|---------------------------|
| **inline trampolines located in loop header** |               |               |                |                           |
| Wall time (ns)                | 37±0          | 39±0          | 159±0          | 162±0                     |
| CPU cycles                    | 108±0         | 115±0         | 476±0          | 498±3                     |
| Instructions                  | 12±0          | 19±0          | 20±0           | 27±0                      |
| **inline trampolines located in called function** |               |               |                |                           |
| Wall time (ns)                | 35±0          | 38±0          | 35±0           | 37±0                      |
| CPU cycles                    | 102±0         | 109±0         | 102±0          | 109±0                     |
| Instructions                  | 11±0          | 18±0          | 11±0           | 18±0                      |
| **outline trampolines located in loop header** |               |               |                |                           |
| Wall time (ns)                | 37±0          | 39±0          | 160±0          | 168±0                     |
| CPU cycles                    | 110±0         | 116±0         | 476±0          | 501±2                     |
| Instructions                  | 15±0          | 24±0          | 23±0           | 32±0                      |
| **outline trampolines located in called function** |               |               |                |                           |
| Wall time (ns)                | 35±0          | 38±0          | 35±0           | 38±0                      |
| CPU cycles                    | 104±0         | 112±0         | 104±0          | 112±0                     |
| Instructions                  | 14±0          | 23±0          | 14±0           | 23±0                      |

can required up to eight (depending on how it is implemented), and in both cases there will be the need to switch rounding modes. Thus our use of intervals to date has been more of a proof of concept rather than a practical large-scale implementation. For carefully rewritten kernels (in developing any tuned kernel care must be taken to ensure that the code still guarantees containment), our work suggests that an interval code will be about one order of magnitude slower than the equivalent floating point code [22, 23]. However, since real estate on the processor die is now less of a bottleneck compared to the rate at which data can be moved on or off the chip, it is possible that future hardware could support native interval operations at a much lower performance penalty. Whether this would be a good option or whether it would be better to move to quadruple precision is partly the rationale for our exploratory work with intervals.

In developing a framework to support adaptive software, use of DynInst to affect change in a running application is an interesting option. Our benchmarking has shown that executing a piece of code that has been modified by DynInst adds between 100 and 500 cycles, depending largely on whether it is necessary to save floating-point state. This factor needs to be considered carefully when using DynInst. What we have not discussed here is the software for deciding when to make modifications to a running code, and what those changes should be. Our current work in this area is focused on use of machine learning techniques, particularly reinforcement learning, in the context of optimal storage formats for sparse matrices [24].

In summary and as mentioned initially, it’s an exciting time for computational science. There are many challenges ahead, but also many opportunities.

**Acknowledgments**

This work was supported in part by Australian Research Council grants Discovery grant DP0558228 and Linkage grant LP0347178 with partners Gaussian Inc. and Sun Microsystems.
References

[1] Frisch M J, Trucks G W, Schlegel H B, Scuseria G E, Robb M A, Cheeseman J R, Montgomery Jr J A, Vreven T, Kudin K N, Burant J C, Millam J M, Iyengar S S, Tomasi J, Barone V, Mennucci B, Cossi M, Scalmani G, Rega N, Petersson G A, Nakatsuji H, Hada M, Ehara M, Toyota K, Fukuda R, Hasegawa J, Ishida M, Nakajima T, Honda Y, Kitao O, Nakai H, Klene M, Li X, Knox J E, Hratchian H P, Cross J B, Bakken V, Adamo C, Jaramillo J, Gomperts R, Stratmann R E, Yazyev O, Austin A J, Cammi R, Pomelli C, Ochterski J W, Ayala P Y, Morokuma K, Voth G A, Salvador P, Dannenberg J J, Zakrzewski V G, Dapprich S, Daniels A D, Strain M C, Farkas O, Malick D K, Rabuck A D, Raghavachari K, Singh W, Pople J A, Gaussian 03, Revision D.02. Gaussian, Inc., Wallingford, CT, 2004

[2] Buck B R and Hollingsworth J K 2000 Journal of High Performance Computing Applications 14 317 – 329

[3] Antony J, Frisch M and Rendell A P in Press International Conference on High Performance Scientific Computing, Hanoi, Vietnam March 2006

[4] Gill P M W, Head-Gordon M and Pople J A 1990 Journal of Physical Chemistry 94 5564–5572

[5] McVoy L and Staelin C 1996 USENIX Annual Technical Conference pp 279–294

[6] J Weidendorf KCachegrind – A Profile Visualisation Tool http://kcachegrind.sourceforge.net/ URL http://kcachegrind.sourceforge.net/

[7] Nethercote N and Seward J 2007 PLDI ed Ferrante J and McKinley K S (ACM) pp 89–100 ISBN 978-1-59593-633-2

[8] Yang R, Antony J, Janes P P and Rendell A P 2008 IISPAV (IEEE Computer Society) pp 31–36

[9] Antony J, Janes P P and Rendell A P 2006 HiPC (Lecture Notes in Computer Science vol 4297) ed Robert Y, Parashar M, Badrinath R and Prasanna V K (Springer) pp 338–352 ISBN 3-540-68039-X

[10] Moore R E 1959 Automatic error analysis in digital computation Tech. Rep. LMSD-48421 Lockheed Missiles and Space Company URL interval.louisiana.edu/Moores\_early\_papers/Moore_Lockheed.pdf

[11] IEEE Computer Society Standards Committee Working group of the Microprocessor Standards Subcommittee and American National Standards Institute 1985 IEEE standard for binary floating-point arithmetic ANSI/IEEE Std 754-1985. (Silver Spring, MD: IEEE Computer Society Press)

[12] Caprani O, Madsen K and Nielsen H B 2002 Introduction to interval analysis URL http://www2.imm.dtu.dk/pubdb/p.php?1462

[13] Kearfott R B and Kreinovich V 1996 pp 1–22 URL citeseer.ist.psu.edu/kearfott95applications.html

[14] P M W Gill 1994 Advances in Quantum Chemistry 25 141–205

[15] McMurchie L E and Davidson E R 1988 J. Chem. Phys. 89 5777–5786 URL http://link.aip.org/link/?JCP/89/5777/1

[16] Obara S and Saika A 1986 J Chem. Phys. 84 3963 URL http://dx.doi.org/

[17] Gill P M, Johnson B G and Pople J A 1991 International Journal of Quantum Chemistry 40 745–752

[18] Janes P and Rendell A P In Press 2008 IEEE 11th International Conference on Computational Science and Engineering

[19] Rendell A P, Clarke B, Janes P, Milthorpe J and Yang R 2007 “International Conference on Computational Science and its Applications (ICCSA), IEEE Conference Proceedings (DOI 10.1109/ICCSA.2007.49) (IEEE Computer Society) pp 457 – 466

[20] Armstrong W, Christen P, McCreath E and Rendell A P 2006 Proceedings of the International Workshop on Integrating AI and Data Mining (AIDM 2006), IEEE Publications (ISBN 0-7695-2730-2, Digital Object Identifier: 10.1109/AIDM.2006.9) pp 18 – 24

[21] Armstrong W and Rendell A P To be published

[22] Rendell A P, Clarke B and Milthorpe J 2006 International Conference on Computational Science (1) (Lecture Notes in Computer Science vol 3991) ed Alexandrov V N, van Albada G D, Sloot P M A and Dongarra J (Springer) pp 218–225 ISBN 3-540-34379-2

[23] Milthorpe J and Rendell A P 2005 Proceedings of the Third Australian Undergraduate Computing Conference, Canberra 2005 pp 34–41 ISBN 0-9757173-1-6

[24] Armstrong W and Rendell A P 2008 The Third international Workshop on Automatic Performance Tuning (IWAPT08), (submitted)