The Development of a Database-driven Application Benchmarking Approach to Performance Modelling

DOI:
10.1109/hpcsim.2014.6903760

The Development of a Data-driven Application Benchmarking Approach to Performance Modelling

A. Osprey*†, G. D. Riley‡, M. Manjunathaiah*, and B. N. Lawrence*†
*University of Reading, UK
†National Centre for Atmospheric Science (NCAS), UK
‡University of Manchester, UK
{a.osprey, m.manjunathaiah, b.n.lawrence}@reading.ac.uk, graham.riley@manchester.ac.uk

***Abstract*—Performance modelling is a useful tool in the lifeycle of high performance scientific software, such as weather and climate models, especially as a means of ensuring efficient use of available computing resources. In particular, sufficiently accurate performance prediction could reduce the effort and experimental computer time required when porting and optimising a climate model to a new machine.**

Yet as architectures become more complex, performance prediction is becoming more difficult. Traditional methods of performance prediction, based on source code analysis and supported by machine benchmarks, are proving inadequate to the task. In this paper, the reasons for this are explored by applying some traditional techniques to predict the computation time of a simple shallow water model which is illustrative of the computation (and communication) involved in climate models. These models are compared with real execution data gathered on AMD Opteron-based systems, including several phases of the U.K. academic community HPC resource, HECToR. Some success is had in relating source code to achieved performance for the K10 series of Opterons, but the method is found to be inadequate for the next-generation Interlagos processor.

The experience leads to the investigation of a data-driven application benchmarking approach to performance modelling. Results for an early version of the approach are presented using the shallow model as an example. In addition, the data-driven approach is compared with a novel analytical model based on fitting logarithmic curves to benchmarked application data. The limitations of this analytical method provide further motivation for the development of the data-driven approach and results of this work have been published elsewhere.

***Keywords*—Performance modelling; benchmarking; multicore, shallow water model**

I. INTRODUCTION

In climate modelling and numerical weather prediction, improved performance over time arises in roughly equal proportions from algorithmic developments and through the purchase of new hardware, which is typically refreshed every four or five years. Over time, computer architectures have become more complex, with current processors containing 6, 8 or 12 cores, and implementing out-of-order instruction execution, deep memory hierarchies, sophisticated communication systems within shared memory nodes, and complex interconnection networks between nodes. As a result, performance is becoming ever more complex to predict.

The ultimate motivation for this work is be able to compare the performance of different runtime scenarios under which a code might execute on a given HPC system. Scenarios may differ in the number of processes (and threads allocated); the domain decomposition used with a particular allocation of processes; and the mapping of processes to specific cores (often called an affinity mapping). These options lead to a large parameter space, and the aim of this work is to develop a performance modelling tool to support the rapid exploration of options, seeking scenarios which provide (near) optimal performance. Generally, climate modelling centres explore this space experimentally, but this is costly in terms of both human effort and computational resources, see for example Edwards [1].

Performance modelling is a useful process for ensuring that high performance computing (HPC) applications, such as climate models, make the best use of available computing resources. Performance modelling can inform choices of new algorithmic developments as well as support performance tuning of codes on existing and new architectures, thus helping target the, typically scarce, development effort available. Furthermore it can be used to reduce the cost of computer time for experimentation to find good deployment configurations (Kerbyson and Jones [2], Barker et al. [3]).

Analytical application models parameterise performance in terms of key application inputs, and run-time deployment options, see Hoefler et al. [4]. Such models break the application down into computational kernels, and communication events, then use empirical or analytical techniques to estimate the time to execute each portion of work.

In this paper, we examine different methods of modelling the computational work of a simple shallow water model [5] that replicates the type of work present in a typical climate model. The target architectures considered are all systems composed of various iterations of the AMD Opteron processor series.

We begin with a detailed analytical model of the processor, that counts the cycles to complete floating point operations

This research was partly sponsored by the EU FP7-Infrastructures-201201 project, IS-ENES2 (GA312979)

978-1-4799-5313-4/14/$31.00 ©2014 IEEE

plus loads and stores to and from cache and memory. It will be established that, even for a simple model, traditional analytical modelling techniques are inadequate, even when supplemented with run-time or benchmarked machine information. The limitations of these techniques motivate the development of a data-driven approach to performance modelling, based on the targeted collection of application benchmark data to support the exploration of runtime scenario choices.

This empirical modelling approach is illustrated by evaluating against measured results, and comparing it with a more sophisticated analytic model, based on fitting logarithmic curves to benchmarked data. The results of these modelling efforts are positive, and provide support for the benchmark-driven method. The authors have found this method to be the most appropriate for capturing both computation and communication behaviour, though the focus in this paper is on the modelling of computation only. The utility of the data-driven approach is demonstrated in a companion paper (Osprey et al. [6]) addressing the evaluation of runtime scenario deployment choices when mapping processes to cores in complex multicore HPC architectures. In ongoing work the approach is being applied to other architectures including IBM Power 7 and IBM Blue-Gene/Q.

The structure of the paper is as follows. Section II describes the shallow water model, Section III describes the AMD Opteron-based target architectures used in the study, Section IV reviews traditional analytical performance modelling techniques based on computational intensity and theoretical peak performance measures and illustrates their use and limitations on the shallow water model. Section V motivates and develops the data-driven empirical model, based on the collection of targeted application benchmark data, and then Section VI develops a more complex, novel model for sthe shallow water example based on fitting mathematical functions to the measured data. Finally, Section VII concludes and discusses future work.

II. SHALLOW WATER CODE

The NCAR shallow water model [5], is a small program that solves the shallow water equations with a second-order finite-difference scheme [7] on a horizontally staggered Arakawa C grid [8]. The shallow water equations are a simplified version of those solved in the dynamical cores of complex weather and climate models such as the UK Met Office Unified Model (UM). The code we use (hereafter called "shallow") originated from the NCAR website [5] but was substantially rewritten for the purpose of this work. The full code structure is outlined in Fig. 1.

Shallow performs calculations over a rectangular domain of size M by N with periodic boundary conditions in both directions to replicate the behaviour on a sphere whilst avoiding the use of poles. Local domains are sized m by n with arrays dimensioned as $m+1$ by $n+1$ to allow for a single halo row and column. [a] There are 13 local array fields and at each timestep the code performs 10 array update loops, 3 array copies, and 7 exchanges of halo data. Similar stencil-based computational work over arrays of data and communications to update halo regions form significant parts of the UM dynamical core.

Listing 1. Structure of shallow code.

```
Initialisation:
* Read inputs (iterations, problem size
   processor decomposition)
* Allocate array space
* Initialise scalar variables for solver
* Initialise velocities (U and V),
   pressure (P) and stream function (PSI)
* Apply periodic boundary conditions

Time-stepping loop:
* Compute CU, CV, Z and H (from U, V and P)
* Apply periodic boundary conditions to
   CU, CV, Z and H
* Compute UNEW, VNEW and PNEW
   (from U, V, P plus CU, CV, Z and H)
* Apply periodic boundary conditions to
   UNEW, VNEW and PNEW
* Time smoothing and update for next tstep
   (UOLD, VOLD, POLD -> U, V, P)
   (U, V and P -> UNEW, VNEW and PNEW)

Finalisation:
* Write timer information
* Deallocate arrays
```

The first 7 loops only update inner points `(1:M,1:N)` as boundary points are updated in a separate step, and the final 3 loops update the full array domain including boundaries `(1:M+1,1:N+1)`. We consider each array update loop individually to emulate a larger scale application with multiple compute kernels. A summary of the operations performed in each loop is given in Table I. Loads account for spatial cacheline reuse, since if two values are adjacent in memory, i.e. `a(i,j)` and `a(i+1,j)`, they will reside in the same cacheline and only a single load will be required.

The "reference flops" is the number of floating point operations listed in the source code for each loop, and is used to calculate the performance in Gflops/s. [b] The use of reference flops provides a baseline for comparison across systems. Flops derived from hardware counters can give different results due to compiler optimisations, speculative branch execution (not an issue here), and packed SSE instructions which may be counted as a single operation.

Here, addition, multiplications and subtractions are defined

[a] This is unlike other dynamical core which often have at least 2 halo rows and columns due to dependencies in all directions.

[b] In this work flops will be used as an abbreviation for floating point operations, and flops/s for the performance metric floating point operations per second.

TABLE I
BREAK DOWN OF OPERATIONS IN EACH ARRAY UPDATE LOOP

	Memory accesses	Stores	Loads with reuse	Reference flops
cu	cu, p, u	1	2	3
cv	cv, p, v	1	3	3
z	z, v, u, p	1	5	13
h	h, p, u, v	1	4	9
unew	unew, uold, z, cv, h	1	6	10
vnew	vnew, vold, z, cu, h	1	6	10
pnew	pnew, pold, cu, cv	1	4	6
uold	uold, u, unew	1	3	5
vold	vold, v, vnew	1	3	5
pold	pold, p, pnew	1	3	5

TABLE II
ACTUAL FLOATING POINT OPERATIONS

	Actual flops
cu	2 *, 1 +
cv	2 *, 1 +
z	2 *, 2 +, 3 -, 1 /
h	4 *, 4 +
unew	3 *, 4 +, 2 -
vnew	3 *, 3 +, 3 -
pnew	2 *, 4 -
uold	2 *, 2 +, 1 -
vold	2 *, 2 +, 1 -
pold	2 *, 2 +, 1 -

as 1 flop, and divides are counted as 5. This is because the Opteron processor used in this work takes 5 times as many cycles to execute a divide (Section III). The precedence for counting non-add/multiply operations as multiples is from the Parkbench project [9]. Compiler optimisations remove some redundant operations that can be carried over to the next loop iteration, and so these are not included in the "actual flops" listed in Table II.

To measure performance, timers are inserted around each individual array update loop using `MPI_Wtime` (the recommended timer for the HECToR architecture at the time). Unless otherwise stated shallow is run for 4000 iterations with all inputs except problem size and parallel decomposition kept constant

III. AMD OPTERON BASED TARGET ARCHITECTURES

Experiments were performed on four systems: Chronos, a small cluster at the University of Manchester, and Phase 2a, 2b and 3 of HECToR, the UK's national academic supercomputing service from 2007 to 2013 [10]. The AMD processors used in Chronos (4-core Shanghai), Hector Phase 2a (4-core Barcelona) and 2b (12-core Magny-Cours) are all from the AMD K10 family of processors. A comparison between these machines is given in Table III along with the compilers used. HECToR Phase 3 used the more complex AMD Interlagos processor.

The K10-based AMD Opteron can concurrently dispatch 11 operations: 3 integer execution, 3 address generation, 3 floating point and multimedia (add, multiply and misc) and 2 loads or stores to data cache [11]. Additionally, with packed SSE instructions it should be possible to perform 2 adds and 2 multiplies simultaneously. Each of these operations takes a number of cycles to complete, but some or all of this can be hidden by prefetching, branch prediction and out-of-order execution. Floating point operations take: 4 cycles for add and multiply, 20 cycles for double precision divide and 27 cycles for double precision square root. There are pipelines so the adds and multiplies can reach a throughput of 1 per cycle. A detailed description of the Opteron architecture and processing of instructions is given by de Vries [12].

Each core has 64 KB 2-way set associative L1 caches for data and instructions, plus a 512 KB 16-way set associative L2 cache. Cachelines are 64 bytes in length. Caches follow a victim-cache regime, where data are loaded directly into L1 cache and evicted to L2 when no longer needed, and then on to L3 when space runs out. L3 cache is fully shared with the other cores on the processor, with no need for duplication [13]. Data are removed from L3 only when no cores require it. (The alternative method, used by Intel processors, is for inclusive caches where the highest cache level replicates data held by the lower caches.) On the Magny-Cours, 1 MB of L3 cache is given over to HT Assist, a system that manages cache coherency, leaving 5 MB cache shared amongst the cores on the die [14].

TABLE III
DETAILS OF THE AMD OPTERON K10 MACHINES USED

	Chronos	Phase 2a (Cray XT4)	Phase 2b (Cray XE6)
Processor	4-core Shanghai	4-core Barcelona	2 x 12-core Magny-Cours
Core speed	2.4 GHz	2.3 GHz	2.1 GHz
L1 cache	64 KB	64 KB	64 KB
L2 cache	512 KB	512 KB	512 KB
L3 cache	6 MB shared	2 MB shared	6 MB shared per 6-core die
Memory	24 GB DDR2	8 GB DDR2	32 GB DDR3
Mem speed	600 MHz	800 MHz	1333 MHz
Compiler	gfortran -O3	pgf90 -fastsse	pgf90 -fastsse crayftn -O3

HECToR Phase 3, a Cray XE6 system, is based on an AMD Opteron Interlagos processor which has a somewhat different architecture. The Interlagos is part of the Bulldozer series of Opteron processors [15]. Processor sockets contains two dies that each comprise four "compute modules". Compute modules are made up of two integer cores that include a load store unit, 16 KB of private L1 cache and an integer scheduler. Both cores share 2 MB of L2 cache and a single floating point unit that is double the width of the K10 series, allowing for a greater amount of flexibility. Applications with greater per-core memory requirements can run with only a single integer unit per module ("core-pair mode") whilst still having access to the full floating point capability. Running with both cores ("compact mode") means more instructions feeding the floating point pipeline and thus generally should provide greater performance.

IV. ANALYTICAL MODELS

Due to their complexity, detailed models of modern processors are generally only applied to small code kernels that follow a single execution pattern. They do, however, provide a means of understanding low-level performance issues, and if the method is systematic, it can be made automatic, see for example the PMaC performance prediction framework (Snavely et al [16], [17]).

Generally the two most important factors to consider are the speed at which data can be accessed from cache or memory, and the speed at which floating point operations can be executed.

A. Related work

Snavely et al., 2001 [16] predict performance in Gflops/s from the number of floating point operations (flops) executed divided by the number of memory operations (mops), scaled by the speed per memory operation. This assumes that the code is "memory-bound" and that any time spent performing floating-point operations is overlapped by time transferring cachelines. Note that the number of flops divided by the number of mops is also known as the "compute intensity". Since Gflops/s are derived from the floating point operations divided by run time, the run time is effectively only dependent on the performance of memory operations. That is, the initial formulation

$$\text{Perf} = \left(\frac{\text{flops}}{\text{mops}}\right) \times \text{bandwidth}, \tag{1}$$

reduces to

$$\text{Time} = \frac{\text{mops}}{\text{bandwidth}}, \tag{2}$$

where the bandwidth is calculated from the location and access patterns of the memory operations.

This is equivalent to the performance estimated from so called balance-metrics, described by Callahan et al., 1988 [18]. They state that the processor efficiency can be calculated from the "machine balance", which is the rate of memory accesses divided by the rate of flops, divided by the "loop balance", which is the number of memory accesses divided by the number of flops. Multiplied by the peak processor speed, this gives the predicted performance as

$$\text{Perf} = \left(\frac{(\text{bandwidth / flop rate})}{(\text{mops / flops})}\right) \times \text{flop rate}. \tag{3}$$

The floating point terms can then be factored out and the remaining terms rearranged to give (2). Note that to be accurate the bandwidth should relate to the location of the data, which may be in cache.

Datta et al., 2009 [19] consider the effect of prefetching on the achieved bandwidth, noting that several cache misses occur before full streaming bandwidth is achieved. The time to update a loop is given by the time to load the first cacheline, plus the time to load the next k cachelines before full streaming is achieved (at some intermediate cost), plus the time to load the remaining cachelines at the full streaming bandwidth. These costs are determined by using a memory benchmark and varying the access patterns and block sizes.

When the data are resident in L1 cache, accesses can be very fast and the run-time limited by the speed of floating point operations. Snavely et al., 2002 [17] account for floating point work by adding an extra term to (2):

$$\text{Time} = \frac{\text{mops}}{\text{bandwidth}} + \frac{\text{flops}}{\text{peak flop rate}}, \tag{4}$$

where the flop rate is the theoretical peak for the processor. As discussed by the authors, this does not account for the overlap of flops and mops and the theoretical peak is a highly unrealistic value. In later work, a more sophisticated convolution is used that considers the overlap between the terms (Carrington et al., 2005 [20]).

Treibig et al [21] present a more detailed method that counts the cycles required to perform the floating point and data transfers. They consider the number of operations that can be performed simultaneously and the cost of performing each one, taking into account the cycles to load data from each cache level. This is found to work well for the processors studied but requires detailed knowledge of the architecture and low level measurements in some cases.

In the following sections we use the methods described here to determine whether the performance of shallow can be predicted from source code operations and machine metrics.

B. Estimating memory-bound performance

The performance when data are resident in main memory is estimated using (1) for each individual loop of shallow. Compute intensity (flops/mops) can be reported from the PGI compiler, but this does not account for cacheline reuse, so it is derived from the operations listed in Table I. As the theoretical maximum bandwidth is unlikely to be achieved by real code, we simply plot compute intensity versus measured performance (Fig. 1). Ideally, a realistic memory bandwidth would be measured from a benchmarking tool.

The compute intensity values show a near-linear relationship with the observed performance, suggesting a model for memory-bound performance based on two machine factors:

$$\text{Perf (Gflops/s)} = \beta \cdot \frac{\text{flops}}{\text{mops}} + \alpha. \tag{5}$$

The parameter values derived for each system are listed in Table IV. Note that if α is ignored, then $(1/\beta)$ is equivalent to the achieved memory bandwidth. Thus on the HECToR systems, the shallow kernel achieves around 1 Gword/s or approximately 8 GB/s.

C. Estimating peak performance

The theoretical peak performance is the maximum performance that could be expected given the operations that need

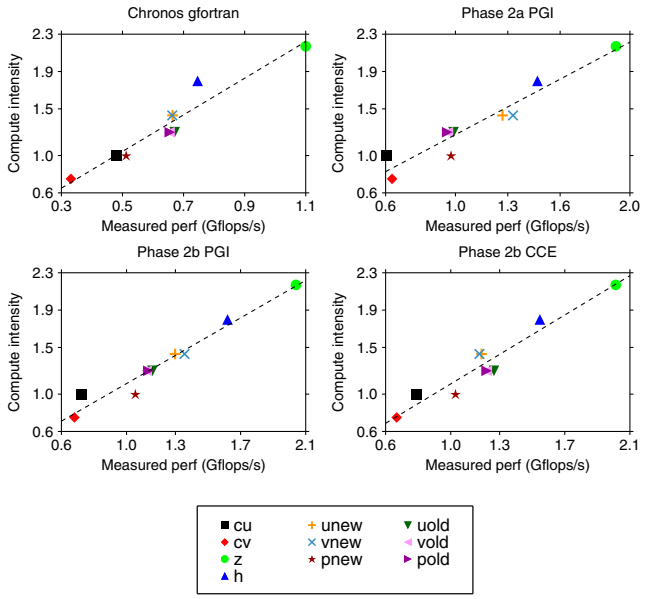

Figure 1. Compute intensity against performance in memory (from a 512 × 512 problem size) for different machine and compiler combinations for each computation loop in shallow.

TABLE IV
PARAMETER VALUES FOR LINEAR DATA FIT OF MEMORY-BOUND PERFORMANCE (5).

System	β	α
Phase 2b PGI	1.005	0.108
Phase 2b CCE	1.054	0.0550
Phase 2a PGI	0.991	0.233
Chronos gfortran	1.961	0.0625

to be executed. A model based on the method described by Treibig et al [21] is developed, considering only the L1 case.

The Opteron processor can perform several tasks concurrently (see Section III) including two memory operations (either two loads or one load and one store) and two floating point operations (one add and one multiply or up to two adds and two multiplies with SSE instructions). From this information, the minimum number of cycles required to update one cacheline is derived from the operations listed in Table I, assuming full pipelining, full use of SSE vectorised instructions where possible and prefetching such that all cachelines are ready in L1 cache when needed.

For example, the `cu` loop requires 1 store and 2 loads (as `p(i+1,j)` will be reused at the next iteration it need only be loaded once). These operations can be completed in 2 cycles. Simultaneously on the floating point registers 2 vectorised multiplies and 1 addition can be executing. For the `z` loop careful consideration must be made to the floating point divide. This takes 5 cycles to execute on the multiply register when fully pipelined, during which time the minuses and adds can execute, then an additional cycle is required to perform two vectorised multiplies for a total of 6 cycles. The memory operations in this case take only 3 cycles and so are fully hidden.

The theoretical peak performance is then given by dividing the number of flops by the minimum execution cycles and then multiplying by the core speed as follows:

$$\text{Perf (Gflops/s)} = \frac{\text{flops}}{\text{cycles}} \times \text{clock speed (GHz)}. \qquad (6)$$

This results in predictions that are out by a considerable factor, but do show a strong correlation to the observed peak performance (Fig. 2).

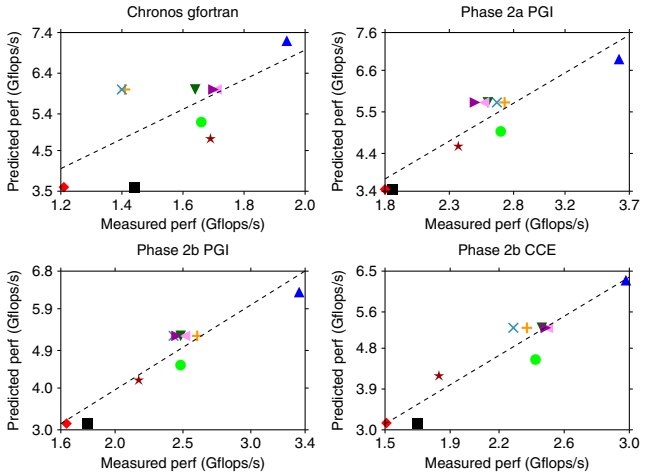

Figure 2. Theoretical peak performance against peak performance measured form a series of runs, for different machine and compiler combinations. Key for the loops is as in Fig. 1.

Again the strong relationship implies a linear model such as

$$\text{Perf (Gflops/s)} = \delta \cdot \frac{\text{flops}}{\text{cylces}} + \gamma. \qquad (7)$$

The derived parameter values for each system are listed in Table V.

TABLE V
PARAMETER VALUES FOR LINEAR DATA FIT OF PEAK PERFORMANCE (7).

System	δ	γ
Phase 2b PGI	0.968	-0.0498
Phase 2b CCE	1.028	-0.0519
Phase 2a PGI	0.872	0.0506
Chronos gfortran	1.518	-0.131

Although the peak metric is based on PGI compiler optimisations, it works well for the Cray compiler too, suggesting both implement the same type of optimisations. The cycle count does not work as well for Chronos, which is likely to be due to the system in general (including the gfortran compiler) being less tuned for high performance scientific applications. In particular, it is hypothesised that the Chronos runs do not include SSE vectorisation, since the achieved performance is nearly two times slower, despite having a slightly faster processor.

A more detailed comparison of the loop performance on Chronos versus HECToR shows that the peak performance occurs for larger problem sizes on Chronos - when the data are in L2 or L3 cache rather than L1 or L2 cache. If it is supposed that much of the performance difference is due to the compiler, then it could be inferred that compiler optimisations have more of an effect when the problem fits into the lower level caches. Therefore the lack of optimisations would mean that small amounts of data would not be sufficient to efficiently utilise the floating point pipeline, and better performance would only be seen when the pipeline was saturated with a large steady stream of data. The shape of the Chronos plots could then be interpreted as having the L1/L2 peak missing. This explains why the theoretical peak metric was not appropriate as it is based on L1 cache performance. A more appropriate model would consider the compiler optimisations actually applied and the data transfer cost from the correct cache level.

D. Conclusions

Some success was achieved with these theoretical metrics. Although we were unable to make accurate predictions, it was possible to identify which loops would perform better than others based on their instruction mix. A realistic memory bandwidth could be estimated from a standard memory benchmarking tool, however it is unclear how to derive the peak performance scaling without directly benchmarking the code. Attempts to extend this work to the Interlagos processor (HECToR Phase 3) were unsuccessful due to i) different compiler optimisations being applied, ii) the complexity of the two integer cores sharing a single wide floating point unit. Thus the conclusion from using this simple model is that realistic performance prediction from the source code is too complex for modern processors due to unpredictable factors, such as the compiler translation to machine code, and techniques such as out of order execution.

V. EMPIRICAL MODEL

As has been shown, it is difficult to make accurate performance predictions without some form of application benchmarking. In this section a simple empirical model is developed based on interpolation between measured performance for different problem sizes. Careful selection of problem sizes, respecting the cache architecture of the target machine, reduces the cost of collecting the data. Once defined, evaluating the model is cheap, making it suitable for exploring different runtime scenarios.

Outline

A series of benchmark experiments of different problem sizes are used to provide a reference performance (P) in Gflops/s. This is then multiplied by the number of floating point operations performed during the run to predict the total runtime for that loop. Thus the time to complete one block of computations over the whole run is

$$T(s) = \text{P (flops/s)} \times \text{flops} \times I_{\text{dim}} \times J_{\text{dim}} \times N_{\text{itr}}, \quad (8)$$

where flops is the number of floating point operations performed on each loop iteration, I_{dim} and J_{dim} are the loop dimensions and N_{itr} is the number of time-stepping iterations. The reference performance is given by a lookup function with the total memory usage as input, which for shallow would be

$$\text{P (flops/s)} = \text{Lookup}\{\,(M+1)(N+1) \times 13 \times 8 \text{ bytes}\,\}, \quad (9)$$

where $(M+1)$ by $(N+1)$ are the array sizes, of which there are 13, all comprising 8 byte real numbers.

Results

Results are reported from HECToR phase 3 with the Cray compiler. Benchmark runs were performed over a set of 23 square problem sizes from 1×1 to 600×600 to provide a range of problem sizes from L1 cache resident to memory resident. Fig. 3 shows the results with a linear interpolation between measurements. As the plot uses a log-scale the interpolated lines appear logarithmic. Performance is plotted against the memory usage for the program in order to estimate where the data reside in cache or memory. The memory usage is calculated from the local array size times the number of arrays in use (13) times the number of bytes per element (8).

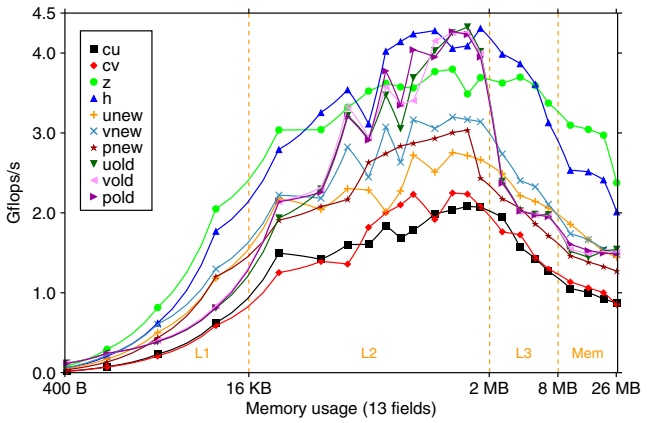

Figure 3. Empirical model for each computational loop in shallow for different problem sizes (based on linear interpolation between measured values).

As an estimate of how well this model characterises the performance over all problem sizes, the estimates are compared to measured performance for a larger set of runs (Fig. 4). Note that these figures are plotted on a linear x-axis, and L2 results are plotted on a smaller scale as otherwise this detail is lost. The loops are split into 3 plots for clarity only.

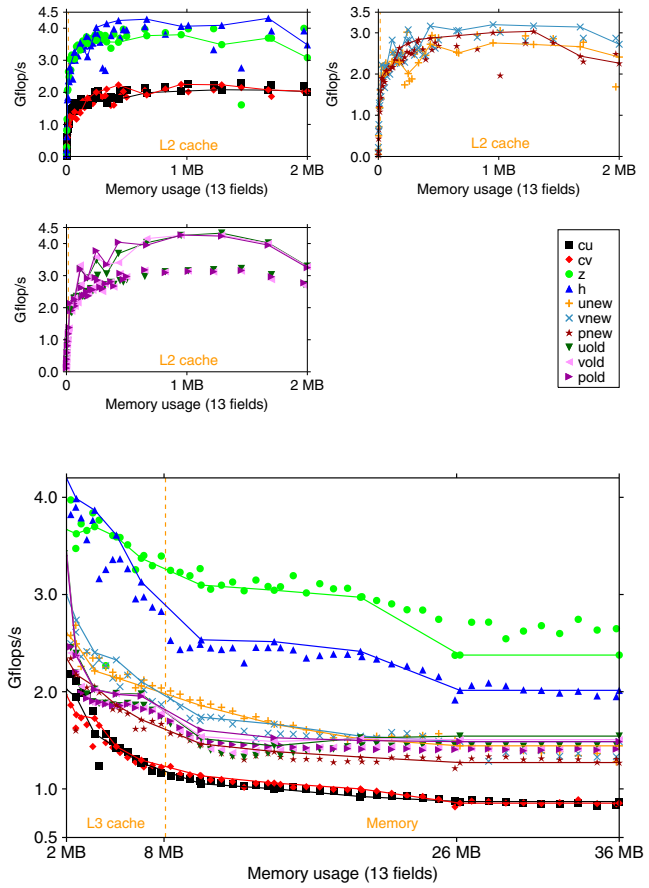

Figure 4. Predicted performance from empirical model plotted against measured results for various problem sizes.

Discussion

The benchmark model agrees reasonably well with the observed performance, apart from some pathological peaks and troughs. It is hypothesised that the dips are due to cache-thrashing. The L1 cache is 2-way set-associative and when the arrays have certain sizes, the same indices of all arrays can be mapped to the same cache location, causing frequent cache eviction. This is only observed when the arrays are dynamically allocated. When the array sizes are fixed, the compiler can avoid these issues by padding and other techniques. Two performance modes appear in the `[uvp]old` loops between the empirical model values (connected by lines), and the values from the validation run. Further investigation showed these pattern to be repeatable and the separation between each set of runs, seems to be simply due to the regular increase between array sizes.

As the problem size increases, clear performance steps can be seen, with a reasonably constant performance in L2, then a slope down towards a performance plateau in L3, then another slope down towards a plateau in memory. This is the usual behaviour seen on cache-based processors, and the behaviour modelled by Kerbyson and Jones [2]. The model doesn't capture this exactly, except for where the values are close to the "corners" of the function, however a smaller set of benchmark values is used. The exponentially spaced problem sizes appear to be well chosen in this regard as the cache sizes also increase on an exponential scale, and so there are benchmark values at all cache levels.

VI. LOGARITHMIC MODEL

The results plotted in the previous Section (Fig. 3 and Fig. 4), infer that the observed performance steps between cache levels could be represented by logarithmic lines. This motivated an attempt to develop a more sophisticated analytical model based on only the peak performance and the performance in memory. The model is presented in this section and compared with the results from the purely empirical approach.

Outline

The shape of the measured results suggest a log interpolation may provide a good fit to the data. If performance was benchmarked at the cache boundaries the log fit may capture the sloped-step effect observed whilst still using only a small number of values. The difficulty in capturing the effect linearly is that the location of the plateaus and slopes are different for different loops.

Here performance is represented by two logarithmic line, one from the origin to the peak performance, and one from the peak to the performance in memory, and a constant memory performance once convergence has taken place. This is described by an analytical model defined by three data points. As logarithmic lines are used, the first data point (X_0, Y_0) cannot be exactly at the origin, therefore a performance of 0 is defined for some very small problem size. The peak performance is described by the point (X_1, Y_1) and the performance in memory by (X_2, Y_2), where X_i is the size in bytes of the memory used, and Y_i is the performance in Gflops/s. Once the performance has reached the convergence point (X_2), then the constant memory performance is used (Y_2).

The performance for any data size is given by

$$\text{Perf} = \begin{cases} A \log x + \lambda, & \text{if } x \leq X_1, \\ B \log x + \phi, & \text{if } X_1 \geq x \leq X_2, \\ Y_2, & \text{if } x \geq X_2, \end{cases} \tag{10}$$

where the constants A, B, λ and ϕ can be derived by plugging the value of the defined data points into the linear equations.

The advantages of this model over the purely empirical model, are that the performance is encapsulated using fewer values, and a good choice of values could allow for smoothing over the observed discontinuities.

Results

The model is evaluated using the measured data from the previous Section. The peak and memory convergence points were hand-selected to avoid outlying points, although this could also be done using formal statistical regression. We choose to ignore the higher peaks for `[uvp]old` to better fit the majority of the data.

Fig. 5 shows the analytical model plotted against the measured data points. Again the results are split into plots for L2 cache, and L3 cache and memory. The larger problem sizes in memory are not plotted as they converge to the same value, allowing for more detail to be seen at smaller scales. The results show that the model reasonably follows the measured performance, to a level of accuracy comparable to the empirical model.

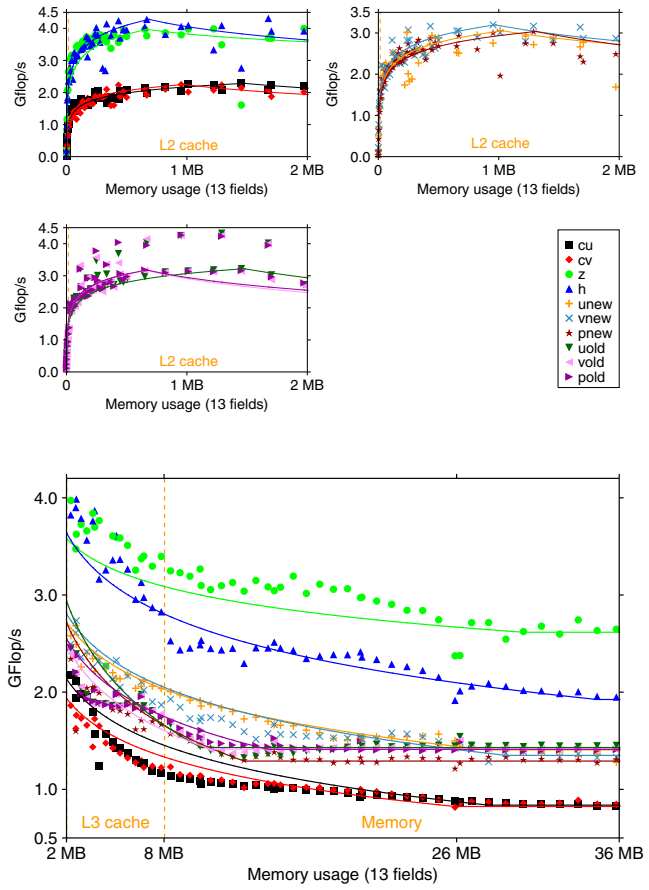

Figure 5. Predicted performance from analytical model plotted against measured results for various problem sizes.

Discussion

The success of the model confirms the hypothesis that the performance over all problem sizes can be represented by logarithmic lines between the origin, peak performance and performance in memory. Whilst this provides a simple and novel representation of performance, the difficulty in practise is in defining the input data points to use. It has already been seen in Section IV that a relationship can be derived from analysis of the source code, but actual values have to depend on application benchmarks. Furthermore the location of the peak and memory convergence need to be determined. Some success may be had by using fixed locations, for example, a peak location near the middle of L2 cache, but this is likely to be error prone. Thus, in practise, this model may be difficult to implement. Further, it remains to be determined whether the performance of shallow on other machines or the performance of other applications can be represented in this way.

VII. SUMMARY AND FUTURE WORK

Using a shallow water benchmark code illustrative of the dynamical cores of weather and climate codes, this paper has explored the use of traditional, source code based methods of the performance prediction of computation costs and demonstrated their limitations on modern, complex architectures based on the AMD Opteron family. These traditional methods are usually supported by some form of machine-based benchmarking. The use of targeted application-based benchmark data collection in a data-driven approach to performance modelling of runtime deployment scenario choices has been presented and compared with a novel technique using logarithmic curves to approximate the benchmarked data.

In ongoing related work the authors have applied the data-driven approach to the typical neighbour-based communication patterns involved in climate models with similar positive results. In other ongoing work, the work with shallow is being evaluated on other architectures, notably a large IBM Power 7 system, as currently used by the Met Office, and an IBM BlueGene/Q machine in order to test the robustness of the approach across architectures.

The utility of the data-driven approach has been demonstrated in a companion paper [6] addressing the evaluation of runtime scenario deployment choices when mapping processes to cores in complex multicore HPC architectures.

REFERENCES

[1] T. Edwards, "Optimising UPSCALE on HERMIT," Cray CoE for HECToR, Report, May 2012.

[2] D. J. Kerbyson and P. W. Jones, "A performance model of the Parallel Ocean Program," *International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 261–276, 2005.

[3] K. J. Barker, K. Davis, and D. J. Kerbyson, "Performance modeling in action: Performance prediction of a Cray XT4 system during upgrade," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[4] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:12.

[5] *NCAR HPC shallow water model tutorial*, UCAR, October 2006, http://www.cisl.ucar.edu/docs/hpc_modeling/.

[6] A. Osprey, G. Riley, M. Manjunathaiah, and B. Lawrence, "A benchmark-driven modelling approach for evaluating deployment choices on a multi-core architecture," in *PDPTA '13: Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications, July 22–25*, 2013.

[7] R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of the Atmospheric Sciences*, vol. 32, pp. 680–689, 1975.

[8] A. Arakawa, "Computational design for long-term numerical integration of the equaions of fluid motion: Two dimensional incompressible flow. Part 1," *Journal of Computational Physics*, vol. 1, pp. 119–143, 1966.

[9] M. Berry, "Public international benchmarks for parallel computers: Parkbench committee: Report-1," *Scientific Programming*, vol. 3, pp. 100–146, June 1994, chairman: Roger Hockney.

[10] *HECToR - UK National supercomputing service*, UoE HPCX Ltd, The University of Edinburgh, 2011, http://www.hector.ac.uk/.

[11] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron processor for multiprocessor servers," *Micro, IEEE*, vol. 23, no. 2, pp. 66 – 76, March-April 2003.

[12] H. de Vries, "Understanding the detailed architecture of AMD's 64 bit core," Chip Architect, September 2003.

[13] A. Bailey, "Barcelona's innovative architecture is driven by a new shared cache," AMD arcticle, August 2007, http://developer.amd.com/documentation/articles/pages/8142007173.aspx.

[14] T. Carver, "Magny-Cours and Direct Connect Architecture 2.0," AMD article, March 2010, http://developer.amd.com/documentation/articles/pages/magny-cours-direct-connect-architecture-2.0.aspx.

[15] D. Kanter, "AMD's Bulldozer microarchitecture," Real World Tech online article, August 2010, http://www.realworldtech.com/bulldozer/.

[16] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatures with application profiles," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 149–156.

[17] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–17.

[18] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 334–358, 1988.

[19] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, pp. 129–159, February 2009.

[20] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336–346, 2006.

[21] J. Treibig, G. Hager, and G. Wellein, "Multi-core architectures: Complexities of performance prediction and the impact of cache topology," *CoRR*, vol. abs/0910.4865, 2009.