# Dynamic Performance Prediction of an Adaptive Mesh Application

Mark M. Mathis and Darren J. Kerbyson

Performance and Architecture Laboratory (PAL)
CCS-3, Computer and Computational Sciences (CCS-3)
Los Alamos National Laboratory
{mmathis,djk}@lanl.gov

## Abstract

*While it is possible to accurately predict the execution time of a given iteration of an adaptive application, it is not generally possible to predict the data-dependent adaptive behavior the application will take and therefore to predict the total execution time for a given execution. To remedy this situation we have developed an executable performance model that can be utilized dynamically at runtime directly from the application of interest. In this manner, the application itself can rapidly predict the expected execution time for its next iteration based on current information on the data layout and level of adaptivity. This enables the application itself to determine: if an optimum level of performance is being achieved (i.e. by comparing measured and predicted times); when to perform a checkpoint (if the next iteration will exceed a predefined time limit between checkpoints); or when to terminate (if the next iteration will exceed the application's system time allocation for instance). The dynamic model is shown to have high accuracy over a number of test cases, even in the presence of interference (system activities that are not a part of application activities).*

## 1. Introduction

Performance modeling is an important tool that can be used by a performance analyst to provide insight into the achievable performance of a system and/or an application. It is only through knowledge of the workload for the system that a meaningful performance comparison can be made. It has been recognized that performance modeling can be used throughout the life-cycle of a system, or of an application, from first design through to maintenance [7] including procurement and system installation.

Recent work at Los Alamos National Laboratory (LANL) has demonstrated the use of performance modeling in many situations, for instance: in the early design of systems; during the procurement of new systems; in exploring possible optimizations in applications prior to implementation [6]; and in verifying the performance of ASCI Q system during installation [9] – which lead to optimizing the performance of the system by a factor of two [14]. Models have also been used to compare the performance of large-scale systems including several of the highest peak-rated terascale systems such as the Earth Simulator and ASCI Q [8].

One of the large-scale applications used in these studies is SAGE (SAIC's Adaptive Grid Eulerian hydrocode). It is a multi-dimensional (1-D, 2-D, & 3-D), multi-material, Eulerian hydrodynamics code with adaptive mesh refinement (AMR). SAGE has been applied to a variety of problems in many areas of science and engineering including water shock, energy coupling, cratering and ground shock, stemming and containment, and hydrodynamic instability problems. SAGE is representative of part of the ASC (Accelerated Strategic Computing) workload at Los Alamos that routinely runs on thousands of processors. An overview of SAGE and its sister code, RAGE, is given in [4].

A performance model of SAGE has been previously developed and validated on a number of systems [6]. What makes SAGE interesting for our current work is its adaptivity. That is, as the application progresses from cycle to cycle, individual spatial cells can be divided or combined to provide greater calculation resolution in areas of interest. Unfortunately, the adaptation taken over the course of an execution cannot be predicted from initial conditions. This means that it is not generally possible to predict the total execution time (i.e. for all cycles) of SAGE. So, the question we seek to answer is, "Can the time for the next cycle be dynamically predicted?".

There are a number of uses for this type of dynamic modeling (e.g. [1,10]). One is to make runtime decisions about the execution of the application. For example, the predicted time could be used to decide when to checkpoint (e.g. [3,19]). That is, if the next cycle will exceed a predefined limit of accumulated time plus the optimal checkpoint interval, then a checkpoint should be done before the next cycle. Alternatively, the predicted time could be used to dynamically determine the number of cycles for a particular execution. For example, in a batch system, it can be determined (to a relatively high degree of accuracy) whether or not another cycle can be completed before the time allocation runs out. This could be particularly useful if the scheduler is not very forgiving of programs that run over their time limits.

Equally important is in determining if the predicted performance is actually being realized. For instance, is the machine working at an optimum level or is it being perturbed by external factors (e.g. operating-system noise [14] or a transient hardware failure such as an un-seated cable in the network). A dynamic performance model is able to identify when the expected performance is not being achieved.

Dynamic performance models are also useful for scheduling in grid environments (e.g. [2]) perhaps in conjunction with network monitoring tools (e.g. [18]). This approach allows long-running applications to be re-allocated as resources wax and wane. The models previously used in these applications are often very high-level or statistical in nature (e.g. [16,17]). Our approach is to apply the detailed analytical modeling approach developed at LANL to these interesting applications of performance analysis.

In order to provide dynamic predictions, one must have an executable version of the performance model. To accomplish this we utilize a modified version of a performance specification language called CHIP$^3$S [13]. This is intended to provide predictions through a discrete event simulation of the performance characteristics of the application code. For our current work, since a high level model is known *a priori*, we can express the model directly in the specification language. This greatly increases the efficiency of the evaluation and therefore the viability of runtime use.

The rest of this paper is organized as follows. In Section 2 we briefly describe key features of the model and its implementation. In Section 3 we validate the dynamic model using a 64 node HP AlphaServer system and a 32 node Itanium-2 cluster. The techniques developed here for dynamic performance prediction are shown to have reasonable accuracy in all test cases. In Section 4 we further test the accuracy of the performance model under dynamic conditions and use the model to determine optimal checkpoint intervals.

## 2. Dynamic Model Implementation

A performance model can be implemented in a number of ways including a straightforward coding in which all the analytical details of the performance model form the basis for the evaluation, and inputs represent characteristics of the system (e.g. communication performance, node size, topology etc.) as well as characteristics of the current data-set being processed (e.g. number of cells, level of adaption etc.). These inputs are dynamic and, in the case of the application parameters change from cycle to cycle.

In this work we implemented the performance model of SAGE using a performance specification language (PSL). A modified version of the CHIP$^3$S language was utilized for this purpose [13]. A CHIP$^3$S model consists of a *hardware specification* (i.e., a system model) and a *parallel template* which implements a *task graph* representation of the application. The nodes of the task graph are then specified by an *application* model. This results with several input files that together form a program that can then be compiled and executed to produce performance predictions. Importantly for this work is that the executable may be linked with, and hence used by, another application using a runtime evaluation interface.

By default a CHIP$^3$S application model is a mapping from the source code to a performance domain. The evaluation system can then take this model specification and predict the performance of the individual tasks. Although it is perhaps useful to facilitate modeling of single-processor performance, we wish to focus primarily on interactions of multiple-processors (and minimize the evaluation process). In this case, it is more expedient to measure the single-processor performance, and predict the performance of the parallel application.

In fact, our approach to performance modeling does not currently address the modeling of single-processor performance. In most cases it is sufficient to benchmark the single-processor time and use the benchmark measurement in the model. In the case of strong scaling, where the problem size is fixed, the time per element will change as the number of elements assigned to each processor changes largely due to memory hierarchy effects. That is, the greatest performance will be seen when each processor's sub-grid fits in cache. In this case a simple piecewise model obtained from benchmark measurements is required to capture the single-processor performance of the application [11].

SAGE, however, normally operates in a weak scaling mode. That is, a equal number of cells is mapped to each processor. In this manner, more processors are used to increase the fidelity of the simulation rather than decrease the execution time. In general, this would allow a single value to be used to model the single-processor performance (since the amount of work per processor remains constant). However, the adaptive nature of SAGE means that the number of cells changes throughout the execution of the application. Although the input-deck specifies a number of cells per processor at the beginning of the execution, the number of cells will typically increase as the mesh is adapted.

To account for this fact, we provide two different modes of operation for the dynamic model:

*online* mode: timers in the application are used to extract a time-per-cell (effectively a grind-time) and by using a window over previous cycles, a prediction of the run-time for the next cycle can be made using knowledge of the number of cells being processed.

*offline* mode: uses a detailed analytical model of the application and pre-measured computation characteristics (application dependent) and communication characteristics (application in-dependent). Thus time-per-cell input is pre-measured.

The online model works well when the goal is to make dynamic decisions at runtime based on current system performance. However, it is not a good approach to take when verifying the health of the system (i.e., "Are we getting the best, expected, performance?"). The time-per-cell input to the offline model can be obtained by varying the number of cells on a single processor benchmark. One way to accomplish this is to run a number of cycles on a single processor with adaption turned on. This will give several performance values which can described by a piecewise linear (or logarithmic) curve (e.g., Figure 1). In this manner, a baseline single processor performance can be obtained that (with high confidence) represents the best achievable performance.

The prediction accuracy of both the online and offline models is compared in Section 3. In section 4 the effectiveness of both models is examined on a system which may be perturbed by un-expected activities.

## 3. Model Validation

It is not our goal here to validate the SAGE performance model itself as it has been previously validated on many systems [6]. Rather, our overall goal is to demonstrate how a performance model can be used dynamically at runtime. To that end, we do need to validate the executable version of the model and show that it can be used with a minimum perturbation to the existing code. We do this using three input-decks (timing_a, timing_b, and timing_c) for SAGE, that are often used to access its performance for a large number of cycles, on several different system configurations. A summary of the input-decks is given in Table 1 in terms of the initial number of cells assigned to each processor (which can change due to adaption), and the type of calculation performed (hydro and/or heat).

**Table 1. SAGE input-decks**

| | Timing_a | Timing_b | Timing_c |
|---|---|---|---|
| Initial cells per PE | 4,000 | 4,000 | 80,000 |
| Hydo | Y | Y | Y |
| Heat | N | Y | Y |

For our experiments, we utilize up to 16 processors of a 32 node Itanium-2 Cluster (IA64) and a 64 node AlphaServer ES40 Cluster. The Itanium-2 cluster consists of two processors per node running at 1.3GHz each with a 256K L1 cache, 3MB L2 cache, and 2GB main memory. The AlphaServer cluster consists of four processors per node running at 833MHz each with an 8MB L2 cache and 2GB main memory. The nodes in both clusters are interconnected using the Quadrics QSnet-I high speed network with Elan3 switching technology.

The MPI uni-directional bandwidth and latency characteristics measured from a micro-benchmark for inter-node communications are listed in Table 2. The performance is considered as a set of tuples in which the latency characterizes the message start-up component and the time per byte characterizes the bandwidth component for given message sizes. It should be noted that even though the same network is used in both clusters, the communication performance can vary due to differences in the node design. Bi-directional MPI characteristics as well as NIC contention are actually used in the model of SAGE [6].

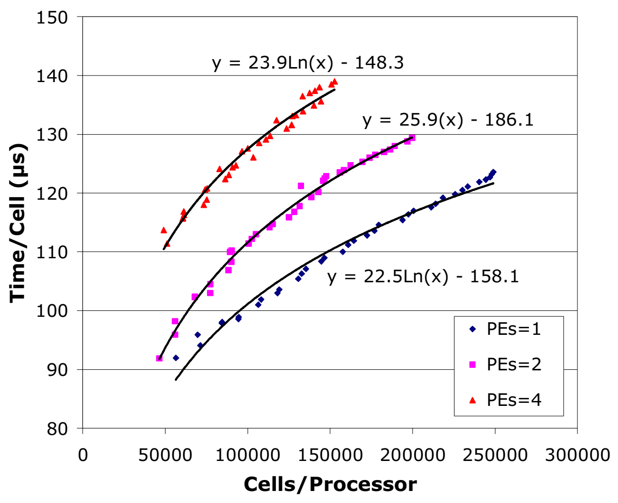
The measured performance as well as a piece-wise performance model for single-processor and single-node performance for one of the input-decks (timing_a) is shown in Figure 1(a) for an AlphaServer ES40 node, and in Figure 1(b) for an Itanium-2 node.

It is worth noting that we obtained single-node models for several cases on each system. This is done in order to capture memory contention effects. The maximum memory contention will be seen when all processors within a node are used. Since, it is possible to use 1-, or 2-processors-per-node (and 4 on the AlphaServer), we provide models for each case. However, we use all processors within a node when the processor-count exceeds the node-size.

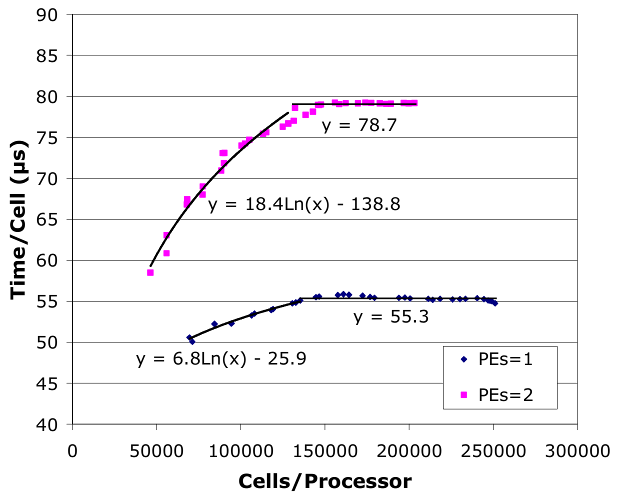
Measured and predicted times versus cycle number (up to 200 cycles) are shown in Figure 2 for the AlphaServer cluster and in Figure 3 for the Itanium Cluster. Note that only 1 and 16 processor runs are shown. On each graph the three curves depict: measured time, online predicted time (using internal application timers for the single-processor time), and offline predicted time (using the piece-wise logarithmic models as shown in Figure 1).

**Table 2. Uni-directional MPI communication characteristics (QSnet-1)**

| | Message Size (B) | Latency (μs) | Time per byte (ns) |
|---|---|---|---|
| AlphaServer ES40 Cluster | ≤ 32 | 5.6 | 0.0 |
| | >32 & <512 | 5.9 | 18.7 |
| | ≥ 512 | 8.1 | 5.0 |
| Itanium-2 Cluster | ≤ 32 | 5.0 | 0.0 |
| | >32 & <512 | 7.6 | 8.9 |
| | ≥ 512 | 11.0 | 4.6 |



(a) AlphaServer ES40 node.



(b) Itanium-2 node.

**Figure 1. Time per cell vs. cell count (timing_a)**

The average errors across all cases when using the online and offline model are summarized in Table 2 for the AlpaServer cluster and in Table 3 for the Itanium-2 cluster. More configurations than shown in Figures 2-3 are considered in this summary. In general, the predicted time tracks the measured time very well. The average error varies from 3.4% to 11.0%. It is also worth noting that the predictions change in discrete steps, although this distinction diminishes as the processor-count increases. This is due to the fact that the predicted time changes when adaption takes place. In fact, it can be inferred from the model predictions precisely when adaption occurred.

## 4. Model Run-time Use

Once we are confident in the capability of the model, it can be used to explore diverse performance scenarios. For the following experiments the first input-deck to SAGE

**Table 3. Average prediction error, AlphaServer**

| | timing_a (%) | | timing_b (%) | | timing_c (%) | |
|---|---|---|---|---|---|---|
| PEs | online | offline | online | offline | online | offline |
| 1 | 5.6 | 5.0 | 3.4 | 4.2 | 3.6 | 4.8 |
| 2 | 5.5 | 3.4 | 4.9 | 3.0 | 4.3 | 4.6 |
| 4 | 5.4 | 3.4 | 4.6 | 3.7 | 6.4 | 5.1 |
| 8 | 4.7 | 4.7 | 3.9 | 5.5 | 5.65 | 6.6 |
| 16 | 4.5 | 9.1 | 3.9 | 7.7 | 5.4 | 10.6 |

**Table 4. Average prediction error, Itanium-2**

| | timing_a (%) | | timing_b (%) | | timing_c (%) | |
|---|---|---|---|---|---|---|
| PEs | online | offline | online | offline | online | offline |
| 1 | 5.9 | 11.0 | 3.8 | 5.9 | 4.9 | 5.0 |
| 2 | 8.0 | 9.6 | 6.7 | 7.2 | 5.7 | 5.9 |
| 4 | 8.3 | 5.3 | 6.8 | 3.5 | 8.3 | 6.2 |
| 8 | 8.1 | 4.7 | 6.0 | 3.3 | 7.45 | 7.6 |
| 16 | 8.4 | 8.6 | 5.8 | 6.0 | 10.2 | 8.4 |

(timing_a) is used on 16 Itanium-2 processors (8 nodes), although the results are applicable to other configurations.

The model is first used to determine the optimal checkpoint interval using Young's equation [19].

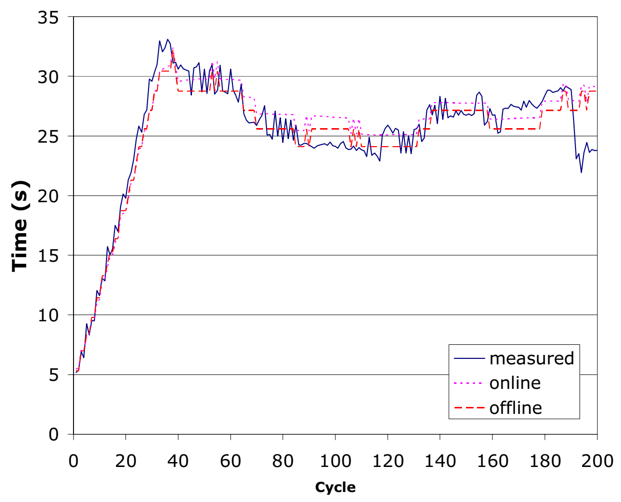
$$\tau = \sqrt{2\delta M} \qquad (1)$$

where δ is the time required to perform a checkpoint and *M* is the mean time between failures.

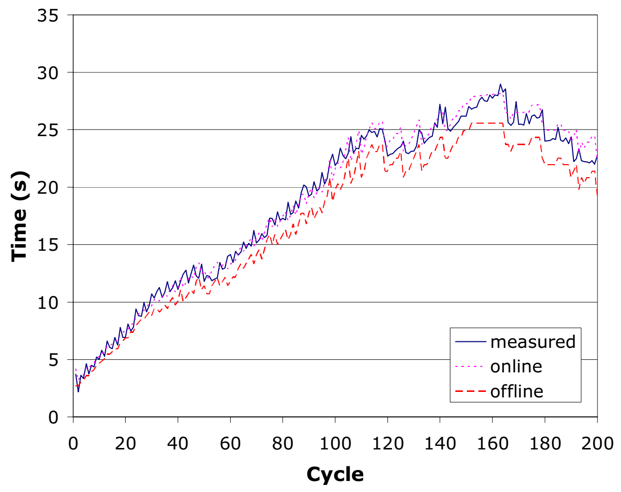
In our case, we want the checkpoint interval τ to be an integral number of cycles. That is, at each cycle, we check to see if the predicted time for the next cycle will exceed τ. The optimal checkpoint intervals calculated using this approach are shown against the accumulated time in Figure 4. For demonstration purposes we have assumed *M*=10 min. Note that this is a much lower value (i.e., higher failure rate) than we can reasonably be expected for most systems, where *M* is more likely expressed in terms of days. For this analysis we have also chosen δ = 10 sec, largely based on the analysis in [15].

It is not surprising that the number of cycles between suggested checkpoints decreases as the cycle number increases. That is, the first checkpoint occurs after cycle 27, but the next one follows cycle 43. This is because the time per cycle steadily increases, as can be seen in Figure 3(b) (which is the same run). For this example, we have assumed a constant value for δ. However, as with the single-processor time, δ will also increase as the number of cells-per-processor increases [15].

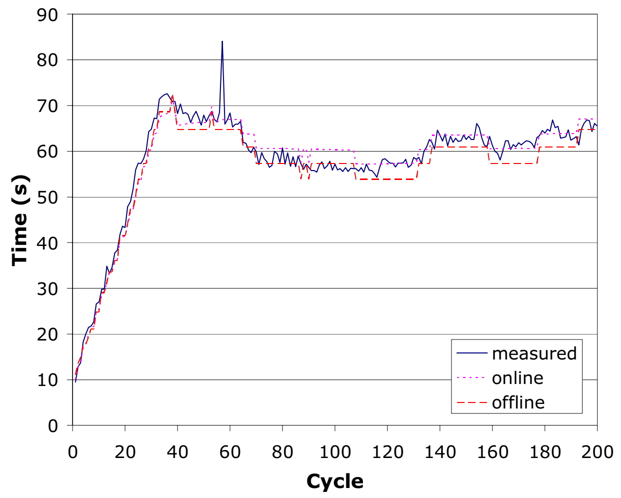
The adaptability of the model is exercised through imposing intentional interference. Interference in this case is an external activity on the system that impacts on the application performance. We accomplish this by bypassing the job control system and logging in directly to one of the compute nodes in the cluster. A memory intensive kernel is then executed on one processor in the compute node to perturb, or slow-down, the application.



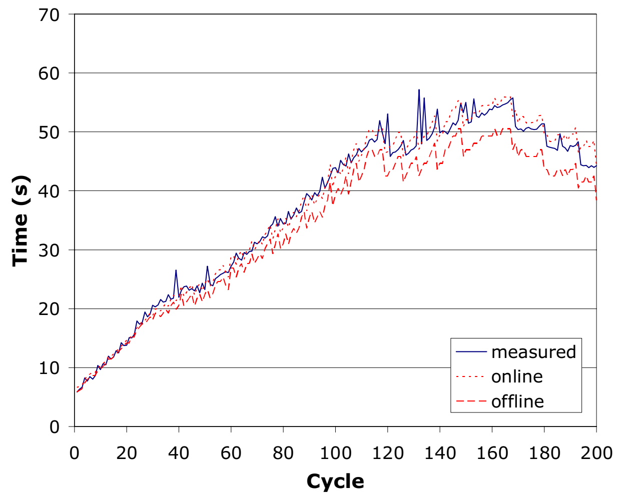
(a) timing_a (1 PE)



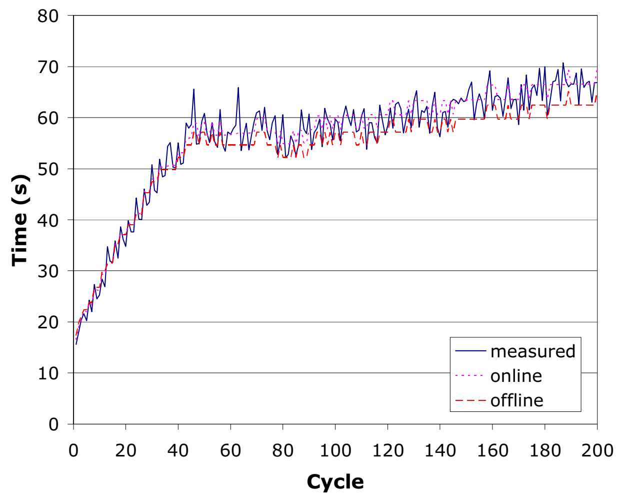
(b) timing_a (16 PEs)



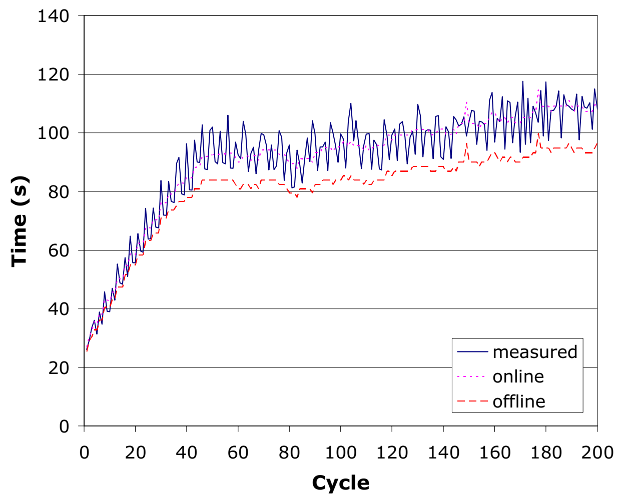
(c) timing_b (1 PE)



(d) timing_b (16 PEs)

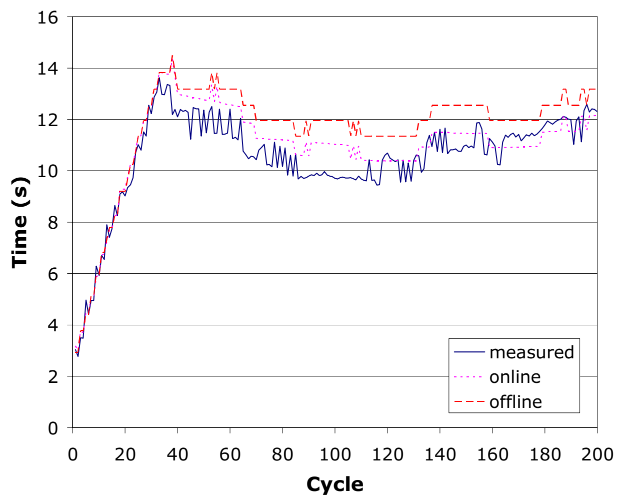


(e) timing_c (1 PE)

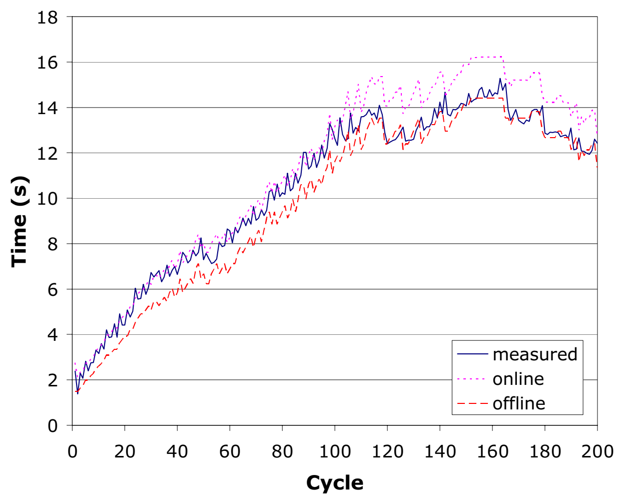


(f) timing_c (16 PEs)

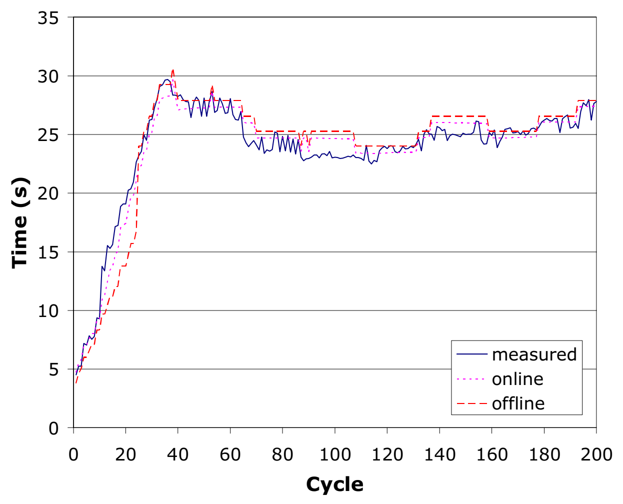
**Figure 2. Measured and predicted performance of SAGE on the AlphaServer ES40 cluster on 1 and 16 processors.**



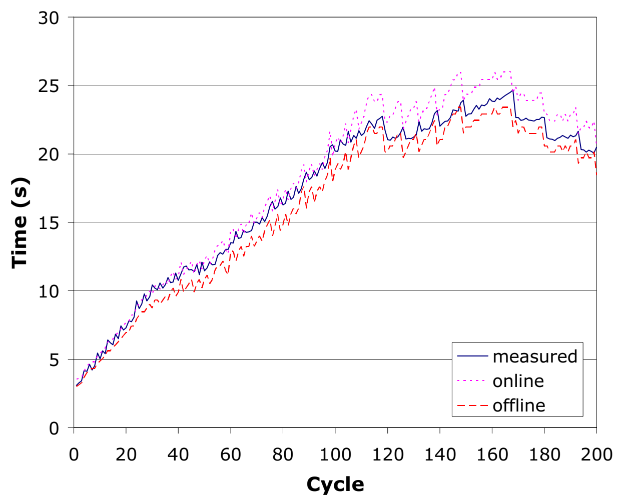
(a) timing_a (1 PE)



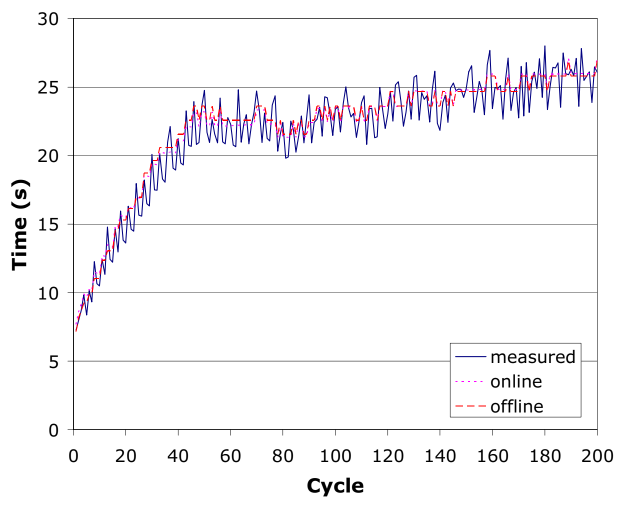
(b) timing_a (16 PEs)



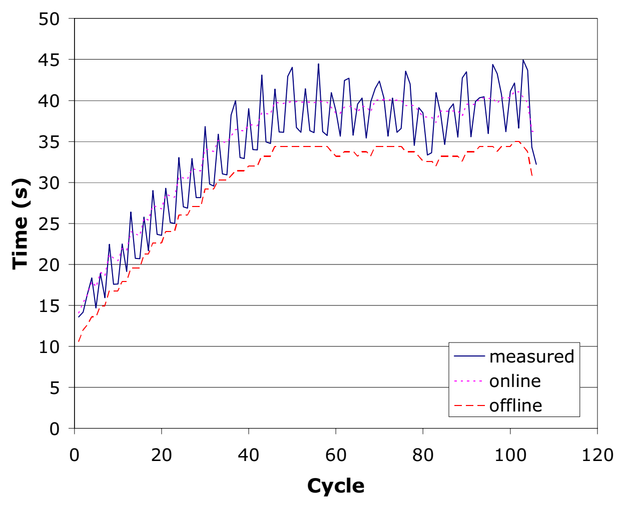
(c) timing_b (1 PE)



(d) timing_b (16 PEs)



(a) timing_c (1 PE)



(f) timing_c (16 PEs)

**Figure 3. Measured and predicted performance of SAGE on the Itanium-2 Cluster on 1 and 16 processors.**

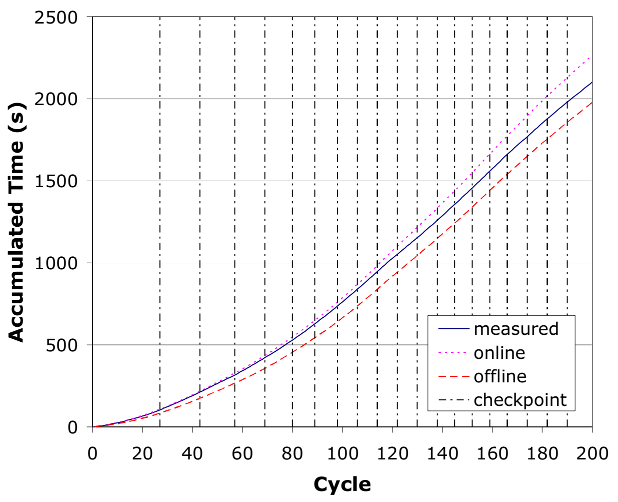
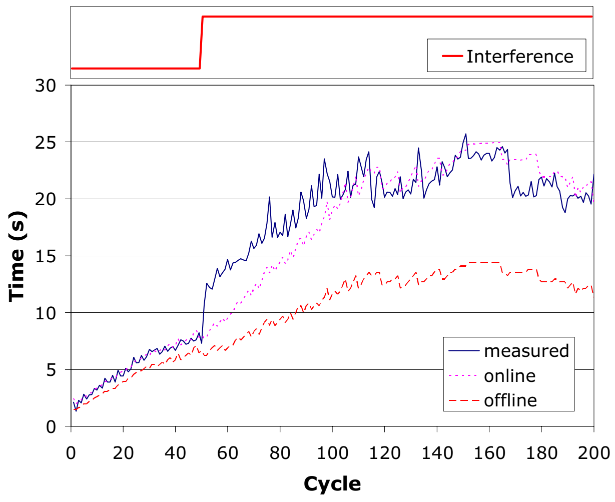


**Figure 4. Accumulated run-time and optimal checkpoint intervals.**

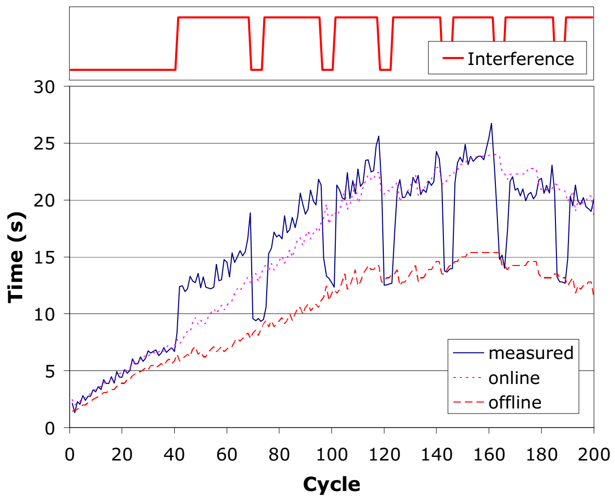
As we see in Figure 5(a) the measured time approximately doubles after the start of the interference from cycle 50 – this is what would be expected since the system is now effectively running two jobs rather than just one. The interference is shown by the two-value curve in the upper part of the graph. As expected the offline model continues to predict the non-perturbed execution time (using pre-measured timing information). The online model, on the other hand, gradually catches up with the measured time. The reason that the online model does not immediately change is that a large window size is used to estimate the time-per-cell. In this case a window that extends back to the first cycle is used.

In Figure 5(b) a similar experiment is undertaken, with interference that consists of multiple phases with a delay between the phases in which the interference is switched on and off. In this case, the online model again gradually catches up to the measured time and is not significantly affected by the periodic return to the non-perturbed state. The measured time oscillates between the online (with interference) and offline (without interference) models. Finally, we illustrate the effect of the window size on the online model in Figure 5(c). Here we have reduced the window size such that only the previous cycle is used to estimate the current time-per-cell. In this case the online model tracks the measured time closely and returns to the non-perturbed execution time (and the offline prediction) once the interference is terminated.

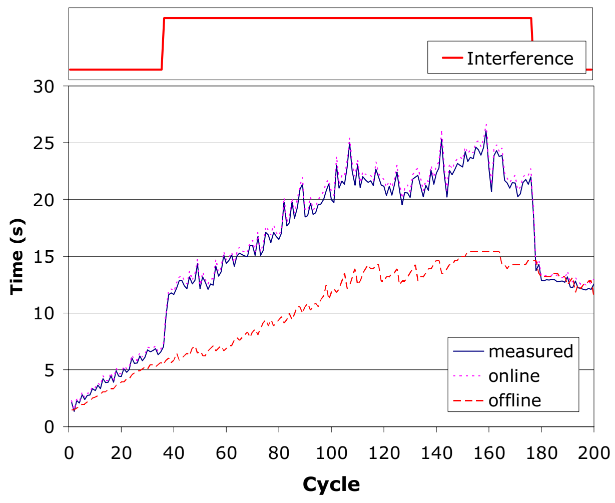
These experiments illustrate that an online model will adapt to changes in the system, and indeed with a small window, will not recognize when the system is not achieving a good performance level. The offline model conversely can be used to recognize when the system is not achieving its optimal performance level and hence be used as part of a diagnostic process in order to determine what part of the system is not performing as expected.



**(a) interference (large window size)**



**(b) intermittent interference (large window size)**



**(c) interference (small window size)**

**Figure 5. Measured and predicted times for a system with interference.**

## 5. Summary

In this work we have presented a method by which a performance model can be used to dynamically predict the individual cycles of an adaptive mesh refinement hydrodynamics code. The dynamic model is shown to perform with high accuracy and has low overhead. We have shown how the dynamic model can be used to determine optimal checkpoint intervals and further demonstrate the adaptability of the model by introducing interference during the application run.

We believe performance modeling is key to building performance engineered applications and architectures. The techniques presented here are general and can be easily retargeted to use other performance models such as our work on structured grid particle transport modeling [5], unstructured mesh particle transport [11], and Monte-Carlo simulation [12]. In particular, an executable model could be used at the beginning of these applications to determine optimal or near optimal partitioning strategies or to configure input parameters. For example, given a set number of processors the Monte-Carlo model could be used to determine the number of particles to simulate.

## Acknowledgements

This work was funded in part by the WSR component of the Accelerated Strategic Computing (ASC) program of the Department of Energy, and by the DARPA High Productivity Computing Systems program in. Los Alamos National Laboratory is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36.

## References

[1] A. M. Alkindi, D. J. Kerbyson, E. Papaefstathiou and G. R. Nudd. Dynamic Optimisation of Application Execution on Distributed Systems. *Future Generation Computing Systems*, 17(8):941–949, June 2001.

[2] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, O. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su and Z. Zagorodnov. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, April 2003.

[3] J. Daly. A strategy for running large scale applications based on a model that optimizes the checkpoint interval for restart dumps. In *ICSE Software Engineering for High Performance Computing System Applications Workshop*, pages 70–74, May 2004.

[4] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Stefan and D. Ranta. The RAGE Radiation-hydrodynamic code. To appear in *Journal of Computational Physics*, 2006.

[5] A. Hoisie, O. Lubeck and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, Winter 2000.

[6] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H. Wasserman and M.L. Gittings. Predictive Performance and Scalability Modeling of a Large-scale Application. In *IEEE/ACM Supercomputing (SC'01)*, Nov. 2001.

[7] D.J. Kerbyson, A. Hoisie and H.J. Wasserman. Modeling the Performance of Large-Scale Systems. *IEE Proceedings (Software)*, 150(4):214–221, Aug. 2003.

[8] D.J. Kerbyson, A. Hoisie and H.J. Wasserman. A Performance Comparison between the Earth Simulator and other Terascale Systems on a Characteristic ASCI Workload. *Concurrency and Computation, Practice and Experience*, 17(10):1219-1238, Aug. 2005.

[9] D.J. Kerbyson, A. Hoisie and H.J. Wasserman. Use of Predictive Performance Modeling During Large-Scale System Installation. *Parallel Processing Letters*, 15(4): 387-395, Dec. 2005.

[10] D.J. Kerbyson, E. Papaefstathiou and G.R. Nudd. Application Execution Steering using On-the-Fly Performance Prediction. High Performance Computing and Networking. In *High-Performance Computing and Networking, Lecture Notes in Computer Science*, 1401:718–727, Apr. 1998.

[11] M.M. Mathis and D.J. Kerbyson. A General Performance Modeling of Structured and Unstructured Mesh Particle Transport Computations. *Journal of Supercomputing*, 34:181-199, 2005.

[12] M.M. Mathis, D.J. Kerbyson and A. Hoisie. A Performance Model of non-Deterministic Particle Transport on Large-Scale Systems. In *Computational Science (ICCS), Lecture Notes in Computer Science* 2659:905-915, June 2003.

[13] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper and D. V. Wilcox. PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, Fall 2000.

[14] F. Petrini, D.J. Kerbyson and S. Pakin. The case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *IEEE/ACM Supercomputing (SC'03)*, Nov. 2003.

[15] J.C. Sancho, F. Petrini, G. Johnson, J. Fernandez and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In *IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2004.

[16] J. M. Schopf and F. Berman. Performance prediction in production environments. In *Merged 12$^{th}$ International Parallel Processing Symposium & 9$^{th}$ Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*, pages 647-653, Apr. 1998.

[17] J.M. Schopf and F. Berman. Performance prediction using intervals. *Technical report CS97-541*, UCSD, 1997.

[18] R. Wolski, N.T. Spring and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757– 768, 1999.

[19] J.W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, Sep. 1974.