Dynamic Performance Prediction of an Adaptive Mesh Application

Mark M. Mathis and Darren J. Kerbyson

July 11, 2005

Abstract

While it is possible to accurately predict the execution time of a given cycle of an adaptive application, it is not generally possible to predict the adaptive steps the application will take and therefore to predict the total execution time for a given run of the application. To remedy this situation we have developed an executable model that can be accessed dynamically at runtime directly from the application of interest. In this manner, the application itself can predict the execution time for its next cycle, after the adaption step has been completed. This allows the application to decide if it is running correctly (i.e., by matching up to predicted times), when to perform checkpoint operations (if the next cycle will exceed a predefined time limit between checkpoints), or when to terminate the application (if the next cycle will exceed the application's time allocation). In this paper we introduce a framework for specifying such an executable model and demonstrate its use in a variety of performance scenarios. The dynamic model is shown to have high accuracy over a number of test cases (even in the presence of interference). We further demonstrate the use of the model to determine optimal checkpoint intervals.

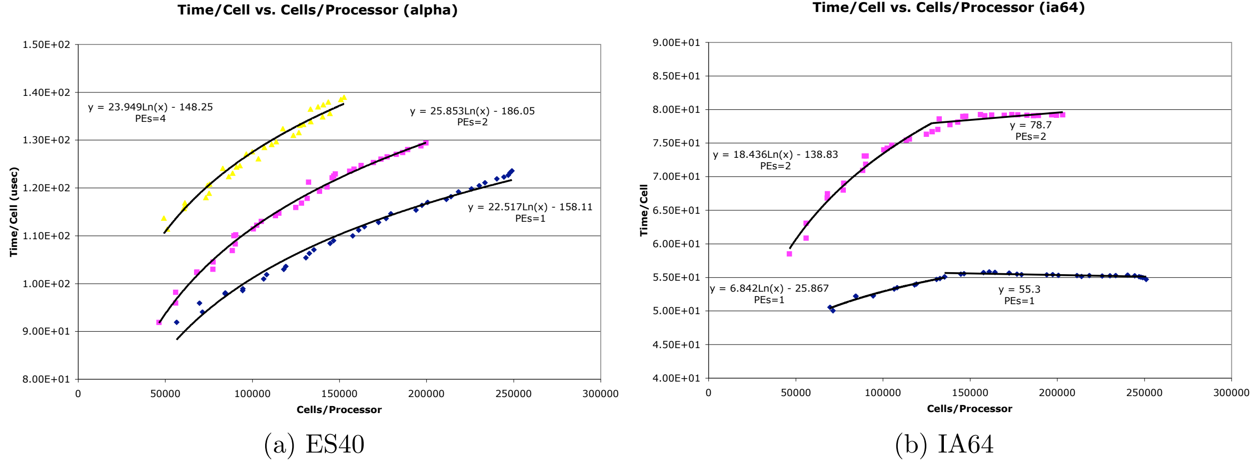
1 Introduction

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 the system is to be used for 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 [6] 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 [5]; and in verifying the performance of the 20-Tflop ASCI Q system during its installation [8]. The ultimate of which lead to system optimizations resulting in a factor of 2 performance improvement [14]. Models have also been used to compare large-scale system performance including a comparison of several of the highest peak-rated terascale systems such as the Earth Simulator and ASCI Q [7].

One of the key applications used in these studies is SAGE (SAIC's Adaptive Grid Eulerian hydrocode). It is a multidimensional (1D, 2D, and 3D), multimaterial, Eulerian hydrodynamics code with adaptive mesh refinement (AMR). SAGE comes from the Los Alamos National Laboratory Crestone project, whose goal is the investigation of continuous adaptive Eulerian techniques to stockpile stewardship problems. SAGE has also 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, early time front end design, explosively generated air blast, and hydrodynamic instability problems. SAGE represents a large class of production ASCI applications at Los Alamos that routinely run on 2000-4000 processors for months at a time.

SAGE has been previously modeled and the model validated on a number of systems [5]. What makes SAGE interesting for our current study is its adaptivity. That is, as the application progresses through its cycles, it can divide and combine cells to focus the computation on areas of interest. Unfortunately, the adaption taken over the course of an execution cannot be predicted from initial conditions. This means



(a) ES40 (b) IA64

Figure 1: Time per cell vs. the number of cells for (a) AlphaServer ES40 and (b) Itanium-2 Cluster.

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 we dynamically predict the time for the next cycle during the execution of SAGE?”.

There are a number of uses for this type of dynamic modeling (e.g., [1], [9]). 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, 20]). 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 starting 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.

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. [17, 16]). Our approach is to apply the detailed analytical performance modeling approach developed at LANL to these interesting applications of performance modeling.

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 modeling specification language called CHIP^3S [13]. The modeling framework provided by CHIP^3S is intended to provide predictions through simulation of the code. In this case, the CHIP^3S language specification is a direct mapping from the source code to the performance domain. Evaluation is then accomplished using discrete event simulation. 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. A complete treatment of the model and the CHIP^3S language implementation are included in Appendices A and B, respectively. 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.

	Itanium-2 1GHz	AlphaServer ES40 833MHz
$L_c(S)(\mu s)$	$\begin{cases} 9.8605 & S \leq 512 bytes \\ 24.076 & S > 512 \end{cases}$	$\begin{cases} 14.838 & S \leq 512 bytes \\ 29.474 & S > 512 \end{cases}$
$1/B_c(S, D)(ns)$	$\begin{cases} 49.5 & S \leq 512 bytes \\ 2.3 & S > 512 \end{cases}$	$\begin{cases} 40.1 & S \leq 512 bytes \\ 3.3 & S > 512 \end{cases}$

Table 1: Hardware parameters for both validation systems.

PEs	timing_a		timing_b		timing_c	
	online	offline	online	offline	online	offline
1	5.56%	5.00%	3.40%	4.19%	3.59%	4.84%
2	5.53%	3.38%	4.88%	2.99%	4.34%	4.55%
4	5.44%	3.37%	4.56%	3.74%	6.38%	5.08%
8	4.69%	4.67%	3.92%	5.53%	5.55%	6.59%
16	4.54%	9.14%	3.93%	7.69%	5.38%	10.63%

Table 2: Average errors (ES40).

2 Dynamic Model Implementation

The complete SAGE model is detailed in Appendix A and the CHIP^{3}S implementation of the model is shown in Appendix B. A CHIP^{3}S model consists of a *hardware specification* layer (i.e., a system model) and a *parallel template* layer which implements a *task graph* representation of the application. The nodes of the task graph are then specified by an *application layer* model. This results in a number of input scripts that together form a program that can then be compiled and executed to produce predictions. Furthermore, the executable model can then be accessed from another application using a runtime evaluation interface.

By default a CHIP^{3}S application layer model is a direct mapping from the source code to a performance domain. The evaluation system can then take this model specification and predict the single processor 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 seldom addresses 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 (e.g. [10]) (largely due to cache effects). That is, the greatest performance will be seen when each processor's partition 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.

SAGE, however, nominally operates in a weak scaling mode. That is, the user specifies a number of cells per processor in the input file. 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 per processor changes throughout the execution of the application. Although the user specifies a number of cells per processor at the beginning of the code, the number of cells will actually (typically) increase as the mesh is adapted.

To account for this fact, we provide two different modes of operation for the dynamic model. First, we use internal timers to extract a time per element for a window of previous cycles and then use that time to predict the total time for the next cycle. This works very well when the goal is to make dynamic decisions at runtime based on model predictions. However, this is not a good approach to take in order to verify the health of the system (i.e., "Are we getting the expected performance?").

PEs	timing_a		timing_b		timing_c	
	online	offline	online	offline	online	offline
1	5.91%	10.98%	3.79%	5.94%	4.89%	5.02%
2	7.99%	9.58%	6.70%	7.20%	5.66%	5.85%
4	8.27%	5.34%	6.82%	3.48%	8.32%	6.20%
8	8.13%	4.71%	6.03%	3.31%	7.45%	7.64%
16	8.41%	8.55%	5.75%	5.96%	10.15%	8.42%

Table 3: Average errors (IA64).

The second mode of operation requires a *model* of the time per element, similar to the strong scaling case. This can be obtained by varying the number of cells on a single processor. One way to accomplish this is to run a number of cycles on a single processor with adaption turned on. This will give several (or many depending on the number of cycles) values which can then be fit to a piecewise linear (or logarithmic) curve (e.g., Figure 1). In this manner, one can obtain a baseline single processor performance that (with high confidence) represents the best achievable performance.

3 Model Validation

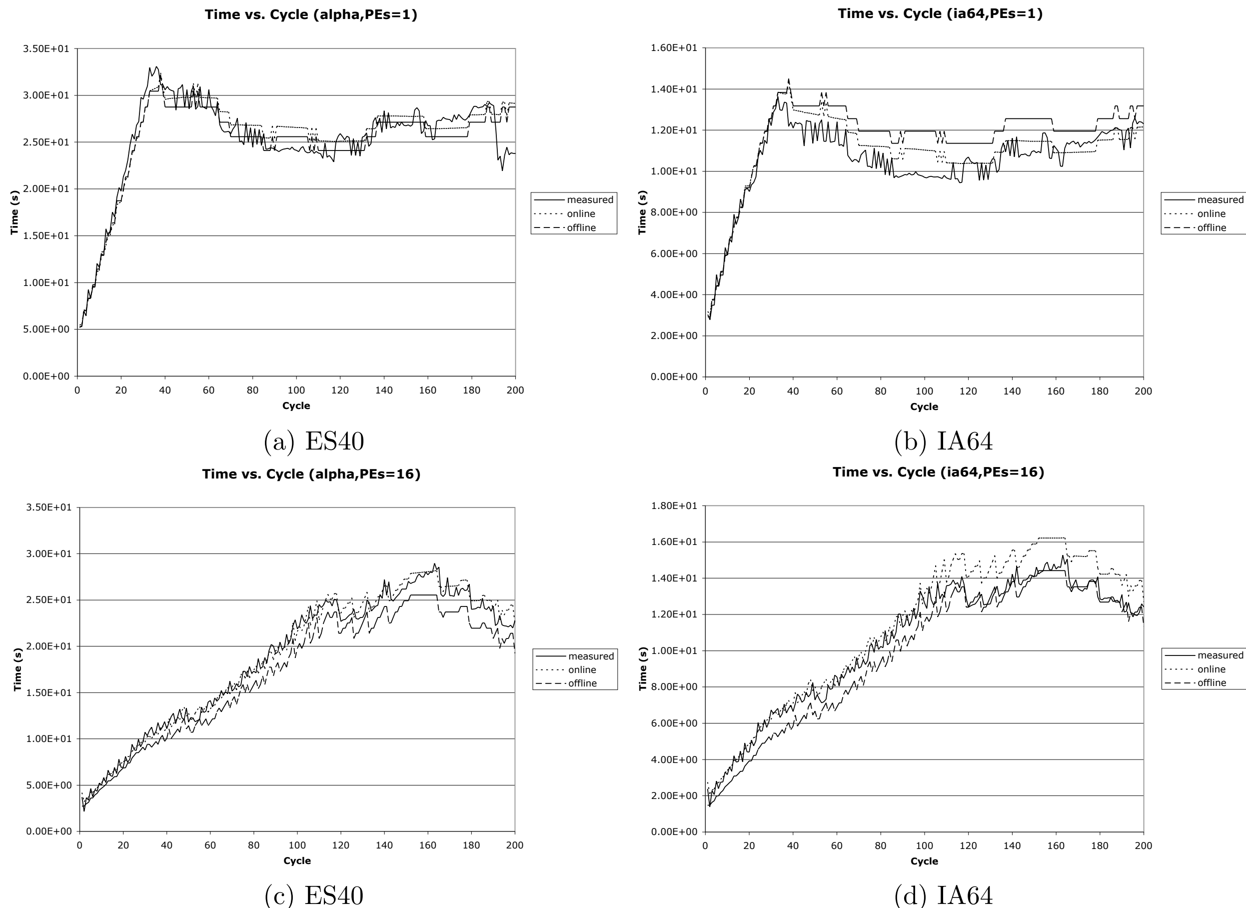
The SAGE model has been previously validated on a number of systems [5]. So, it is not our goal here to validate the model itself. Rather, our overall goal is to demonstrate how a performance model can be used dynamically at runtime. To that end, we do need to validate the executable version of the model and show that it can be used with minimum perturbation to the existing code. We do this for several inputs (timing_a, timing_b, and timing_c) for a large number of cycles on several different system configurations.

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 2 processors per node running at 1.3GHz each with a 256K L1 cache, a 3MB L2 cache, and 2GB main memory. The AlphaServer cluster consists of 4 processors per node running at 833MHz each with an 8MB L2 cache and 2GB main memory. The nodes in both the clusters are interconnected using the Quadrics QSnet-I high speed network with Elan3 switching technology. The performance characteristics of these systems are listed in Table 1. The models for single processor performance for one of the inputs (timing_a) are displayed on their respective graphs in Figure 1.

It is worth noting that we have obtained single processor models for several different cases on each system. This is done in order to capture memory contention effects. For the HP AlphaServer, each node has four processors. Therefore, the maximum memory contention will be seen when all four processors on a node are used. Likewise, the Itanium-2 cluster has two processors per node leading to maximum contention when two processors per node are used. Since, it is possible to use 1, 2, or even 4 processors/node, we provide models for each case. However, for our experiments we use all processors on a node when the total number of processors exceeds the number of processors per node.

Measured and predicted times versus cycle number (up to 200 cycles) are shown in Figures 2- 4 (only 1 and 16 processor runs shown). On each graph we have three lines for each of our test systems: measured time, online predicted time (using internal timers for the single processor time), and offline predicted time (using the logarithmic models). For clarity we have removed individual point markers and focus instead on the general trends.

The average errors for all of the cases are summarized in Tables 2- 3. In general, the predicted time tracks the measured time very well. The average error varies from 3.37% to 10.98%. It is worth noting that the predictions change in discrete steps (although this distinction diminishes as the number of processors increases). This is due to the fact that the predicted time changes when adaption takes place. In fact, it can be inferred from this precisely when adaption occurred.



(a) ES40 (b) IA64

(c) ES40 (d) IA64

Figure 2: Measured and predicted times for 1 PE (a-b) and 16 PEs (c-d) on the AlphaServer ES40 (a,c) and the Itanium-2 cluster (b,d), respectively (timing-a).

4 Model Exploration

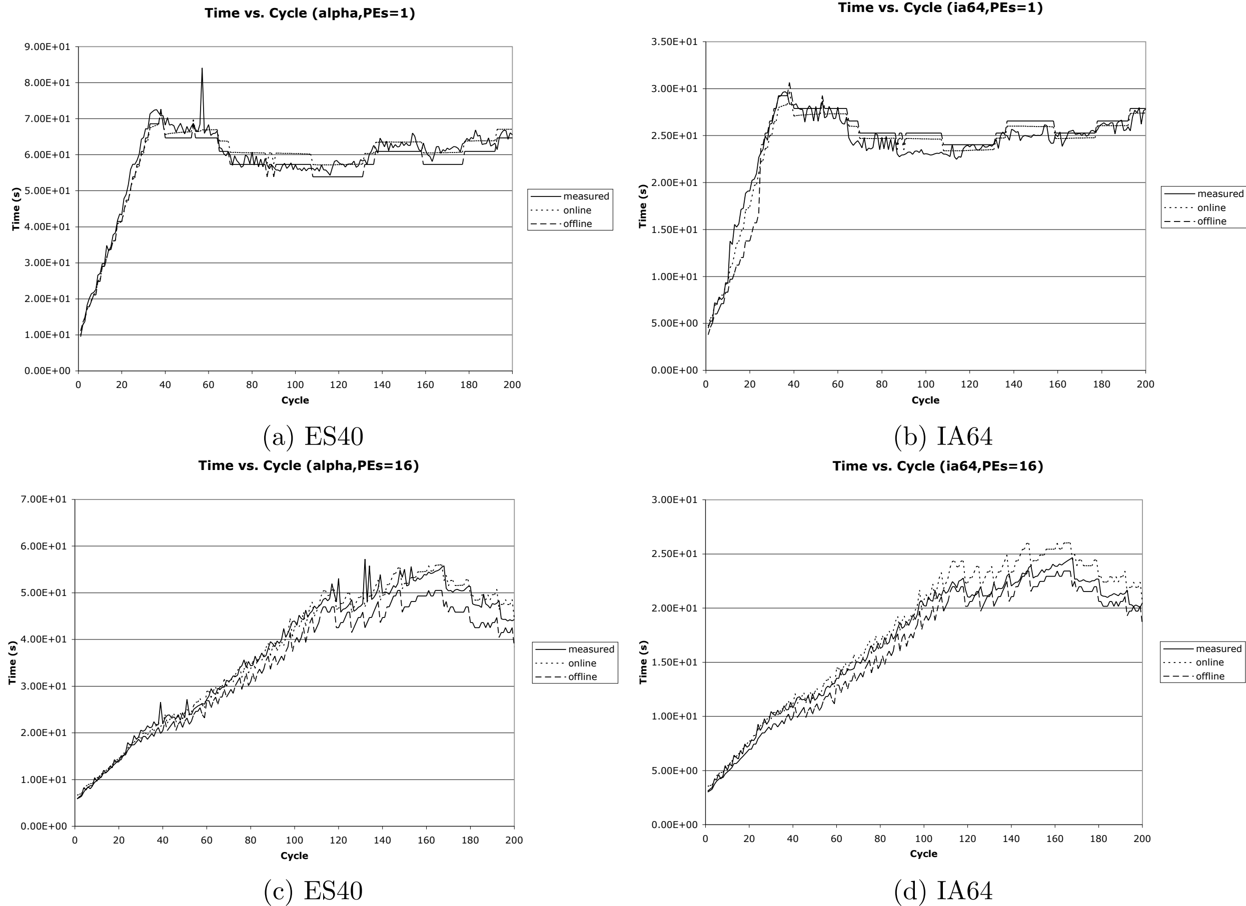
Once we are confident in the capability of the dynamic model, we can then use the model to explore diverse performance scenarios. For the following experiments we are using the first input (timing_a) with 16 of the Itanium-2 processors (8 nodes), although the results should generalize to other inputs/configurations. We first use the model to determine the optimal checkpoint interval using Young's equation [20].

$$\tau = \sqrt{2\delta M} \tag{1}$$

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 5(a). For demonstration purposes we have assumed $M = 10min$. Note that this is a much lower value (i.e., higher failure rate) than we can reasonably expect for most modern systems, where M is more likely expressed in terms of hours. For this analysis we have also chosen $\delta = 10sec$, 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 has been steadily increasing, as can be seen in Figure 2(d) (which is the same run). For



(a) ES40 (b) IA64

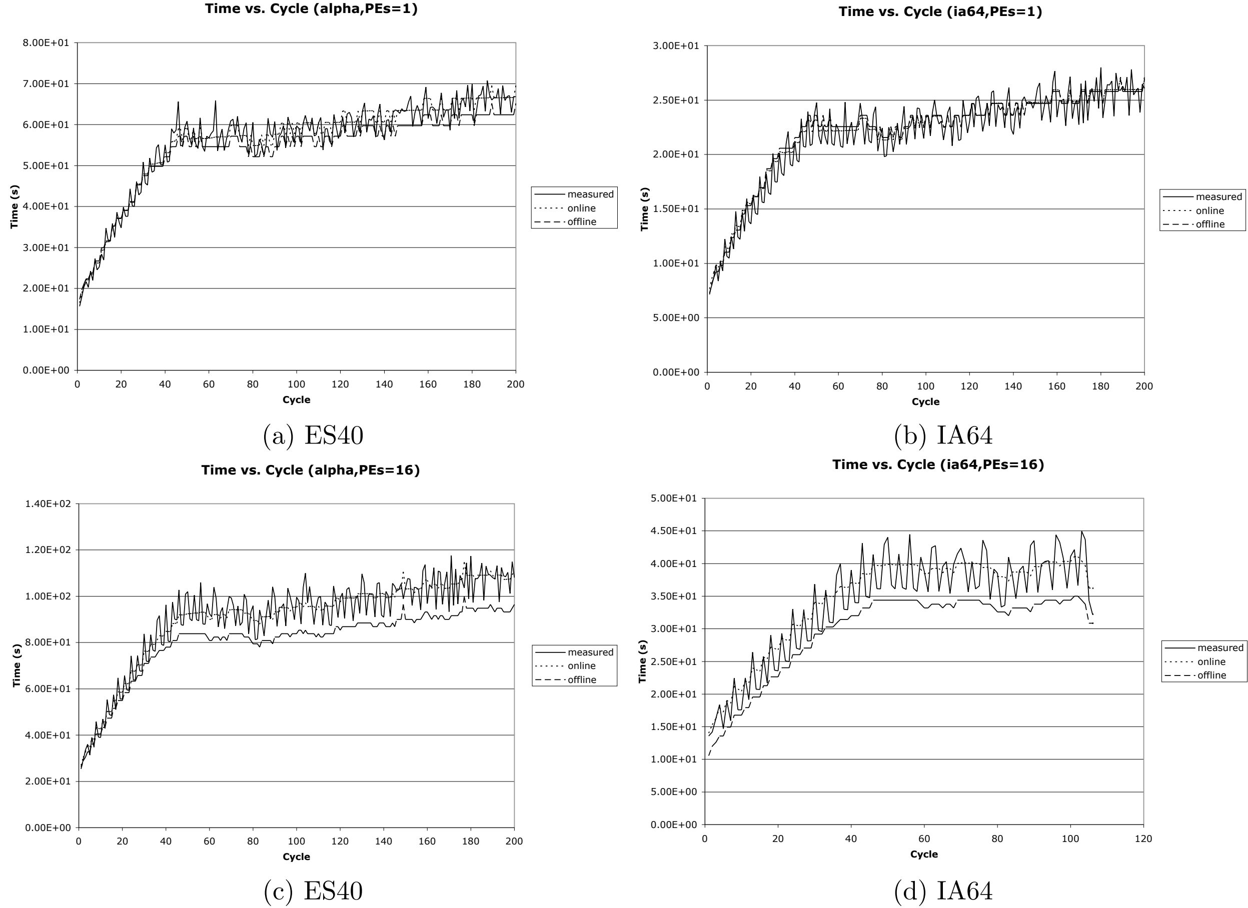
(c) ES40 (d) IA64

Figure 3: Measured and predicted times for 1 PE (a-b) and 16 PEs (c-d) on the AlphaServer ES40 (a,c) and the Itanium-2 cluster (b,d), respectively (timing-b).

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].

Next we exercise the adaptability of the model through imposing intentional interference. We accomplish this by bypassing the job control system and logging in directly to one of the compute nodes. We then run the cachebench [12] benchmark on the local compute node in order to perturb the overall execution time. As we see in Figure 5(b) the measured time approximately doubles after the start of the interference (around cycle 50). As expected the offline model continues to predict the non-perturbed execution time. 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 it is using a large window size to estimate the time per cell. In fact, it is using a window that extends back to the first cycle of the run.

In Figure 5(c) we perform the same sort of experiment, except with a delay between the different phases of the cachebench application. In this case, the online model again gradually catches up to the measured time and is not affected by the periodic return to the non-perturbed execution time. 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(d). Here we have reduced the window size such that only the previous cycle is used to estimate the current time per cell. Now the online model tracks the measured time quite closely and returns to the non-perturbed execution time (and the offline prediction) once the interference is terminated.



(a) ES40 (b) IA64

(c) ES40 (d) IA64

Figure 4: Measured and predicted times for 1 PE (a-b) and 16 PEs (c-d) on the AlphaServer ES40 (a,c) and the Itanium-2 cluster (b,d), respectively (timing-c).

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 low overhead. We have shown how the dynamic model can be used to determine optimal checkpoint intervals and further demonstrated 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 [4], unstructured mesh particle transport [10], and Monte-Carlo simulation [11]. In particular, an executable model could be used at the beginning of these applications to determine optimal or near optimal partitioning strategies or for configuration of 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.

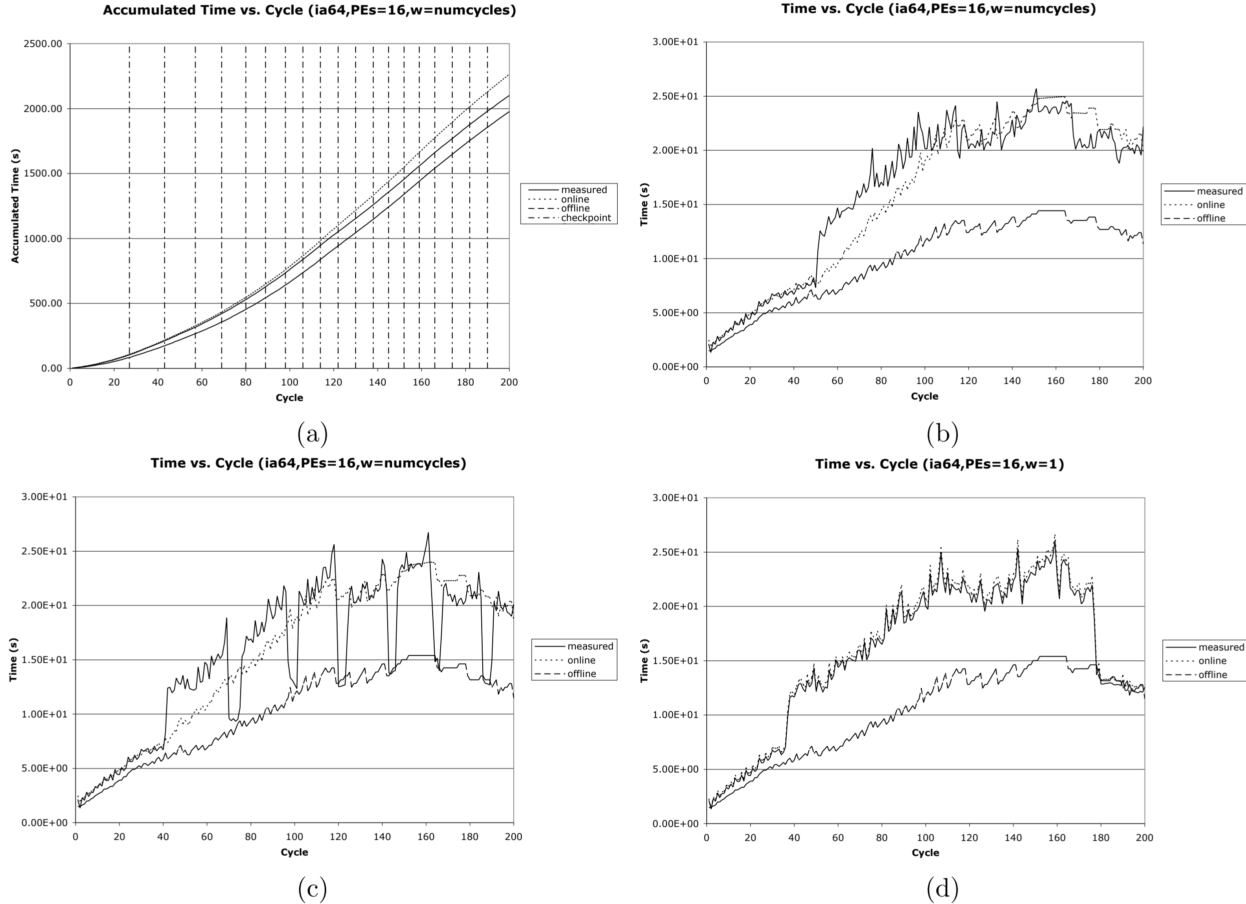


Figure 5: Measured and predicted times for various performance scenarios: (a) accumulated time and optimal checkpoint intervals, (b) perturbed with large window size, (c) intermittent perturbed with large window size, and (d) perturbed with small window size.

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] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Obertelli Obertelli, Jennifer Schopf, Gary Shao, Smallen Smallen, Neil Spring, Alan Su, and Zagorodnov 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 *Proceedings of the ICSE Software Engineering for High Performance Computing System Applications Workshop*, pages 70–74, Edinburgh, Scotland, May 2004.

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

[5] 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 *Supercomputing*, Denver, Nov. 2001.

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

[7] 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. *To appear in Concurrency and Computation, Practice and Experience*, 2004.

[8] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. Use of Predictive Performance Modeling During Large-Scale System Installation. *To appear in Parallel Processing Letters*, 2004.

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

[10] M. M. Mathis and D. J. Kerbyson. Performance Modeling of Unstructured Mesh Particle Transport Computations. In *IPDPS*, Santa Fe, NM, Apr 2004.

[11] M. M. Mathis, D. J. Kerbyson, and A. Hoisie. A Performance Model of non-Deterministic Particle Transport on Large-Scale Systems. In *Proc. Int. Conf. on Computational Science (ICCS)*, Melbourne, Australia, June 2003.

[12] P. J. Mucci and K. S. London. Low Level Architectural Characterization Benchmarks for Parallel Computers. Technical Report ut-cs-98-394, University of Tennessee Knoxville Department of Computer Science, 1998.

[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. *Journal of High Performance Applications*, 14(3):228–251, October 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 *SuperComputing*, Phoenix, 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 *IPDPS*, Santa Fe, NM, April 2004.

[16] J. M. Schopf and F. Berman. Performance prediction in production environments. In *Proc. IEEE Symp. Par. Dist. Proc. (SPDP)*, 1998.

[17] J.M. Schopf and F. Berman. Performance prediction using intervals. Technical report, University of California, 1997.

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

[19] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *Programming Language Design and Implementation*, 2003.

[20] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17:530–531, September 1974.