Identification of Performance Characteristics from Multi-view Trace Analysis

Daniel Spooner[1] and Darren Kerbyson[2]

[1] High Performance Systems Group, Dept. Of Computer Science,
University of Warwick, Coventry, UK
dps@dcs.warwick.ac.uk

[2] Performance and Architectures Laboratory (PAL), CCS-3
Los Alamos National Laboratory, Los Alamos, USA
djk@lanl.gov

Abstract. In this paper, we introduce an instrumentation and visualisation tool that can be used to assist in analytical performance model generation. It is intended to provide a means of focusing the interest of the performance specialist, rather than automating the entire formulation process. The key motivation for this work was that while analytical models provide a firm basis for conducting performance studies, they can be time-consuming to generate for large, complex applications. The tool described in this paper allows trace files from different runs of an application to be compared and contrasted in order to determine the relative performance characteristics for critical regions of code. It is envisaged that the tool will develop to identify and summarise specific performance issues such as communication strategies through the use of novel visualisation techniques.

1 Introduction

The design and implementation of high-performance systems is a highly complex problem requiring knowledge of many factors. The peak performance of a system is a result of the underlying hardware architecture including the processor design, memory hierarchy, inter-processor and communication system, and their interaction. Moreover, the achievable performance is dependent upon the workload applied to the system, and how this workload utilises the resources within the system.

Performance modelling is a key approach that can provide information on the expected performance of a workload, given a particular architectural configuration. It is useful throughout the entire system life-cycle: starting at the design stage where no system is available for measurement, through comparison of systems and procurement, to implementation, installation and verification, and finally to examine the effects of system updates over time. At each stage, a performance model can provide an expectation of the achievable performance of the workload with reasonable fidelity.

Performance models are widely used: from large-scale, tightly-coupled systems through to dynamic and distributed Grid based systems. For instance, performance modelling is being used to validate the performance during the installation of ASCI Q

at Los Alamos National Laboratory (LANL) [1], to compare the performance of large-scale systems such as the Earth Simulator [2], and has been used in the procurement of ASCI purple (expected to be a 100Tflop system). Performance models have also been applied in dynamic and distributed 'Grid type' environments to consider service-orientated metrics in the provision of resource management services [3] and in the mapping of business applications to resources [4].

The accuracy of a model, and hence, its effectiveness lie in its ability to capture an application's performance behaviour. It is considered advantageous to parameterise a model in terms of system configuration (e.g. for scalability analysis), and calculation behaviour (e.g. input data-set size). This allows for the exploration of the performance space without being specific to a particular 'performance point'.

It is, however, generally acknowledged that the formation of a performance model is a complex task. It may involve a thorough code analysis, inspection of important data structures and analysis of profile and trace data. It can therefore be time-consuming to generate a detailed model given the large size of many scientific applications and the relative complexities of advanced data structures and optimised communication strategies. Several semi-automated approaches have been proposed that aim to make the formation of a performance model a simpler task using 'black-box' techniques in which individual performance aspects are observed but not necessarily understood. Examples include modelling the scaling behaviour of basic-block performance [5] and modelling the memory behaviour of basic-blocks and extrapolating to other systems [6]. These approaches tend to be specific to a particular processor configuration and/or problem size.

In this work we consider an approach that aims to simplify the process of generating a performance model, but not to automate it entirely. The purpose is not to simplify the resultant performance model, nor detract from the skill-set required by the modeller, but rather remove unnecessary steps during formulation. While the answer to this question lies, in part, with the experience of the performance-modeller; we believe that tools can be developed (or adapted) to help locate and *focus* on the performance critical regions. Such regions are typically those whose execution behaviour changes when the system configuration or application input-data is varied.

While there are a number of post-analysis and diagnosis tools that can assist with identifying performance constraints such as bottlenecks [7] and communication patterns [8], many are aimed at resolving *problems* with the application rather than trying to *characterise* the application's behaviour. In this case, it is useful to identify the differences in particular idiosyncratic behaviours as well as to detect problems.

A tool is introduced here that uses a combination of static and dynamic call-graph analysis to attempt to identity regions of code that are sensitive to data-set and scalability variations in order to reduce the time-to-model. It provides a compact view of multiple executions of an application using colour cues to draw attention to areas of interest. Although the current implementation is a prototype; it is envisaged that other methods of visualisation could be employed to summarise large-scale performance characteristics "at a glance", such as those used already in code maintenance [9].

The paper is organised as follows: Section 2 describes the approach taken that can lead to a performance model and identifies areas where tools can assist. Section 3 introduces a tracing tool that can create call-graph trace files for post-analysis using

source-code instrumentation. In Section 4, we describe a further tool that uses the trace files to create multi-view visualisations. Conclusions and future work is discussed in Section 5.

2 Identification of Performance Characteristics from Multiple Executions.

The performance modelling work at Los Alamos National Laboratory to date has primarily focussed on applications representative of the ASCI (Accelerated Strategic Computing Initiative) workload where analytical techniques are employed to develop entire-application models for the large-scale ASCI computing resources. This differs from a number of other performance activities that tend to focus on smaller applications in distributed computing environments such as [10]. The Los Alamos models are used most prominently to explore the scaling behaviour of applications on existing and speculative future architectures.

The approach to developing a performance model is based upon a detailed understanding of the performance effects that occur when changes to the system and application configuration are applied. In the initial stages of formulation, the application is typically executed with fixed input data sizes and a varying number of processing elements (PE) to observe changes in the overall execution-time and resource use. This can reveal basic information, such as whether the program scales weakly or strongly. Likewise, observations can be taken by fixing the PEs and varying the data-set sizes.

Instrumentation and profiling are subsequently used to obtain an understanding of the code. Highlighting the changes in communication patterns between processors as the PE count changes can, for example, provide insight into the type and method of domain decomposition. Instrumentation of this type can reveal message size changes in SAGE [11], an adaptive mesh hydro code, due to its 1D decomposition, and the difference in neighbouring processors in Tycho [12], radiation transport code, due to its use of an unstructured grid.

The problem partitioning, and related messaging, revealed through instrumentation typically provides a strong indication of the arrangement of the application's data structures, which can be confirmed by thorough code analysis of the relevant regions. Generating static call-graphs can assist with identifying the functional dependencies of the code sections and dynamic call-graphs (through the use of traces collected at runtime) can illustrate the flow of execution. In addition, comparing the relative number of instructions issued for a given subroutine in different runs/iterations can highlight the impact of configuration change on calculation and computational areas.

The overall objective of these activities is to obtain a model based on timings of sequential elements parameterised by expressions that are subject to input parameters and differing levels of parallelism. By identifying messaging, data placement and computational sensitivity to configuration and data-set sizes, it is possible to locate regions of the code that can be described by a single timing (for a given architecture) or by an analytical expression that captures the computation/communication characteristics with respect to the input parameters. While the approach is based on understand-

ing the application's behaviour, it is apparent that much of the initial work is based upon observing the performance effects of input and configuration variation (the dotted region in Figure 1). It is when these effects are compared across configurations that areas of interest can be brought to the attention of the performance specialist.

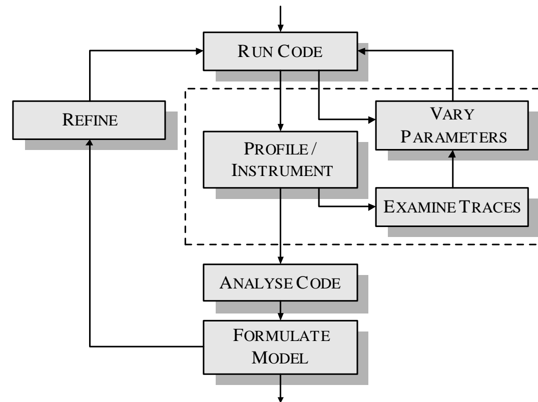

Figure 1. The key stages of model development. Initially the code is run with a set of different input parameters (e.g. number of assigned processors, data-set sizes) to get a view of its scaling behaviour. Typically instrumentation or profiling is used to produce trace files that identity performance critical regions in the code. This is followed by detailed code analysis to formulate an initial performance model. This can then be refined until a satisfactory level of fidelity (predicted vs. measured) is obtained

Without losing the depth of detail that the models provide, a suitable tool can assist with instrumentation and call-graph generation producing a visualisation of the key differences to indicate "areas of interest". This can limit the performance modelling process, allowing a performance specialist to focus on a subset of the application rather then the entire program.

3 Call-graph Collection

In order to assist the instrumentation, an automatic source-code level modification tool is used to indicate where and when a subroutine is entered and exited. The tool currently supports Fortran-77/90 and C. A lightweight profiling library is also used and linked with the application, which stores subroutine "events" into a page-based list whenever a subroutine is called. To minimise the overhead, each event records a limited amount of information including a source-file identifier, source-line number, and a field to denote that a 'context' has been entered into or exited from. As with nested subroutines, contexts are linked so that when subroutine *main* calls subroutine *init*, the instrumentation library is in the context of both *main* and *init* and any events that occur are associated with both of these contexts. This property is used to reduce the storage space of the event list as it is constructed dynamically in memory. The event-

list is written to file when the application exits, although it is possible to allow on-line paging. The trace-file can then be processed by a subsequent utility.

In a similar manner to Paradyn [13], the instrumentation library utilises the concept of inclusive and exclusive metrics, where inclusive sums a metric for a subroutine and all its children, while exclusive returns the metric for a subroutine without its children. Current recordable metrics include subroutine duration and the number of issued instructions (if PAPI [14] is available). The instruction count is not used to determine the amount of work, rather to relate code density between different processors in the same application, or differences between application runs for the same architecture. In the case of a particular application run, each subroutine will issue a given number of instructions; if the input data or configuration is varied it is possible that the instruction counts will change for a particular subroutine. It is then possible to examine the differences, or error, to ascertain how the application was effected by the change.

When the application is started, the instrumentation is initially disabled which results in virtually no overhead. An explicit "PROFILE_START" call is required to enable the library and store the context changes as necessary. In addition to context changes, MPI calls are logged as events through the profiling MPI (PMPI) interface which is connected to the capture library and can assist with identifying the communication patterns that occur. The relevant parameters (source, destination, collective, size, type) are stored as part of an MPI event. Currently, only a limited number of MPI calls are wrapped which covered our test cases; it is reasonably straightforward to include further commands or to employ a third party tool such as VampirTrace.

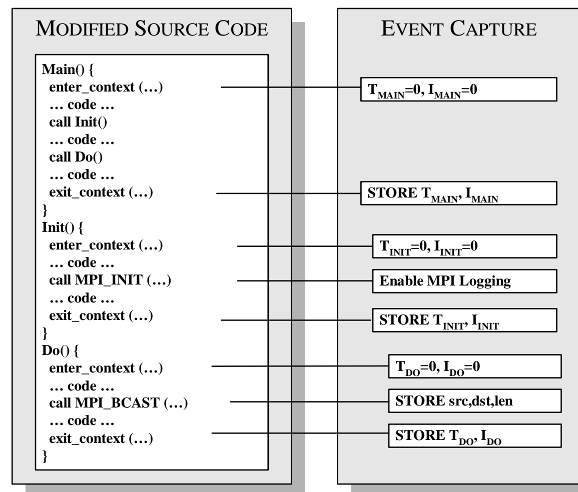

Figure 2. The instrumented source consists of calls to a backend event capture library. This library links events together based on context and writes out a text trace file when the application quits. On entering a context, a timer and instruction counter is reset. These are then stored on exit of the context for post-analysis

The process allows rapid instrumentation of the source code with a subsequent compile, link and run sequence to obtain the required tracing information. The events

that appear are essentially a call-graph of the program and can be subjected to a wide variety of analysis tools.

The instrumentation method is lightweight, consisting of a few array operations to store events in order to minimize the performance impact on the application. Depending upon the page size, periodic memory allocation is required which will incur a slight performance penalty. However, the principal purpose of this instrumentation utility is to illustrate the functional operation of the application which can be achieved by examination of the call-graph and the number of instructions issued, as opposed to performance-timings which are usually sensitive to experimental error.

4 Multi-view Visualisation

There are a number of visualisation tools that can assist with understanding an application's runtime behaviour. For MPI applications, Vampir provides a large suite of views that provide a good level of detail. It is essentially possible to "playback" the application and determine where communication occurs (and which processors are involved) and where periods of computation occur. However, these tools tend to produce views of a single application, albeit across many processors. In order to develop analytical performance models, it is useful to visualise the application behaviour over various iterations, processors or different runs.

Placing two traces "side-by-side" and allowing a tool to highlight the difference between the code densities (number of instructions in a subroutine) for two application-runs (presumably with different input-data sizes) is useful when attempting to identify critical regions of code. Using this approach, a performance modeller can rapidly determine if a code region is worth analysing or whether a single timing can be taken.

The tool developed in this work provides a multi-view visualisation of the trace files. Loading an event file generated by the instrumentation library described in Section 3, it reconstructs a complete call-graph of the application and compares it with subsequently loaded event files. The tool utilises an algorithm that searches for the largest groups of call-chain (or grouped call-chain) entries to locate similar code-regions. Entries in the trace files are deemed similar if the call-graph nodes match and that the instruction counts are close to a given sensitivity. Where entries differ or are non-existent, colour cues are used to highlight changes to the performance modeller.

When event-files are loaded, they can be linked together using a control panel to associates relevant view-options. The tool permits some filtering which allows the user to view MPI collectives, point-to-point functions and normal context (subroutines). A 'hotspot' slider allows the user to highlight portions of the trace where the inclusive number of instructions as a percentage of the overall total, is above the threshold percentage. The 'sensitivity' slider allows the user to set the extent to which the number of issued instructions in the first trace can deviate from the second. In tests, we found that a value of around 5% highlighted the regions of the call graph that altered significantly during a change in configuration. The current implementation places limitations on the size of the trace-files that can be examined at any given time. Schemes to partition the trace-files are employed to reduce the computational and

memory overhead of examining the traces. This includes viewing a single iteration of an application (which requires a small manual modification to the source code) or post-processing the files to follow the chains to a given depth.

The screenshots shown in Figures 3 and 4 illustrate traces that were loaded into the tool and visualised. Figure 3 demonstrates the comparative algorithm attempting to identify the largest similar run within the call-graph and creating links to the next trace. Where calls have not been made, they are highlighted in red and the link cursor between the traces points to where they should exist in the second trace. Where calls exist that are not in the second trace, they are highlighted in yellow and again link to where they are expected. The common colour, blue, indicates sections that exist in both traces but are different (i.e. instructions issued differ by a given threshold). The tool allows a chain of comparisons to be established, so that trace 1 is compared with 2, 2 with 3 and so on. It is assumed that the call-chains are broadly similar (such as differences in loop counts, conditionals, and communication patterns). Where multiple traces are entirely different, the tool effectiveness would be limited.

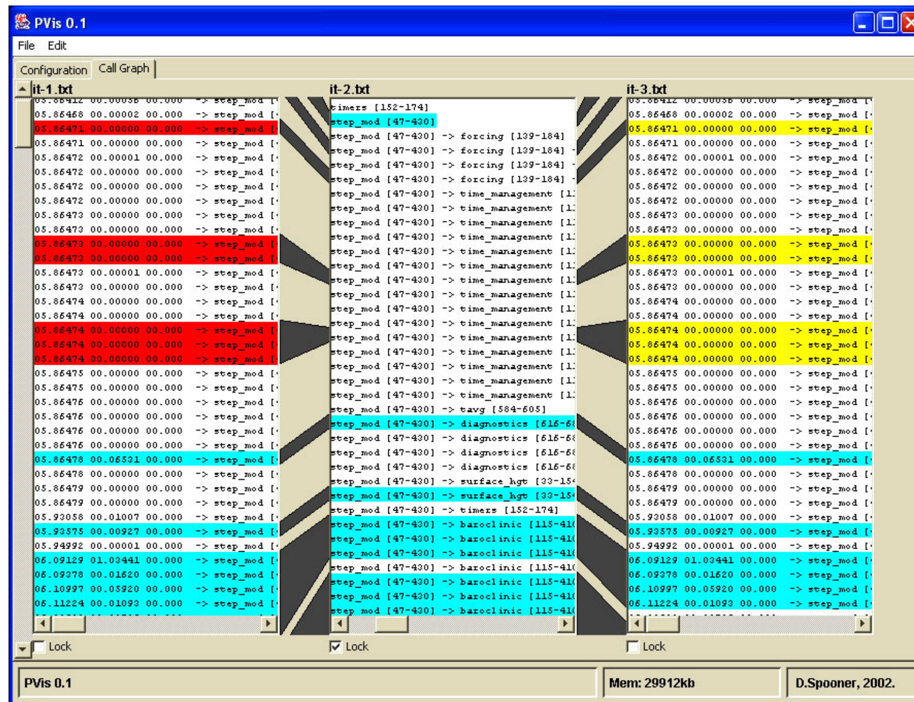

Figure 3. Visualisation screenshot comparing three iterations of the same application on the same processor. Where regions match, but have differences (such as the number of issued instructions), the tool connects and highlights the sections using colour cues. Regions that have been removed or added are also identified.

In addition to viewing the call graph, the tool is able to group call points together to obtain a frequency view of the application's trace. This is useful for grouping similar communications together and fits well with the combined analytical expressions that describe the overall application. Figure 4 illustrates this effect with traces from the Sweep3D ASCI demonstrator application. These are also subject to the same highlighting, so it is immediately obvious where one application has spent more time in a particular subroutine or exhibited different communication behaviour.

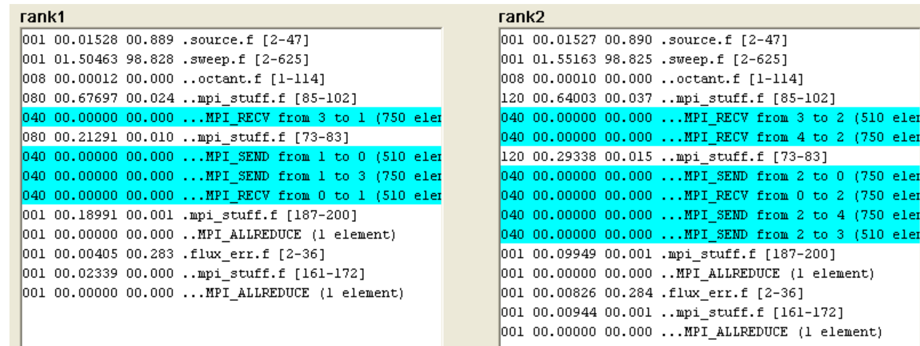

Figure 4. Visualisation screenshot comparing consolidated traces for two processes in a single iteration of Sweep3D. The tool has highlighted an area where change is identified: in this case, the communications patterns are different due to rank placement in a 2x6 processor grid.

Table 1. Example identifiable performance features from visualization tool.

Example Characteristic	Tool Assistance
Calculation impact due to data-set change.	• Highlighting of relative difference in instruction counts in similar regions of the call-graph. • Highlighting of relative difference in message sizes between PEs.
Data-placement differences due to variation in PE count.	• Highlighting of changes in communication patterns (source, destination) as topology is changed to account for configuration.
Functional differences in application iterations.	• Highlighting of new/deleted regions in call-graph and differences in frequency views.

Together the tools allow a performance modeller to rapidly profile a code, run it under different configurations (such as on a 4x4 processor network, and then on 8x8 network) and then load in the traces to analyse the result for particular performance characteristics (see Table 1). In the case of moving from one processor arrangement to

another, it is likely that the communication patterns will be different and, if the application scales weakly, large differences in the computation sections and message sizes. These visual cues aim to provide an effective summarised view of the program's dynamic operation. Once alerted to a particular region of code that appears sensitive to a particular change it is possible to direct attention to that part of the program which can ultimately assist in constructing the performance model.

The problem with viewing traces in this format is that screen estate is typically limited and it only takes a few traces to become impractical. In order to see more detail, the traces must be reduced into a more compact form and we envisage using individual pixels to visualise events in the trace-files. Displaying traces at this density should an observer to rapidly distinguish areas of interest and then "drill down" to view the relevant code.

5 Future Work and Conclusions

The approach described in this paper utilises dynamic trace files and a multi-view visualisation tool to highlight areas of interest when input parameters, data-sets and resource configurations are modified. By *focusing* on the areas of a code that are sensitive to configuration and input data, some of the drudgery is removed in terms of isolating the critical regions that govern the performance characteristic of an application.

Future developments of this approach include developing the visualisation tool to utilise the Vampir trace facility which is widely utilised in parallel performance studies. Additional work will focus on utilising exotic forms of visualisation to obtain compact views of multiple trace files relating to different performance scenarios.

Acknowledgements

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the US Department of Energy.

References

1. Kerbyson, D.J., Hoisie, A., Wasserman, H.J.: Use of Predictive Performance Modelling During Large-Scale System Installation, in Proc. PACT-SPDSEC02, Charlottesville, VA. (September 2002)
2. Kerbyson, D.J., Hoisie, A., Wasserman, H.J.: A Comparison Between the Earth Simulator and AlphaServer Systems using Predictive Application Performance Models, Computer Architecture News, ACM (December 2002)

3. Spooner, D.P., Jarvis, S.A., Cao, J., Saini, S., Nudd, G.R.: Local Grid Scheduling Techniques using Performance Prediction, IEE Proc. - Computers and Digital Techniques (2003)
4. Perry, S.C., Grimwood, R.H., Kerbyson, D.J., Papaefstathiou, E., Nudd, G.R.: Performance Optimisation of Financial Option Calculations, Parallel Computing, **26**(5), Elsvier (2000) 623-639
5. Mellor-Crummey, J., Marin, G.: Building Parameterized Models for Black-Box Applications, in Proc. Los Alamos Computer Science Institute Symposium (LACSI), Santa Fe (October 2002)
6. Snavely, A, Carrington, L., Purkayastha, A., et. al.: A Framework for Application Performance Modeling and Prediction, in Proc. SC2002, Baltimore (2002)
7. Cain, H.W., Miller, B.P., Wylie, B.J.N.: A Callgraph-based Search Strategy for Automated Performance Diagnosis, Concurrency and Computation: Practice and Experience **14** (2002) 203-217.
8. Vampir, Pallas. `http://www.pallas.com/e/products/vampir/index.htm`
9. Eick, S.C., Steffen, J.L., Sumner, E.E.: Seesoft-A Tool for Visualizing Line Oriented Software Statistics, IEEE Trans. on Software Eng., **18**(11) (1992) 957-968
10. Nudd, G.R., Kerbyson, D.J., et.al.: PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems, Int. J. of High Performance Computing Applications **14** (2000) 228-251
11. Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.L.: Predictive Performance and Scalability Modeling of a Large-scale Application, in Proc. SC2001, Denver (2001)
12. Kerbyson, D.J., Hoisie, A., Pautz, S.D.: Performance Modeling of Deterministic Transport Computations, in Performance Analysis and Distributed Computing, Kluwer (2003)
13. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.E., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool, IEEE Computing **28**(11) (1995) 37-46
14. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors, Int. J. of High Performance Computing Applications **14**(3) (2000) 189-204