An Annotated Bibliography of Interactive Program Steering

Weiming Gu (weiming@cc.gatech.edu)

Jeffrey Vetter (vetter@cc.gatech.edu)

Karsten Schwan (schwan@cc.gatech.edu)

GIT–CC–94–15

Revised November 1, 1994

Abstract

This annotated bibliography reviews current research in dynamic and interactive program steering. In particular, we review systems-related research addressing dynamic program steering, raising issues in operating and language systems, mechanisms and algorithms for dynamic program adaptation, program monitoring and the associated data storage techniques, and the design of dynamically steerable or adaptable programs. We define *program steering* as the capacity to control the execution of long-running, resource-intensive programs. Dynamic program steering consists of two separable tasks: monitoring program or system state (monitoring) and then enacting program changes made in response to observed state changes (steering).

College of Computing

Georgia Institute of Technology

Atlanta, Georgia 30332–0280

1 Introduction

The intent of program steering may be illustrated with the following quote from [114]:

> Scientists not only want to analyze data that results from super-computations; they also want to interpret what is happening to the data during super-computations. Researchers want to steer calculations in close-to-real-time; they want to be able to change parameters, resolution or representation, and see the effects. They want to drive the scientific discovery process; they want to *interact* with their data.

This bibliography reviews current research in dynamic and interactive program steering. In particular, we review systems-related research addressing dynamic program steering, and we raise issues in operating and language systems, mechanisms and algorithms for dynamic program adaptation, program monitoring and the associated data storage techniques, and the design of dynamically steerable or adaptable programs. Most of this research is performed in the context of distributed and parallel machines. This bibliography does not review research concerning information display, interactive user interfaces, scientific visualization, and program animation. We refer the reader to articles by Stasko [116], Malony [119], Heath [120], Pancake [117], Myers, [111], Francioni [121], or Madhyastha [122] (cited in Section Additional References) for more information on these topics.

We define *program steering* as the capacity to control the execution of long-running, resource-intensive programs. Such online control of execution includes modifying program state, managing data output, starting and stalling program execution, altering resource allocations, etc. Dynamic program steering consists of two distinct tasks: monitoring program or system state (monitoring) and enacting program changes made in response to observed state changes (steering). Past research in program monitoring has extensively addressed off-line monitoring (e.g., using trace files) and intrusive monitoring as performed for parallel or distributed debugging. We briefly review such work, but this survey focuses on the on-line monitoring necessary for program steering. Past work on program modification has typically concerned program reuse, maintainability, etc. This survey concerns only the dynamic modification of programs, primarily addressed used by real-time and distributed applications, including process migration and load balancing for enhanced system performance or fault tolerance.

Program monitoring is essential for most high-performance parallel and distributed codes, in part because of the inherent complexity of large-scale parallel and distributed programs, and because of their non-deterministic nature caused by the concurrent execution of different program components (including the monitoring system). As a result, effective tools are necessary to help users understand program performance and run-time behavior. Monitoring is an integral component of any such tool. However, to be useful for on-line program control and interaction, monitoring tools must also limit their degrees of intrusiveness compared to tools designed for program debugging (e.g., they should not require that programs be stopped in order to be monitored), offer a variety of program interfaces (e.g., not just trace files), and in general, offer a variety of mechanisms both for information acquisition, analysis, transmission, and storage. With this variety, a user/developer should be able to satisfy the different requirements of low- versus high-latency program control that they might want to use for monitoring. While these comments certainly do not constitute a complete list of requirements imposed on monitoring when steering programs, they limit the extent of the literature reviewed in this bibliography.

When parallel or distributed application programs execute from several hours to several weeks, it can be important to have human beings 'in the loop'. Specifically, human users can assist and guide long running computations using their domain knowledge, which is often difficult, if not impossible, to encode in automatic algorithms performing on-line program control. We call this process *interactive program steering*. The result of such steering is the configuration of the program and/or its execution environment (e.g., the configuration of underlying operating system functions). The purpose of such configuration is to affect the program's execution behavior. Therefore, interactive program steering implies (1) monitoring – obtaining and analyzing information about the running program and its execution environment, (2) information presentation – presenting the information to the human user, and (3) steering – enabling the user to affect the execution of the running program. The use of graphical user interfaces (GUIs) is often essential [114] for performing tasks (2) and (3).

In addition to interactive program steering, systems may be controlled by on-line algorithms. This method entails processing of monitoring information by algorithms which decide on necessary changes in program configuration and then, enact such changes via the steering mechanisms. This implies that on-line program monitoring and steering must offer well-defined application program interfaces (APIs) to such 'control' or 'adaptation' algorithms. Systems able to react to state changes using on-line adaptation algorithms are often called 'reactive' systems.

This bibliography has several limitations that merit discussion. First, we focus primarily on system performance rather than reliability. Second, we narrow our focus in areas, like instrumentation, to those papers relevant to interactive program steering. Third, for ease of access to cited references, we cite only reviewed, published articles with an occasional, auxiliary reference to a PhD thesis or technical report.

This bibliography has seven sections. Following the introduction, Section 2 reviews the different conceptual models used to express program states, information about programs, and the program configuration information required for on-line steering. Several systems that use these models are described. Section 3 reviews work that deals with specific issues in program or system instrumentation and data collection (or capture). Section 4 addresses data analysis techniques, including post-mortem data analysis, on-line data analysis, perturbation analysis, and static program analysis. Issues in dynamic and adaptable systems are reviewed in Section 5. Interactive program steering is addressed in Section 6. Research not easily classified into this survey's sections is reviewed in Section 7.

2 Modeling and Systems

A variety of conceptual models and prototype systems have been constructed to address the monitoring, debugging, and steering of high-performance programs. Representative models and systems are reviewed in this section. Early systems tend to offer simple conceptual models of program information, typically derived from prior research on sequential programs, whereas recent systems employ more sophisticated models using complex events derived from database research and object oriented programming methods.

2.1 Program Profiling

Unix gprof is a commonly used performance tuning tool for sequential programming [1]. Gprof directs the programmer's attention to program components (procedures) that may be the cause of performance problems, by providing call graphs, statistics on procedure and function calls, and information about the average time spent in a program's various routines.

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.

Several commercial tools for parallel performance profiling are derived from gprof. They measure the execution times spent in a parallel code's different procedures. The Parasight profiler by Encore Corp. [2] and the Quartz profiler developed at the Univ. of Washington [3] are both based on the ideas found in gprof. The Parasight profiler's implementation extends gprof using interactive and non-intrusive instrumentation. *Profview*, for Silicon Graphics machines, offers high quality user interfaces to gprof.

[2] Ziya Aral and Ilya Gertner. Non-intrusive and interactive profiling in Parasight. In *Proceedings of the ACM/SIGPLAN PPEALS*, pp. 21–30, Marlborough, Massachusetts, July 1988.

The experimental tool, Quartz [3], extends the sequential profiling of gprof with a new metric for parallel program performance, called *normalized processor time*. For each procedure, this metric calculates the total processor time spent in the procedure divided by the actual number of processors that are busy when this procedure is being executed. This measure attempts to direct the programmer's attention toward poorly parallelized code.

[3] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 115–125, Boston, May 1990.

One major limitation of profiling-based tools is the lack of depth in the information they provide. From profiling data alone, it is difficult to derive insightful information about a program's execution, such as how data is shared during run time, how synchronization among different components is achieved, etc. More sophisticated approaches are required to provide such information. For the same reasons, profiling information alone is likely to be insufficient for program steering, which tends to require detailed run-time performance and program information from the target application. However, when used with only selected program procedures, profiling has the advantages of low overhead and relatively small degrees of intrusiveness. More importantly, profiling is easily automated by compilers which make it an important auxiliary source of information for on-line program steering. Additional references concerning profiling include:

[4] Jonathan D. Becher and Kent L. Beck. Profiling on a massively parallel computer. In *Proceedings of CONPAR'92*, pp. 97–102, Lyon, France, September 1992.

[5] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel program performance metrics: A comparison and validation. In *Proceedings of Supercomputing'92*, pp. 4–13, Minneapolis, Minnesota, November 1992.

[6] Mark E. Crovella and Thomas J. LeBlanc. Performance debugging using parallel performance predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 140–150, San Diego, California, May 1993.

2.2 Event-based Approaches

An *event* is a characteristic atomic program behavior [7]. Event-based modeling approaches support the construction of higher-level program behaviors from a stream of events which represent the activities of the target system. In an event based monitoring system, basic events are captured by sensors which are inserted into the target systems, and more complex program behaviors are constructed from these basic events. In more recent systems, visualizations and animated graphical displays are used to show program behaviors visually [120].

Advantages of event-based monitoring include: events are easily formulated and interpreted by users, and recent systems have begun to standardize event formats, so that event-based monitoring information may be shared between different analysis or display systems.

Bates [7, 8] is one of the first researchers to formulate complex notions of events for the debugging of distributed systems. In [7], he proposes an *Event Definition Language (EDL)* for describing program behaviors (termed *behavioral abstraction* in the paper). The process of behavioral abstraction is achieved by event *filtering* and event *clustering*. Filtering removes all but a selected subset of events from the event stream, while clustering forms complex events from lower-level events. In essence, event-based debugging is a process of building models of program's behavior from the program events and comparing these to the models of the program's intended behavior. Differences discovered during this comparison indicate errors in the target program. In [8], Bates discusses a prototype debugging system based on the *Event Based Behavioral Abstraction (EBBA)* debugging approach.

[7] Peter C. Bates and Jack C. Wileden. Event definition language: An aid to monitoring and debugging complex software systems. In *Proceedings of 5th Hawaii International Conference on System Sciences*, January 1982.

[8] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pp. 11–22, Madison, Wisconsin, May 1988.

Miller's IPS [9] [10] parallel program measurement system is based on a hierarchical program model, assuming that most distributed and parallel programs are structured hierarchically. Events are captured and performance is measured at several levels: at the level of the entire program, at the machine level, at the process level, at the procedure level, and at the primitive activity level. The system performs two types of analysis on trace data: critical path analysis and phase behavior analysis. Critical path analysis identifies the program parts and sequences of events comprising the largest amount of program execution time, while phase behavior analysis identifies the different phases of computation performed by the program.

[9] Barton P. Miller and Cui-Qing Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 482–489, Berlin, West Germany, September 1987.

[10] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

Reed's *Pablo* system [11, 12] is an event-based performance analysis environment providing performance data capture, analysis, and presentation. Three classes of events differing in their degrees of detail and perturbation are supported: *trace*, *count*, and *time interval* events (much like the different classes of events identified earlier by Snodgrass [24]). Trace events record all occurrences of a specified event, which may produce a large amount of data and can result in substantial perturbation of the target program's execution. Count events generate a comparatively small amount of data, by recording only the number of occurrences of each event. Time interval events associate an event with a pair of source code points, and each occurrence of an event contains the time that elapsed during execution of the source code fragment.

[11] D. A. Reed, R. D. Olson, R. A. Aydt, T. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty. Scalable performance environments for parallel systems. In *Proceedings of Sixth Distributed Memory Computing Conference*, pp. 562–569. IEEE Computer Society Press, 1991.

[12] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment.* Department of Computer Science, University of Illinois, Urbana, Illinois, November 1992.

Bruegge's *BEE* system [13] is a platform for building event-based environments to monitor and debug the performance of distributed and heterogeneous application programs. It supports predefined events such as the execution of a function, as well as user-defined events which can be used to specify any application-specific monitoring information. BEE distinguishes four kinds of activities concerning events: event sensoring, event generation, event handling, and event interpretation. The independent implementation of each activity contributes to system portability. A later version of BEE, called *BEE++* [14], is an object-oriented implementation.

[13] Bernd Bruegge. A portable platform for distributed event environments. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 184–193, Santa Cruz, California, May 20-21 1991.

[14] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In Andreas Paepcke, editor, *The Proceedings of OOPSLA 1993*, pp. 65–82. October 1993.

In ZM4/SIMPLE, Dauphin et al. [15] propose an approach that integrates performance monitoring with the process of program construction. For each program, a *functional model* and a *functional implementation model* are built, based on a specification of the problem and on the selection of an algorithm solving the problem. Next, a *program implementation* and a *monitoring model* are derived from the functional implementation model. The advantages of coupling performance monitoring with program construction are that program instrumentation can be performed automatically by using program construction information, and performance information obtained from a program's execution can be used to improve the program itself. The disadvantages of this approach are that it only supports applications constructed with the system, and defects in program construction may result in defects in program monitoring and therefore, may generate faulty and misleading performance information.

[15] Peter Dauphin, Richard Hofmann, Rainer Klar, Bernd Mohr, Andreas Quick, Markus Siegle, and Fanz Sotz. ZM4/SIMPLE: a general approach to performance measurement and evaluation of distributed systems. In T. L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design.* IEEE Computer Society Press, 1992.

The PICL [16] [17] library, developed at Oak Ridge National Laboratory for message-passing parallel programming with the PVM system[123], captures PVM message send and message receive events. Trace data obtained by the embedded monitor in the PICL library may be used by other tools including ParaGraph, which displays run-time information graphically. An interactive interface to PVM, called XAB, offers some of the Paragraph displays for on-line program monitoring. The implementation of such on-line monitoring relies on a central recipient of all monitoring events (also called a central monitor in [27]) captured in the distributed application and runtime system.

[16] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. *PICL – A Portable Instrumented Communication Library – C Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[17] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. *A Users' Guide to PICL – A Portable Instrumented Communication Library.* Oak Ridge National Laboratory, Oak Ridge, TN, May 1991.

One interesting, and perhaps, problematic, attribute of event-based modeling of monitoring information is that event-based models are necessarily based on the control flow of the target application. Therefore, event-based models are often specific to the way in which a target application is implemented [14]. For example, the primitive events for monitoring programs implemented on distributed memory paradigms always include message-send and message-receive [16] [17] events, while the primitive events for programs implemented on shared memory paradigms include shared-resource-lock and shared-resource-unlock events. A second attribute of event-based approaches is that there is no clear boundaries between different layers of abstraction concerning event-based descriptions of program behaviors, which can make event-based information modeling quite complex.

Additional references related to event based modeling include:

[18] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed systems. In *Proceedings of the 5th IEEE International Conference on Distributed Computing Systems*, pp. 515–522, May 1985.

[19] Dieter Haban and Dieter Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[20] D.C. Marinescu, J.E. Lumpp, T.L. Casavant, and H.J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, June 1990.

[21] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The Ariadne debugger: Scalable application of event-based abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 85–95, San Diego, California, May 1993.

JEWEL [22] is an application-specific, distributed measurement system. JEWEL offers a set of generic and extendible components to the developer. The components include a graphical presentation system for online visualizations and an *interactive experiment control system* that provides central control of the distributed measuring system.

[22] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurement System, *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657-671, Nov. 1992.

2.3 Relational Approaches

The event-based modeling approach describes program behavior in terms of a program's control flow. A different approach based on information modeling was first proposed by Snodgrass in his PhD thesis [23]. This approach treats monitoring information (runtime data, states of processes, states of processors, messages, etc.) as relations, as with data in a relational database. The program states and activities observed by the monitor, including those based on control flow, are specified by a high-level query language called *TQuel* (considerably refined in [24]). The query language is a superset of the relational database query language *Quel*. It is based on a formal temporal algebra, in which a time-stamp is included as a special implicit attribute for each relation. New syntax and semantics for retrieve statements utilize the new temporal capabilities of such 'monitoring relations'. Temporal operators include **overlap**, **extend**, **begin of**, **end of**, etc. The relational approach to monitoring complex systems is further refined in [25], and resulting research on temporal databases (addressing the efficient storage and retrieval of temporal information) is leading to international efforts to standardize the semantics of temporal databases and their query languages.

[23] Richard Snodgrass. *Monitoring Distributed Systems: A Relational Approach.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, Carnegie-Mellon University, December 1982.

[24] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[25] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

Based, in part, on the relational approach originated by Snodgrass, Ogle et al. [27, 26] explore the application-dependent and on-line or dynamic monitoring of parallel and distributed applications. Monitoring information is modeled using two different languages, an *attribute language* and a *view language*. Application-dependent information (e.g., the length of a shared queue) is first described as attributes using the attribute language. Based on the specified attributes, the view language is used to specify higher-level views of potentially distributed monitoring information as entities, relationships, and sets of both. Each such view specifies (1) the involved entities (or relationships) and attributes, (2) the time at which the view is considered active, (3) performance and correctness criteria such as latency and perturbation constraints, and (4) the action to be taken when the view is active.

[26] David M. Ogle. *Real-Time Monitoring of Parallel and Distributed Programs.* PhD thesis, Department of Computer and Information Sciences, The Ohio State University, July 1988.

[27] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

Kilpatrick and Schwan [29, 30] further refine the language-based approaches for specifying application-dependent monitoring requests, and they further advance the idea of making the monitoring system independent of the application to be monitored and of its run-time environment. Kilpatrick also explores the idea of integrating all components of a parallel programming environment using the uniform Entity-Relationship information model (also see [115]), with special attention paid to using monitoring information for graphical views [31].

[28] Carol Kilpatrick, Karsten Schwan, and David Ogle. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. In *International Conference on Computer Languages*, pp. 180–189, New Orleans, LA, March 1990.

[29] Carol E. Kilpatrick. *Capture and Display of Performance Information for Parallel and Distributed Applications.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, March 1991.

[30] Carol E. Kilpatrick and Karsten Schwan. ChaosMON – application-specific monitoring and display of performance information for parallel and distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 57–67, Santa Cruz, California, May 20-21 1991.

[31] Jim Matthews and Karsten Schwan. Graphical Views of Parallel Programs. OSU-CISRC-TR-85-11, Computer and Information Science, The Ohio State University, September, 1985.

3 Instrumentation and Data Collection

Instrumentation of a target application and its run-time system is the first step toward application steering. Hardware monitoring and data collection require instrumentation of the hardware platform on which the target application is running. Alternatively, software monitoring and data collection require instrumentation of the program's source code, the system libraries, the compiler, or any combination of the above. Hardware monitoring is attractive in offering a low degree of intrusiveness, resulting in low overhead and low perturbation of the execution of the target application. However, its cost, inherent inflexibility and its inability to provide high-level monitoring information limit its usefulness for application-dependent monitoring and for on-line program steering. Software monitoring or hybrid hardware-software monitoring is used by a majority of the existing parallel and distributed performance tools.

Program instrumentation for software monitoring can be performed automatically, semi-automatically, or manually. Profiling systems [1, 2, 3] perform automatic instrumentation (of the application code or of the underlying system libraries) to obtain run-time execution information. Automatic instrumentation can also be used if the monitoring system is interested only in a limited number of predefined events (e.g., as in Pablo [12] and PICL [16]) or if instrumentation points can be derived from existing information maintained by compilers, linkers, or loaders (e.g., as in IPS-2 [10] and ZM4/SIMPLE [15]).

Application-specific monitoring [33, 25, 27, 29, 14] requires extensive compiler and/or user involvement in program instrumentation. In such systems, high-level specification mechanisms (e.g., a temporal query language in [24], a script language in [33], and specification languages in [26] and [30]) are provided to the user to describe what to monitor. Specification language compilers generate 'sensor' code, which is then semi-automatically or manually inserted into the target application's code. The user must provide the code locations to be used for sensor insertion.

When the instrumented application executes, trace data records or sampling data are generated by the inserted sensors, and are collected and stored into a data structure, trace file, or a database. In more sophisticated systems, trace data collection is assisted by the OS [25] or by a dedicated monitoring component, called a *local monitor* in [29] or a *resident monitor* in [33] [25] [27]. A second monitoring component is used in addition, termed *central monitor* in [29] or *relational monitor* in [33]. This component possibly executes on another machine or processor and communicates with the local monitor(s) over a network or through shared memory. Advantages of using two monitoring components include: system-specific issues are localized to the local (resident) monitors, and more than one local monitor might be used to monitor a large-scale parallel or distributed application. The events collected by local monitors may be analyzed locally (see [27]) or simply sent to the central monitor for analysis and storage.

Hollingsworth et al. [32] implemented a dynamic instrumentation technique, which defers inserting instrumentation until the target application is in execution. In this instrumentation model, users associate simple operations (called *primitives*, e.g. changing values of a counter or a timer) with insertion *points*, and provide additional *predicates* to control the execution of the primitives. Advantages of this instrumentation approach includes its ability to dynamically control the insertion points and its efficiency in instrumentation execution because of operating at binary code level. However, application-specific monitoring is not easily supported because instrumentation takes place at the binary code level.

[32] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, pp. 841–850, Knoxville, TN, May 1994.

Segall et al. [33] present an *Integrated Instrumentation Environment (IIE)* for experimentation on multiprocessors. In this environment, an experiment is described as a script, called a *schema*. The results of an experiment are captured by *schema instances*, which store the output from the execution of a program and the results from its performance measurements. A user specifies the behavior of his parallel program in a high-level behavior description language, called *B-language*, as a directed data flow graph. Each node in the graph represents a subtask, which is a parallel program component that executes in parallel with other subtasks. A buffer is associated with each arc to contain data or control tokens flowing from one subtask to another. Instrumentation is achieved by adding sensors, either built-in or user-specified, to the B-language script. When the instrumented code, translated from the script, is running, event records are captured by embedded sensors and collected by a *resident monitor*. The result data from schema instances is analyzed to detect errors in the program.

[33] Zary Segall, Ajay Singh, Richard T. Snodgrass, Anita K. Jones, and Daniel P. Siewiorek. An integrated instrumentation environment for multiprocessors. *IEEE Transactions on Computers*, C-32(1):4–14, January 1983.

Additional references concerning instrumentation and data collection include:

[34] Devesh Bhatt and Michael Schroeder. A comprehensive approach to instrumentation for experimentation in a distributed computing environment. In *Proceedings of the 3rd International Conference on Distributed Systems*, pp. 330–340, October 1982.

[35] Zary Segall and Larry Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, 11(11):22–37, November 1985.

[36] D. Bhatt, A. Ghonami, and R. Ramanujan. An instrumented test-bed for real-time distributed systems development. In *Proceedings of the 8th Real-Time Systems Symposium*, December 1987.

[37] Raymond R. Glenn and Daniel V. Pryor. Instrumentation for a massively parallel MIMD application. *Journal of Parallel and Distributed Computing*, 12:223–236, 1991.

[38] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978, November 1992.

4 Data Analysis

The amount of trace data generated by inserted sensors and collected by the run-time monitoring mechanism is usually too large, and the information is too low-level to be directly useful to any human user. Alternatively, sampling sensors can limit the amount of information generated by the monitoring system; however, this depends on the sampling rate for the sensor. Trace data filtering and analysis must be performed to generate information interesting to end users. In addition, the monitoring mechanism may perturb the execution of the target program, so that trace data may not be accurate. Perturbation analyses are performed on trace data to generate accurate traces.

4.1 Post-mortem Trace Data Analysis

Many existing parallel and distributed performance debugging tools use the trace-and-replay approach: trace data is captured when the target program executes; trace data is then analyzed to detect anomalies (e.g., data races) and performance bottlenecks in the program, or it is used to reproduce the execution. If sufficient information is captured during the program's initial execution, the same bugs or performance problems can be reproduced during program re-execution (from its trace), thereby assisting users in finding and correcting them.

Several techniques have been proposed by researchers to minimize the amount of trace data that is required for reproduction of a program's execution: Choi [45] and Mellor-Crummey [50] use compile-time program information; Miller [41, 45] proposes an incremental tracing technique; Netzer [49] and Griswold [46] propose optimal trace algorithms.

Instant Replay [39] is a debugging approach that is designed to reproduce the execution behavior of a parallel program. The prototype debugging tool based on this idea captures the relative order of all accesses to shared objects in parallel codes from the program's initial execution. From this information, a partial order of accesses to such objects can be established. During program replay, the actual program is re-executed, and each program process repeats its computations. However, the recorded trace data forces the accesses to shared objects to be performed in the same order as done in the initial program run. As a result, the initial program's execution behavior is reproduced faithfully, and program trace files remain comparatively small.

[39] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471-481, April 1987.

Helmbold et al. [40] presents algorithms to compute possible event orderings that can be derived from the trace of a program execution. These algorithms can be used to detect potential data races in a parallel program which uses shared resources.

[40] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):827–840, July 1993.

Additional references on post-mortem trace data analyses include:

[41] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pp. 141–150, Madison, Wisconsin, May 5-8 1988.

[42] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[43] C. W. Oehlrich and A. Quick. Performance evaluation of a communication system for transputer-networks based on monitored event traces. In *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 202–211, Toronto, May 27-30 1991. ACM SIGARCH, 19(3).

[44] Jong-Deok Choi, Barton P. Miller, and Robert H.B. Netzer. Techniques for debugging parallel programs with flow-back analysis. *ACM Transactions on Programming Language and Systems*, 13(4):491–530, October 1991.

[45] Jong-Deok Choi and Janice M. Stone. Balancing runtime and replay costs in a trace-and-relay system. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 26–35, Santa Cruz, California, May 20-21 1991.

[46] Victor Jon Griswold. Core algorithms for autonomous monitoring of distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 36–45, Santa Cruz, California, May 20-21 1991.

[47] Madalene Spezialetti. An approach to reducing delays in recognizing distributed event occurrences. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 155–166, Santa Cruz, California, May 20-21 1991.

[48] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 167–174, Santa Cruz, California, May 20-21 1991.

[49] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1–11, San Diego, California, May 1993.

[50] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 129–139, San Diego, California, May 1993.

[51] Diane T. Rover and Abdul Waheed. Multiple-domain analysis methods. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 53–63, San Diego, California, May 1993.

4.2 On-line Trace Data Analysis

One of the main disadvantages of trace based analysis methods is their complexity in part, because most problems these algorithms try to solve are NP-complete. In addition, trace-and-replay techniques may not be suitable for debugging large-scale parallel and distributed applications, simply because repeated execution of these applications may be too expensive. One possible solution is to use on-line performance monitoring and debugging tools. On-line trace data analysis is required for an on-line tool to reduce the amount of trace data generated, to reduce the perturbation to the execution of the target program, and to present monitored information to the end user on-the-fly.

Snodgrass [23] proposes a technique called *update networks* to process monitored information. The information to be monitored is modeled by temporal relations in a hierarchical structure [24]. At the bottom of the structure are primitive relations and at the top are relations composed of primitive relations and other composite relations. The hierarchical relations are transformed into a directed acyclic graph, in which the tuples of the primitive relations enter the nodes at the bottom and the tuples of the composed relations flow out of the nodes at the top. There are two types of nodes in an update network: access nodes and operator nodes. Information in the form of tuples flows out of an access node, while an operation node takes tuples from one or more lower level nodes and produces tuples to be used by higher level nodes.

The technique used by Ogle and Schwan [27, 26] to meet real-time and perturbation requirements is to distribute on-line trace data processing to different layers of the monitoring mechanism. The analysis of data being collected may be performed either by individual sensors, by the resident monitor, by the central monitor, or by any combination thereof. Tradeoffs in the use of such techniques include computational versus communication costs in monitoring, the amounts of computation to be performed by resident versus central monitors, etc. In [27], a method is provided for generating distributed codes for trace data analysis: (1) generate all possible view implementation plans that preserve the semantics of the target list and of the action predicates, (2) discard those plans that violate stated latency constraints by estimating the maximum latency of a given plan, (3) choose the plan from among the remaining plans that minimizes monitoring perturbation.

Additional references concerning on-line trace data analyses follow:

[52] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing'90*, pp. 74–81, New York, December 1990.

[53] John M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pp. 24–33, November 1991.

[54] Yong-Kee Jun and Kern Koh. On-the-fly detection of access anomalies in nested parallel loops. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 107–117, San Diego, California, May 1993.

[55] Doug Kimelman and Dror Zernik. On-the-fly topological sort – a base for interactive debugging and live visualization of parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 12–20, San Diego, California, May 1993.

4.3 Perturbation Analysis

Malony and Reed [56] have studied the perturbation caused by instrumentation. Two perturbation models are the *time-based* model and the *event-based* model, each used to estimate a different kind of perturbation, *sequential perturbation* and *parallel perturbation*. Sequential perturbations are those effects that can be contained to a local region of the instrumented code. However, in concurrent execution mode, instrumentation may change the execution order of events, affect shared resource allocation, or alter the scheduling of tasks. Such effects are called parallel perturbations. Sequential perturbation can be easily modeled by time-based modeling: perturbation is simply the total amount of time spent on executing instrumentation code. However, for modeling the execution of concurrent program parts, the authors approximate their execution times by the execution times of their critical paths, and the perturbation of the concurrent part is approximated by the time spent in instrumented code on the critical path.

Event-based perturbation analysis attempts to derive the approximate time at which each event occurs. A simple model for event-based perturbation model for sequential programs is easy to understand: the actual time when an event occurs is the measured time less the perturbation introduced by the instrumentation points before this event in the execution sequence. Approximation of event times for concurrent traces is much more difficult than for sequential traces, because the perturbation of each event is determined by the perturbation of all events on the critical path to this event. The analysis presented in [56] make several simplifying assumptions to estimate concurrent perturbation.

[56] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

In [57], Williams et al. use timed petri nets (TPNs) to model perturbations in trace data obtained from message-passing parallel programs. Generation of a petri net for a parallel program is performed in a top-down fashion, beginning by representing each process by a petri net with a single token, and adding inter-process communication links as places. This procedure continues by replacing process tokens by a sub-petri net that contains routines as places and inter-routine communication links as places. The procedure stops when the petri nets are refined to program statements. The recovery of a true trace is achieved by analyzing the TPNs. Two analysis techniques are explored by the group: *finite perturbation analysis* (FPA) and *perturbation tracking* (PT). FPA explicitly picks out incorrectly ordered events and delays, corrects them, and computes the actual trace. PT is adapted from FPA.

[57] K. J. Williams, M. S. Andersland, J. A. Gannon, Jr. J. E. Lumpp, and T. L. Casavant. Necessary conditions for tracking timing perturbations in timed Petri nets. In *Proceedings of the 13th Allerton Conference on Communication, Control, and Computing*, 1992.

4.4 Static Analysis

Static analysis of programs has been widely used in compiler techniques for program optimization and in the automatic transformation of sequential programs to vector or parallel programs. Commonly used static program analysis techniques include interprocedure analysis [112], data flow analysis, and data dependence analysis [113, 118]. The information from static analysis can also be used in parallel and distributed program debugging. In [45], statically derived program information is used to balance runtime tracing cost and trace replay time. Other uses of static program analysis include detecting abnormal program states such as deadlocks [61, 59], detecting race condition [60], and computing global predicates [63, 62].

A concurrent history graph is a representation of all possible concurrency states that a parallel program can enter. Although such a graph is very useful (e.g., to detect potential data races in the program), it can also be very large. In [60], Helmbold and McDowell propose a new abstraction mechanism in which multiple states are folded into one node, to reduce the size of concurrency graphs. Three basic types of folding techniques are supported: *resolution folding*, *value folding*, and *attribute folding*. The first folding technique ignores an attribute of a state and represents those program states that differ only in the discarded attribute by a single node. Value folding considers those program states as one if they differ only in one attribute, and their values of that attribute are in a specified set. Attribute folding represents all possible concurrent program states over a set of attributes and a set of values by one node. Another folding method called *entity folding* uses a DAG (Directed Acyclic Graph) to represent a set of objects (e.g. tasks) and disjoint paths in the DAG to represent possible program states.

[58] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 21–30. March 1990.

In [59], a polynomial-time algorithm is proposed to statically detect deadlocks in a subset of the Ada language. The program representation is extended to include nearly all of the Ada rendezvous primitives. The paper also presents preliminary experimental results for an implementation of their algorithm.

[59] Stephen P. Masticola and Barbara G. Ryder. A model of ADA programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 97–107, Santa Cruz, California, May 20-21 1991.

Additional references on static program analyses include:

[60] David P. Helmbold and Charles E. McDowell. Computing reachable states of parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 76–84, Santa Cruz, California, May 20-21 1991.

[61] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 85–96, Santa Cruz, California, May 20-21 1991.

[62] Alexander I. Tomlinson and Vijay K. Garg. Detecting relational global predicates in distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 21–31, San Diego, California, May 1993.

[63] Michel Hurfin, Noel Plouzeau, and Michel Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 32–42, San Diego, California, May 1993.

5 Dynamic Systems

While the main goal of on-line program steering is configuring a program to achieve or maintain high levels of performance or reliability, related research in automatic process migration, dynamic system (re-)configuration, and dynamic program tuning and adaptation attempts to achieve similar goals while using slightly different performance measures. Process migration attempts to balance system loads when moving processes from loaded to less loaded nodes. Dynamic program adaptations in real-time systems may be performed to judge program performance versus timing reliability, often in response to changes in the program's run-time environment.

This section reviews research in 'dynamic' systems. We first review program configuration and control, followed by program tuning and adaptation. These two categories are distinguished by the kinds of dynamic changes they support: program components are internally modified in dynamic tuning and adaptation systems, whereas dynamic system control and configuration only addresses changes to the collection of these components, not the components themselves.

5.1 Dynamic Program Configuration and Control

Papers in this subsection deal with issues in dynamic reconfiguration at the operating system level, the process level, and the application level. The main topic of these papers is to facilitate adding or removing an application program, one of its components, or a process during program execution. Compared with the alternative static techniques, in which the system or application program has to be stopped before any changes are made, dynamic reconfiguration avoids recompilation, reloading of compiled code, loss of state, and restarting the stalled system.

Dynamic reconfiguration is justified when stopping an application is too costly or infeasible, even when such reconfiguration is intended to improve program performance, etc.

Kramer and Magee present a model for dynamic reconfiguration in distributed systems [64], emphasizing that system configuration should be separated from the implementation of the system components themselves. Dynamic changes are performed at the level of system components. One main contribution of this work is the compiled list of essential and desired properties of:

- *programming languages* used for implementing the configurable system components,
- *configuration and change specifications* used for specifying the system configuration and its changes,
- the *operating system* on which the system is configured,
- the *validation process* used to verify the system,
- and *configuration management* used to manage the evolving configuration.

A detailed patient monitoring system is used to illustrate the dynamic configuration model.

[64] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

In [66], Kramer and Magee further develop dynamic change management based on the separation of structural concerns from component application considerations. They argue that this separation of concerns permits the formulation of general structural rules for changes at the configuration level without the need to consider application state, and it permits the specification of application component actions without prior knowledge of the actual structural changes to be introduced. In addition, the changes can be applied such that the modified system is left in a consistent state, and so that no disturbance is caused for the unaffected part of the operational system. The model is applied to an sample problem, *evolving philosophers*.

[65] Jeff Kramer. Configuration programming - a framework for the development of distributable systems. In *Proceedings of IEEE International Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, May 1990.

[66] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, SE-16(11):1293–1306, November 1990.

Concurrent with the work of Kramer and Magee, Schwan et al. develop conceptual models, algorithms, and implementation of system configuration for distributed and for parallel systems, specific emphasis on real-time systems[67, 68]. For distributed systems, programming primitives based on the object model of parallel software are developed[69, 71]. Using several distributed and parallel applications, these primitives' use for rapid prototyping and tuning of software is shown feasible[71, 68]. The programming system consists of a syntax-directed editor for an object-based language for describing configurable programs, a stub generator intergrated into code generation for the object-based language, a distributed monitoring system[27], and an adaptation controller[71] supervising static and dynamic program tuning on-line. Runtime support for the object model is constructed for a local area network of SUN workstations. Runtime system, monitoring, adaptation controller, and the runtime-present compiler support are integrated via an incore database able to represent and contain all information shared among tools and used for on-line program tuning[71]. Further work focussing on real-time systems is reviewed below.

[67] Karsten Schwan and Rajiv Ramnath. Adaptable Operating Software for Manufacturing Systems and Robots: A Computer Science Research Agenda. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, December 1984, pp. 255-262.

[68] Karsten Schwan, Thomas E. Bihari, and Ben Blake. Adaptive, Reliable Software for Distributed and Parallel, Real-Time Systems. In *Proceedings of the Sixth Symposium on Reliability in Distributed Software, Williamsburg, Virginia*, March 1987, pp. 32-44.

[69] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A System for Parallel Programming. In *Proceedings of the 9th International Conference on Software Engineering, Monterey, CA*, March 1987, pp. 270-282.

[70] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS – Kernel Support for Objects in the Real-Time Domain. *IEEE Transactions on Computers*, C-36(8):904-916, July 1987.

[71] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A Language and System for Parallel Programming. IEEE Transactions on Software Engineering, April 1988, 14(4):455-471.

[72] Brian Bershad, Edward Lazowska, Henry Levy, and David Wagner. An Open Environment for Building Parallel Programming Systems. In *Proceedings of the ACM SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems, SIGPLAN Notices, 23,9* , September 1988, pp. 1-9.

[73] Prabha Gopinath and Karsten Schwan. CHAOS: Why One Cannot Have Only An Operating System for Real-Time Applications. *SIGOPS Notices*, July 1989, pp. 106-125.

[74] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles"*, December 1989, pp 191-201.

[75] Prabha Gopinath, Rajiv Ramnath, and Karsten Schwan. Entity-Relationship Datase Support for Real-Time Applications. In *Proceedings of PARBASE-90*, 1990.

[76] M.L. Scott and T.J. LeBlanc and B.D. Marsh. Multi-Model Parallel Programming In Psyche In *2nd ACM SIGPLAN Symposium on Principles and Practice or Parallel Programming, SIGPLAN Notices, 25/3*, March 1990, pp. 70-78.

[77] Bodhisattwa Mukherjee and Karsten Schwan. A Survey of Multiprocessor Operating System Kernels. GIT-CC-92/05, College of Computing, Georgia Institute of Technology. January 1992.

[78] Prabha Gopinath, Tom Bihari, and Karsten Schwan. Operating System Constructs for Managing Real-Time Software Complexity. *Mission Critical Operating Systems, Studies in Computer and Communications Systems, Vol. 1.*, IOS Press, Netherlands, ISBN: 90 5199 069 3. March 1992.

[79] Ahmed Gheith and Karsten Schwan. CHAOS-Arc – Kernel Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Applications. *ACM Transactions on Computer Systems*, 11(1):33-72, April 1993.

[80] Prabha Gopinath, Peter Wiley, and Karsten Schwan. What Price Objects? Evaluating a Real-Time, Adaptable, Object-based System. *Proceedings of Sixth IEEE Workshop on Real-Time Operating Systems and Software, Pittsburgh, PA*, May 1989, pp. 29-34.

[81] Kaushik Ghosh, Bodhi Mukherjee, and Karsten Schwan. A Survey of Real-Time Operating Systems. *GIT-CC-93/18, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332*, February 1994.

In [82], Feeley et al. describes a node reconfiguration facility for *Amber*, an object-based parallel programming system for networks of multiprocessors. Amber allows an application to expand when new nodes become idle, and to contract when nodes become busy. What contrasts this work with others is the support of process migration at user-level, instead of at kernel-level. The support of logical node and large sparse virtual space makes node reconfiguration easy to implement. A node set can *shrink* in size, stay the same but *change* membership, or *grow*. The program expands by migrating objects and threads to a newly recreated logical node; the program contracts by forcing a logical node to terminate after migrating its objects and threads to other logical nodes.

[82] Michael J. Feeley, Brian N. Bershad, Jeffrey S. Chase, and Henry M. Levy. Dynamic node reconfiguration in a parallel-distributed environment. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, pp. 114–121, April 1991.

Schwan et al. [85, 83, 84] experiment with reconfigurable operating system components in order to improve program performance. The results from these experiments show that reconfigurable micro-kernels [85], configurable thread packages [83], adaptive locks, and adaptive objects and invocations [84] can be used to develop high-performance operating systems and applications for parallel and distributed programs.

In [83], a model and associated mechanisms are presented for implementation of kernel-level adaptive objects. Based on this model, adaptive locks are implemented and evaluated for parallel application programs. To improve application performance, adaptive locks change their waiting strategies to suit changing application locking patterns. Lock adaptation is specified by 'policies' associated with each lock. When changes are detected in the application locking pattern, a run-time mechanism determines whether to adapt the locking strategy and, effects the change, if it decides to do so.

[83] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pp. 59–66, July 1993.

[84] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. Ktk: Configurable objects and invocations. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, August 1993. (Position Paper).

In [85], Mukherjee and Schwan propose an architecture for a micro-kernel that can be reconfigured at both compile-time and run-time to suit an application's performance requirement. They argue that by matching operating system kernel functionalities with specific application characteristics and specific hardware, application performance can be significantly improved.

[85] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable micro-kernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pp. 45–60, September 1993.

Process migration is commonly defined as a transfer of a sufficient amount of a process' state from one machine to another for the process to execute on the target machine. Process migration is a straightforward example of steering an application by providing improved performance or reliability. For a review of process migration mechanisms and design issues, consult [86].

[86] J. M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, pp. 28-40, July 1988.

[87] Jason Gait. Scheduling and process migration in partitioned multiprocessors. *Journal of Parallel and Distributed Computing*, 8(3):274–279, March 1990.

[88] Fred Douglis and John Ousterhout. Transparent process migration. design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[89] David W. Glazer. On process migration and load balancing in time warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):318–328, March 1993.

Wheater, et al., [90] develop a reconfiguration facility to dynamically change the structure of an application in order to improve its performance. To improve performance, the system controls clustering of persistent objects of the application. Reconfiguration can range from no clustering (where all objects treated as independent) to maximum clustering (where all objects are grouped together).

[90] Stuart Wheater and Santoch Shrivastava. Exercising Application Specific Run-time Control Over Clustering of Objects. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pp. 25-35, March 1994.

Additional references on configurable operating systems employ compiler techniques for code reorganization in the Synthesis system[74], reorganize at the procedure level, and perform configuration at the level of program or operating system objects[72, 76]. Additional references on this topic appear in [77, 81].

5.2 Dynamic Program Tuning and Adaptation

The primary goal of dynamic program tuning and adaptation is to achieve high performance. For instance, real-time software may require dynamic adaptation to maintain required levels of performance (or timeliness) in face of changes in its operating environment. Similarly, for complex parallel application programs running on large-scale machines, dynamic program tuning may be necessary due to changes in input data or changes in the program's computational behavior on the parallel machine.

A prerequisite to dynamic program tuning and adaptation is program monitoring, where monitors can vary from simple hardware counters to sophisticated application-specific monitoring systems. Optional graphical displays present on-line program information visually to the user. An adaptation manager stores a user's adaptation specifications, processes run-time information from the monitor, and triggers and enacts the stated adaptations. In some systems, a user may be directly involved in tuning by sending adaptation commands to the adaptation manager through a user interface. A well-defined adaptation specification facility must be provided to let users describe the conditions under which specified adaptations should be performed.

In [91], LeBlanc and Markatos describe a real-time system that is adaptable in a real-world environment. Adaptability is achieved by allowing multiple real-time process models, with different properties and timing constraints, to be used in a single system and application. A prototype system system is built on a multiprocessor to control a behavioral system with vision and manipulation capabilities. A robot is described by *reflexive* tasks and adaptive *cognitive* tasks. Techniques such as user-level scheduling, user-defined process, and multiple communication models are used in the construction of the robotics applications.

[91] Thomas J. LeBlanc and Evangelos P. Markatos. Operating system support for adaptable real-time systems. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pp. 1–10, Charlottesville, VA, May 1990.

Bihari and Schwan [92] present a *RE*al-time *S*oftware *A*daptation *S*ystem (RESAS), its uniform model of adaptable software, and the tools necessary for programmers to implement adaptation algorithms. RESAS includes a *Programming Model*, a *Representation Framework*, and an *Adaptation Control System*. The Adaptation Control System is composed of a *Monitoring Mechanism*, a *Data Management System*, an *Adaptation Controller*, and an *Adaptation Enactment Mechanism*. The programming model and representation framework are used by the programmers to design and construct their application programs. All of the information during program construction, such as symbol tables, source code, etc., are stored in the data management system. During program execution, the monitoring mechanism retrieves information of the target program into the data management system. Adaptations are performed by manipulation of data in the data management system, which in turn trigger the adaptation enactment mechanism. As a result, adaptations are specified in terms of program-independent manipulations on the data model, thereby enabling programmers to formulate adaptations independently of details of program implementations.

[92] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

Bihari's work is extended and applied to real-time operating systems by Gopinath et al., including the CHAOS real-time operating system. In this system, the runtime representations of both objects and of object implementations are made configurable, resulting in an object model where operations are executed by multi-grain tasks ranging from procedures executed synchronously in the caller's address space, to single or multiple execution threads, which may be executed asynchronously and in parallel with the invoking task. Multi-grain tasks are complemented by multi-grain invocations, which range from reliable invocations that maintain parameters and return information (or even communication 'streams'[70]) to invocations that implement unreliable 'control signals' or 'pulses'[70]. Furthermore, invocation semantics can be varied by attachment of real-time attributes like delays and deadlines. Programming and tool integration support for runtime adaptation are described further in [73, 78, 75].

The construction of efficient adaptable objects is pursued further by Gheith who develops the notion of 'policies' associated with objects that intercept object invocation to make runtime decisions on invocation and object implementation[79]. In contrast to subcontracts in the Spring operating system[81], policies can accept and interpret runtime parameters, called 'attributes'. Attributes expose selected aspects of object and invocation implementations and therefore, act somewhat like meta-objects in object-oriented operating systems.

In [93], [94], and [95], Marzullo and Wood propose a *reactive system*, which is comprised of a *control program* interacting with an *environment*. The environment is instrumented with *sensors* and *actuators* that enable the control program to obtain information from and enforce control over the environment. While the focus of [93] and [95] is on fault-tolerance of such a reactive system, [94] describes a UNIX-based toolkit, called the *Meta* reactive system, its architecture, and its implementation. In the *Meta* system: (1) the programmer instruments the application and its runtime environment with sensors and actuators (some of them are provided by default), (2) an object-oriented data model is used to describe the application, and (3) the control program is based on that model (much like in RESAS). The object-oriented modeling language is called *Lomita*; it is used to describe the actions to be performed when certain application behaviors occurs.

[93] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. *ACM Operating Systems Review*, 25(1):45–48, January 1991.

[94] Keith Marzullo and Mark D. Wood. Tools for constructing distributed reactive systems. Technical Report TR 91-1193, Department of Computer Science, Cornell University, Ithaca, New York 14853, February 1991.

[95] Mark Dixon Wood. *Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture.* PhD thesis, Cornell University, January 1992.

Mills, et al. [96] introduces a language construct for parallel programming that constrains the relative rates of progress of the tasks executing in parallel. Logical clocks provide the necessary information to the scheduling system so that tasks lagging behind the group of executing processes are given priority. This construct is demonstrated, in part, with the real-time simulation in [102], and it is especially useful with load-balancing problems.

[96] Peter H. Mills, Jan F. Prins, John H. Reif. Rate-Control as a Language Construct for Parallel and Distributed Programming. In *Proceedings of The Workshop on Parallel and Distributed Real-Time Systems* (April 1993), pp. 66–70.

Kindberg, et al.[97] describes adaptively parallel computations under Equus. Reconfigurations occur dynamically during run-time. This reconfiguration-based expansion and contraction uses a communication system that programmers use to implement services as reconfigurable collections of server processes. Because the synchronization primitives for reconfiguration are embedded in the application code, the application processes are easily disconnected and reconnected without harm to the application's integrity. Clients do not require re-compilation when a reconfiguration occurs.

[97] T. Kindberg, A. V. Sahiner, and Y. Paker. Adaptive Parallelism under Equus. In *2nd International Workshop on Configurable Distributed Systems*, pp 172-183, March 1994.

6 Interactive Program Steering

As discussed earlier, *interactive program steering* implies that human users interpret program data and provide feedback to the program during execution. This section lists only those systems that allow users in the feedback process. Earlier sections on dynamic systems, also discuss feedback and adaptation; however, the feedback is usually the product of an algorithm. On-line program steering is a comparatively immature area of research that includes a variety of efforts addressing both program steering and interactive program visualization.

Tuchman, et al. [98] created the Vista system for simulation-time visualization of data. Vista provides a window into the application by showing program data automatically during execution. The system architecture is designed for a distributed or remotely executing application. The Vista model allows a trace file to replace the executing application, providing a visualization 'data browser' for data from past simulation runs. Data from the executing application are interactively selected and displayed.

[98] Allan Tuchman, David Jablonowski, and George Cybenko. A System for Remote Data Visualization. CSRD Report No. 1067. June 1991. University of Illinois Urbana-Champagne.

Program directing is investigated in [99]. Program directing is synonymous with program steering. Dynascope monitors a program, presents the data to a user or program, and allows for possible feedback actions. *Dynascope* provides basic monitoring and controlling in distributed environments. The system is integrated with existing programming tools and uses a few generic operating system and networking primitives.

[99] R. Sosič. Dynascope: A Tool for Program Directing. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 27(7):12-21, July 1992.

The VASE system [100] presents an abstraction for a steerable program and offers tools that create and manage collections of steerable codes. VASE annotates existing Fortran code to create a high-level model of the application; therefore, users do not have to work at the source code level. Software developers must annotate the existing code, however. Once the source code is annotated, VASE coordinates the execution of these codes in the distributed environment. VASE supports only the

SPMD model of parallel execution. A powerful 'C'-like scripting language provides flexibility for data selection and steering during execution. The SGI Iris Explorer renders output data visualizations.

[100] David Jablonowski, John Bruner, Brian Bliss, and Robert Haber. VASE: The Visualization and Application Steering Environment. In *Proceedings of Supercomputing 93*, pp. 560–569.

DYNA3D and AVS (Application Visualization System from AVS Inc.) are combined with customized interactive steering code to produce a time-accurate, unsteady finite-element simulation in [101]. Rudimentary steering is demonstrated in a distributed environment consisting of a supercomputer and multiple graphics workstations.

[101] David Kerlick and Eliabeth Kirby. Towards Interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations. In *Proceedings of Visualization 93*, October 1993, pp. 374-377

[102] describes challenges for a real-time visualization of a complex physical simulation. The goal of this real-time visualization is a virtual world where human users interact with the visualization in a 3D environment. The implementation spans a network of several specialized computer systems.

[102] Mark Parris, Carl Mueller, Jan Prins, Adam Duggan, Quan Zhou, Erik Erikson. A Distributed Implementation of an N-body Virtual World Simulation. In *Proceedings of The Workshop on Parallel and Distributed Real-Time Systems* (April 1993), pp. 66–70.

Eisenhauer, Gu, and Schwan et al. [103] explore interactive program steering with a molecular dynamics simulation (MD) program. The physical model of MD is a time-stepped simulation of the behavior of interacting particles. The parallelization of MD is achieved by domain decomposition applied to those particles. The steering target is to achieve load balancing among domains. On-line steering is accomplished by three steps: (1) the workload information concerning domains is monitored by an on-line monitoring tool called *Falcon*, (2) the monitored load information is processed and displayed to the user on-the-fly, (3) when the user finds that some threads perform significantly more computation than others, domain boundaries may be adjusted to shift particles among program threads.

[103] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing*, San Francisco, CA, August 1994.

[104] focuses on research leading toward a marriage of two areas: visual programming languages and steering. Visual programming languages are at least partially graphical and, hence, they provide one framework for interacting with scientific visualization and graphical steering tools. [104] also develops a taxonomy for interactive steering and visualization systems. Within this taxonomy, numerous systems are classified. System reviewed include AVS, Vista, VASE, SCENE, VPL, Khoros, Forms/3+, and ThingLab.

[104] Margaret Burnett, Richard Hossli, Tim Pulliam, Brian VanVoorst, and Xiaoyang Yang. Toward Visual Programming Languages for Steering Scientific Computations. *IEEE Computational Science & Engineering*, Winter 1994, pp. 44-62.

[105] presents SPI which provides a programming environment, based upon the event-action model, for developing instrumentation functions for parallel and distributed systems. This programming environment includes run-time support for distributed event-action execution, a library of standard actions, and tools for mapping/loading actions onto target system processors. This allows the SPI users to specify/develop real-time instrumentation for parallel applications that itself executes in parallel and can be configured to math the architecture of the (potentially heterogeneous) parallel platform and the user's application. A SPI prototype is currently being developed for the Intel Paragon.

ESL is the high-order language for expressing the event-to-action mapping used to instrument a system-under-study. ESL is a set of preprocessable extensions to a target programming lanuage such as C or Ada. Its syntax is flexible and may be adapted to the style of the target langauge. The ESL syntax and semantics are described in ESL Reference Manual [106].

[105] Devesh Bhatt. Scalable Parallel Instrumentation (SPI): an Environment for Developing Parallel System Instrumentation. *Proceedings of the Intel Supercomputer Users Group Conference (ISUG94)*, June 1994, pp. 98-104.

[106] Devesh Bhatt, Rashmi Bhatt, Rakesh Jha, Todd Steeves, and David Wills. Experiment Specification Language (ESL) Reference Manual. *Technical Report, Honeywell Technology Center, Minneapolis, MN*, August 1994.

Opportunities in monitoring and steering high performance parallel systems are explored in [107]. Advances in networking, visualization and parallel computing signal the end of the days of batch-mode processing for computationally intensive applications. The ability to control and interact with these applications in real-time offers both opportunities and challenges. [107] examines two computationally intensive scientific applications and discusses the ways in which more interactivity in their computations presents opportunities for gain. It briefly examines the requirements for systems trying to exploit these opportunities and discusses Falcon, a system that attempts to fulfill these requirements.

[107] Greg Eisenhauer, Weiming Gu, Thomas Kindler, Karsten Schwan, Dilma Silva, and Jeffrey Vetter. Opportunities and Tools for Highly Interactive Distributed and Parallel Computing. *Proceedings of The Workshop On Debugging and Tuning for Parallel Computing Systems. Chatham, MA.*, October 1994.

[108] describes Falcon, a system for on-line monitoring and steering of large-scale parallel programs. The purpose of such interactive steering is to improve its performance or to affect its execution behavior. The Falcon system is composed of an application-specific on-line monitoring system, an interactive steering mechanism, and a graphical display system. In this paper, we present a framework of the Falcon system, its implementation, and evaluation of the system performance. A complex sample application – a molecular dynamics simulation program (MD) – is used to motivate the research as well as to evaluate the performance of the Falcon system.

[108] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffrey Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia. February 1995.

Progress [109] is a toolkit for developing steerable application. Users instrument their applications with library calls and then steer parallel applications with Progress' runtime system. Progress provides steerable objects which encapsulate program abstractions for monitoring and steering during program execution. Once created, steering objects are know to and manipulated by Progress' runtime server. This toolkit provides sensors, probes, actuators, function hooks, complex actions, and synchronization points.

[109] Jeffrey Vetter and Karsten Schwan. Progress: a Toolkit for Interactive Program Steering. *Proceedings of the International Conference on Parallel Processing (ICPP95)*, August 1995, pp. 1-1.

7 Miscellaneous

The Zeus algorithm animation system is noteworthy in this bibliography because its design allows user feedback into the executing algorithm. Zeus provides multiple views of an algorithm, where each view is maintained by a separate thread of control. Each of Zeus' multiple views interprets user gestures and initiates feedback events to the algorithm. The algorithm updates the common data structures and sends output events to all views. Each view updates itself in response to the output events of the algorithm.

[110] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 4-9, Kobe Japan, October 1991.

Additional References

[111] Brad A. Myers. INCENSE: A system for displaying data structures. *Computer Graphics*, 17(3):113, July 1983.

[112] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Language and Systems*, 9(4):491–542, October 1987.

[113] Jeanne Ferrante, Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language and Systems*, 9(3):319–349, July 1987.

[114] B.H. McCormick, T.A. DeFanti, M.D. Brown (eds.). Visualization in Scientific Computing. *ACM SIGGRAPH Computer Graphics*, 21(6), November 1987.

[115] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and David Ogle. A language and system for the construction and timing of parallel programs. *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[116] John T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[117] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of Supercomputing '89*, pp. 627–636, November 1990.

[118] Michael Wolfe. Data dependence and program restructuring. *Journal of Supercomputing*, 4:321–344, 1990.

[119] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. Traceview: A trace visualization. *IEEE Software*, pp. 19–28, September 1991.

[120] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pp. 29–39, September 1991.

[121] Joan M. Francioni, Larry Albright, and Jay Alan Jackson. Debugging parallel programs using sound. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 68–75, Santa Cruz, California, May 20-21 1991.

[122] Tara M. Madhyastha. A portable system for data sonification. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.

[123] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, 1990.