Instrumentation in Parallel Program Interaction

Oral Comprehensive Examination Position Paper

Sameer Shende
Department of Computer and Information Science,
University of Oregon, Eugene, OR 97403
Abstract

Parallel programs have complex behaviors that are difficult to interpret. We look at techniques that help in building tools that present debugging and performance data in ways that are meaningful to the user. In doing this, we study the areas of instrumentation, performance profiling, tracing, event-based debugging, replay, checkpointing and visualization. We then look at issues of optimizations and how tools can deal with code transformation techniques and still present views of the data that represent abstractions that the user can benefit from. Several of the above areas have overlapping issues and looking at these areas together, we believe, gives a holistic view of the problems that need to be addressed for building parallel performance and debugging tools.
Chapter 1: Introduction

Understanding the behavior of parallel programs is a daunting task. Typically, it is undertaken with one of two goals in mind: debugging for correctness, or debugging for performance. In the case of correctness, the aim is to locate and correct logical errors; in the case of performance, the aim typically, is to locate and remove performance bottlenecks; the user needs to ascertain that instructions are executed in the correct order, at appropriate locations and the variables in the program are assigned correct values. To locate performance bottlenecks, the user needs to ascertain whether the amount of time spent executing certain code segments exceeds what may be reasonably expected. This requires understanding the behavior of the program in terms of its state and its performance characteristics. In either case, to understand the behavior of the parallel program, we must first make interesting aspects of its behavior observable.

To achieve observability, we regard the execution of a program as a sequence of events, each representing some significant activity such as the entry into a routine, the execution of a line of code, a message communication or a barrier synchronization. The execution of an event interrupts the program, stopping one or more of its processes and initiates one of three possible responses:

1. Attributes of the event are recorded, or
2. Statistics are collected.

After an event response is completed, control returns to the program. The response to an event, depends on its use; we describe three common uses here: interactive state-based debugging; event-based debugging and tracing; and performance profiling.

Common to all areas is the question of instrumentation or how we can capture information that we need to analyze. We discuss that in Chapter 2. Chapter 3 discusses the issues of debugging. Chapter 4 focuses on tracing and Chapter 5 discusses application of trace based techniques to event-based debugging, replay and checkpointing. Chapter 6 highlights the issues of profiling. Whether tracing or profiling, it is important to present data that the user can relate to easily. Scientific applications are often constructed using libraries, application frameworks and high-level parallel languages that try to “hide” the implementation details from the users. However, tools go to great lengths to peel through layers of abstraction to show precisely the details of the
implementation layers. This leads to confusion and frustration for the users. Chapter 7 focuses on techniques for profiling and tracing high-level data-parallel languages. Chapter 8 looks at future research directions. This paper ends with an unsolved problem: the problem of profiling in the presence of optimizations achieved with runtime asynchronous execution of parallel programs, and presenting users with meaningful, easy to understand performance data that can be correlated to high-level abstractions that the user is familiar with.
Chapter 2: Instrumentation

Instrumentation is the addition of instructions to a program to measure or record some aspect of its behavior. These additional instructions collect data from the running program in order to characterize the entire execution for purposes of debugging and performance analysis. Executing these additional instructions can perturb the application and introduce the “probe-effect”. Inspite of this, being able to observe the program behavior in terms of significant events can help the user gain valuable insights. In this chapter, we discuss the techniques of instrumentation employed in debugging and performance analysis. We see when and how events can be used to gain insight in the behavior of the parallel program.

2.1. What does the instrumentation do?
The execution of instrumentation instructions at runtime triggers events. The trigger can be one of the following actions.

- **When some point is reached during an execution.** For an event based debugger, it could involve the execution of an instruction that specifies an event in a high level language, or a synchronization operation such as the sending or receiving of a message. For performance profiling or tracing, an event could be triggered at routine entry or exit.

- **When some internal condition is satisfied.** For profiling based on sampling, an event could be triggered by an interrupt that occurs when some constraint is satisfied. This interrupt could arrive after an amount of time has elapsed, or when some system specific constraint is satisfied (overflow of a preset threshold for hardware performance counters). In debugging, when some exceptional condition arises or some pattern of events is matched, an event could be triggered as well.

- **When some external condition is satisfied.** An external trigger such as a signal sent by a debugger or a user could run a signal handler routine that records a caliper point.

The execution of instrumentation code may require the program to be modified with extra instructions for specialized purposes. Techniques for adding instrumentation in a program are discussed in the next section.

2.2. When is instrumentation added to the program?
Instrumentation can be added to the application process at any stage during the compilation/
As shown in Figure 1, moving down the stages from source code instrumentation to runtime instrumentation, the instrumentation mechanism changes from language-specific to platform-specific. At one extreme, making an instrumentation API specific to a language with source code instrumentation may make the instrumentation portable across multiple compilers and platforms that support the language. However, it won’t work with other languages. At the other extreme, targeting the instrumentation at the executable level ensures that...
applications written in any language would benefit from the instrumentation on the specific platform. However, it is more difficult to map performance data back to source-code entities, especially if the application’s high-level programming language (such as HPF or ZPL), is translated to an intermediate language (such as Fortran or C) during compilation. There are often trade-offs between portability, what level of abstraction the tool can provide, and how easily the instrumentation can be added.

At the highest level, instrumentation can be added manually to the source code using an API as is done in TAU[23], JEWEL [48] or Ariadne [8]. In TAU the focus is on identifying the segment of code that is currently executing and the instrumentation can be in the context of profiling, tracing or monitoring. In Ariadne the instrumentation identifies interprocess communication synchronization operations as well as user defined events. These events are then logged in a trace file by each process. In Jewel the instrumentation triggers events that are sent to a central instrumentation manager process that merges the event stream and traces them in event-logs. Adding annotations to the source code manually requires user intervention; often it can be aided by a preprocessor that can automatically insert annotations. This approach is used in SvPablo [24] and AIMS [40] for C and Fortran, by Sage preprocessor with TAU for HPC++ [75], or PDT[96] with TAU for C++. In SvPablo and AIMS, the instrumentation is inserted around code segments such as routines, statements and loops using Sage. Using Sage and TAU, the instrumentation is added as annotations of the abstract syntax tree (AST). Sage is a compiler restructuring toolkit where instrumentation is inserted in the form of objects in the AST. An unparse phase generates the source code from the AST after the preprocessor has expanded the macros and header files. Using PDT, a C++ front-end parses the source code, and generates an intermediate language (IL) file that represents the abstract syntax tree. Then an IL Analyzer [96] extracts a subset of AST entities and passes it to a library interface. DUCTAPE that provides an API to these entities. TAU instrumentor then identifies the location of classes, templates, routines from this database and inserts instrumentation macros in the source code by a source to source tranformation. This last approach is independent of any compiler or platform as it targets a source language for the instrumentation and the instrumented sources are not preprocessed.

Access to information is often tied with the stage at which the instrumentation is inserted in the program. For example, the advantage of source code instrumentation either at source code or
preprocessor level is that inlined functions can be profiled using statement level timers [36] at this stage. After a preprocessor expands an inlined function, it may not be available for inserting instrumentation once the compilation process is completed. The disadvantage of a source code instrumentation approach is language specific instrumentation. For a compiler to compile the modified source code, the added instrumentation needs to conform to the target language.

As shown in Figure 1 instrumentation can also be added by a modified compiler as is done with Cray MPP Apprentice for Fortran and C [33]. The compiler adds instrumentation code for performance profiling or tracing during the compilation after code optimizations have been performed. This approach taps into a wealth of information that is available during compilation such as data and loop dependencies. After compilation, object files are linked together by the linker to form the executable. Another place where instrumentation may be added is during linking. Using instrumented runtime libraries instead of original libraries allows the instrumentation to be inserted in the application flexibly. This approach is adopted by the Cedar tracing [47], IPS-2 [45] and a number of libraries that use the MPI Profiling interface [92] such as VampirTrace [60], PICL [32], TAU [23] and NUPSHOT [32]. When an inter-process communication call is executed, the instrumented library acts as a wrapper that calls the profiling and tracing routines before and after calling the corresponding uninstrumented library call. This is achieved by weak library bindings.

Often program sources and libraries are unavailable to the users and it is necessary to examine the performance of prebuilt binaries. Pixie [37] and ATOM [100] are tools that rewrites binary executable files by adding instrumentation for basic block counting. The modified executable generates profile data on execution. Speedshop [37] can also profile binaries by tapping into symbol table information to relate program counter and callstack information to source code constructs and uses sampling or pixie when the application is spawned using Speedshop. The advantage of binary rewriting is language independent instrumentation. However, this approach is platform specific and may not be portable.

Sometimes, re-execution of an application using the performance analysis tool is not a viable alternative while searching for performance bottlenecks. This is true for long-running tasks such as database servers. Runtime instrumentation is the last level where a running program may be instrumented to generate performance data. Paradyne [51] automates the search for bottlenecks in
an application at runtime by selectively inserting and altering instrumentation using the DynInst [27][46] dynamic instrumentation package. It allows us to combine the advantages of low data volume that are typically found in sampling with the accuracy of tracing. The instrumentation is used to precisely count and time events. The counters and timers are then periodically sampled and passed to the search model that identifies performance bottlenecks. DCPI [25] tool from Compaq (Digital) can profile long running programs using periodic sampling based on hardware performance counter overflow interrupts. It is a low overhead profiling facility targeted to profiling not just the application program, but kernel level routines as well for long running programs. Typically, the executable program can be run by the operating system. In the case of interpreted programs, an interpreter runs under the operating system and the interpreted code is executed by the interpreter. This is the model of execution for the Java programming language. TAU profiles and traces Java applications by loading the TAU shared object library in the context of the Java Virtual Machine (JVM). JVM then invokes TAU callback routines when interesting events, such as thread creation, loading of a class, method entry and method exit, take place. This way, there’s no need to modify the virtual machine, the bytecode or the Java source program to generate the performance data.

As we go down the layers from source code instrumentation to binary instrumentation, sometimes, information is lost (such as information regarding inlined functions in source code instrumentation, and information about loops and data dependencies in the modified compiler instrumentation) due to preprocessing or optimizations that result in restructuring of the original code. Also, as we go down the stages, we move from a language specific instrumentation mechanism to a platform specific instrumentation mechanism.

2.3. Conclusions
This chapter covered the issues of instrumentation regarding when the instrumentation is triggered and at what stage in the compilation/execution process this can be done. While data collection is a critical phase in any tool, effective presentation of this information is vital to the success of the tool. The tool needs to provide abstractions that are meaningful to the user and help in understanding the behavior of the parallel program, whether it is for debugging the program for correctness or debugging the program for performance problems.
Chapter 3: Debugging

3.1. Interactive State-based Debugging
When an event is triggered and it is sent to another process with other data, it could be used for debugging [1], steering [43][42][57], visualization of arrays [56] or visualization of performance data [35]. For state-based techniques for debugging, when a breakpoint is reached, a trigger (in the form of a signal) is sent to an external debugger process which takes control of the process. The user can then modify the dataflow or control flow of the debuggee task. For debugging parallel programs, this process is complicated by a number of problems such as scalability, asynchrony, non-determinism, non-transparency, consistent break-points and a large state space [8]. These problems are partially addressed by state-based debuggers targeted to specific high-level parallel languages such as Sneaky for pC++ [1] where barriers are used for breakpoints, kdb for uC++/COOL [2] where multi-threaded debugging is possible for user level threads, HDD [9] for message passing environments and various extensions of gdb for parallel debugging [59].

Interactive state-based debuggers allow users to examine the state of a system to an arbitrary level of detail and make it easier to relate the detected anomalies to source code errors. They rely on the break-examine-re-execute strategy, where the user sets a breakpoint at some location in the source code, stops the execution, inspects and possibly modifies the program state, and re-executes the program. Such a strategy is suitable and is commonly used for debugging sequential programs. The focus here is on the mechanism for stopping the program in a useable state. That is usually, a state that represents a consistent point in some execution that is meaningful to the user. State-based debugging techniques have limited success while debugging parallel programs.

3.2. What are the interesting research questions in this area?
The following questions summarize the interesting research problems posed by interactive state-based debugging:

- What is a consistent global state to stop the parallel execution? Extending the state-based debugging approach to parallel executions is ineffective because the program is composed of multiple loci of control. Do we stop each process independently or do we look at some global condition? Should these be stopped at a barrier or at any arbitrary code location? How can this be done? [99][8] discuss the problem of defining a consistent global state for parallel executions.

- How is such a global state specified? To apply the break-examine-reexecute strategy to parallel programs requires
the user to specify a global state to set a meaningful breakpoint in the source code. [22][8] discuss how such a state can be specified.

- How can parallel programs be made to re-execute in a reproducible manner? After specifying the breakpoint, the parallel program needs to reach such a consistent global breakpoint by re-executing the program. Replaying parallel programs is difficult due to their non-deterministic behavior [19][20].

- Are the breakpoints meaningful in the context of the high-level parallel language used for writing the program? Object-parallel and other high-level parallel languages encounter problems in specifying globally consistent breakpoints. The breakpoints may not be meaningful to the user in the context of the parallel language. [1] discusses this issue.

- Are classical debugging models applicable to object-parallel and other higher level languages? Do we need to come up with debugging abstractions that are meaningful in the context of parallel language? How can we build such abstractions? Sneaky [1], a state-based debugger addresses this issue.

The same techniques and issues are applicable in the area of computational steering as well.

3.3. Why interactive state-based debugging fails for parallel programs?
State-based techniques allow the user to examine the state of the execution to an arbitrary level of detail in terms of the source code constructs. However, state-based techniques do not address the following issues that are discussed in depth in [8]:

1. **Consistent global state.** In parallel computations, it is difficult to specify and detect a consistent global state that is meaningful to the user. Enumerating a set of local states of multiple loci of control does not generate a meaningful global state because the behavior of a parallel program is determined by the inter-dependence among the different processes that participate in the parallel computation. A meaningful global state is one that can occur in a computation. Specifying such a state is difficult; it is even harder to stop the computation at such a consistent state once it is specified.

2. **Asynchronous execution.** When a globally synchronized real-time clock is not used, processes in a distributed computing environment do not execute in lock-step. Events that take place on different processes may appear to interleave in unexpected ways to an external observer. The correctness of the program depends on ordering of events imposed by inter-process data dependencies that are described by the Lamport’s happened-before relationship [11]. By ignoring the temporal relationships between events that take place on different processes, it is difficult for a state-based debugger to filter out perturbations due to the asynchronous behavior.
of parallel programs.

3. **Large state space.** Multiple loci of control produce a large state space. State based techniques present this huge state space to the user who is overwhelmed with information. State based approaches do not allow the user to filter out the low level details of the program by concentrating on the salient features of the behavior. They do not provide any basis for specification of the behavior the parallel program.

4. **Non-determinism.** In the traditional break-examine-repeat approach, that is employed in sequential debugging, a user executes the program repeatedly, sets breakpoints in the code, stops the execution to examine the state of computation at a breakpoint. This approach cannot be directly applied to debugging parallel computations due to their non-deterministic nature. We discuss this problem of irreproducible executions in the next section in greater detail.

5. **Scalability.** State-based debuggers find it difficult to visualize debugging data from massively parallel programs where the number of threads involved in the computation exceeds the pixel resolution of standard displays [62].

For all these reasons, state-based debuggers are not sufficient when dealing with parallel programs.
Chapter 4: Tracing

Tracing is the activity of capturing an event to observe actions that take place in the program. It normally but not always involves generation of a log of the events that characterize the execution. Event tracing is commonly employed in debugging and performance analysis. An event trace is a set of event records. An event is typically an ordered tuple that consists of the event identifier, the timestamp when the event occurred, where it occurred (the location could be specified by the node and thread identifiers) and an optional field of event specific information. In addition to the event-trace, a table that maps the event identifier to an event name and its characteristics (the event class, the meaning of the optional parameter, etc.) may be maintained separately or be part of the same trace as records defining an event [24]. Tracing involves storing data associated with an event in a buffer and periodically writing the buffer to stable storage. Issues include event logging, timestamping, compensation for perturbation, trace output, or how the traces are stored, and post-processing of traces to extract information about program behavior.

4.1. What are event timestamps?
The timestamp used in the event could be based on logical time or physical time. The logical time could be a counter that is incremented locally in a process when an event takes place. It is useful in establishing temporal relationships between tasks that do not execute in lock-step, based on Lamport’s happened before relationship and is applied in debugging [8][11][12] of parallel programs. The physical time could be the reference of time obtained from a globally synchronized real-time clock [47].

4.2. What about perturbation?
Event tracing can perturb the application and even change the relative ordering of events. Any kind of instrumentation can have such a probe-effect that was introduced in the first chapter, but in the case of tracing, the volume of data generated is quite large and can slow down the application. Malony showed that perturbation analysis can be done in two phases. Firstly, given the measured costs of instrumentation, the trace event times can be adjusted to remove these perturbations. Secondly, given instrumentation perturbations that can reorder trace events, the event sequences need to be adjusted based on knowledge of event dependencies, maintaining causality. While analyzing the traces, some systems such as AIMS [40][39] and [34] can use such a scheme to compensate for the intrusion caused by the instrumentation and ensure that the traced event
orderings are preserved while removing the perturbation (to ensure that a message is not received before the corresponding send operation).

4.3. Where are event logs stored?
Trace buffer allocation involves ordering and merging the logged events. This can be done using a shared or private trace buffer. On shared memory multiprocessors, multiple threads can access a shared trace buffer using mutual exclusion (using locks) and appending the trace record. The advantage of this approach is the implicit ordering and merging of trace records and maximizing the buffer memory utilization. The other approach involves a private trace buffer and is used in both shared memory and distributed memory systems. Since the buffer is private, there is no contention for the buffer, but the trace buffer memory is not optimally utilized due to varying rates of event generation in each task. The trace buffer needs to be periodically flushed to disk and this can be done by either each task locally or it can be done by an external trace collector task that shares the trace buffer with the rest of the tasks. The trace buffer can be in the form of a dequeue or a circular buffer (ring). In TAU [36] a private trace buffer in the form of double ended queue is used without a trace collector task. In BRISK [49], a shared circular trace buffer is used with a collector task. In Cedar’s tracing system [47], a user can select between a statically allocated fixed size trace buffer or dynamically allocated trace buffers using linked buffer blocks, and the user can choose where the trace buffers will be stored in the Cedar’s memory hierarchy. Also, runtime trace I/O can be selected that causes the trace collection task to run concurrently with the program task and the trace I/O can be sent to a file or to a remote computer over a network.

The trace data can be stored to a file when the trace buffer overflows as in TAU [36] or when a barrier is reached as in the Split-C tracing system [26] and when the program terminates.

4.4. What is done with traced output?
After the events are logged, these traces often need some form of post-processing. This could take the form of merging event traces from multiple tasks, and/or conversion to another trace format. Tracing the execution of a parallel program can be used for debugging, online-monitoring, replaying and checkpointing as well.
Chapter 5: Event-based debugging

An event-based debugger [8] allows the user to model the behavior of parallel programs in terms of salient events and to match the intended behavior to the actual behavior as captured in execution traces. When the behaviors do not match, it can relate the cause of the mismatch back to events in the source code and allows the user to execute functional queries, to detect the cause of the anomalous behavior, in a domain-specific programming language. Event-based debugging focuses on reaching some control point in the program that indicates that some event has happened. The visible attributes of the event may be less important and may be fewer (the size of a message may be the attribute associated with a message send operation that is an event).

5.1. What are the interesting research problems in this area?
The following questions summarize the interesting research problems posed by event-based debugging:

• How do we define the set of events that we need, to effectively characterize the parallel execution? Should the user be exposed to the internal workings of the language and runtime-system or should he/she be shielded from this level of detail? The different alternatives include using low-level runtime system events [8], or application specific events, meaningful in the context of the high-level parallel language, or using a combination of the two [1]?

• How do we deal with the size of the information collected and what techniques can we use to filter this to represent some meaningful aspects of the execution? Event-based techniques generate a huge volume of trace data. Filtering this information requires the debugger to match patterns of intended program behavior with the logged events. These patterns or abstract events may be specified at a local process level, at a process level that involves matching local patterns on a set of processes or at a global level that involves matching the patterns on one or more processes over time, to capture the temporal aspect of these abstract events. [54] discusses a three tier hierarchy to specify such patterns to filter the event-data.

• How do we specify the events in a high-level language that is meaningful to the user during analysis? What events can effectively capture the effect of actions in the high-level language? [1] discusses how these events may be specified by the user in the context of a high-level object parallel language.

• Is event-based debugging alone sufficient as a debugging strategy or is a combination of state- and event-based debugging approaches more relevant in the context of parallel programs? Using either of the two debugging strategies in isolation rarely fulfills the debugging requirements of parallel programs. [8] discusses the effectiveness of a comprehensive debugging strategy using both event- and state-based approaches.
• How do we replay the execution to reproduce the bugs? Non-determinism that characterizes shared memory and message passing parallel programs creates problems when a combination of event and state-based approaches is used. Event-based approaches can specify a globally consistent breakpoint and state-based approaches can effectively examine the state space when this breakpoint is reached. However, reaching the breakpoint requires the underlying system to guarantee that parallel executions can be replayed successfully. It also raises questions about reducing the overhead associated with replay and how optimal tracing strategies could be employed to minimize this overhead. [17][19][4][3][10][19][20] discuss the issue of replay in parallel programs.

5.2. Event-based pattern matching
Event-based behavioral abstraction allows the user to describe the behavior of the program at a higher level of abstraction. This behavior is then matched with the actual execution, as captured in event-traces to locate bugs. Event-based debuggers differ in the flexibility they provide in modeling abstract events, in their respective event specification language, how they model concurrency, in terms of partial and total order modeling constraints, and the level of feedback on a mismatch. Here we discuss four event-based modeling approaches, EDL, DPE, PEDL and Ariadne and compare the choices made by each.

EDL or the event description language, developed as part of the event-based behavioral abstraction (EBBA) toolkit [105], employs a string representation of program behavior. Temporal constraints on events are then described using a regular expression based syntax. It employs filters on event attributes to specify constraints. Using a string representation for matching abstract events has its limitations. It cannot distinguish between partial order (graph representations based on Lamport’s happened before relation) and total order effectively [8]. It models parallelism using a shuffle operator which matches concurrent two concurrent events by accepting them in either order. This concurrency semantics is weak and cannot distinguish between two possible matches of the model “W parallel with R” where W and R are unrelated and when they are the write and read events of the same message. For modeling such abstract events, EDL’s notion of concurrency, based on the interleaving semantics, requires the user to specify constraints based on process identifiers, which makes the model non-scalable. It requires specification of $O(n^2)$ constraints on process identifiers for modeling true concurrency among $n$ events. The models need to be changed when the number of processors changes, as concurrency between abstract events can only be specified using the process identifiers. Another limitation of EDL is in imprecise feedback on a mismatch of the model. It employs pattern extraction approach where
events can be skipped, as opposed to the parsing approach, which requires the model to match all events and flag an error at the first mismatch. A parsing approach can detect a mismatch accurately, whereas a pattern extraction approach cannot do so, because it allows multiple finite state machines to be simultaneously match the input string. So, EDL suffers from limitations in modeling concurrent events and providing precise feedback on an error.

DPE, or data path expressions extends the notion of path expressions by allowing data events to appear in the path expression. It is implemented in the Meld debugger [106]. It can model parallelism more effectively than EDL, as data path expressions operate on the partial order graph of the execution trace using Lamport’s happened before relationship. However, it requires that all event relationships be known at compile time and its implementation can be exponential in the size of the regular expression used for matching the abstract event. Another limitation of DPE is that it defines a precedes relationship between abstract event which is inflexible. Two abstract events are said to be sequential if and only if all primitive events that make up the first abstract event preceed all primitive events in the second. This imposes a total order among events which makes it difficult to apply to MIMD computations.

PEDL or parallel event definition language introduced by Al Hough [107] overcomes some limitations of EDL and DPE. It uses Lamport’s happened before relationship to model communication events and defines a flexible precedes relationship between abstract events that allows only part of one abstract event to preceed part of the other. However, it cannot distinguish between a happened before relationship imposed by program order (on the same process) from one imposed by inter process communication. Also, PEDL uses a parallel closure operator to model concurrency which is ill-defined and cannot provide a meaningful semantics in the presence of an error [8].

Ariadne [8] builds upon the shortcomings of previous event-modeling approaches and provides unambiguous semantics. It uses Lamport’s happened before relationship to filter the effects of asynchrony and defines a flexible precedes relationship similar to the one employed by PEDL. It provides precise feedback on errors using a two-phase matching algorithm. It can detect whether an abstract event is missing or if the event is present but the required temporal relationships with other abstract events is missing. It does this by first matching behaviors at the process level, and then asserting temporal relationships between abstract events that hold on the process level. This
avoids building of product automation and is efficient to implement.

5.3. **Event and State-based techniques**

Event-based debugging provides a viable alternative to debugging large-scale parallel applications. Here, patterns of inter-process interactions are examined for anomalous behaviors using event-based modeling. This guides the user to examine the local states by stopping the computation at a consistent global state that is specified in terms of abstract events. This is done by comparing the expected program behavior with the actual program behavior, as captured in execution traces. Using an event-abstraction mechanism, event-based techniques are better equipped to handle the large amount of trace data generated by multiple loci of control by filtering the event data.

A combination of event and state based approaches provides a comprehensive debugging solution that has been demonstrated in Ariadne [8]. Event-based debugging uses logical time, instead of physical time, to filter the effects of asynchrony. It uses a replay mechanism to capture and contain the non-determinism present in the parallel programs. To address the concerns of scalability of visualization of debugging data, AVE [54], or the Ariadne Visualization Engine provides an interface for the event-based debugger, Ariadne [8] for a scalable hierarchical visualization of abstract events. Ariadne is used to match models of intended program behavior with actual program behavior, as captured in event traces. It combines a simple modeling language with functional queries. AVE shows the structure of user-defined behavioral models by providing visualizations of the match tree as shown in Figure 2. Program behavior is characterized in terms of significant events which are described in the modeling language in terms of a three level hierarchy of chains, p-chains and pt-chains. Chains are patterns that represent local views of events that take place on a single process. P-chains extend this abstraction to concurrent execution of the chain by one or more processes. Pt-chains are “patterns that represent the logical, temporal composition of a set of p-chains and they are matched in two steps: first the individual p-chains are matched and then the asserted temporal relationships between them are verified. When a pt-chain is matched or partially matched, its events are returned as a match tree in which internal nodes represent user-defined abstractions and leaves represent primitive events. The levels of the match tree correspond to the hierarchical levels of the model definition with the root corresponding to the outermost level of the pt-chain. AVE automatically compresses trees for visualization. The compression occurs both horizontally and vertically: horizontally by eliding
sibling nodes and vertically by pruning levels. Suppressed sibling nodes are represented by a single node with buttons that can be used for scrolling through hidden constituents; an integer label gives the number of these constituents. Vertical compression is indicated by a summarizing box labeled with an integer pair: the first component indicates the total number of behaviors and the second component indicates the number of different behaviors” [54]. The pop-up window shown below the match tree gives a further description of the summarizing box showing the matched pattern and the set of process identifiers of the processes that participated in the matched behavior.

Figure 2. AVE shows the match tree for a pC++ Binary Image Compression program [TAU-Ariadne].

The issues of instrumentation that are relevant to this mixed state and event debugging approach involve:
1. The generation of event-traces, which are used in the analysis of events to understand the behavior of the parallel program for setting of a globally consistent breakpoint,
2. The generation of replay traces annotated with the breakpoint information, and
3. The use of a replay facility to stop the parallel program at such a breakpoint.

The source code is annotated with instrumentation that generates event traces when the program executes. These traces are then read by Ariadne and are used in event based modeling. After the user sets a globally consistent breakpoint, Ariadne writes the trace data back and annotates the events with breakpoint information that specifies when the computation is stopped on the process; before or after the execution of the specially marked event in each process during re-execution. A replay facility ensures that effects of non-determinism can be contained and the program executes the events in same order as it did during the initial generation of event-traces. We discuss the Replay facility next.

5.4. Capturing events enables replay and checkpointing
Replay involves generation of an event trace that captures events that could potentially lead to non-deterministic behavior and on a subsequent run, enforcement of the same event orderings during the re-execution phase. During program execution, the resolution of non-deterministic choices is logged in the form of an event trace, and this log is consulted during re-execution to ensure that it follows the same state transitions as before. The size of the traces can be significantly large and so techniques to reduce the trace size are important.

5.4.1 Tracing for Deterministic Replay
For the purpose of controlling non-determinism that stems from parallelism, Netzer’s work has contributed significantly to reducing the size of traces for capturing information related to race conditions. Two messages race with each other when either could arrive at a receive operation first. It is not necessary to trace the order of delivery of every message communication but [20][21] showed that it is sufficient to trace only racing messages. Similarly, a data race [3] occurs in shared memory parallel programs when the order in which any two accesses to a shared variable (where at least one modifies it), or the order in which any two synchronization operations execute is not guaranteed by the program’s synchronization. [3] showed that locating every data race condition is an NP-hard problem but tracing only a subset of all races is sufficient for debugging. Tracing non-transitive dynamic data dependencies is sufficient to guarantee a
successful replay [19], however [10] shows how adding artificial dependencies among events can reduce the trace size without slowing the program (by guaranteeing that these additions are not on the critical path).

### 5.4.2 Re-execution phase
When an event log is read for replay, we try to enforce the same behavior in the program as was recorded during a previous run. Parallel scientific applications are often composed of layers. Typically, these are shown in the Figure 3 below. Replay instrumentation can be introduced at any one of these layers and various tools do this.

Chorus distributed operating system provides a mechanism for deterministic replay [18]. Chorus implements a software instruction counter that counts the number of instructions executed by a thread for each context switch operation. The scheduling of threads is done based on this
information during replay to guarantee the same order of accesses to the shared memory regions. Also, the interaction with the operating system (syscalls) is recorded and is reproduced in the same manner during replay. This provides a viable solution for replay of both shared memory and distributed memory message passing parallel programs running on multiprocessors.

For uniprocessor shared memory applications, [15] shows how multi-threaded applications running on uniprocessors can be replayed in the Mach operating system. Their scheme uses a software instruction counter that is used to record the number of instructions executed between events and controlling how many instructions are executed before an event is recreated with respect to asynchronous events.

At the next level, message passing programs can be instrumented for deterministic replay by linking with the modified libraries. The Annai programming environment demonstrates how this can be done [14] in the context of the MPI [92] library while the Pangaea [58] tool demonstrates this with the PVM library. In MPI, race conditions can exist due to blocking receives and probes (synchronous and asynchronous). The Annai replay engine uses vector timestamps of logical time and uses optimal tracing techniques [20] to capture non-determinism. Their extension to the algorithm records when a non-blocking probe operation is successful and guarantees that non-blocking probes return the same result during replay as they did during the tracing phase based on an internal counter. However, schemes for instrumenting a specific message passing library (PVM or MPI) that work on a target platform are insufficient in providing portable replay facilities for a language. Instant Replay [13] shows how to instrument for deterministic replay by recording the relative order of events and defines protocols for accessing shared objects independent of interprocess communication primitives. Optimal tracing algorithms [19][10][20] improve on this.

To provide replay facilities at the runtime system level independent of the underlying interprocess communication paradigm allows languages that are built atop these runtime systems to provide a seamless integration of replay for applications written for such languages. [17] shows how this is implemented for the Tulip runtime system [88] used in the pC++ [89] and HPC++ [90][91] object parallel languages. Replay schemes proposed for shared memory parallel programs usually
require the distinction between a read and a write operation for shared objects. This information
could be obtained by parsing the program [19] but is not typically available at the runtime system
layer. [17] implements an element access scheme that keeps track of the order of accesses and
during replay, forces a thread to poll till its local access count matches the traced value. For
message passing systems, the logical time in terms of unsuccessful probe counts [1] is used to
ensure that the logged value matches the current value in determining if a given probe is to return
with a success or a failure. If a match occurs, a non-blocking probe could be invoked to
successfully reproduce the behavior of the original program. In case it doesn’t match, the probe
can return with a failure, incrementing its probe count.

5.5. Checkpointing
During a checkpoint, the entire state of the program is logged onto a stable storage. It is
impractical to replay the entire execution of a long running program to reach a consistent
breakpoint while debugging. Incremental replay [4][7] allows the state of the program to be
recorded periodically, while the tracing the content of a subset of messages. In addition, the order
of messages needs to be traced and as discussed earlier, there are optimal tracing algorithms
[10][19][20] that can be used for this. Checkpointing is useful in debugging, as it allows the
program to be re-executed from a checkpoint instead of starting it from the beginning. When
replaying, it is important to ensure that it receives the same messages during replay as the original
execution did. A globally consistent checkpoint can be obtained by using a “co-ordinated
checkpoint” scheme where each process checkpoints simultaneously based on a central
coordinator task or by using an “independent checkpoint” scheme where each task decides when
to take a local checkpoint independently and selectively traces those messages that cannot be
reproduced quickly. [6] describes technique for efficient logging of messages for incremental
replay for independent checkpointing that avoid problems associated with earlier work (domino-
free algorithms).

5.6. Conclusions
We discuss how capturing event traces is useful for event-based debugging, replaying and
checkpointing. We look at different event-based debuggers and how they address issues of
modeling events, concurrency, and feedback on errors, integration of event- and state-based
debugging approaches and issues of scalability. We discussed how the problems of non-
determinism is addressed by replay, how the problem of asynchrony is addressed by using logical
time and using globally synchronized real-time clocks, how the problem of a lack of a consistent global state is addressed by event-based debugging by using causality constraints in setting a globally consistent breakpoint.
Chapter 6: Profiling

Profiling shows the summary statistics of performance metrics that characterize the performance of an application. The statistics are maintained at runtime as the process executes and are stored in profile data files when the program terminates. Examples of metrics include: the CPU time associated with a routine; the count of the secondary data cache misses associated with a group of statements, and the number of times a routine executes. Techniques for adding instrumentation needed to collect these values were discussed in Chapter 2. Here, we discuss general profiling issues: what performance metrics can we measure and how can we use them to effectively characterize the performance of a parallel program. We then discuss some specific issues in profiling object-parallel computations.

6.1. General Profiling issues
In this section, we examine what can trigger profiling instrumentation, what quantities can be measured once it is triggered, what performance metrics can be computed on these measurements and how the resulting data can be presented.

6.1.1 What triggers profiling instrumentation?

As shown in Figure 4, the two main approaches to profiling are interrupt-based sampling and location-based measurement.

6.1.1.1 Sampling

In sampling, a hardware interval timer periodically interrupts the execution. Timers are the most
common means of generating the interrupt that triggers the profiling instrumentation as seen in well-known tools such as prof [101] and gprof [102]. When an interrupt occurs, the state of the program counter (PC) is sampled and a histogram that represents the frequency distribution of PC samples is maintained. The program’s image file and symbol table are used post-mortem to calculate the time spent in each routine as the number of samples in a routine’s code range times the sampling period. Instead of the PC, the callstack of routines can be sampled too. In this case, a table that maintains the time spent exclusively and inclusively (of called child routines) for each routine, is updated, when an interrupt occurs. The **inclusive** time is the total value of the time spent in the given routine (block) summed over one or more invocations, and of any routines called by it, whereas the **exclusive** time represents the total value of the time spent executing the given routine and does not include in it the contributions from other blocks that are called by it.

When the callstack of routines is sampled, the exclusive time of the currently executing routine and the inclusive time of the other routines that are on the callstack are incremented by the inter-interrupt time interval. Additionally, in gprof, each time a parent calls a child function, a counter for that parent-child pair is incremented. Gprof then shows the sorted list of functions and their call-graph descendents. Below each function entry are its call-graph descendents, showing how their times are propagated to it.

When time-based sampling is used, we can regulate the slowdown (perturbation) that occurs in the application. The timing interval typically varies from 1 to 30 milliseconds. As the processor speeds continue to increase, using a static sampling interval could lead to sampling errors and produce inaccurate profiles. In this case, sampling based hardware counters may be more accurate.

Most modern CPUs provide on-chip hardware performance counters that can record several quantities such as the number of instructions issued, floating point operations performed, number of secondary and primary data and instruction cache misses, etc. and often the operating system provides library calls [41] to read and reset these counters. Most CPUs provide two or three registers that can be used for the measurements (implying two or three counters can be simultaneously measured) and provide some mechanism for multiplexing these counters. Tools such as Intel’s VTune [28], SGI’s SpeedShop [37] and Digital’s DCPI [25] use interrupts based on the number of instructions issued, which is independent of processor speed. SpeedShop also
provides interrupts based on other quantities that can be measured by the R10000 processor hardware performance counters (such as generation of an interrupt after a given number of secondary data cache misses). It provides two preset experiment types for each of the counters that use prime numbers for the elapsed counts and they claim that if two profiles of the application that use different prime numbers are similar, then the experiment is statistically valid.

When an interrupt is generated, either the program counter (and thus the currently executing routine) is sampled or the callstack is sampled. When a program counter is sampled, the program counter translates to the currently executing block of code and the number of samples associated with that block of code is incremented. Based on the total number of samples recorded, the interval between the interrupts and the number of samples recorded when a particular code segment is executing, statistical techniques are employed to generate an estimate of the relative distribution of a quantity over the entire program. This approach is used in Speedshop [37]. When the callstack is sampled, typically, the block of executing code (at one end of the callstack) has its exclusive time (quantity) incremented based on the interval associated with the interrupt. The other blocks of code have their inclusive time (quantity) incremented by traversing the callstack.

Besides time, other quantities can be measured as well, as shown in Figure 5. DCPI, for example, can produce “an accurate accounting, down to the level of individual instructions, of where time is being spent. When instructions incur stalls, the tools identify possible reasons, such as cache misses, branch mispredictions, and functional unit contention”. Based on the number of samples taken in a given block, the time spent in the block is estimated using the sampling frequency and the processor clock frequency using a processor model [25].

6.1.1.2 Measurement
In this approach, profiling instrumentation is triggered at block entry and exit. A block could be a routine or a group of statements (segment of code) that has start and stop triggers for the instrumentation. A basic block is defined as a sequence of instructions that is entered only at the beginning and exits only at the end. When the instrumentation is triggered, a timestamp is recorded, and profiling statistics are updated. As with the earlier approach, exclusive and inclusive times or counters are measured for each block of code. TAU [36] implements statement level timers for profiling a segment of code. Pixie [37][38] shows ideal time based on counting the
execution of each basic block. This ideal time represents a “best-case” execution; actual execution may take longer as Pixie includes predicted stalls within a basic block, but not actual stalls that may occur entering a basic block. Pixie also assumes that all instructions and data are in cache and excludes the delays due to cache misses and memory fetches and stores.

6.1.2 What quantities can be measured?

As shown in Figure 5, the quantity that is measured could be time and/or one or more of hardware performance counters that keep track of CPU activity. When the measured quantity is time, it could be the wallclock time, process virtual time, or user time (same as the CPU time). The wallclock time is the total time a program takes to execute including the time when it is waiting for resources. It is the time measured from a real-time clock. The process virtual time is the time spent when the process is actually running. It does not include the time spent when the process is swapped out waiting for CPU or other resources and it does not include the time spent on behalf of the operating system (for executing a system call, for instance). The user time includes both the time the program is running (process virtual time) and the time the system is providing services for it (such as executing system calls).

As shown in Figure 5, the quantity that is measured could be time and/or one or more of hardware performance counters that keep track of CPU activity. When the measured quantity is time, it could be the wallclock time, process virtual time, or user time (same as the CPU time). The wallclock time is the total time a program takes to execute including the time when it is waiting for resources. It is the time measured from a real-time clock. The process virtual time is the time spent when the process is actually running. It does not include the time spent when the process is swapped out waiting for CPU or other resources and it does not include the time spent on behalf of the operating system (for executing a system call, for instance). The user time includes both the time the program is running (process virtual time) and the time the system is providing services for it (such as executing system calls).
6.1.3 What performance metrics can be computed during profiling

Measurement based profiling that triggers instrumentation at block entry and exit, allows for accurate measurement of the exclusive and inclusive quantities for the block. In addition, data for computation of other statistics (such as standard deviation) could be recorded too. Figure 6 shows what metrics we can compute during profiling. Other quantities that could be stored include the count of the number of times a block executes (numcalls), the count of the number of blocks that it calls (numsubrs), and other callstack related information. This approach is used in TAU as shown in the pprof tool [36] in Figure 7. Pprof provides a character based display of the profile and allows the user to sort the profile data using different keys. TAU can also keep track of user defined events.

6.1.4 User Defined Events
Besides timing based instrumentation that is triggered at block entry/exit, TAU provides profiling of user defined events. A user defined event has a name associated with it and is triggered by the execution of some statement that associates a data value with the event identifier. This could be used to record any user defined entity such as the error in an iteration or the size of a message. The data is then recorded and aggregate statistics of the variation of the data are kept at runtime. TAU provides standard deviation, maxima, minima, number of samples, sum, and mean values of the event. Figure 8 shows the use of profiling user defined events in tracking memory allocations used in the PaRP project [73].
Figure 7. pprof sorts and displays profiles

Figure 8. Profiling user defined events in TAU
6.1.5 What techniques are employed to present performance metrics?
Parallel programs produce vast quantities of multi-dimensional data. Representing this data without overwhelming the user with unnecessary detail is critical to the tools’ success. It can be represented effectively using a variety of visualization techniques [63] such as scatter plots, Kiviat diagrams, histograms, Gantt charts, interaction matrices, pie charts, execution graphs and tree displays. ParaGraph [61][53] is a trace visualization tool that incorporates the above techniques and has inspired performance views in several other tools. Pablo [24] provides a user directed-analysis of performance data visualization by providing a set of performance data transformation modules that are interconnected graphically to create an acyclic data analysis graph. Performance data flows through this graph and metrics can be computed by different user-defined modules.

There are a number of visualization techniques that have been employed with varying degrees of success to provide insights into program behavior. Unless tools can visualize data in ways that are meaningful to the user, and are consistent with the user’s mental model of abstractions, their success will be limited. As the complexity of software increases and scientific and engineering applications are composed of layered components, the “semantic-gap” that exists between higher-level components visible to the user, and lower level entities that relate to implementation layers will continue to increase. For the visual representations of the parallel programs to be truly meaningful, the underlying data represented must be meaningful as well.

6.2. Higher-level features
Some tools provide features for profiling object-parallel languages. This section talks about the problems involved in profiling object-oriented computations.

6.2.1 Function Identification Problem
Profiling object oriented languages such as C++ using source code instrumentation techniques highlights the problem of unique identification of functions and templates. In this section, we see how it was solved in TAU.

The first approach considered was to associate a unique integer with each function. This worked well in the case of a parallel program restricted to one source file where contiguous function numbers would make it possible to use efficient, simple array indexing to access profiling statistics for a given function. With multiple source files, however, the function table must allocate
a range of numbers for each source file, creating fragmentation, and posing problems when functions are added or removed from files. Contiguous function numbers could be maintained by TAU [31] which has a program database for each source file, but recompilation of all sources would still be necessary when functions are added or removed. A more serious drawback to the assigned integer approach, is the problem of template instantiation. Each instantiation must be identified. Since template instantiation occurs in a prelinking phase, this requires tighter integration with the compiler (for unparsing the profiled sources and supporting the prelinking phase) than their portability concerns allowed.

The second approach was to use the address of each function as its identifier. This eliminates problems with multiple source files (as each template instantiation has a unique address in the executable) but it incurs higher run-time overhead for hash table/map lookup on function invocation. It also requires access to the symbol table in order to map function and template addresses to names, something compilers generally provide only in debug (nonoptimized) mode. Another serious limitation is that it would not allow access to the address of constructors and destructors of a class as specified in ANSI C++, which would preclude supporting the complete C++ language.

The third alternative considered was to use the function names themselves, solving the problem of identifying constructors and destructors of each class, as well as the problem of uniquely identifying a function across multiple files. Template instantiations can be identified by constructing a unique string at run-time based on the types of the return value and the function parameters. This approach requires no access to the symbol table, and thus is portable across multiple compilers, operating systems and platforms. It also supports profiling in the presence of optimization. Potentially, however, this approach is inefficient: if the string containing the function name is passed as an argument to an instrumentation routine at each function entry and exit, there is a significant run-time overhead associated with the string processing and hash table lookup needed to locate the appropriate profiling data. To avoid this, TAU uses a one-time function name registration with subsequent, low overhead function invocations. It provides a
static object of class FunctionInfo having the function name and type information as parameters to its constructor which is inserted into the function. It is responsible for function name registration and, since it is static, name registration occurs only once per function. The address of this object is then passed as an argument to another object of the class Profiler. This object’s constructor and destructor are invoked at each function entry and exit, but require just a pointer dereference, an inexpensive operation, to access the function profile data. Furthermore, the API is implemented as a set of macros that expand to these objects when profiling is enabled, and to the default of null when profiling is disabled.

This third approach solved the problem of unique identifiers. In addition, however, it handled a number of other problems: runtime inefficiency of access to profile data, function identification and name registration issues, template instantiation based on runtime type information of objects [74], support for multiple source files, profiling constructors and destructors of classes, profiling in the presence of optimization, and portability across compilers and platforms.

### 6.2.2 Per Object Profiling

While conventional profiling techniques generate profiles for member functions summed over all objects, they do not address the problem of associating the performance data with individual objects. Object oriented techniques allow the user to design and understand the application in terms of objects that are instances of classes. Traditional procedural languages such as Fortran and C gave users a procedure centric view and the profile data reflected this view. As we transition to object oriented thinking, it may be beneficial to interpret performance entities such as number of member function invocations on a per object basis. Parallel execution of member functions poses another problem as the profiling data is inherently multi-dimensional and objects are typically spread over one or more nodes, contexts and threads. They term these statically defined distributed objects as “aggregates”.

For profiling member functions on a per object basis, the objects needed to be identified by the user and name registration procedures maintained the tuple of object address and its type information. Inside member functions, the instrumentation checked the address of the object and using a map [74] associated the statistics such as the counts associated with predefined events (representing member functions) with the address of the object. In Figure 9., we see the aggregate
profiles in the main racy\textsuperscript{1} window showing activity on a set of distributed objects. By clicking on an individual object we get the view presented in Figure 10., where we see details of the distributed object “z” which is an instance of the distributed vector class and has significant events (member functions) associated with it. We see the distribution of these events on different nodes, contexts and threads where the colors represent each event.

One drawback with using per object profiles is the significant cost associated with each map lookup inside the member functions for associating the profiling data with each object which can significantly perturb [34] the parallel program.

\textsuperscript{1} racy is the TAU profile browser tool.
6.3. Conclusion
Profiling performs an action of updating summary statistics of execution when an event occurs. It uses the occurrence of an event (such as a routine transition) to keep track of performance metrics. The amount of data collected for profiling is significantly less than that collected for event-based debugging. The following questions summarize some of the interesting research problems posed by performance profiling:

- **What metrics should we be summarizing?** Can one metric effectively characterize all bottlenecks in an application? Should a tool use a fixed set of metrics or allow the user to build application specific performance abstractions using expressions of underlying metrics? How can one define new metrics? [45] studies this aspect.

- **How do we deal with multiple layers at which this data is collected?** Can the user selectively focus on a set of layers and associate the entities to the layers? Grouping related methods and routines from each layer into profile-groups allows the user to selectively enable or disable the instrumentation associated with the layer and focus on it. [36] explains how this can be done.

- **How do we specify the events that we profile?** Specifying events such as routine transitions or user-defined events, requires a mechanism for doing so. This may be implemented in the form of a static profiling API [23] or a dynamic one [27][46].

- **How do we relate the performance information to source code constructs or bottlenecks?** Relating source performance information back to source code gives the user flexibility in identifying code segments responsible for poor performance. [24][40] discuss this issue.

- **What are the performance bottlenecks and how do they relate to changing the source code?** The presence or absence of a bottleneck is ascertained by testing the observed performance of the application against thresholds for acceptable performance. [45] looks at the problem of searching for bottlenecks in parallel executions.
Chapter 7: Profiling High-Level Parallel Languages and Application Frameworks

It is important for users to see performance data at a level of abstraction that the parallel programming model of the application framework/language provides. Transformation of source code by optimizing compilers often makes it harder for tool developers to present meaningful performance data to the user. This necessitates a tighter integration between application frameworks, languages and tools.

In this chapter, we present two common types of optimization techniques employed in application frameworks and high-level data-parallel languages. Expression templates, and compile time array optimizations, in the context of ZPL, hinder profiling due to code restructuring and transformations. We discuss mechanisms we’ve developed for profiling in the presence of these optimizations.

7.1. Expression Templates
Expression templates [86] allow logical and algebraic expressions to be passed to functions as arguments that are inlined in the function body. Traditionally, operator overloading in C++ permits a natural notation for building expressions with arrays using pairwise arithmetic overloaded operators. Evaluating these expressions, however, requires the creation of temporaries, which results in performance degradation. Expression templates are used instead to build a representation of the parse tree for the right hand side of an expression that avoids the high cost of binary arithmetic operators. The expression is parsed at compile time, and stored as a nested template argument of an expression type. This technique can be used to evaluate vector and matrix expressions in a single pass without temporaries.

Expression templates are difficult to profile. Existing tools are useless when profiling such a program because they would chart the time spent in internal routines pertaining to the parse tree (i.e., the functions resulting from the full expression template expansion). Further, their representation of the type for the expression template for $A + B \times C$, say, would be something that is hard to interpret like “Expression<OpAdd,Array,Expression<OpMultiply,Array,Array>>.”
Figure 11 shows typical routine names that result from standard tools, when using expression templates. It is taken from the POOMA particle simulation toolkit, showing the profile of Conejo, a multimaterial hydrodynamics code that uses expression templates implemented using the Portable Expression Template Engine (PETE) [96]. The routine names in the profile are difficult
TAU gives the user the flexibility to choose the level at which profiling takes place. Furthermore, the profiling information can be represented in a form specific to the meaning of the nested types. With the knowledge of what the nested type means, the string given to TAU can be of the form “Array+Array*Array,” resulting in a more readable report. This is implemented in the Blitz++ [78][79][84] numerical library that implements efficient operations involving array in C++. In Figure 13, TAU shows a high-level profile representation of the performance of the expression template code from Figure 12.

### 7.2. Compile-time Optimizations

Compilers for high-level data parallel languages use extensive optimizations to generate efficient code. The resulting transformed code is difficult to profile effectively. In this section, we discuss profiling in the presence of compile-time optimizations in the context of the ZPL language.

ZPL [76][77] is a data-parallel array programming language designed for scientific computations. It uses optimization techniques that transform the high-level ZPL code to optimized C code that has a runtime efficiency of handcoded message passing programs while taking the advantage of
high-level abstractions that hide low-level implementation details. This however, results in difficulties while profiling ZPL programs, similar to that of debugging in the presence of optimizations[16][104].

In [77], the authors discuss how ZPL data-parallel normalized statements are decomposed into factors, (a factor corresponds to a different communication or computation structure) and subsequently how optimizations are achieved by combining, or joining these factors. Factors preserve high-level information about the array operations and the contributions of factors are employed in “mapping” the performance data gathered for low-level operations to the high-level statements that these factors represent. Each factor represents an elementary array operation involving either local computation (C-factor) or interprocessor data transfer (T-factors). Array iterations form m-loops (C-factors) and they represent the local portions of any statement, and include operations such as assignment, reduction or scan. These m-loop factors are joined and it results in fused loops in the corresponding C code. The criteria for deciding whether or not to fuse these factors is similar to data dependence analysis for fusing loop nests. The optimizations used in ZPL involve joining m-loops (loop fusion), array contraction (replacing an array by a scalar in a loop), and communication optimizations (message vectorizations, combining communication portions).
26 XVel := DeltaXPos/DeltaT;
27 YVel := DeltaYPos/DeltaT;
28 Vel := sqrt(XVel * XVel + YVel * YVel);
29 minvel := min << Vel;
30 maxvel := max << Vel;

Figure 14. Original ZPL source

for (i=mylow; i<myhi; i++)
    XVel[i] = DeltaXPos[i] / DeltaT[i];
for (i=mylow; i<myhi; i++)
    YVel[i] = DeltaYPos[i] / DeltaT[i];
for (i=mylow; i<myhi; i++)
    Vel[i] = sqrt(XVel[i]*XVel[i] + YVel[i]*YVel[i]);
temp = DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = min (temp, Vel[i]);
<... data transfer code here ...>
<... assignment to minvel ...>
temp = -DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = max(temp, Vel[i]);
<... data transfer code here...>
<... assignment to maxvel ...>

Figure 15. Naive Loop generation

temp1 = DBL_MAX;
temp2 = -DBL_MAX;
for(i=mylow; i<myhi; i++) {
    xvel = DeltaXPos[i] / DeltaT[i];
    yvel = DeltaYPos[i] / DeltaT[i];
    vel = sqrt(xvel*xvel + yvel*yvel);
    temp1 = min(temp1, vel);
    temp2 = max(temp2, vel);
}
<... data transfer for 29, 30 ...>
<... assignment to minvel ...>
<... assignment to maxvel ...>

Figure 16. Optimized code generated by the ZPL compiler
TAU_PROFILE_TIMER(t26, "XVel := DeltaXPos / DeltaT;", " line 26 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(t27, "YVel := DeltaYPos / DeltaT;", " line 27 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(t28, "Vel := sqrt(XVel^2 + YVel^2);", " line 28 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(t29, "minvel := min << Vel;", " line 29 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(t30, "maxvel := max << Vel;", " line 30 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(tcomm, "Communication", "line 29 and 30 vel.z", TAU_DEFAULT);
TAU_PROFILE_TIMER(tloop, "loop overhead", "line 26, 27, 28, 29 and 30 vel.z", TAU_DEFAULT);

TAU_PROFILE_START(t29);
  temp1 = DBL_MAX;
  TAU_PROFILE_STOP(t29);
TAU_PROFILE_START(t30);
  temp2 = -DBL_MAX;
  TAU_PROFILE_STOP(t30);
  TAU_PROFILE_START(tloop);
for(i=mylow; i < myhi; i++) {
  TAU_PROFILE_STOP(tloop);
  TAU_PROFILE_START(t26);
    xvel = DeltaXPos[i] / DeltaT[i];
    TAU_PROFILE_STOP(t26);
    TAU_PROFILE_START(t27);
    yvel = DeltaYPos[i] / DeltaT[i];
    TAU_PROFILE_STOP(t27);
    TAU_PROFILE_START(t28);
    vel = sqrt(xvel*xvel + yvel*yvel);
    TAU_PROFILE_STOP(t28);
    TAU_PROFILE_START(t29);
    temp1 = min(temp1, vel);
    TAU_PROFILE_STOP(t29);
    TAU_PROFILE_START(t30);
    temp2 = max(temp2, vel);
    TAU_PROFILE_STOP(t30);
  TAU_PROFILE_START(tloop);
}
TAU_PROFILE_STOP(tloop);
TAU_PROFILE_START(tcomm);
<... data transfer for 29, 30 ...>
   TAU_PROFILE_STOP(tcomm);
TAU_PROFILE_START(t29);
<... assignment to minvel ...>
   TAU_PROFILE_STOP(t29);
TAU_PROFILE_START(t30);
<... assignment to maxvel ...>
   TAU_PROFILE_STOP(t30);

Figure 17. Profiling optimized ZPL code using TAU
To illustrate optimizations in ZPL, Figure 14, shows a portion of a ZPL code that computes approximate minimum and maximum velocities of a particle from a vector of sampled positions and times [77]. Naive code generation would result in Figure 15. The ZPL compiler optimizes this code by joining factors and employing communication optimizations to result in code as shown in Figure 16. This leads us to the problem of profiling in the presence of these optimizations: How can we profile this many-to-many mapping between reductions in ZPL to segments of code in C? Should we attribute the time spent in the implementation routine in C to the group of ZPL statements?

If commercially available profilers are used to measure the performance of a ZPL transformed program, they can successfully display the time spent in C routines but fall short of relating it to the contributions of individual ZPL statements known to the user. This “semantic-gap” in understanding the performance data of optimized execution is representative of the problems that profiling in the presence of optimizations poses.

It is possible, however, to do better. By tracing the transformations associated with the factors, to generate the optimized code, we can “map” the higher level ZPL constructs into the optimized code. This aids in the process of instrumentation which can relate this information in terms of the original ZPL code.

The ZPL compiler preserves information about the set of C statements that are generated from a ZPL statement. Even after optimizations such as loop fusion (joining m-factor), we can correlate the part of C statement (shown in bold typeface) back to the originating ZPL statement. This represents a “many-to-many mapping”. This problem is dealt with in [98]. The author suggests reducing this mapping by aggregating the cost of all low-level (C) statements to one function and treating it as a one-to-many mapping. The cost of this aggregate function can then be split evenly over all higher-level (ZPL) statements, or it can be computed by merging all higher-level statements into one set and by assigning the cost of the aggregate C-function to the set of higher-level (ZPL) statements. This would result in a profile that shows the total cost of executing the five ZPL statements. TAU tries a different approach where the cost of “individual” ZPL statement can be computed using information about the transformations. Figure 17 shows how the contribution of the optimized statements can be related back. This approach works for optimizations such as array contraction, loop fusion, and communication optimizations too. In short, when the
“contribution” of one ZPL statement is traced down to one or more C statements (after optimizations), the cost of each individual C-statement is attributed back to the original ZPL statement. This helps us generate descriptive, meaningful profiles using TAU that highlight the contributions of each ZPL statement and provides a better and more accurate abstraction as shown in Figure 18.

Other approaches to mapping performance data from one layer of abstractions to another have been studied in detail in [98]; it defines SAS (Set of Active Sentences) and shows how it can be used to deal with mapping through layers of abstraction. The SAS is a data structure that “records the current execution state of each level of abstraction similar to the way a procedure callstack keeps track of active functions. Any two sentences contained in the SAS concurrently are

Figure 18. Profiling in the presence of optimizations: profiling ZPL programs
considered to dynamically map to one another” [98]. The techniques proposed here are useful in profiling parallel applications that use one-to-one mapping and employ a “synchronous” model of execution. We feel that one can do better (as illustrated in the ZPL example) in terms of accuracy and abstraction, while treating “many-to-many” mappings if information as shown in the ZPL example is available. A second limitation of this approach is in profiling asynchronous execution.

Similarly, profiling in the context of lazy functional languages has been studied for Haskell [103]. Lazy evaluations are similar to futures and use the asynchronous execution model. Their approach annotates particular source expressions (which they are interested in profiling) with “cost-centres” (a label under with they attribute execution costs) that identify an expression. “During execution, statistical information is gathered about the expressions being evaluated and attributed to the appropriate cost centre”. They keep track of the cost centre of the current expression being evaluated. As costs are incurred, they are attributed to it. The process of annotation identifies precisely the mapping of the expression to be evaluated with the label of the entity to which the cost of the expression is to be attributed to. Lazy evaluation ensures that only the expressions that are “demanded” are evaluated and when the evaluation takes place, these costs are mapped directly.

7.3. Conclusions
In this chapter we studied the problem of performance analysis of optimized executions. It shows two different systems where compile-time optimization techniques are used to restructure and generate efficient code. These are representative of the problems faced by tools to show performance data based on program entities that the user can understand. In Blitz++, expression templates make it difficult for the user to understand the template names and their association with the structure of the expression. In ZPL, mapping ZPL constructs to transformed C constructs represents an efficient many-to-many mapping that is used to accurately attribute the time spent in different ZPL statements that were written by the user. Generating and effectively representing meaningful and accurate performance data is the of paramount importance in dealing with optimization techniques to reduce the “semantic-gap” that exists in tools today.
Chapter 8: Research Directions

Optimization techniques require an efficient mapping mechanism to relate high-level application framework entities to low-level implementation level routines to better represent the performance of optimized executions. In this chapter, we continue to examine the problem of performance analysis of optimized programs with asynchronous model of execution. We show an example of the type of problem environment that we are targeting.

8.1. Profiling in the presence of runtime optimizations using asynchronous execution

Object-oriented numerical frameworks implemented in high-level languages such as C++ show great promise as they present domain-specific data structures and algorithms with better abstraction than traditional languages such as Fortran [81][83][84]. Several inefficiencies of C++ are addressed with techniques such as Expression-templates [86] that help achieve near- or better than Fortran performance for C++ codes [84] that use algebraic expressions composed to data-parallel implementations of arrays. These techniques have been implemented in POOMA [82][72][71], A++/P++ [70][64][65], and Blitz [78] frameworks that illustrate the viability of using object oriented techniques in numerical computations.

The complex organization of todays parallel computers with deep memory hierarchies has forced numerical analysts to seek help from runtime system experts for optimizing the performance of data-parallel statements on parallel shared memory multiprocessors. The runtime-system layer presents a common interface to higher level application or library layers and uses platform specific optimizations for better utilization of communication or memory resources. Parallelism stems from concurrent execution of independent loop iterations in data-parallel statements. The discovery of these independent loop-iterations at runtime, and their efficient execution based on data locality and cache reuse optimizations leads to better utilization of memory hierarchies. Conventional synchronous model of execution uses “horizontal” execution of a set of data-parallel statements, where the execution of a statement does not commence till the previous statement has finished execution. Multiple processors simultaneously apply the same set of operations to different parts of the data associated with the execution of such a statement. In contrast, the concurrent “vertical” execution of multiple data-parallel statements while maintaining data dependencies improves data locality and cache reuse. This concept has been
implemented in SMARTS (Shared Memory Asynchronous Runtime-System) [87][95] that exploits both vertical and horizontal execution using the “asynchronous” model of execution.

Performance analysis tools find it challenging to deal with asynchronous execution of statements [98].

There are several problems that need to be solved before tools can accurately represent the profile of each data-parallel statement and correlate it back to the source code constructs that the user is familiar with. The first problem is “mapping” between elements of one layer of abstraction such as the user specified code constructs to those of underlying “implementation-centric” layers. The second problem is building a profiling framework that provides facilities for accurately attributing the time spent executing the high-level statement. These two problems are described in greater detail in the following sections.

8.1.1 Mapping

Scientific applications are often constructed using multiple layers, components and through the composition of such components. Each of these layers represents an abstraction level for performance information. The user often sees the frameworks through user-level entities (classes and functions comprising the API) and behind this high level layer lies one or more implementation layers. Mapping performance entities across layers is a challenge. To illustrate this, we use an example from an Array class library POOMA that provides mechanisms for the user to construct expressions involving arrays. For example.

\[
\begin{align*}
S1: & \quad A = B + C + D; \\
S2: & \quad B = 2 \times A; \\
S3: & \quad C = D - E + A; \\
S4: & \quad \text{Pooma::blockAndEvaluate();}
\end{align*}
\]

Using PETE (Portable Expression Template Engine), statement S1 is transformed using compile time optimizations to the equivalent of:

\[
\begin{align*}
\text{for (i=0; i<N; i++)} \{ \\
& \quad A[i] = B[i] + C[i] + D[i]; \\
\} \quad \text{// N is the size of the arrays}
\end{align*}
\]
This is followed by decomposition of the three data parallel statements into Iterates, or work packets that are scheduled on one or more processors in a manner that optimizes cache reuse for vertical execution. This allows array chunks (that fit into the cache memory) of array A when S1 executes, to be reused for subsequent computations (statements S2, S3). This is done by “delaying” the execution of S1, S2 and S3 by just registering the statements and composing the work packets instead of executing these synchronously. Data dependence analysis is done at runtime, when S4 executes, and the graph of iterates is created and partitioned on one or more processors to achieve parallelism.

Mapping the execution of statement S1 to a) registration of statement, b) work involved in generating chunks or iterates and c) the execution of chunks that belong to S1, would allow us to build tools that would aggregate time spent (or cache misses or other quantities) in a through c and attribute it to statement S1 to present the user with a high level view of the performance data that shows the “cost” associated with the execution of each source code statement. It might prove useful to be able to provide other views of the performance data as well, which might allow the user to see the “implementation view” focussing on one or more layers of implementation (runtime-system) or see a combination thereof.

**8.1.2 Performance Instrumentation Framework**

We need a performance instrumentation framework to work in conjunction with the mapping information that we hope to generate. To show aggregate summary statistics (profiling) or detailed event trace information representations with processor timeline displays (tracing) requires an instrumentation framework that:

a) Works with source level instrumentation in C++ and can handle all the intricacies of the language.

b) Provides user-level timers that provide the function/basic block abstraction to regions of code for instrumentation.

c) Provides access to hardware performance counters on platforms where available (for extracting cache miss information) and to low-overhead timers for minimal intrusion.

d) Provides components of performance instrumentation (callstack, exclusive and inclusive timers, function database) that can be put together to form modules for profiling, tracing, tracking statistics, and monitoring. The framework needs to be flexible enough to allow new
performance abstractions (user defined events) to be added to it as modules and be robust and portable enough to handle multiple platforms, compilers, operating systems, and languages.

e) Provides access to dynamic information that cannot be captured by traditional static instrumentation techniques for tracking runtime type information in dynamic languages such as C++.

f) Works reliably in shared memory environments with multiple thread packages such as Tulip [88], pthread, and SMARTS threads consistent with other requirements mentioned in this section. It should be capable of showing aggregate statistics across all nodes, contexts and threads of execution for each data-parallel statement that is instrumented.

g) Provides selective instrumentation capabilities that allow instrumentation in one or more components to be disabled at runtime to reduce the intrusion caused by the instrumentation. This is particularly useful when the user needs to focus on performance data from one or more components.

h) Provides mechanisms to automatically or manually instrument the source code. It would be desirable if it could plug into other means of instrumentation as well (such as dynamic instrumentation [45][5]). An ability to instrument a program written in a language such as C++ mandates the use of robust parsers to interface with such a framework for instrumentation.

i) Provides mechanisms to relate this performance information to source code constructs familiar to the user.

If such a framework were constructed for performance analysis, it would aid in showing the user performance data for the asynchronous optimized program at a higher level of abstraction, if the problem of mapping, as discussed above, was solved. The above steps exemplify the research problem in the context of C++. Our research aims to be more general and widely applicable.

8.2. Conclusions

This research takes a two pronged approach to solving the problem of profiling in the presence of optimizations for asynchronous executions. Firstly, it attempts to solve the problem of mapping performance data between layers of a vertically integrated parallel scientific application to show the user performance data while peeling through layers of abstraction and presenting it with the constructs meaningful to the user. Secondly, it aims to build a performance analysis framework
that can be effectively used by such mappings to generate the performance data for the applications. The combination of these two strategies would provide a comprehensive performance analysis environment that would work in the presence of static and runtime optimizations with synchronous and asynchronous models of parallel execution.
Chapter 9: Conclusions

We conclude this paper by reviewing how we can better understand the behavior of parallel programs. We studied the areas of instrumentation, and how it helps to generate data for a diverse set of areas such as performance profiling, event-tracing, debugging, monitoring, replaying and checkpointing. Once the data is generated, we need efficient ways to represent this to the user in the form of visualizations that show the mapping for low-level data to high-level program entities that are meaningful to the user.

9.1. Summary and Contributions

This work highlights:

- The issues of instrumentation. The compilation/execution stage at which the instrumentation is inserted in the parallel program determines the nature of data that can be captured as well as whether the instrumentation system is language specific or platform centric. To be portable, the instrumentation system needs to work across languages on a number of platforms.

- The problem of profiling and the performance metrics that can be used to characterize the performance of an application. It is difficult to anticipate all metrics that can be useful for profiling. It would be useful to build a portable, modular and extensible profiling infrastructure that could be extended with new metrics and could build upon previous profiling abstractions.

- The problem of capturing events in the form of execution traces. These traces are useful in both event-based debugging as well as performance debugging. We discussed how the problems of non-determinism is addressed by replay, how the problem of asynchrony is addressed by using logical time and using globally synchronized real-time clocks, how the problem of a lack of a consistent global state is addressed by event-based debugging by using causality constraints in setting a globally consistent breakpoint, and how the problem of the probe-effect is addressed by perturbation analysis in compensation of intrusion due to the instrumentation.

- The techniques used to profile in the presence of optimizations. Representation of meaningful data is just as important as meaningful representation of data. We looked at the problem of mapping and how it can be used to peel through layers of abstraction to correlate performance data with program entities that the user can understand easily.

- Our future research directions. We look at the problem of analyzing the performance of parallel programs that use runtime-optimization techniques to improve data-locality using an asynchronous model of execution.

Unless tools can present performance and debugging data in ways that are meaningful to the user, and are consistent with the user’s mental model of abstractions, their success will be limited.

9.2. Acknowledgements

I gratefully acknowledge the support of all the members of the TAU team at the University of...
Oregon and Peter Beckman and Steve Karmesin from Los Alamos National Laboratory. I thank Todd Veldhuizen for his work in profiling expression templates in Blitz++. I’d like to thank the members of the POOMA team at LANL for their support and encouragement.
References


