Event and State-Based Debugging in TAU: A Prototype

Sameer Shende, Janice Cuny*, Lars Hansen, Joydip Kundu†
Stephen McLaughry, and Odile Wolf

Department of Computer and Information Science
University of Oregon
Eugene, Oregon

Abstract. Parallel programs are complex and often require a multilevel debugging strategy that combines both event- and state-based debugging. We report here on preliminary work that combines these approaches within the TAU program analysis environment for pC++. This work extends the use of event-based modeling to object-parallel languages, provides an alternative mechanism for establishing meaningful global breakpoints in object-oriented languages, introduces the TAU program interaction and control infrastructure, and provides an environment for the assessment of mixed event- and state-based strategies.

1 Introduction

Parallel programs can have complex behaviors that are difficult to debug. We advocate a two-level strategy in which an event-based approach is first used to investigate high-level patterns of process interactions and then, as the focus of attention narrows, a state-based approach is used to investigate lower-level program details. Event-based techniques focus the user's attention on manageable portions of the state space, provide the basis for establishing consistent and meaningful breakpoints, permit replay, and allow manipulations that filter the effects of asynchrony. State-based techniques offer a familiar environment for examining an execution to an arbitrary level of detail and make it easier to relate detected anomalies to source code errors. The combination of these techniques provides comprehensive debugging support.

We report here on preliminary work that combines event- and state-based debugging approaches within the TAU program analysis environment [3]. We build on an earlier prototype [15] which combined our event-based debugger, Ariadne, with Intel's state-based debugger, *ipd* [11]. Combining Ariadne with a debugger in TAU provides us with a more realistic programming environment for assessing the strengths of mixed event- and state-based strategies. TAU is an environment for programming in the pC++ data-parallel language [6]; in this paper, we introduce extensions to event-based behavioral modeling that support object-parallel languages and provide new mechanisms for establishing meaningful breakpoints in object-oriented code. In addition, we introduce the TAU program interaction and control infrastructure that allows the integration of debugging and visualization tools as cooperating clients of a single agent.

In Section 2, we briefly describe the Ariadne debugger. Ariadne was designed for use with explicitly parallel languages and required some modifications to its modeling and replay/breakpointing facilities for pC++. The modifications are described in Sections 3 and 4. In Section 5, we introduce the TAU program interaction and control infrastructure. Finally, in Section 6, we present our conclusions and outline future directions, focusing on the debugging facilities that will be needed for more complex object-parallel languages that incorporate task parallelism.

2 The Ariadne Debugger: An Overview

Ariadne is a *post mortem*, event-based debugger [13, 16]. It matches user-specified models of intended program behavior against actual program behavior as captured in event traces [2]. The traces record low-level interprocess communication, pre-defined language specific-events, and user-defined events. They are stored internally as *execution history graphs* with nodes representing events and edges representing orderings imposed by Lamport's *happened before* relation [17]. Ariadne matches user-models against the execution history graph, extracting matched or partially matched subgraphs for further investigation. The extracted subgraphs can be explored through additional modeling, functional queries, and manipulations of logical, temporal orderings.

Ariadne's modeling language for describing intended behavior is quite simple. It was designed to allow efficient matching and precise feedback on errors. Communication patterns are described in terms of a three-level hierarchy of *chains*, *p-chains*, and *pt-chains* with a notation based on regular expressions.

- *Chains* are patterns that represent "local views" of communication. They are described by extensions of regular expressions. When matched against an execution history graph, all events in the chain must occur exactly in the specified order (with the exception of communication events that are not physically realized because of "edge effects" on process array boundaries).

*Support for this work was provided by ARPA under Fort Huachuca contract ARMY DABT63-94-C-0029 and by the National Science Foundation under grants CCR-9023256 and STI-9413532.

†Joydip Kundu is currently employed by Digital Equipment Corporation, Nashua, NH.

- *p-Chains* are patterns that represent the concurrent execution of a chain by one or more processes. They are described by binding a chain to a process set. When a p-chain is matched, the chain is matched on each element of the associated process set.

- *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 relations between those events are verified. (Since the temporal relations are not used in the examples in this paper, we do not discuss them further here but refer the reader to existing literature [8].)

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 a match tree correspond to the hierarchical levels of its model.

Ariadne's matching language is quite simple and, as a result, it is not always expressive enough; often its patterns are too coarse, matching program behavior in unexpected ways. To compensate, Ariadne provides a scalable, spreadsheet like interface for exploring match trees, called AVE (Ariadne Visualization Engine) [14]. AVE features the automatic compression of hierarchical visualizations, dynamic attribute calculation, and the partitioning of matched behaviors and attributes. Together these features allow a user to answer questions about a match such as: Did the number of events decrease on each iteration of a loop? Which processes were active during the third iteration? Which processes executed successful comparisons? and Why did the match fail on process 12?

Ariadne was designed for explicitly parallel programming languages in which all interprocess communication occurs through read and write operations. In the next section, we discuss the modifications we made so its modeling language would be appropriate for pC++.

3 Event-Based Behavioral Abstraction for pC++

pC++ is an object-parallel programming language intended for both shared and distributed memory machines. It extends the notion of data-parallel execution to *aggregates*; that is, structured sets of objects distributed over the processors of a parallel system. Application-specific aggregates are described in pC++ as *collection* class types. Parallelism is implicit and results from the concurrent application of a function to the elements of a collection. pC++ provides a shared name space; references to shared data compile to GET_ELEMENT routines that generate (at runtime) explicit sends and receives for nonlocal data.

Event traces from a pC++ program include these generated events, each tagged with the type of operation it implements (for example, W_AskForData, R_StoreBlockMsg, and W_AckReplyMsg[1]). Thus the trace exposes the workings of the runtime system to the programmer, which is sometimes useful in debugging. For the most part, however, it is not. The majority of pC++ programmers understand their code only at the higher level of data-parallel operations and those operations must be accessible during debugging.

[1] The actual event names that Ariadne currently uses are much less readable and they have been changed for clarity purposes throughout this paper.

Ariadne's predefined event set can be extended dynamically, without requiring a recompilation of the debugger. This enables us to add higher-level pC++ events as predefined user events, leaving the debugger itself independent of the source language. For pC++, the predefined events include

BeginCreateCollection	EndCreateCollection
BeginBarrier	EndBarrier
BeginRpc	EndRpc
BeginParallel	EndParallel
BeginServiceRpc	EndServiceRpc

We also allow ordinary user events which can be inserted into pC++ source code in standard functions, in methods of elements (invoked by each element of a collection), or in member functions (invoked on a per process basis). This gives us three levels of abstraction in traced events: runtime communication events, pC++ predefined events, and user events. Ariadne filters unwanted events from a match, making it possible to use events from different levels separately or in combination.

Example: Binary Image Compression. To illustrate the use of Ariadne and the extended modeling language, we debug a parallel version of an algorithm to compress black and white images stored as *bintrees* [22]. Working up from the leaves, the algorithm compresses homogeneously colored siblings into single nodes represented at their parent as shown in Figure 1. The parallel version of the algorithm distributes entire subtrees of the initial bintree to processes. We ran our code on images with 256 nodes. It ran to completion but frequently produced compressions that were too small.

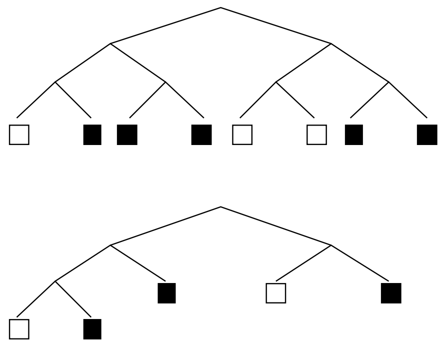

Figure 1: Original bintree representation of an image with its compression.

To begin debugging this program with Ariadne, we added user-defined events UserMerge and UserNoMerge to trace the decisions made about each pair of siblings. The behavior of a single node during a step in which it is active was then described by one of the two chains

ch Merge = UserMerge wrt [ ];
ch NoMerge = UserNoMerge wrt [ ];

where ch is a keyword that begins the definition of a chain and the wrt clause indicates that event types not explicitly mentioned in the pattern are to be ignored.[2]

pC++ distributes the elements of a collection across the processors and uses an "owner computes" rule. In this case, elements of the collection correspond to the active nodes of the bintree; initially the leaves are active. Processors iterate through the elements assigned to them. To model the combined behavior of the nodes across a level of the tree, we first define the successful completion of a barrier synchronization as

ch Barrier = BeginBarrier EndBarrier wrt [ ];

and then model the complete behavior at a single level as

pch CompressLevel =
  Barrier* ((+ Merge NoMerge +)) * Barrier
  onsome Procset;

where pch is a keyword that begins the definition of a p-chain, (+ a b +) denotes the regular expression a + b , * denotes the Kleene star, and onsome indicates that the pattern occurs on some, but not necessarily all, of the specified processes. Since we ran our code on an SGI Power-Challenge with 8 processors, we used the set of process ids Procset = {0..7} for this example. It should be noted that onsome was quite useful here because at the later stages of the algorithm, subsequent steps involve fewer processes and thus it allows us to specify a single model for all steps.

The behavior of the algorithm as a whole is modeled as

match BIC = CompressLevel* ;

where match is a keyword indicating that the specified pattern (0 or more instances of CompressLevel) is to be matched in the execution history graph and the resulting subgraph is to be assigned to the specified variable, BIC. Note that the model is scalable: it is completely insensitive to the input size and only the definition of procset depends on the machine size. Figure 2 shows a match of this model to an execution.

As mentioned above, matching imposes a tree structure on the matched subgraph that corresponds to the model's structure. It is this *match tree* that is displayed by AVE in the figure. For scalability, it is automatically compressed both horizontally and vertically: horizontally by eliding sibling nodes and vertically by pruning levels. The horizontal compression is straightforward: omitted siblings are represented by a single slider. In our example, the match tree included 8 instances of the CompressLevel p-chain but only the first and last are explicitly shown in the initial picture. Vertical compression is achieved by pruning subtrees below each p-chain. The pruned subtrees are partitioned into equivalence classes and represented by an icon labeled with a pair of integers that summarizes the partitioning: the first element is the total number of chain subtrees represented and the second is the number of equivalence classes of those trees. In the example, the leftmost box is labeled with **8,1** which indicates that 8 processes initiated a CompressLevel event and all of them matched the same p-chain pattern. The pop-up window below (obtained by clicking on the icon) gives a further description of this subtree: a

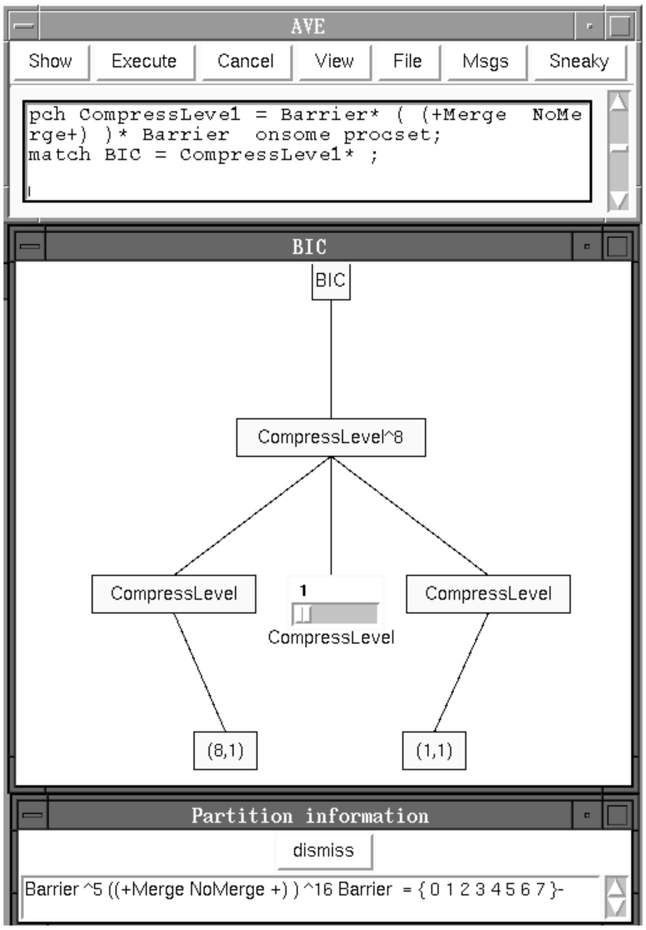

Figure 2: AVE output showing the match tree for an 8 processor compression of a 256 node binary image as originally displayed with pop-up window for further explanation of icon labeled **8,1**.

pattern matching 5 barrier events, followed by 16 compression events (either Merge or NoMerge), followed by another barrier was matched on processes 0 through 7. This is as expected. In the first step of the compression, each process handles 32 leaf nodes of the image, performing 16 compression operations. In the last step, only one process is active as evidenced by the **1,1** label on its node.

Intermediate nodes of the tree can be viewed as in Figure 3. It shows the third node also expanded. The third node again has all eight processes active because they are still handling local subtrees. Eventually, the local subtrees are compressed and at the later stages of the program, processes become inactive; if we looked at the events of the seventh and eighth stages, for example, we would have seen that only two processes were active in the seventh and only one in the eighth.

So far, our debugging output is as expected, the model matched the traced behavior. Ariadne's models, however, are quite coarse and it is often useful to investigate a match further using its query facilities. In this case, because we knew that the resulting compressed image was too small, we were suspicious that erroneous, extra Merge events were occurring. To check this, we wanted to determine the number of elements executing Merge events in each CompressLevel. First, we defined an additional attribute, MergElts, for Merge nodes using

[2] The wrt clause in each of these patterns would actually have to specify the set {Merge,NoMerge,BeginBarrier,EndBarrier} but we omit this detail throughout the paper to make the examples more readable. We continue to work on improving the usability of Ariadne's interface.

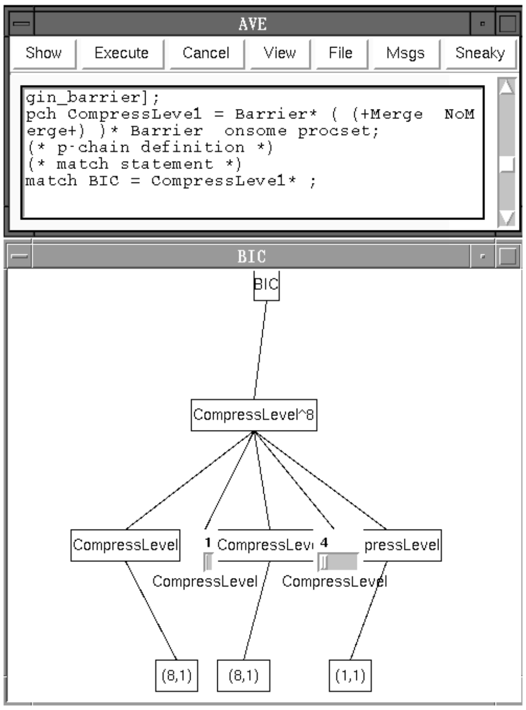

Figure 3: AVE output showing of the match tree for an 8 processor compression of a 256 node binary image after the user has expanded some of the nodes.

with BIC foreach Merge
compute MergElts = Elemid(UserMerge);

which computes the element ID attribute (Elemid) from the corresponding matched trace events. Our query was then

with BIC foreach CompressLevel compute
NumMerges = sizeof(MergElts(Merge));
with BIC show NumMerges (CompressLevel)
wrt CompressLevel;

which first counts the number of elements performing Merge events in each CompressLevel (that is, computes the size of the set of all elements IDs) and then displays the output as in Figures 4(a) and 4(b). These plots were not as expected. At each level in the bintree, only those nodes that were successfully merged at the previous level are available for merger; thus, the number of successful merges at any level should be no more than half the number from the previous step. Figure 4(a) shows roughly the decreasing values that we would expect but looking at the actual values in Figure 4(b), we see a problem. 32 events were compressed at the third level but this is more than half of the 63 that had been compressed at the second level. There were too many Merges on the third level.

Perhaps the program was incorrectly merging siblings with different color values. To check out this possibility, we replayed the execution (as discussed in the next section) annotating the traced Merge events with the color values of the subtrees being merged. We then rematched the model against the trace and asked if Merge events only occurred at like-colored siblings. The answer was "yes." We next investigated the possibility that some of the mergers were invalid because one of the siblings represented a subtree where compression had already failed at a lower level. This would result in a NoMerge event being followed by a subsequent Merge event *on the same element*. To check this, we first computed

with BIC foreach NoMerge compute
NoMergElts = Elemid(UserNoMerge);

as above and then computed the set of elements having each outcome as

with BIC foreach CompressLevel compute
MergeSet = MergElts(Merge);
with BIC foreach CompressLevel compute
NoMergeSet = NoMergElts(NoMerge);

None of the elements with Merge events on a level should have participated in the NoMerge events on the previous level. In terms of our sets, this means that MergeSets should be disjoint from the NoMergeSets on the previous level. We checked this with the query

with BIC foreach CompressLevel compute
MergeFlag = disjoint(MergeSet(CompressLevel),
NoMergeSet(left(CompressLevel)));
with BIC show MergeFlag(CompressLevel)
wrt BIC;

where left here refers to the immediately preceding CompressLevel event (that is, its left sibling in the match tree). This query produced the output in Figure 5 showing that the sets on the third level were not disjoint as they should have been. (We knew this because the output value for the third level was "0" indicating that the disjoint predicate was false.) Thus some element on level three did a compression on a subtree that had not been compressed at level two.

Further investigations with Ariadne were not productive. This, however, is typical: event-based debugging is useful in narrowing the focus of attention during the early stages of debugging but it is not as useful in relating the detected anomalies back to source code errors in the later stages of debugging. As the result of our event-based analysis, we know that the code merged nodes representing trees when it should have only merged nodes representing single, fully compressed leaves. We also know that this was first detectable on level three. We don't, however, know why it occurred. For this, we switched to a state-based debugger.

4 Setting Consistent Breakpoints from Event Trace Analysis

To examine the state of a program, it is necessary to set breakpoints that stop the computation at locations that are both consistent and meaningful to the programmer. Standard mechanisms set breakpoints in terms of program locations or combinations of local state predicates, but these are problematic: How are the breakpoints to be specified across multiple processes? Must they be globally consistent or are local breakpoints sufficient? If local, how and when should

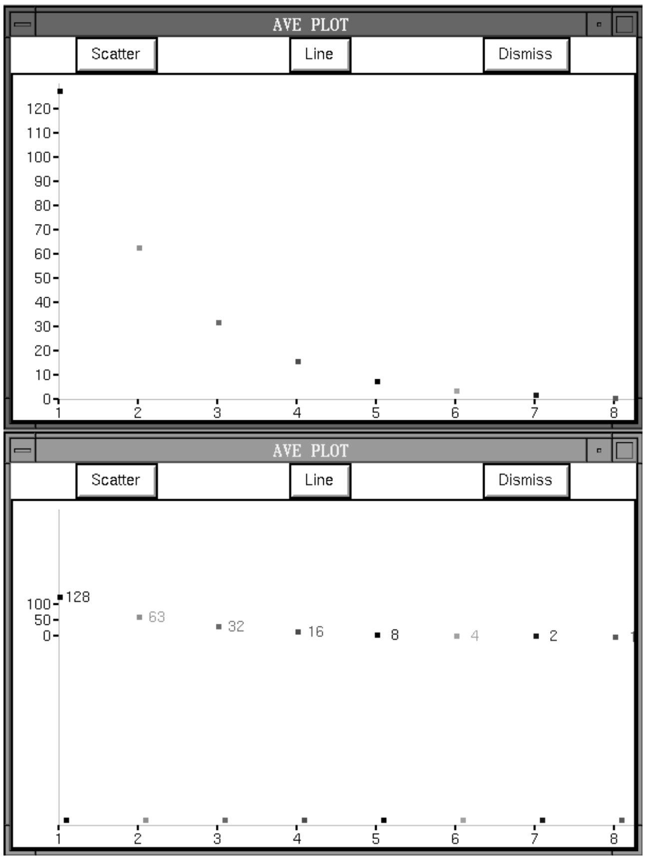

Figure 4: AVE output showing (a) a line plot of the number of merges that occurred at each level of the tree, and (b) a scatter plot of the same data.

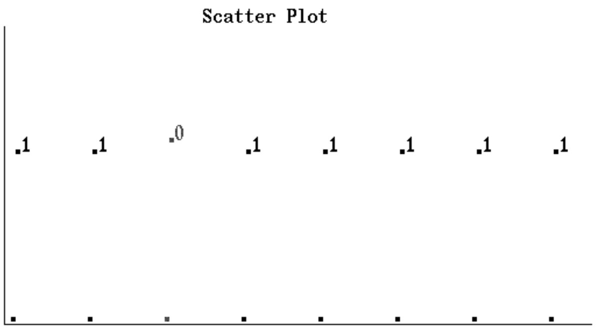

Figure 5: AVE output showing the comparisons between the sets of element ids on merged and nonmerged nodes.

the remaining processes be stopped in order to provide a meaningful state?

One possibility is to allow breakpoints to be set dynamically but only at barrier synchronizations as was done in the TAU *Breezy* debugger [4]. This, however, will be overly constraining in multithreaded and multitasked extensions to pC++, in which programs can have more of an MIMD flavor, with considerable amounts of computation occurring between global synchronization points. In addition, it does not exploit any event-based analysis that the user might have already done. As an alternative, we use the results of the ongoing event-based modeling to automatically halt the program at a consistent state during replay. Ariadne allows the user to specify breakpoints in terms of the same abstract events that he/she has already matched in the execution history graph. Once breakpoints are set, Ariadne annotates the trace and then re-executes the program to the latest consistent state preceding those annotations. Thus each process is stopped at the last point at which it could have influenced the event(s) triggering the breakpoint. This is the consistent state that is most meaningful to the programmer.

In related work, Manabe and Imase [19] also set breakpoints that were used to find consistent states during replay. They, however, used a state-based approach, specifying breakpoints with conjunctions and disjunctions of local state predicates. It is often difficult to determine the appropriate local predicates initially: errors observable only as incorrect output are hard to relate back to individual predicates in specific processes.

We implemented replay at the level of polling in the runtime system: pC++ uses polling in servicing requests for element values and the polling introduces nondeterminism. We record the number of unsuccessful polls as the "logical time" of a read event and, during replay, receive events are allowed only when the identical poll count is reached. This ensures that events take place at their respective logical times and preposted receives are introduced into the computation at the appropriate point.

During replay, selected matched (or partially matched) abstract events are used to partition the processes of a computation into three equivalence classes: the *core* set containing the processes that executed parts of the abstract event; the *influence set* containing the processes that could have causally affected a component of the abstract event; and the *other set* containing the remaining processes. In order to halt a computation, the state most relevant to the abstract event is restored [15]:

1. Processes in the *other set* are stopped at their earliest state;

2. Processes in the *influence set* are stopped immediately after their latest events that could have causally influenced the abstract event; and

3. Processes in the *core set* are stopped immediately before the earliest (or after the latest) local event that is a part of the abstract event (depending on the user request).

In some cases, this procedure results in a state that is not *consistent*, that is, a state that could not have occurred in an actual execution. We define consistency as in [7]. Whenever a user's breakpointing request would result in an inconsistent state, our debugger automatically constructs an earlier, consistent state using a greedy algorithm [13].

For the binary image compression example, we had already used event-based analysis to show that there was some node that was not compressed at level two in the tree that was subsequently compressed at level three. It makes sense to stop the program immediately after level two is processed. With Ariadne, this was done by selecting the second CompressLevel event from the match tree and issuing the command "break after (select)." This stops the program at exactly the right state, allowing us to examine the program variables immediately after the level two compressions complete but before the level three compressions start on each element. Note that this particular breakpoint would have been very difficult to set without first performing the event-based analysis. The user could not have known to set the breakpoint after the second level of the tree, as opposed to the third or fifth or seventh. The "values" that we examined in coming to our choice of breakpoints, that is the count of Merge events and the elements of the MergeSet and NoMergeSet, were not values computed in the program. The conditions of interest were global properties of the computation, not expressible as local state predicates. Event-based analysis often makes it possible to narrow the user's focus to exactly the right set of states for breakpointing.

After the program is stopped, control is transferred to the *Sneaky* breakpoint debugger. This let us examine the local state variables as shown in Figure 6.

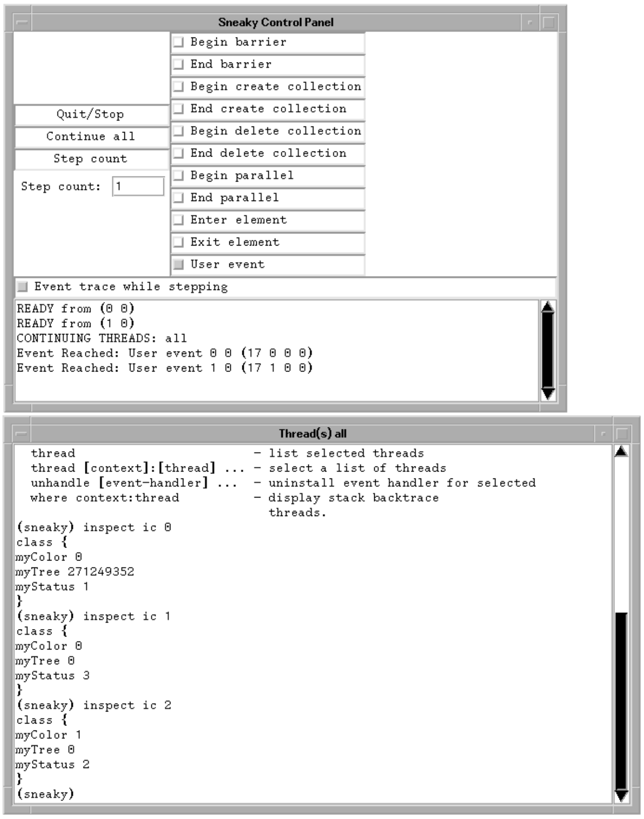

Figure 6: Sneaky output showing the values of the `ic` collection.

Looking at the second element of the collection, we saw an inconsistency: `myColor` was set at 1 indicating that this node represents a single, compressed, black tree but `myStatus` was set to 2 indicating that the merge had failed. Since it is the color of the node that determines its candidacy for merging at the next level, the color should have been set to a third value indicating "black and white, no merge." This failure to set `myColor` after an unsuccessful merge attempt was the error we were looking for.

Sneaky itself is a programmable, state-based debugger that we are currently developing; it is in the style of Acid [23], Dalek [20], and Parasight [1]. Many debugging abstractions are context and application dependent, making it difficult to anticipate all of the debugging needs of the user ahead of time. A programmable debugger alleviates this problem by allowing the user to build new abstractions, synthesizing complex events from sequences of simpler ones, as in Dalek [20]. Our programming language will be high-level, dynamically typed, with powerful data structuring capabilities and closures; the initial prototype uses Scheme [10]. Sneaky will include a large and ever-growing library of packaged functionality to aid rapid debugger extension.

5 Integration within the Tau Debugging Architecture

5.1 Overview

To integrate Ariadne and Sneaky, we made use of the TAU program interaction and control infrastructure, called *Sneezy*. Sneezy was motivated by the desire to support Sneaky, which needed an API to control the target program. For reasons of tool portability, interoperability, and reuse, we chose to isolate the entire debugger—as opposed to just its programming language—from the details of program control. To do this, we built an agent-based infrastructure in which Sneaky acts as the client of an agent as pictured in Figure 7.

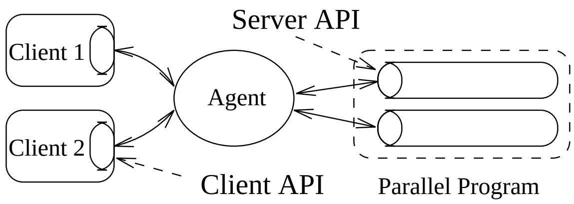

Figure 7: The structure of Sneezy.

Sneezy supports inspection and control of multi-threaded programs with multiple address spaces. The agent handles all communication between the client and the target program, making it possible, for example, for Sneaky (or any other client) to be unaware of whether it is running on the same machine as the target program or whether they are communicating via message-passing or shared memory. The infrastructure is quite general and it can be used to support a variety of tool interactions. In particular, Ariadne could also integrate into the environment as a Sneezy client, along with a variety of trace and data visualization tools.[3] Sneezy will support these tools as multiple *co-operating* clients as discussed in Section 5.3 below. Co-operation is important for replay debugging in particular, since a replay control module and a debugger both need to be present, but it is not natural for either to be part of the other. Other uses might include running a data visualizer along with a load monitor or debugger. Together, the clients of Sneezy will provide a comprehensive debugging environment. Figure 8 shows the beginning of that environment with a number of TAU tools working in concert with Ariadne and Sneezy during a debugging session.

[3] In the current prototype, Ariadne does not function as Sneezy client for performance reasons; we are investigating this aspect of the design further.

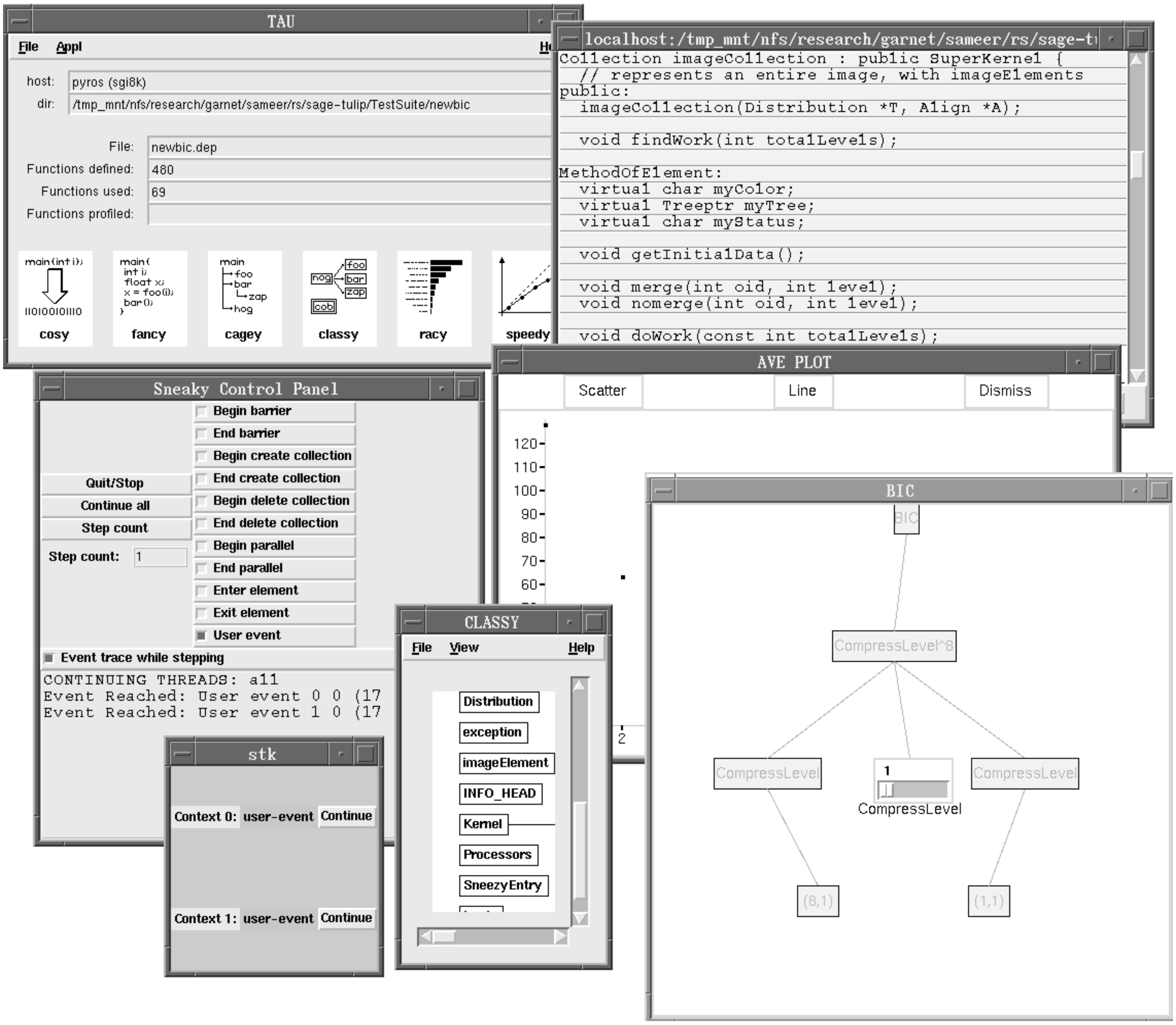

Figure 8: Debugging session using TAU tools. The figure shows a variety of tools in use. The Ariadne window shows a match tree and the Sneaky window a list of pC++ events with available event handlers. In the upper right corner the *Fancy* window shows the source code of the program being debugged. The upper left corner shows the TAU tools window which allows other tools (like *Cosy*, *Speedy*, *Racy*, *Cagey*, etc.) to be launched.

Sneezy's design is minimalist, making it lightweight, relatively easy to implement, and potentially simple to port and verify. Implementations of complex but common functionality will exist as portable libraries available to the clients. Other systems with functionality similar to that of Sneezy include Dynascope [21] and OMIS [18].

5.2 Event Model

The *Sneezy event model* defines the manner in which events, data, and control flow between the program, the agent, and the tools.

The computational model targeted by Sneezy is SPMD/MIMD. A computation is divided into *contexts* (address spaces), each of which has one or more *threads* of computation. In the current system, we assume that a barrier synchronization is program-wide,[4] but by adopting the concept of a *collector* as originally proposed for HPC++ [9], this restriction could be relaxed. Events occur at (thread,location) pairs. Clients enable and disable event pairs by invoking procedures in the agent, which performs instrumentation of the running program.

When an event occurs, notification is sent via the agent to the client. If the event carries data local to the thread, it is collected and sent along with the notification. The BeginRpc event, for example, carries a global pointer to the element which is the target of the service request, a global pointer to the parameter buffer, an integer which is the length of that buffer, and the function code of the remote function. Once an event is signaled, the running thread suspends its computation and enters a command loop controlled by the client (through the agent). The client can further instrument the thread, inspect its address space, and call functions in the thread's context. Eventually the client passes control back to the thread, which continues its computation.

Sneezy events most often correspond to language-level events in the source language (pC++), although they do not necessarily correspond to events in the source text because event points may be hidden in the runtime libraries. For example, if the BeginBarrier event is enabled, it will be signaled even if the barrier is inside a system subroutine. The use of "bracketed" (Begin/End pairs) events makes it possible to filter unwanted events. Thus, for example, a client may want to ignore all events between BeginCreateCollection and EndCreateCollection on the assumption that they are system-internal.

The commands currently supported by Sneezy include CONTINUE, TERMINATE, INSTALL-HANDLER, UNINSTALL-HANDLER, READ-DATA, WRITE-DATA, and CALL-USER-FUNCTION. Of particular interest is INSTALL-HANDLER, which takes a context ID, a thread ID, an event type, and an event ID. INSTALL-HANDLER enables the event in the given context and thread in such a way that when the event is signaled, its ID is sent to the client that installed the handler.

5.3 Multi-Client Event Model

We are in the process of implementing support for multiple, concurrent clients in Sneezy. There are a number of issues that must be addressed: how should we arbitrate when multiple clients have registered an interest in the same event? To what extent should a client be allowed to change the state of the computation? How should we balance the need for clients to be protected from one another's manipulations and with the need for some clients to co-operate? How can we insure that simple clients with simple needs (a visualizer, for example) can be implemented without being required to support complex inter-client protocols?

Initially, we addressed the issues of arbitration and co-operation by allowing only one of the clients to be the "master" of the program at any time, with other clients being slaves. Signaled events were reported only to the master who could choose to forward them on to the slaves. This approach was determined to be too coarse-grained and we refined it to allow master/slave designations on a thread-by-thread basis. While this was sufficient for most situations, the resulting design forced even simple clients to use the full multi-client protocol. After looking more carefully at the behaviors of our tools, we have now concluded that a multi-level mechanism that allows simple clients to ignore others and lets complex tools manipulate the event delivery mechanism in the agent would best serve our needs.

A Sneezy client can be classified along two dimensions: what it does to the program state, and what its intentions are when it issues the CONTINUE command. The full classification is beyond the scope of this paper, but in general, "nice" clients don't modify the state and don't care what happens after a CONTINUE command. Consider, for example, the situation where a visualizer, an event-based debugger, and a state-based debugger are all running together. The visualizer is a "nice" tool which does not need to know about the others; it can receive events as they occur, retain control briefly, and then pass the event and the control on to the next client (perhaps the event-based debugger) by issuing a CONTINUE command. The event-and state-based debuggers, however, may not be as "nice" in the sense that they interact in complex ways. Thus, for example, the event-based debugger may modify or filter events, passing them to the state-based debugger only as it sees fit. The state-based debugger, in turn, may actually modify program state variables and execution but may allow its command stream to be monitored and intercepted by the event-based debugger. The tools set up the priorities and the event and command filtering by manipulating data structures within the agent. The agent does not provide high-level mechanisms for computing the priority/filtering structure, but one can implement a Sneezy client which does, allowing other clients to communicate with this client to set up arbitration.

5.4 Implementation and Performance

The current Sneezy implementation supports SPMD-style systems with the client and agent communicating via sockets using either raw binary data or textual Lisp-syntax data. It is our intention that such distributed-object frameworks as CORBA [5] and ILU [12] could serve as implementation vehicles for Sneezy.

The programmer-visible part of Sneezy is realized as two APIs, one for the parallel program (*server API*) and one for the client (*client API*). These APIs depend on the agent implementation and the client language, respectively. For example, we have two client APIs: one for single-threaded C, and one for multi-threaded Scheme; their behaviors are quite different.

On the server side, The Sneezy/pC++ agent assumes that both the runtime system and the user program are appropriately instrumented; in the case of the user program, the pC++ compiler is responsible for adding calls to the appropriate server API procedures. When an event point is

[4] The current system works even if barriers are not program-wide, but there is no support for discovering the process set to which a partial barrier applies.

reached, the event signaling procedures are called, and the event is signaled to any interested clients. Alternatively, we could have had the agent perform the instrumentation at runtime by setting breakpoints in the program. This would have led to better performance and would have allowed the same program to be used both in debugging and production without recompilation. It would, however, have meant that the run-time system would need access to the symbol tables produced during compilation and it would have limited the attribute information typically carried by Sneezy events. Some of this attribute data might not be available at the event point without additional computation.

Our implementation allows communication optimization in various forms. We estimate that eventually, the cost of moving an event from the program to the client on a machine which supports shared memory will be a small number of procedure calls (< 5), some bookkeeping, and a memory-to-memory copy of a small structure. For further performance improvements, we will need to exploit the "closeness" of a client to its target program. In general, the client and the target do not even have to be on the same machines. Moving them to the same machine will allow the agent to optimize communication costs; moving them to the same address space would allow even further improvements. Thus in specializing the agent implementation we can expect to improve performance.

The performance of Sneezy on larger systems (where the client might be a bottleneck in terms of being a focus for messages from many nodes) is somewhat unclear. At some point we will have to consider a mechanism like that of Parasight [1], where lightweight parasites can be moved into the program and execute there for event filtering and generation. While such a mechanism is complex, it is general and can serve many purposes.

6 Conclusions and Future Directions

In this paper, we discuss the integration of an event-based debugger and a state-based debugger within the TAU program control and debugging infrastructure. The combination of approaches is quite powerful. It permits the programmer to use event-based techniques initially in investigating high-level patterns of process interactions and then, as the focus of attention narrows, to switch to a state-based approach in investigating lower-level execution details and relating them to source code constructs. This work extends the use of event-based modeling to object-parallel languages and provides an alternative mechanism for establishing meaningful global breakpoints in object-oriented languages. In addition, it introduces the TAU interaction and control infrastructure which enables Ariadne and Sneaky to interoperate as co-operating clients of a single agent.

The integration of Ariadne and Sneaky provides us with an environment for the assessment of mixed event- and state-based debugging strategies and for the evaluation of event-based modeling techniques in an object-parallel programming environment. In addition, it provides a basis for considering the effects of proposed extensions to object-parallel languages that include multithreading and multitasking.

In particular, multitasking exacerbates the difficulties of using standard mechanisms for setting breakpoints and makes it even more important to have the ability to set breakpoints based on high-level behavioral analysis. At the same time, replay and modeling become more difficult. Replay, in fact, will be prohibitively expensive in many cases and it will be difficult, if not impossible, for programs that make extensive use of library routines. We are currently investigating the possibility of replaying not the entire program but only partial traces of its behavior, limiting the observability of the code. Breakpointing will be relative to this observable behavior. Finally, the addition of tasking to the pC++ programming model will necessitate changes in Ariadne's behavioral modeling. Currently, parallelism is available only at the process level in Ariadne. pC++ contains parallelism at the element level as well but we were able to handle it within our models using iteration. This will not be possible in a more general thread-based model and it will not be possible with multitasking. Further extensions to Ariadne's modeling language that enable the natural expression of levels of parallelism is needed. In addition, we continue to work on the usability of Ariadne's interface and modeling language.

Acknowledgements. Bernd Mohr and Allen Malony designed and implemented much of TAU. Darryl Brown was involved in the early stages of the design of Sneezy; Kurt Windisch wrote much of Sneaky. George Forman, Alfred Hough, Calvin Lin, Lawrence Snyder, and David Stemple all contributed to the initial design of Ariadne.

References

[1] Z. Aral and I. Gertner, "High-Level Debugging in Parasight," *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN Notices 24(1):151-162, January 1989.

[2] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD Thesis, University of Massachusetts, Amherst, MA 01003, 1986.

[3] D. Brown, S. Hackstadt, A. Malony, and B. Mohr, "Program Analysis Environments for Parallel Language Systems: The TAU Environment," *Proceedings of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing,* pp. 162-171, May 1994.

[4] D. Brown, A. Malony, and B. Mohr, "Language-Based Parallel Program Interaction: The Breezy Approach," *International Conference on Performance Computing*, December 1995.

[5] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Framingham, MA, 1994.

[6] D. Gannon and J.K. Lee, "Object Oriented Parallelism: pC++ Ideas and Experiments," *Proceedings 1991 Japan Society for Parallel Processing*, pp. 13-23, 1991.

[7] A. P. Goldberg, A. Gopal, A, Lowry, and R. Strom, "Restoring Consistent Global States of Distributed Computations," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 144-154, 1991.

[8] A. A. Hough and J. E. Cuny, "Perspective Views: A Technique for Enchancing Visualizations of Parallel Programs," *1990 International Conference on Parallel Processing*, pp. II 124–132, August 1990.

[9] The HPC++ working group. "HPC++ Whitepapers and Draft Working Documents."

[10] *IEEE Standard 1178-1990. Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[11] Intel Supercomputer Systems Division. Paragon Interactive Parallel Debugger Reference Manual. Paragon Documentation, 1994.

[12] Inter-Language Unification. Available from `ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html`

[13] J. Kundu. *Integrating Event- and State- Based Approaches to Debugging of Parallel Programs.* PhD Thesis, University of Massachusetts, Amherst, MA 01003, 1996.

[14] J. Kundu and J. E. Cuny, "A Scalable, Visual Interface for Debugging with Event-Based Behavioral Abstraction," *Frontiers of Massively Parallel Computing '95.* pp. 472-479, 1995.

[15] J. Kundu and J. E. Cuny, "The Integration of Event- and State-Based Debugging in Ariadne," *Proceedings 1995 International Conference on Parallel Processing*, pp. II 130-134, August 1995.

[16] J. Kundu and J. E. Cuny, Integrated Event- and State-Based Debugging of Parallel Programs. Technical Report, University of Oregon, 1996.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), pp. 558–565, 1978.

[18] T. Ludwig, R.Wismüller, V. Sunderam, and A. Bode. "OMIS: On-line Monitoring Interface Specification, Version 1.0." February 1, 1996. Technische Universität München, Germany. Available from `http://wwwbode.informatik.tu-muenchen.de/~omis`.

[19] Y. Manabe and M. Imase, Global "Conditions in Debugging Distributed Programs," *Journal of Parallel and Distributed Computing*, Vol. 15, pp. 62-69, 1992.

[20] R. A. Olsson, R. H. Crawford, and W. Wilson Ho, "A Dataflow Approach to Event-based Debugging" Software–Practice and Experience 21(2), pp. 209-229, February 1991.

[21] R. Sosič. "Design and Implementation of Dynascope, a Directing Platform for Compiled Programs." *Computing Systems.* Vol. 8, 2, pp.107-134, Spring 1995.

[22] P. Scheuermann, A. Yaagoub, and M. Aris-Ouksel, "Compression of Binary Images on a Hypercube Machine," *Journal of Parallel and Distributed Computing*, Vol 23, pp. 49-59, 1994.

[23] P. Winterbottom, "ACID: A Debugger Built from a Language," *Proceedings of the Winter 1993 USENIX Technical Conference*, pp. 192-212, 1994.