A Thread-level Distributed Debugger[⋆]

João Lourenço José C. Cunha

{jml,jcc}@di.fct.unl.pt
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal

Abstract. *In order to address the diversity of existing parallel programming models, it is important to provide development environments that can be incrementally extended with new services. Concerning the debugging of process-based models, we have previously designed and implemented a basic interface that can be accessed by other tools as well as by debugging modules associated with high-level programming languages.*
In this paper we describe our work towards the support of further debugging functionalities for parallel and distributed programs, by discussing a model to support thread-based debugging services. We then show how those services are supported on top of a distributed monitoring and control software architecture.

1 Introduction

In order to ease the task of parallel and distributed application development, a debugging service must support the following aspects:

1. Inspection and control of the computation state;
2. Tool interfacing;
3. Heterogeneity.

There are several difficulties regarding the development of debugging services. On one hand, due to the large diversity of programming and computational models, it is not possible to define a universal debugging interface that can meet the requirements of all such models. On the other hand, there is an increasing number of applications which are composed of multiple separate components, each based on its own computational model, be it sequential or parallel.

So aspect (1) depends on each specific computational model, e.g. process-based, object-based, multi-threaded, as well as the underlying programming paradigm, e.g. imperative or declarative. At a basic level, as far as parallel and distributed debugging is concerned, the following entities should be modeled: processes, threads, and their interactions. Efforts such as the one from the HPDF initiative [BFP97] are currently trying to establish a standard interface for the most common basic debugging functionalities, that can hopefully improve the current situation.

Aspects (1) and (2) were addressed in our previous work [CL97,KCD$^+$97,LCK$^+$97], when we have developed a distributed process-level debugger (**`DDBG`**) for *`C/PVM`* programs. The **`DDBG`** debugger was integrated in a parallel software engineering environment within the scope of an European project [S$^+$94].

In both of the above situations, a debugging service must be able to handle the requirements of very distinct models, and this can be achieved through heterogeneous debuggers (aspect (3) above).

We have recently implemented a process-level debugging interface on top of a very flexible monitoring and control software architecture (**`DAMS`**) [CLV$^+$98]. One important aspect of this architecture is that it can be easily extended with new services and functionalities, such as for debugging, profiling, and distributed resource management. This allows an incremental development of tools and their experimentation with rapid prototyping.

In this paper we extend such debugging functionalities with a thread-based service, and show how it is implemented on top of the mentioned architecture.

The organization of the paper is as follows. Section 2 discusses process and thread-based debugging services, and Sec. 3 discusses implementations on top of the DAMS architecture. Then we discuss related work and present some conclusions.

2 Process and Thread-oriented Debugging Services

In order to provide debugging functionalities for process- and thread-based models, we must identify the basic concepts and mechanisms supporting inspection and control of the computation state. We define a model that is intended to be neutral concerning the diversity of semantics of existing process and thread-based models.

[⋆] In "*The 3rd International Meeting on Vector and Parallel Processing*," Porto, Portugal.

2.1 The Components of the Model

The model defines the following basic entities:

- *Processes.* A process is a passive entity, a kind of "capsule" supporting contexts for the concurrent execution of multiple threads. A context is defined by a non-empty set of cells. A process is completely specified by four types of "contexts":
  - *Process Memory Context.* It corresponds to the process address space which is represented by a set of values of accessible memory cells. Code, data and stack regions are mapped onto such memory cells.
  - *Process Synchronization Context.* It contains cells representing synchronization variables such as locks and mutexes, as well as condition variables. Of course they are also mapped onto memory cells but we prefer to separate them for greater clarity of the model.
  - *Process Communication Context.* It is represented by the values of the input/output ports and the communication channels (such as message queues). Such communication channels and input/output ports can also be modeled by associated memory cells, but they are explicitly identified here, because they describe the process interaction with its outside environment.
  - *Process Execution Context.* This is defined by the set of threads that execute within the scope of the process. Each such thread has a precise logical specification in terms of specific contexts, as described below. Additionally, each process has an associated Scheduling Context which describes the status of the physical processor scheduling for all threads (this is not further detailed in this paper).
- *Threads.* A thread is an active entity which executes some code within the contexts defined by its enclosed process. It is specified by two types of "contexts":
  - *Thread Memory Context.* It is defined by the set of values of the memory cells containing the code, the data, and the stack regions that were specified for each thread. Of course, both the code and data regions are shared by all threads in a single process, unlike the stack regions which must be kept private.
  - *Thread Execution Context.* This is defined by the status of the Virtual Processor that is associated with each thread in order to model its logical behavior. The status of a virtual processor is defined by the set of values of the processor registers, and by a logical state, a cell containing one of the values T_Running, T_Blocked, T_Stopped, T_Terminated.

The thread logical state transition diagram presented in Fig. 1 identifies the possible state transitions allowed to a thread, identifying at the same time some of the debugging functionalities that trigger each state transition. Associated with each transition in the state diagram there is a set of labels naming the possible causes of the transition. Their name suggest the associated functionality. Labels between angle braces, such as <`T_Exit`>, define actions resulting of the thread execution and generated internally or by the system. Other labels, such as `T_step`, identify transitions forced by an external agent, such as the debugger.

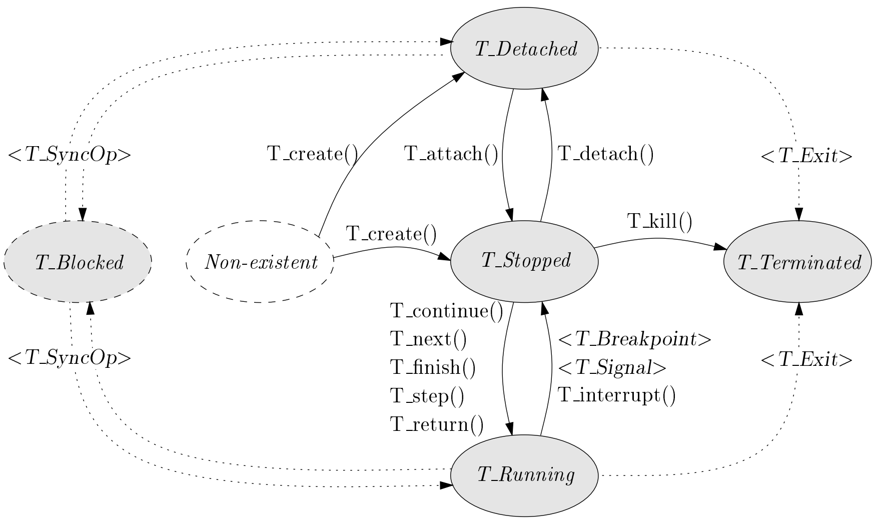

Fig. 1. The thread state transition diagram

– T_DETACHED. The thread is running free and it is not under control of the debugger.
– T_RUNNING. The thread is running under control of the debugger.
– T_STOPPED. The thread is stopped as a result of a debugger command or due to the occurrence of some exception.
– T_BLOCKED. The thread has invoked a blocking call and is temporarily blocked until that request is satisfied.
– T_TERMINATED. The thread has terminated due to a debugger or system command, or because it has reached its exit point.

2.2 Events

Using such a model, we are able to precisely identify the events which are relevant to describe and control a concurrent computation with multiple processes and threads.

In the following we briefly illustrate how this model can help in the process of precisely specifying the operational semantics of debugging primitives in terms of events.

Generally, given a specific Context (as previously defined above) an event is defined by a modification in a single value of a cell contained in that context. This corresponds to the basic notion that an event describes a transition from one state to another state.

Process-level Events These events describe all modifications made to any of the contexts defined in the process (Memory, Synchronization, Communication, and Execution). For example, events are triggered by modification of global process variables, by modifications of the state of a mutex, by the arrival of a message, or by the creation or destruction of a thread in a process.

Thread-level Events These events describe the modifications in the thread Memory and Execution contexts. For example, the modification of a local thread variable, or a physical processor register. Thread-level events are also triggered by any change in the logical state of its virtual processor.

2.3 Actions

An action is responsible for the state modification that triggers each event. We identify two classes of actions:

– *Internal Actions.* They are enforced by the virtual processor associated with a given thread in a process. The sequence of all pairs (Internal_action, Generated_Event) that are produced during thread execution, precisely specify the computation path followed by the thread. Such internal actions may correspond to physical processor instructions or to higher level instructions, for example C code statements.
– *External Actions.* They are enforced by external controller entities such as the debugger, acting upon the contexts defined within a process. The sequence of all pairs (External_action, Generated_Event) gives the history of a debugging session.

2.4 The Debugging Activity

Debugging functionalities fall into two categories: inspection commands, and controlling commands. On the other hand, they can refer to individual processes or threads. They can also refer to process interactions or thread interactions. The core of the debugging activity amounts to observe and/or enforce well-defined sequences of events so that deviations from the program specification can be localized and corrected. Our model provides a foundation to develop a mechanism that controls the detection and registering of events. Basicly event detection can be enabled for a well-defined class of action/event pairs. For example:

– Detect events in a given process/thread, associated with its Memory context, which were generated by internal actions only. It is possible to detect events associated with a given memory cell.
– Detect events in a given process, associated with its Synchronization context, and generated by internal actions of a given thread.
– Detect events in a given process/thread, associated with the logical state of its Execution context, and were generated by external actions.

In general it is possible to selectively enable/disable event detection for specific types through the specification of the following elements:

– Which class of action triggers the event (External, Internal).

– Which entity should be monitored (Process, Thread, Context, Cell).

Well-known debugging primitives can easily be represented in terms of this model. For example, concerning threads, a command “set_var()” of a local variable in a given thread would generate an event related to the Thread Memory Context. A command “set_breakpoint()” in a given thread would relate to the Thread Execution Context. Concerning processes, a command “kill_thread()” would relate to the Process Execution Context (and also to the Thread Execution Context because it also changes the thread state).

By monitoring the occurrence of events of a certain type, it is possible to construct event histories that contribute to a better understanding of the concurrent computation. For example, in order to implement a deterministic replay facility concerning process interactions only (i.e. message exchange), one needs to enable the detection of events related to the Process Communication Context. A replay facility for thread interactions internal to a single process depends on the enabling of events related to Process Memory and Process Synchronization Contexts.

2.5 Asynchronous Event Notification

Several types of debugging commands provide an immediate response, e.g. as in a “set_var()” or “set_breakpoint()”, which give a success or failure indication, and possibly return some result (e.g. a breakpoint identification).

Other types of debugging commands typically act upon Thread Execution Contexts in such a way that it is not possible to obtain immediate meaningful imformation, besides knowing that the command was successfully applied. For example, commands like “continue()” or “next()” immediately originate a logical state transition in a thread (from T_STOPPED to T_RUNNING), but it might take an unpredictable amount of time for the thread to reach a point that should be inspected during debugging, e.g. to reach a breakpoint. In general, the debugger interface or the application that is invoking debugging commands should not be forced to wait until the desired event is reached. Instead, an asynchronous event notification mechanism must be provided by the debugging interface, allowing a thread to explicitly register its interest in receiving *event notifications* through the declaration of an event handler.

This declaration is achieved by calling the service

```
T_sethandler (process_thread_list, event_type, handler)
```

which defines the function `handler` as an handler of events of the given type (according to the previous subsection) which are originated from any of the processes or threads from `process_thread_list`. Multiple threads in the same or different processes can register handlers for a specific type of events. If such event occurs, a notification is sent to all the registered threads. When a thread receives an event, its current execution is suspended while the associated handler function is executed.

This event mechanism is also used to support tool synchronization and coordination in an integrated software development environment where multiple tools (for debugging, testing, visualization, etc.) concurrently observe and control the evolution of a computation. This coordination is achieved by having some tools, e.g. a graphical user-interface or a thread-based visualization tool, registering handlers related to the occurrence of some types of events, that may be originated by internal and/or external actions (e.g. setting breakpoints). On event occurrence, such tools can react and update the graphical view that is being presented to the user, consistently with the evolution of the computation and with the actions triggered by the debugging tool.

2.6 Summary on the Debugging Functions

In this section we have discussed how an event-based model can provide the foundation to develop process and thread based debugging services. In this paper we have not presented the interface of process and thread debugging primitives. Our goal is to be able to support distinct and evolving interface primitives so that our debugging framework can be used to support experimentation and building of prototypes.

3 A Process- and Thread-oriented Debugging Tool

In this section we discuss implementation issues, including the support for multiple connections from concurrent client tools, as well as the infrastructure for implementing the debugging functionalities that we have outlined in the previous section.

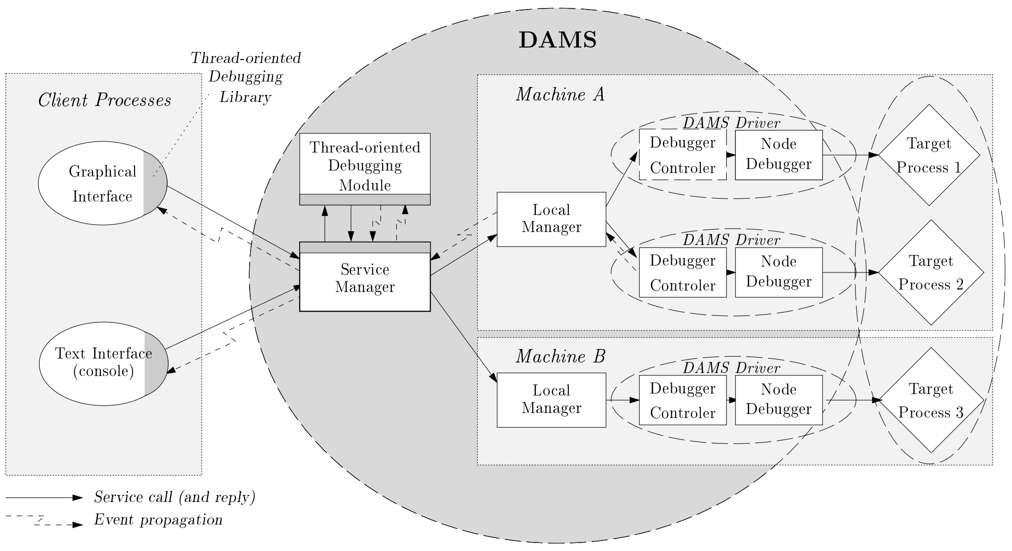

Fig. 2. The **TDBG** architecture

3.1 The **DAMS** system

The **DAMS** (*Distributed Application Monitoring System*) system provides the basic layer to support the incremental development of parallel and distributed monitoring and control services, such as debugging, profiling and resource management. Its design and implementation are neutral regarding the programming and computational models of the target application.

The processes related to **DAMS** can be classified in one of three classes (see Fig. 2):

– *Target application processes*. The set of processes that will be controlled/monitored by the **DAMS** system.
– *Client application processes*. The set of independent client tools, that may operate concurrently over the *Target application processes* by issuing requests to the **DAMS** system, through a service interface library.
– *The **DAMS** processes*. The set of internal processes that implement the **DAMS** system and its services. This set includes:
  • *System processes*. This includes a single Service Manager and several Local Managers, one per physical node of the target architecture. These processes manage the internal **DAMS** resources and provide an architecture independent communication layer that allows the *Client application processes* to control and inspect the evolution of the *Target application processes*.
  • *Service processes*. Each class of service (*e.g.* debugging, resource management, profiling) requires a **DAMS** configuration which includes a set of specific components: one Service Module, to handle the *Client application* service requests and their high-level system-independent parts; and a set of *Driver processes*, usually one per process of the *Target Application*, to implement the low-level system-dependent control and monitoring aspects.

The most important aspects of **DAMS** are: its extensibility; its neutrality concerning the target application models; its builtin support for multiple concurrent connections from client tools; and its functionalities for tool coordination and synchronization using events.

3.2 The **TDBG** tool

In [CLV+98] we have described the implementation of **PDBG**, a process-level debugger as a **DAMS** service. Here we describe how thread-level debugging (the **TDBG** debugger) is implemented as a service on top of the **DAMS** system by the provision of an adequate set of *Service processes*.

For a better understanding of how the **TDBG** components interact, we present an example, which also refers to Fig. 2. There are three *Target application processes*; two *Client applications*: the Graphical Interface and the Text Interface; and, for simplicity, the pictured **DAMS** configuration is providing the **TDBG** service only.

Let us consider that a client application (*e.g.* a Text Interface) issues a debugging command by calling a debugging library function, that sets a breakpoint in a given line of a given thread *e.g.* `set_break(t 12345, l 98)`. This function establishes the communication with the Service Manager, which identifies the type of requested service (related to debugging), and forwards it to the appropriate component: the Debugging Module.

The Debugging Module parses the received data, identifies the type of request, and then sends the (possibly) transformed request to the Debugging Driver. The **DAMS** system internally manages the routing tables to assure that the request reaches the desired Debugging Driver which is associated with the identified target process.

In order to allow easy plug-in of existing commercial or public-domain Node Debuggers, the Debugging Driver includes a front-end process, called a Controller, which is responsible for all interactions with the actual Debugger. The Controller acts as a kind of "user", as far as the Debugger is concerned[1].

After parsing the data that was sent by the Debugging Module, the Controller identifies the target process, and issues an adequate sequence of commands conforming to the existing Debugger interface *e.g.* `select_thread 12345`, `break_line 98`. The Controller waits for the completion of each command before issuing the next one. When the sequence is terminated, the results of the command, *e.g.* `local_brkpt_id=2`, are sent back to the Debugging Module.

The Debugging Module parses the received data, and does the necessary post-processing, for example converting a local breakpoint identifier into a global breakpoint identifier, *e.g.* `global_brkpt_id=14`. Afterwards, it sends the results back to the *Client process* in the form of return values of the invoked library call.

3.3 Summary on DAMS and TDBG

By describing how the interfacing between the client tools and the **TDBG** debugger is done, we have illustrated the great flexiblility of the **DAMS** architecture in order to support extended functionalities. Namely, it is possible to integrate multiple heterogeneous target debuggers, for processes and threads, in a single **DAMS** configuration.

4 Related Work

There are many current efforts on the field of parallel and distributed debugging (with and without thread's support) and related topics [LWSB97,Zho94,MB94,Lum95,XWZS96,PHK91,DJ88,HS88]. Because we cannot cover them all here, we have chosen two related approaches that are briefly presented and compared with our own approach. The first one concerns the specification of debugging functionalities and the second concerns a distributed design supported by an existing tool.

4.1 The HPDF (*proposed*) Standard

The High-Performance Debugging Forum (**HPDF**) [BFP97] is a collaborative effort between researchers and industry, aiming to define a standard for parallel debuggers. As of Version 1 of the standard, a command based (non-graphical) interface has been prepared, specifying either syntax and semantics of the proposed services. The definition of graphical interfacing and complex I/O operations are still under work.

According to **HPDF**, a parallel debugger is either a *thread-oriented debugger*, a *process-oriented debugger* or a *hybrid debugger*, and sets of required and recommended services have been defined for each of them. Our design can easily accommodate most of the **HPDF** proposed functionalities for hybrid debuggers.

In this regard, the tool integration features of **TDBG**, presenting an unified event-based model for the *internal* and *external* actions, is a distinct contribution to the integration of parallel debuggers in more complete and complex program development environments [KCD$^+$97,LCK$^+$97].

4.2 The p2d2 Distributed Debugger

The **p2d2** distributed debugger [Hoo96] is a *process-oriented debugger*. It uses a client-server approach, with a well defined interface, promoting portability by isolating the system dependent code into a debugger server. There is an user-interface capable of displaying and controlling many processes, individually or associated in groups. The GNU gdb is used as a Node Debugger, and a call-back method supports asynchronous interactions between gdb and the user-interface.

The distinctive feature of our approach (*i.e.* **TDBG**+**DAMS**) is to support multiple concurrent client tools and to offer the necessary mechanisms to implement client tool coordination.

[1] From an implementation point of view, the existing Node Debugger must provide an interface library to be accessed by the Controller front-end.

5 Conclusions and Ongoing Work

In this paper we have discussed a model to support the development of process and thread debugging functionalities, and their implementation as services of the **DAMS** distributed architecture. This work is part of our experimentation towards the incremental building of tool support services for parallel and distributed processing.

There is a prototype of **DAMS** running on our Ethernet LAN with Linux/PC's nodes, and a cluster of FDDI-interconnected Alpha processors under OSF/1. A process-level debugger (**PDBG**) runs as a **DAMS** service, and uses the GNU gdb as the target debugger. The efficieny of this prototype suffers because gdb is very *heavy*.

This prototype is being extended to implement **TDBG** which provides a thread-based debugging service according to the description in Sec. 3.2. A different Node Debugger is used, namely **SmartGDB** [Hal92], which is a *thread-oriented debugger*, extending GNU gdb with **TCL** scripting capabilities and debugging support for user-level threads.

An ongoing related project focus on the integration of **TDBG** and a visualization tool for thread-based programs. In this project we are experimenting with the **TDBG** tool integration and coordination support mechanisms.

Acknowledgements

Thanks are due to João Vieira, Bruno Moscão e Daniel Pereira for their work in the **DAMS** system.

The work reported in this paper was partially supported by the Portuguese CIÊNCIA Programme, by the PRAXIS XXI SETNA-ParComp (Contract 2/2.1/TIT/1557/95), and by the DEC EERP PADIPRO (Contract P-005).

References

[BFP97] J. Brown, J. Francioni, and C. Pancake. White paper on formation of the high performance debugging forum. Available in "http://www.ptools.org/hpdf/meetings/mar97/whitepaper.html", February 1997.

[CL97] J. Cunha and J. Lourenço. An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger. *Journal of Systems Architecture, 2^{nd} Special Issue on Tools and Environments for Parallel Processing, Elsevier Science*, 1997.

[CLV+98] J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, and D. Pereira. A framework to support parallel and distributed debugging. In *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98), Springer, LNCS vol. 1401*, pages 708–717, Amsterdam, The Netherlands, April 1998. Springer.

[DJ88] Thomas W. Doeppner, Jr. and David D. Johnson. A multi-thread debugger. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 295–297, Madison WI, May 1988. [Extended abstract].

[Hal92] Sudhir Halbhavi. Thread debugger—implementation and integration with the SmartGDB debugging paradigm. Master's thesis, University of Mysore, India, 1992.

[Hoo96] Robert Hood. The p2d2 project: Building a portable distributed debugger. In *Proceedings of the 2^{nd} Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia PA, USA, 1996. ACM Press.

[HS88] Gil Hansen and Andy Sheppard. Debugging multithreaded programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 310–312, Madison WI, May 1988. [Extended abstract].

[KCD+97] P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing, Elsevier Science*, 22(1997):1747–1770, 1997.

[LCK+97] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewsk. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23^{rd} Euromicro Conference (EUROMICRO'97)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.

[Lum95] Steve S. Lumetta. Mantis: A debugger for the split-C language. Technical Report CSD-95-865, University of California, Berkeley, March 1995.

[LWSB97] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Technical report, Lehrstuhl fur Informatik, Technical University of Munich (LRR-TUM), Munich, Germany, July 1997.

[MB94] John May and Francine Berman. Designing a parallel debugger for portability. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 909–915, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[PHK91] M. Krish Ponamgi, Wenwey Hseush, and Gail E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, 8(3):37–43, May 1991.

[S+94] S.Winter et al. Software Engineering for Parallel Processing, copernicus programme. Progress report 1, University of Westminster, London, UK, October 1994.

[XWZS96] Jianxin Xiong, Dingxing Wang, Weimin Zheng, and Meiming Shen. BUSTER: an integrated debugger for PVM. In IEEE, editor, *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICA PP '96*, pages 11–13, Singapore, June 1996. IEEE Computer Society Press.

[Zho94] W. Zhou. A layered distributed program debugger. In *Symposium on Parallel and Distributed Systems (SPDP '93)*, pages 665–668, Los Alamitos, Ca., USA, December 1994. IEEE Computer Society Press.