Grid-Level Computing Needs Pervasive Debugging

Rashid Mehmood, Jon Crowcroft, Steven Hand and Steven Smith
Computer Laboratory,
University of Cambridge, Cambridge, UK.
Email: {firstname.lastname}@cl.cam.ac.uk

Abstract— **Developing applications for parallel and distributed systems is hard due to their nondeterministic nature; developing debugging tools for such systems and applications is even harder. A number of distributed debugging tools and techniques exist; however, we believe that they lack the infrastructure to scale to large-scale distributed systems, systems with hundreds and thousands of nodes, such as Grids. In this paper, we introduce PDB, our prototype debugger, which is based on a hierarchical, scalable architecture. We explain the design of the PDB, highlight its functionality, and demonstrate its usability with two case studies. Before concluding, we discuss portability and extensibility issues for PDB, and discuss some solutions.**

I. Introduction

Developing debuggers for parallel and distributed systems is inherently difficult. The difficulty lies in the nondeterministic nature of these systems, caused by their autonomous structure, lack of a global clock, and communication delays. Distributed systems are typically sensitive to timing and synchronisation errors. Debugging them may even mask such errors due to a phenomenon known as the *probe effect* [1] or *Heisenberg's uncertainty principle* [2]. The heterogeneous structure of Grid and pervasive computing environments have exacerbated these problems.

Today, there exist a number of tools and techniques for debugging and profiling of distributed systems. With the increasing availability of large-scale distributed resources such as clusters and Grids, extensibility and scalability of these tools is becoming even more important. Many of these tools provide useful features, such as distributed system state and data visualisation, and program replay. We argue, however, that current distributed debugging tools lack an infrastructure which can be extended to the debugging of large-scale distributed systems comprising thousands of processors. This is due to the fact that most debugging tools are based on a single, centralised, *frontend* which controls and coordinates a number of *backend* debuggers or monitors. These conventional architectures usually attach a backend server on a per-process basis; a hundred processes need a hundred backends. Such architectures, inherently, are not suitable for debugging and profiling large-scale distributed systems; they can easily be overwhelmed by the volume of events or data produced by even a few tens of processes.

In [3], [4], the pervasive debugging approach for debugging distributed systems was introduced. It is pervasive in that it allows the complete state of the distributed system under test to be inspected and controlled by virtualising all the resources of a single physical machine. It utilises the machine virtualisation provided by the Xen VMM [5].

In this paper, we consider how to generalise the pervasive debugging approach to support debugging of large-scale systems, with hundreds and thousands of machines, as might be found in today's clusters and Grids. Our approach is a *hierarchical* one, which we believe allows the system to scale to much larger number than traditional systems. Moreover, our PDB incorporates an event-based approach to verify user-specified high-level properties of the distributed system under test.

The rest of the paper is organised as follows. In Section II, we briefly survey the related work, establishing the motivation for this work. We describe the functionality and architecture of the pervasive debugger in Section III and Section IV respectively. Section V discusses portability and extensibility issues for PDB and presents some solutions. Section VI presents two case studies to demonstrate PDB usability. Section VI reports the implementation status for PDB. Finally, we conclude and give future directions in Section VIII.

II. Background

A number of tools and techniques have been designed for debugging concurrent and distributed systems. In 1993, Pancake and Netzer [6] compiled a bibliography of parallel debuggers which included 293 entries. These techniques can be classified into four distinct categories [7]. At the forefront of these is the *traditional* parallel debugging techniques which are a natural extension of the *cyclical* approach to debugging sequential programs. Debuggers of this kind are also referred to as *interactive* or *breakpoint* debuggers. These debuggers permit users to repeatedly stop the execution of the program under test, examine the program state, and to continue with the execution. These are usually designed as a collection of sequential debuggers (e.g. GDB) with a single console which controls and coordinates the activities of independent sequential debuggers. Many well-known debuggers fall into this category, for example, research prototypes such as p2d2 [8] and Net-Dbx [9], and commercial debuggers, for example, TotalView [10] and DDT [11].

Another prominent approach to distributed debugging is the *event-based* approach (e.g. [12]–[17]) which relies on the generation and analysis of program *events* (also called *event history*). The definition of "event" varies, from memory accesses to MPI [18] send and receive functions. A number of directions within this category have been taken. Some systems present the collected raw event data to the user for direct inspection, and some [13] use graphical visualisation techniques (such as space-time diagrams [19]). Some event-based tools use event histories to guide re-execution of programs in the hope of reproducing program errors (e.g. [16]). Another direction taken is to compare event histories with a set of predicates in order to detect anomalous behaviour. Examples are the program behaviour model approach of [15], Event Based Behavioural Abstraction (EBBA) approach of [14], [20], and the IDD debugger [17].

Both the traditional and the event-based techniques suffer from the main problems associated with debugging of concurrent software: that is, the probe effect, nondeterminism, and the lack of global state visibility. A distributed system may not have a single notion of time, and hence precise ordering of events in the individual, concurrently executing processes may not be possible. Latencies and unpredictability in networks is a common cause of nondeterminism in distributed programs; the distributed computation cannot always be exactly reproduced for debugging. The autonomous behaviour of concurrent processes may cause *race condition* even in the absence of a network. Monitoring of a system may change the behaviour of a system, this is common in distributed system debugging due to the sensitivity of these systems to timing and synchronisation errors.

By contrast, static analysis techniques do not require program execution, do not suffer from the probe effect, and can be used for data and control analysis of a program. Model checking [21] is a formal static verification method to algorithmically verify finite state systems. This is achieved by verifying that a system model satisfies a logical specification, often written as a set of temporal logic formula.

All these debugging techniques have been augmented with graphical user interfaces and visualisation tools. An architecture which represents many of these conventional tools is given in Figure 1. Every process in the distributed system under test is attached to a sequential debugger or monitor, usually called the *backend* or *server*. These backends are controlled from a single, centralised console, called the *frontend* or *client*. The backends receive commands through the frontend to carry out debugger functions. In event-based debuggers, these backends may also deliver interesting events to the frontend.

A. The PDB Approach

In the previous section, we surveyed the main techniques and tools for the debugging of distributed systems. We observe that the current trend in parallel debugging tools is to combine multiple debugging techniques. Since each debugging method has its own deficiencies and advantages, it is only natural to integrate these techniques into a single tool. The integration of multiple techniques into a single distributed debugging tool is a challenging task, however, designing such a tool which is scalable to large-scale distributed systems debugging is even more challenging.

Consider the TotalView debugger [10], for example. In addition to the traditional debugging method, it provides a facility for dynamic call graph visualisation of a program. Furthermore, the user can set various actionpoints, e.g. *conditional watchpoints*. A watchpoint is typically created on a memory location to *watch* its contents. It generates an event when the contents of the memory location are changed. When the debugger receives such an event, it evaluates an expression (the condition) and reports to the user only if there is a change in the value of the expression. It is interesting to note here that, before the actual condition becomes true, the debugger may have to stop the execution of the program, and re-evaluate the expression, potentially thousands or even millions of times. This situation is very typical in a distributed debugging session. For a debugger which allows evaluation of expressions global to the entire set of processes in a distributed computation, this can create a huge traffic load within the distributed system.

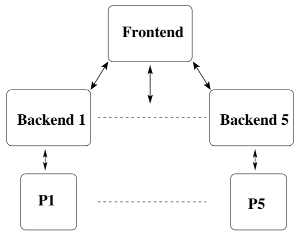

Fig. 1. A Conventional Distributed Debugger

To illustrate this further, suppose that we have a distributed application running 100 processes, and that we set a distributed assertion G, a logical conjunction of some function f on all the 100 distributed processes. That is, $G = f_0 \wedge f_1 \cdots \wedge f_{99}$, where f_n is an evaluation of the function f on the process n. Each time the value of the function changes on any process, the centralised debugger may have to request that each of the other processes report the current evaluation of the function. This may result in interrupting the execution of the whole distributed computation, and create a huge traffic load on the distributed system.

The PDB described in this paper, as we will see in Section IV, avoids this scenario by decomposing and distributing an assertion on distributed system to hierarchically placed intermediate or backend servers. To the maximum extent possible, these servers evaluate assertions locally, and only forward "interesting" events further up hierarchy. For instance, an intermediate server i might be responsible for nodes $0 \cdots 9$, and so could evaluate the distributed assertion $G_i = f_0 \wedge f_1 \cdots \wedge f_9$, forwarding the event to the main server only if the assertion G_i changes. Such a hierarchical approach localises and reduces the overall traffic, affords efficiency, and hence allows the system to scale to much larger number than traditional systems.

The pervasive debugging approach [3], [4] leverages the Xen VMM (see below) to virtualise the system resources (including network and disk) of a single machine. This approach can eliminate the probe effect, and can reproduce the exact behaviour of (i.e. can deterministically replay) a distributed system. In Section VIII, we discuss how can the PDB described in this paper accomplish the above-mentioned pervasive debugging features for large-scale distributed systems.

1) The Xen Virtual Machine Monitor: We are developing the PDB debugger prototype on a cluster of (physical) machines, each running the Xen virtual machine monitor (VMM) [5]. Xen is a VMM, or *hypervisor* for the x86 processor architecture. It permits execution of multiple virtual machines on a single physical system, and live migration of virtual machines between physical nodes of a cluster [22]. Xen is a paravirtualised [23] VMM which allows high performance but requires operating systems to be ported to execute on top of Xen. Currently, Xen supports Linux 2.4, Linux 2.6, NetBSD, FreeBSD, and Plan 9; a ReactOS port is in progress. A survey of machine virtualisation can be found in [24].

III. Functionality

We present here an overview of the functionality offered by the PDB. This will help us in explaining the PDB architecture in the next section.

The overall process of debugging works as follows. The user starts a debug session by starting the main PDB server, subsequently connecting to the backend servers, and attaching to processes of interest. Furthermore, the user, typically, registers to be notified of certain events. Hopefully, some interesting events happen and the user can examine the state of the distributed application to further the debugging process. The user may also specify assertions or high-level properties of distributed computation, which are verified against the actual execution of the computation; upon request, PDB notifies the user or performs arbitrary actions in case the assertion is violated. PDB supports the following:

A. Process View and Control

As in any standard debugger, the user can view or modify the memory of any process which is part of the distributed computation under test. Similarly, the user can stop, continue, or move through the distributed application instruction by instruction.

B. Primitive Events

PDB supports a set of primitive events:

- A *breakpoint* event is generated whenever the execution attains a certain point in a program.
- A *watchpoint* event is generated whenever a read or write access to a certain memory location in the process is made. PDB also allows watchpoint events on program registers such as the stack pointer.
- The user can also request *blocking* and *unblocking* events. These events are generated whenever the operating system (OS) kernel sets a process into a blocked or unblocked state. The kernel might put a process into a blocked state because the process is waiting for I/O or some event. These events can be used, for example, to track *deadlock* states of a distributed system (see Section VI-A).
- We are currently working on introducing primitive *send* and *receive* events. This will allow the user, for instance, to get notifications whenever a send or receive operation is initiated or completed. Events on Inter-Process Communication (IPC) mechanisms such as semaphore and shared memory are also planned.
- We also intend to support *timer* events which can generate events at a particular instance in time or after a specified amount of time has passed. These events, for instance, can be used to keep *soft state* of components or processes in a system, in order to detect their liveness properties or connection status.

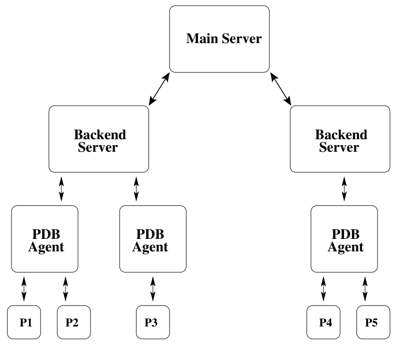

Fig. 2. Architecture of the PDB

Internally, PDB makes use of a reasonably generic publish/subscribe network. The user can request creation of *event sources* (i.e. publication), and can define arbitrary consumers. Consumer can subscribe to event sources, or can act as an event source for other consumers. When an event is generated, its consumers are notified, and they may then take any appropriate action, which may include generating further events, emailing the administrator or retrying the computation from a saved snapshot.

C. Distributed Assertions

PDB allows the user to specify logical assertion on a distributed system, e.g. "*no more than one process believes itself to be the leader*", and can verify the assertion against the actual execution of the system. An assertion is a high-level event composed of primitive events. To verify an assertion, PDB creates primitive event sources which generate events during the program execution. PDB processes these events to detect if the assertion is violated and informs the user-designated consumers. We will illustrate the use of high-level composite events to specify and verify distributed assertions in Section VI.

IV. PDB Architecture

The pervasive debugger, PDB, consists of three main types of components; a *Main Server*, one or more *Intermediate* or *Backend Servers*, and one or more *PDB Agents*. PDB places these components hierarchically in a distributed system. We

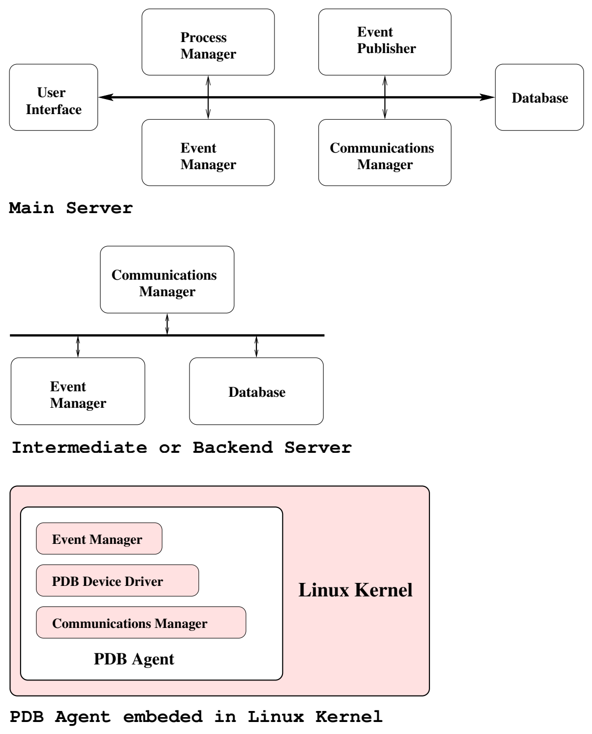

Fig. 3. The main PDB components

consider hereon the Backend Servers alone, and talk about Intermediate Servers in Section IV-B. Figure 2 depicts high-level PDB architecture and Figure 3 illustrates each PDB component.

The Main Server talks to the user and is responsible for overall management of the distributed processes and events. The Backend Servers process and manage events on a certain lower-level of the distributed system hierarchy. The PDB agent actually realises the user requests and enables creation of events.

Both Main and Backend Servers run as user-level processes. The Main Server can run anywhere, not necessarily on top of the Xen VMM. However, the Backend Server runs in a privileged virtual machine on top of the Xen hypervisor. The PDB agent is the only component of the debugger which is part of the OS kernel.

All the PDB components communicate asynchronously with each other. The Main and the Backend Servers talk to each other using TCP/IP sockets. The Backend Server and PDB agent(s) communicate using the mechanisms provided by the Xen VMM (see Section IV-B).

A. Main Server

The Main Server is a multi-threaded module written in Python. As depicted in Figure 3, it consists of six sub-components. The User Interface is responsible for accepting commands from the user and passing these to the appropriate sections. The Process Manager is responsible for initiating and controlling the processes. The Event Publisher is responsible for the creation of events, and for publishing them on request, i.e. making them available to potential consumers or subscribers. The Event Manager is responsible for the processing of received events, i.e. evaluating if these are of any concern to the consumers, dispatching the events to the consumers if appropriate, and informing the Process Manager in case there is an interesting change in the state of the processes. In addition, it controls subscription to events.

Main Server accepts commands from the user through user interface and routes them to appropriate Backend Server(s), if necessary. Backend Servers reply to the Main Server after carrying out the request. In addition to the request-reply sequence of messages, Main Server has to respond to the events delivered by the Backend Servers. It delegates communications related tasks, not surprisingly, to a submodule called Communications Manager. As mentioned earlier, the Main Server and the Backend Servers communicate using TCP/IP sockets. All the global data structures are kept in the central Database, access to which is via synchronised mechanisms.

B. Intermediate or Backend Server

Each physical machine participating in the distributed application runs an instance of the Backend Server. A Backend Server is in many respects a distributed debugger in its own right, except that it communicates primarily with the Main Server rather than directly with the user.

It is a multi-threaded module written in the C programming language which runs as a user-level process. Each instance of the Backend Server is responsible for keeping track of all the PDB agents and processes which are attached within its domain. The information about processes, PDB agents, various events on individual processes and possibly the relations (e.g. disjunction) among these events is kept in the Database. The Backend Server's Event manager is analogous to the one in the Main Server. The Communications manager in a Backend Server has to converse with the Main Server and to one or more PDB agents. Its responsibilities include:

- receiving requests from Main Server (e.g enable blocking events on process p),
- if necessary, forwarding requests to a PDB agent after translating the request to a form suitable for the PDB agent, and collecting replies from PDB agents and forwarding back to the Main Server.
- In addition, the Communications manager receives and buffers events from PDB agents, and dispatches outstanding events to the Main Server.

The Backend Server communicates to the Main Server over TCP/IP sockets. However, communication between the Backend Server and the PDB agents is over a Xen-specific medium. Xen allows virtual machines to communicate via *Descriptor Rings* and *Event Channels*. These two mechanisms collectively provide fast and secure means for asynchronous communication between virtual machines. Space constraints prevent us adequately discussing these here; see [25] for details. We note that the only part of the Backend Server which is specific to the Xen architecture is the Communications manager.

It would be possible to generalise the Backend Server design to make Intermediate Servers, such that the Intermediate Servers would be responsible to the Main Server, each coordinating and controlling several Backend Servers. This would allow a much deeper hierarchy to be constructed, potentially providing better event aggregation and hence improving traffic locality. In a Grid environment, for example, an Intermediate Server can be placed on the head node of a cluster to coordinate the debugging activities related to the processes executing in that cluster. This is a focus for ongoing work.

C. The PDB Agents

Each virtual machine in Xen which is participating in the debugging session runs an instance of the PDB agent. A PDB agent runs as a module within the Linux kernel, and therefore can provide enhanced visibility and fine-grained control of the processes. It is responsible for:

- providing information about processes,
- instrumenting processes to enable event generation, and
- actually controlling the execution of processes.

The PDB device driver exports virtual machine specific information to the Backend Server upon request. The Event and Communications managers within a PDB agent are analogous to their counterparts in the higher levels of the hierarchy, except that they have less work to do.

Note that a PDB agent is not the same as the backends (or servers) found in conventional debuggers: most conventional debuggers attach a backend to a single process, whereas a PDB agent can manage multiple processes.

V. Extensibility And Portability

Portability and extensibility are important characteristics of a distributed tool within large-scale resource sharing environments such as Grids. In this section, we evaluate these characteristics of the PDB.

A single, centralised main server is not sufficient if we wish to debug and control a distributed computation comprising hundreds and thousands of processes. We have introduced the notion of Intermediate Servers in Section IV-B. These servers can be placed between the Main Server and the Backend Servers to provide a second layer of event decomposition and aggregation. We plan to explore our PDB architecture based on multiple layers of Intermediate Servers.

An issue with a design based on multiple layers of Backend Servers is defining the namespace within the Main Server. Currently, a process is located with a 3-tuple (A, B, C), where A is the number given to the physical machine, B is the virtual machine number on the physical machine A, and C is the process running in virtual machine B of the physical machine A.

We now consider the portability issues. We have observed earlier in this paper that the only Xen-specific parts of the PDB design are the Communications manager submodule of the Backend Server and the PDB agent. The Backends communicate with the PDB agents using descriptor rings and event channels, provided by Xen as an efficient means for communication between virtual machines. It would be possible to support multiple modes of communication, for example TCP/IP sockets, between a Backend Server and the PDB agents. Similarly, the PDB agent can be structured as a self contained kernel module, allowing it to be introduced to existing operating systems with minimal disruption.

```
1. forever
2.     $sleep()$
3.     set waiting
4.     $pick\_left\_chopstick()$
5.     $pick\_right\_chopstick()$
6.     reset waiting
7.     $eat()$
8.     $put\_left\_chopstick()$
9.     $put\_right\_chopstick()$
```

Fig. 4. Algorithm: the dining philosophers

VI. Case Studies

In this section, using two case studies, we demonstrate how PDB could be applied to debug concurrent programs. Our first case study is the classical Dining Philosophers problem, which is often used to illustrate various concurrency related problems such as starvation and deadlock. Our second example is a *single program multiple data* (SPMD) solution for large sparse linear equation systems. Our experiences with developing such applications suggest that many times the errors in such programs are data related, for example, an invalid memory access. Deadlock tends to be less common, but is obviously possible and needs to be handled. Another issue in these programs is the fairness in interprocess communication. Usually, in these programs, data is partitioned and distributed among concurrent processes. In order to make progress, each process needs remote data from other processes, and hence sends the local data to other processes. If the algorithm is not designed carefully, some processes might wait longer than others. This may cause performance penalty for the overall program. Ability to resolve such problems is also important.

A. The Dining Philosophers

A dining philosopher algorithm is given in Figure 4. It is briefly explained as follows. There are an equal number of philosophers and chopsticks, where each philosopher has access to two chopsticks, left and right. In order to eat, a philosopher needs access to both chopsticks. Each philosopher (see the figure) first picks up the left chopstick and then the right. It sets a variable `waiting` before attempting to pick up the chopsticks, and resets the variable afterwards. After eating, it puts the chopsticks back on the table. This algorithm will sooner or later reach a deadlock state.

In the following, we give example Python code which can be used to specify a distributed assertion "philosophers_are_alive" on three philosophers. PDB can verify this assertion against the program execution and can take arbitrary action if the assertion is violated.

```
def phil_is_blocked($phil$):
    return $phil$.blocked() and $phil$.waiting == 1

def deadlocked():
    return phil_is_blocked($phil_1$)
        and phil_is_blocked($phil_2$)
        and phil_is_blocked($phil_3$)

def philosophers_are_alive():
    return $\neg$ deadlocked()

Alive = Assertion(philosophers_are_alive)
```

At the core of the example code above is the primitive Assertion. When a new assertion is created on a particular function, it first evaluates the function, noting as it does which aspects of the remote process state (e.g. memory locations) are being accessed. We then create primitive event sources (e.g. watchpoints) on each piece of state, and subscribe to them. When one of these primitive events are generated, we re-evaluate the condition. In this way, it is possible to specify a completely arbitrary assertion in a powerful, high-level language.

In this case, a blocking event source will be created for each process, and a watchpoint will be created on each `waiting` location. The assertion will be violated if at any point all three processes are blocked and waiting to acquire a chopstick: deadlock. Note that the above code assumes that the three processes ($phil_1, phil_2, phil_3$) are already attached to the PDB with these symbolic names.

In the following, we give example code to create a Consumer object "MyConsumer" on the function "ring_email_consumer". Then in the last line, we subscribe "MyConsumer" to the Assertion "Alive" which was created in the code above. Consequently, if the assertion is violated, PDB will *ring the bell* and will *email Alice*.

```
def ring_email_consumer():
    ring the bell, email Alice

MyConsumer = Consumer(ring_email_consumer)
MyConsumer.subscribe(Alive)
```

B. Solving Large Sparse Linear Equation Systems

The solution of large systems of linear equations is at the heart of scientific computing. Many problems such as forecasting, estimation, approximating non-linear problems in numerical analysis and integer factorisation, give rise to linear equation systems. Another example is automatic verification of probabilistic systems against some temporal logic specifications. We are actively involved in developing out-of-core and distributed solution of large sparse linear equation systems which arise from automatic verification problems, and we are using these as one of the test cases for our prototype PDB. We have reported steady state solutions of CTMC systems containing over a billion states [26]–[28].

```
1.  while error $> \epsilon$
2.      for $j = 1$ to $p$
3.          if $A_{ij}$ is not a $zero$ block
4.              send request for $x_j$
5.      accumulate sub-MVP $A_{ii}x_i$
6.      while true
7.          wait for incoming $message$
8.          if $message$
9.              if a request for $x_i$ from processor $j$
10.                 send $x_i$ to processor $j$
11.             else
12.                 receive $x_j$ from processor $j$
13.                 accumulate sub-MVP $A_{ij}x_j$
14.         if $my$ computations finished, break
15.     serve any remaining requests
16.     update $x_i$
17.     collectively calculate error
```

Fig. 5. An SPMD algorithm for node i

In Figure 5, we give an SPMD Jacobi iterative algorithm for the distributed solution of a linear equation system $Ax = 0$, taken from [29]. In this paradigm, all parallel processes run the same program but operate on different data. We implemented this program using an implementation of the MPI standard. The figure gives pseudo code for node i. We use *non-blocking* communication primitives to allow overlapped communication and computation. Each process remains in a loop while the convergence indicator `error` is larger than some predefined precision value ϵ. Note that the calculation of this variable is an expensive operation in a distributed system because its value depends on the newly calculated iteration vector which partially resides on each process. Given P processes, the iteration vector x is partitioned into P blocks; process i keeps and updates block x_i. The matrix A is *row-wise striped* partitioned into P^2 blocks and is also distributed among the processes; process i keeps all A_{ij}.

We now set a simple test case to demonstrate how PDB can be used to trace unintended behaviour of the progressing computation given in Figure 5. The Python code for the PDB test case is given in Figure 6. The user considers that a successively increasing or constant value of `error` is an indication of program malfunction. She creates a watchpoint ("WP") on the convergence variable `error` for a process p. Furthermore, She defines the function "count_set" to count the number of times, the convergence indicator successively increased or remained constant. She creates the consumer "MyConsumer" on this function ("count_set") and subscribes to the watchpoint "WP". In short, the overall process works as follows. Each update to the convergence indicator `error` generates an event. This event invokes an execution of the function "count_set". This function emails Alice if the convergence variable successively increased or remained constant for 100 times. Note that, in this case, `error` is only updated once per iteration, and has the same value on every process. This knowledge allowed the user to set a single watchpoint on a single process.

```
$global.error\_t = 10$
$global.count = 0$
def count_set():
    if $p$.error $\geq global.error\_t$:
        $global.count += 1$
        $global.error\_t = p$.error
    else:
        $global.count = 0$
    if $global.count == 100$:
        *Email Alice*

WP = WatchPoint($p$.error)
MyConsumer = Consumer(count_set)
MyConsumer.subscribe(WP)
```

Fig. 6. A test case to trace malfunction in the SPMD program

VII. Implementation Status

We have experimented with debugging distributed applications comprising a couple of dozen processes on a number of virtual machines. There are two main directions we are pursuing to advance the state of our PDB.

Firstly, work is being carried out to improve various functional and non-functional aspects of the PDB. The prime among these is our work on formalising and realising a scalable framework for specifying and evaluating distributed assertions. The work involves identifying a formal semantics for distributed assertions and formalising a consistent model for time and order. The notion of time and order is a major concern in distributed systems, and it is a nontrivial task. In the current implementation, an assertion in PDB is modelled as a tree where leaves represent events, primitive or composite, and nodes represent relations between the events. The Main Server decomposes an assertion into subtrees and distributes these to the relevant Backend Servers. The backends evaluate their individual subtrees and forward interesting events to the Main Server, which on receiving these events, evaluates its own (high-level) tree. We are investigating other computational models for this purpose, and the work is in progress. We are also working on the syntax and semantics of the specification language presented in Section VI, and it is likely to change in future.

The other direction of our research is to increase the size and variety of distributed applications, in order to make real progress in debugging large-scale distributed system. We are working with the CamGrid project [30]. The aim of the CamGrid project is to build a university-wide Grid across the University of Cambridge, UK. From this collaboration, we expect to develop PDB on large-scale CamGrid resources, and expect to find diverse distributed applications which can be tested on our PDB.

VIII. Conclusion and Future Work

In this paper, we introduced PDB, our prototype debugger, which provides a hierarchical, scalable architecture by delegating tasks to hierarchically placed intermediate or backend servers. Distributed assertions are decomposed and distributed to the backend servers, and to the maximum extent possible, these servers evaluate assertions locally, and only forward "interesting" events further up hierarchy. This approach localises and reduces the overall traffic, and affords efficiency and scalability.

We discussed the functionality and architecture of PDB, demonstrated its usability and discussed portability and extensibility issues. We also surveyed a number of debugging techniques. The most promising approach we believe is the event-based assertion checking technique. In this technique, the user specifies an assertion and the debugger verifies it against the program behaviour. In this context, we presented two example test cases in Section VI. In order to make the PDB tool attractive to a casual user, it will be necessary for us to build a library of commonly used test cases, i.e. consumers and distributed assertions. It is anticipated, however, that this will occur naturally as we continue to develop and test the tool using varied distributed applications. Currently, our assertion language and the underlying computational model are not powerful enough for the user to completely specify the intended program behaviour. This will be considered in the future.

A problem with the traditional debugging approaches is that these can only be used to detect the presence of errors but not their absence. In recent years, model checking has emerged as a promising approach for automatically verifying that a system meets its specification. However, it suffers from the *state space explosion* problem, i.e. the size of the state space of a system can be exponential in the size of the system. Another disadvantage of the model checking approach is that it verifies a model, not the actual system, and that it requires substantial efforts to construct a model of a system. The approach of building and verifying program behaviour models as demonstrated in this paper has the advantage that it works on the actual program. In future, we intend to investigate a combination of the two approaches.

A pervasive debugger for large-scale distributed systems in itself is a complex distributed system overlayed on top of the host distributed platform. It requires usual services such as communication, concurrency, time, synchronisation, loadbalancing, reliability and quality of service (QoS). For example, the ability to deterministically replay a distributed system and to capture a consistent global state of the system at an arbitrary point in time are desirable features of a distributed debugger. These features, however, are extremely difficult to attain, if not impossible, for a large-scale distributed system. Given a time model of a distributed system, the system can be executed in virtual time, and if necessary, checkpointing and replay can be used to globally synchronise its distributed state. This can also eliminate the probe effect. If QoS is also available, the core debugger (control) messages can be given a higher priority, and possibly an upper bound on transmission delay for these messages can be determined. This can minimise the amount of computation to be rewound, and/or the states to be logged. Similarly, a distributed debugger should be able to automatically instantiate and administer its components and adapt to the dynamics of the host distributed system (in the current implementation, PDB components are statically instantiated and managed).

The pervasive debugging approach advocates that such services and functions, along with the core PDB software, should be embedded within a single debugging and verification framework. Essentially, this involves building a virtual distributed system overlayed on top of a distributed host platform. We appreciate the scale of such work, and therefore intend to leverage existing tools and services to build our PDB for large-scale systems. In this respect, some interesting ideas and work on network overlays and distributed virtualisation can be found in [31]–[33], being realised into PlanetLab [34]. The PlanetLab – a virtual testbed as well as a deployment platform – provides a promising approach towards developing and deploying global-scale tools, services, and infrastructures. PDB can also take advantage of the PlanetLab and its existing services, making it possible to develop, test, and deploy a large-scale distributed pervasive debugger. Similarly, PDB can also utilise tools and services generated from Grid computing research.

Acknowledgement

This work is supported by the EPSRC "Pervasive Debugging" Grant GR/S63113/01 and by an Eclipse Innovation Grant from IBM.

References

[1] J. Gait, "A probe effect in concurrent programs," *Softw. Pract. Exper.*, vol. 16, no. 3, pp. 225–233, 1986.

[2] C. H. LeDoux and J. D. Stott Parker, "Saving traces for ada debugging," in *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*. New York, NY, USA: Cambridge University Press, 1985, pp. 97–108.

[3] T. Harris, "Dependable Computing needs Pervasive Debugging," in *Proceedings of the 2002 ACM SIGOPS European Workshop*, 2002.

[4] A. Ho, S. Hand, and T. Harris, "PDB: Pervasive Debugging With Xen," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, 2004.

[5] P. Barham, B. Dragovic, K. Fraser, S. H. T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[6] C. M. Pancake and R. H. B. Netzer, "A bibliography of parallel debuggers, 1993 edition," in *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*. New York, NY, USA: ACM Press, 1993, pp. 169–186.

[7] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, 1989.

[8] R. Hood and G. Jost, "A debugger for computational grid applications," in *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*. Washington, DC, USA: IEEE Computer Society, 2000, p. 262.

[9] P. Neophytou, N. Neophytou, and P. Evripidou, "Debugging mpi grid applications using net-dbx." in *European Across Grids Conference*, ser. Lecture Notes in Computer Science, M. D. Dikaiakos, Ed., vol. 3165. Springer, 2004, pp. 139–148.

[10] "TotalView. Product Brochure, Natick MA, 2003."

[11] "DDT, The Distributed Debugging Tool. Product Brochure, Allinea Software Ltd, Warwick, UK."

[12] D. Kranzlmuller, "Dewiz - event-based debugging on the grid," in *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP 2002)*, 2002.

[13] C. Schaubschlager, D. Kranzlmuller, and J. Volkert, "Event-based program analysis with dewiz," in *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003.

[14] P. C. Bates and J. C. Wileden, "High-level debugging of distributed systems: The behavioral abstraction approach," *Journal of Systems and Software*, vol. 3, no. 4, pp. 255–264, 1983.

[15] M.Auguston, "Building program behavior models," in *Proceedings of the European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning*, Brighton, England, August 1998, pp. 19–26.

[16] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. 36, no. 4, pp. 471–482, 1987.

[17] P. Harter, D. Heimbigner, and R. King, "Idd: an interactive distributed debugger," in *Proc 5th International Conference on Distributed Computing Systems*, Denver, CO, May 1985, pp. 498–506.

[18] "The Message Passing Interface (MPI) standard," http://www-unix.mcs.anl.gov/mpi/index.htm.

[19] B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan, "The paradyn parallel performance tools and pvm," in *Environments and Tools for Parallel Scientific Computing*. SIAM Press, 1994.

[20] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM Trans. Comput. Syst.*, vol. 13, no. 1, pp. 1–31, 1995.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[22] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, May 2005, boston, MA.

[23] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," University of Washington, Tech. Rep. 02-02-01, 2002.

[24] J. E. Smith and R. Nair, "An overview of Virtual Machine Architectures," 2003.

[25] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams, "Safe Hardware Access with the Xen Virtual Machine Monitor," in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004, boston, MA.

[26] R. Mehmood, "Serial Disk-based Analysis of Large Stochatic Models," in *Validation of Stochastic Systems: A Guide to Current Research*, ser. Lecture Notes in Computer Science, vol. 2925. Springer-Verlag, 2004, pp. 230–255.

[27] ——, "Disk-based techniques for efficient solution of large markov chains," Ph.D. dissertation, Computer Science, University of Birmingham, UK, October 2004.

[28] M. Kwiatkowska, D. Parker, Y. Zhang, and R. Mehmood, "Dual-processor parallelisation of symbolic probabilistic model checking," in *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, 2004.

[29] R. Mehmood, "Out-of-Core and Parallel Iterative Solutions for Large Markov Chains," School of Computer Science, University of Birmingham, UK," PhD Progress Report 3, October 2001.

[30] M. Calleja, B. Beckles, M. Keegan, M. A. Hayes, A. Parker, and M. T. Dove, "CamGrid: Experiences in constructing a university-wide, Condor-based, grid at the University of Cambridge ," in *Proceedings of the 2004 UK e-Science All Hands Meeting*, 2004.

[31] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," in *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[32] L. Peterson, A. Bavier, M. Fiuczynski, S. Muir, and T. Roscoe, "Towards a Comprehensive PlanetLab Architecture," PlanetLab Consortium, Tech. Rep. PDN–05–030, June 2005.

[33] L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet Impasse Through Virtualization," in *Proceedings of the 3rd ACM Workshop on Hot Topics in Networks (HotNets-III)*, November 2004.

[34] "Planetlab: An open platform for development, deploying and accessing planetary-scale services," http://www.planet-lab.org/.